UNIVERSIDAD DE MÁLAGA

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

# Optimization Techniques for Automated Software Test Data Generation

Ph.D. Thesis Dissertation in Computer Sciences


*Author*

***Francisco Javier Ferrer Urbano***

*Supervisors*

***Dr. Enrique Alba***
*and*
***Dr. Francisco Chicano***

*Department of*
**Lenguajes y Ciencias de la Computación**

**UNIVERSITY OF MALAGA**


January 2016



AUTOR: Francisco Javier Ferrer Urbano

http://orcid.org/0000-0002-1074-0139

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga

Departamento de Lenguajes y Ciencias de la Computación
Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

El Dr. **Enrique Alba Torres** y el Dr. **Francisco Chicano García** pertenecientes al Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga,

**Certifican**

que, D. **Francisco Javier Ferrer Urbano**, Ingeniero en Informática por la Universidad de Málaga, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo su dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada:

# *Optimization Techniques for Automated Software Test Data Generation*

Revisado el presente trabajo, estimamos que puede ser presentado al tribunal que ha de juzgarlo. Y para que conste a efectos de lo establecido en la legislación vigente, autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

En Málaga, Enero de 2016

Fdo: Dr. Enrique Alba Torres y Dr. Francisco Chicano García

*A mis padres, Virgilio y María José,*
*por la educación y el cariño*
*que me han dado siempre*

## Agradecimientos

Para comenzar me gustaría agradecer a todas las instituciones e investigadores que han contribuido en mi desarrollo hasta ver plasmada esa evolución en este volumen de tesis doctoral.

Agradezco en primer lugar a mis directores Enrique y Francis, que me han guiado de la mejor forma posible para conseguir este objetivo tan deseado. Me alegra mucho haber podido ser capaz de devolver parte de lo que la sociedad me ha brindado en forma de conocimiento científico. Han sido muchos los momentos que he vivido junto a ellos en este periodo; por encima de todo me gustaría resaltar su faceta personal, ya que el trabajo siempre se ha realizado en el mejor ambiente posible.

Durante este tiempo han sido muchos los compañeros que han pasado por el laboratorio desde donde estoy escribiendo estas líneas. Cada uno de ellos me ha aportado lo mejor, y sólo espero haber podido captar lo máximo de ellos, puesto que no existen mejores compañeros que los que he tenido. Agradezco a Gabriel, Paco, Guillermo, Juanjo y José Manuel, siempre dispuestos a ayudarte y de los que he aprendido mucho. A los compañeros con los que empecé esta aventura Briseida, Pablo, y Martín, a los cuales les deseo lo mejor. También agradezco a los que nos hemos unido durante este tiempo y compartimos nuestro laboratorio: Jamal, Yesnier, Daniel, Raúl y recientemente Christian y Francisco. Con todos ellos este camino recorrido ha sido mucho mejor, y espero que lo siga siendo en esta nueva etapa como doctor. Quisiera mencionar también a Juan Miguel y Antonio, que me han prestado su ayuda con amabilidad siempre que lo he necesitado.

NEO es un grupo donde acogemos cada año a una gran cantidad de investigadores externos, lo que me ha dado la posibilidad de conocer a Sergio, Franco, Martín, Carolina, Andreas, Mauricio, Javier, Sofiene, Karel, Roberto, Darrell, Manuel, Zakaria, David y muchos otros, todos excelentes profesionales y grandes personas. A ellos les doy las gracias por abrirme las puertas de mundo global desde un laboratorio de Málaga.

Mi más sincero agradecimiento a Joachim y Peter, que me acogieron en mi estancia en Berlin en lo que ha sido una de las mejores experiencias de mi vida. Esta estancia supuso un crecimiento profesional y sobre todo personal.

En estos momentos me acuerdo del sacrificio que hicieron mis padres, Virgilio y María José, para enviarme a estudiar a Málaga. Espero haber aprovechado bien la oportunidad y que estéis orgullosos de mí, porque yo no tengo palabras para agradeceros todo lo que habéis hecho. También agradezco a mis queridos hermanos Eva, Fernando, Alejandro y Virginia, cuyo ejemplo me ha servido desde el día en que nací. A mis abuelos y tíos, en especial a mi abuela María y mi tía Mercedes que son tan importantes para mí, gracias por hacerme sentir tan querido.

Ahora que cierro una etapa, quiero agradecerle especialmente a una persona que ha estado a mi lado desde que empecé esta aventura del doctorado, mi prometida Myriam. Voy a emprender un nuevo camino lleno de incógnitas en la vida profesional, sin embargo no me preocupa, ya que estando contigo ese camino estará lleno de felicidad.

## Acknowledgements

This PhD thesis has been partially funded by the Spanish Ministry of Economy and Competitiveness (MINECO) and the European Regional Development Fund (FEDER), under contract TIN2014-57341-R moveON project (http://moveon.lcc.uma.es). It has been also partially funded by the University of Malaga, under contract UMA/FEDER FC14-TIC36. Finally, the author is supported by a FPI grant with code BES-2012-055967 from MINECO.

# Contents

**Acknowledgements** **iv**

**1 Introduction** **1**
1.1 Motivation . . . 1
1.2 Objectives and Phases . . . 2
1.3 PhD Thesis Contributions . . . 3
1.4 PhD Thesis Organization . . . 4

**I Fundamentals of Software Testing and Metaheuristics** **7**

**2 Fundamentals of Software Testing** **9**
2.1 Structural Testing . . . 10
2.2 Structural Testing Problems Addressed in this Thesis . . . 11
2.2.1 Test Data Generation Problem . . . 11
2.2.2 Multi-Objective Test Data Generation Problem . . . 14
2.3 Functional Testing . . . 15
2.4 Functional Testing Problems Addressed in this Thesis . . . 17
2.4.1 Prioritized Pairwise Test Data Generation Problem with Classification Tree Method . . . 17
2.4.2 Test Sequence Generation Problem with Extended Classification Tree Method 20
2.4.3 Pairwise Test Data Generation Problem in SPL . . . 23
2.4.4 Multi-Objective Test Data Generation Problem in SPL . . . 26
2.5 Conclusions . . . 27

**3 Fundamentals of Metaheuristics** **29**
3.1 Formal Definition . . . 29
3.2 Classification of Metaheuristics . . . 32
3.2.1 Trajectory Based Metaheuristics . . . 33
3.2.2 Population Based Metaheuristics . . . 34
3.3 A Methodology for Evaluating Results . . . 36
3.3.1 Quality Indicators . . . 36
3.3.2 Statistical Analysis Procedure . . . 41

**4 Algorithms** **45**
4.1 Mono-objective Metaheuristics Used in this PhD Thesis . . . 45
4.1.1 Genetic Algorithm . . . 45
4.1.2 Evolutionary Strategy . . . 47
4.1.3 Ant Colony Optimization . . . 48
4.2 Multi-objective Metaheuristics Used in this PhD Thesis . . . 49
4.2.1 Non-dominated Sorting Genetic Algorithm II . . . 49
4.2.2 Strength Pareto Evolutionary Algorithm 2 . . . 49
4.2.3 Multi-Objective Cellular Algorithm . . . 50
4.2.4 Pareto Archived Evolution Strategy . . . 51
4.2.5 Random Multi-Objective Algorithm . . . 52

**II Structural Testing** **55**

**5 Test Data Generation in Object-Oriented Software** **57**
5.1 Introduction . . . 57
5.2 Test Data Generator . . . 58
5.2.1 Objective Function . . . 59
5.2.2 Instrumentation Tool . . . 60
5.3 Distance for instanceof operator . . . 61
5.4 Experimental Setup . . . 63
5.4.1 Algorithm Details . . . 63
5.4.2 Mutation Operator . . . 63
5.4.3 Benchmark of Test Programs . . . 64
5.5 Experimental Analysis . . . 64
5.5.1 Preliminary Results . . . 65
5.5.2 Uniform vs. Distance-based Mutation . . . 66
5.5.3 Adaptive Mutation . . . 68
5.6 Conclusions . . . 69

**6 Estimating Software Testing Complexity** **71**
6.1 Introduction . . . 71
6.2 Static Measures . . . 72
6.3 Branch Coverage Expectation . . . 76
6.3.1 Markov Chain . . . 76
6.3.2 Definition of the Branch Coverage Expectation . . . 77
6.4 Validation of the Branch Coverage Expectation . . . 80
6.5 Empirical Validation Setup . . . 82
6.5.1 Algorithms Details . . . 82
6.5.2 Program Generator Tool . . . 83
6.5.3 Benchmark of Test Programs . . . 85
6.6 Empirical Results . . . 87
6.6.1 Analysis of the Correlation Between the Static Measures . . . 87
6.6.2 Correlation Between Coverage and Static Measures . . . 89
6.6.3 Another use of the Branch Coverage Expectation . . . 92
6.6.4 Validation on Real Programs . . . 94
6.7 Conclusions . . . 96



**7 Multi-Objective Test Data Generation** **99**
7.1 Introduction . . . 99
7.2 Experimental Methodology . . . 100
7.2.1 The *MM* Approach . . . 100
7.2.2 The *mM* Approach . . . 101
7.2.3 Benchmark of Test Programs . . . 103
7.3 Experimental Analysis . . . 103
7.3.1 Evaluation of the *MM* approach . . . 104
7.3.2 Evaluation of the *mM* approach . . . 107
7.3.3 *MM* vs. *mM* approaches . . . 110
7.3.4 Validation on Real Programs . . . 114
7.4 Conclusions . . . 115

**III Functional Testing** **119**

**8 Combinatorial Interaction Testing using Classification Tree Method** **121**
8.1 Introduction . . . 121
8.2 Prioritized Pairwise Test Data Generation using CTM . . . 122
8.2.1 Solution Approaches . . . 123
8.2.2 Experimental Benchmark . . . 125
8.2.3 Comparison between GS, PPC and PPS . . . 127
8.2.4 Comparison between Genetic Solver and other existing algorithms . . . 129
8.3 Test Sequence Generation using ECTM . . . 131
8.3.1 Algorithms Details . . . 132
8.3.2 Experimental Setup . . . 135
8.3.3 Test Sequences' Quality . . . 137
8.3.4 Test Suite Coverage versus Test Suite Size . . . 139
8.4 Conclusions . . . 141

**9 Pairwise Testing in Software Product Lines** **145**
9.1 Introduction . . . 145
9.2 Parallel Prioritized Pairwise Testing . . . 146
9.2.1 Algorithm Description . . . 146
9.2.2 Weight Priority Assignment Methods . . . 147
9.2.3 Experimental Setup . . . 149
9.2.4 Experimental Analysis . . . 150
9.3 Seeding Strategies for Multi-Objective Pairwise Testing . . . 152
9.3.1 Seeding Strategies . . . 153
9.3.2 Evaluation . . . 156
9.3.3 Experimental Analysis . . . 158
9.4 Optimal Multi-Objective Pairwise Testing . . . 160
9.4.1 Mathematical Linear Program . . . 161
9.4.2 Algorithm Details . . . 162
9.4.3 Experimental Setup and Analysis . . . 163
9.5 Conclusions . . . 164



**IV Conclusions and Future Lines of Research** **167**

**10 Conclusions and Future Work** **169**
10.1 Conclusions . . . 169
10.2 Future Work . . . 171

**Appendices** **173**

**A Publications Supporting this PhD Thesis Dissertation** **177**

**B Resumen en Español** **181**
B.1 Introducción . . . 181
B.2 Organización de la Tesis . . . 182
B.3 Fundamentos . . . 184
B.3.1 Pruebas de software . . . 184
B.3.2 Metaheurísticas . . . 184
B.4 Problemas Abordados en esta Tesis . . . 185
B.5 Generación de Datos de Prueba en Programas Orientados a Objetos . . . 186
B.6 Estimando la Complejidad de Probar un Programa . . . 187
B.7 Generación de Datos de Pruebas Multi-objetivo . . . 188
B.8 Pruebas Combinatorias usando el Método de Clasificación de Árboles . . . 189
B.9 Líneas de Productos Software . . . 190
B.10 Conclusiones . . . 191

**List of Tables** **195**

**List of Figures** **198**

**List of Algorithms** **201**

**Index of Terms** **202**

**References** **205**



# Chapter 1

# Introduction

## 1.1 Motivation

Most countries in the world depend on complex computer-based systems which govern the main infrastructures and utilities, as well as most electrical devices used daily. Therefore, producing and maintaining software is essential for the actual society and means always a major challenge for computer scientists [220]. Research on computer science is an expanding field that involves the understanding and design of computers and their software. The main objective in this field is the study of automating algorithmic processes that scale to eliminate bottlenecks. One of the most relevant aspects of research in computer science is the design and development of new efficient algorithms able to solve complex problems decreasingly running times [160]. Moreover, researchers are determined to tackle problems that were uncomputable in a reasonable amount of time in the past, as their contribution to the field.

Real world problems are in general hard problems (NP-hard in most cases), what means in practice that the computation time spent in finding the optimal solution is growing exponentially with its size [105]. When exact techniques are used to solve these problems, they ensure to find the optimal solution for relatively small problems, but they are extremely slow in medium to large size problems. Most interesting real world problems have large solution spaces subject to a set of restrictions and uncertainties, then difficult to solve for exact algorithms. With this aim in mind, researchers have applied emerging techniques such as metaheuristic search techniques which have been proven to be successful facing hard problems [2, 35]. Metaheuristic approaches are able to provide a near-optimal solution in a moderate time lapse, so they offer a good trade-off between quality and cost, what is in fact, the main goal of industrial companies.

Software Engineering is an engineering discipline that is concerned with all aspects of software production, from the early stages of system specification to maintaining the system after it has gone into use [192]. Developing trustworthy software is a key challenge for the Software Industry, therefore the testing phase is very important in the software development process. In fact, it is estimated that half the time spent on the software project development and more than half its cost, is devoted to testing the product [158]. The automation of test generation could reduce the cost of the whole project. This explains why both, Software Industry and Academia, are interested in automatic tools for testing. As the generation of adequate tests implies a big computational effort, *search-based* approaches are required to deal with this problem.



In this thesis dissertation we apply metaheuristic search techniques to optimization problems derived from the automation of the testing process, particularly the automatic generation of test data for finding bugs in the source code [147]. Throughout this PhD thesis we try to encompass the most important testing paradigms, white-box as well as black-box testing. In white-box, also called structural testing, the testers require the source code of the implemented software. In contrast, in black-box or functional testing the testers are focused on what the software does, instead of how it does it.

Overall, we propose metaheuristic techniques to solve the different variants of the automated software test data generation problem. We analyze several techniques with the aim of obtaining optimal results in quality and cost, both aspects very important in the software development because of the lack of resources. Due to the intrinsic character of this problem, we have proposed single-objective optimization techniques to maximize the quality of the test suite and multi-objective techniques considering at the same time the quality and the cost of the test suite. The understanding of the insights of a problem is a very goal. In this particular problem, project managers have to estimate the effort needed to perform this important task of the software development. Hence we propose a new complexity measure to predict in a better way the difficulty to test a piece of source code. In this way, this PhD thesis contributes with experimental and theoretical proposals to the domain of modern software testing.

## 1.2 Objectives and Phases

The main objectives in this PhD thesis are the following: study the main problems in software testing, apply metaheuristics for solving the software testing problems, analyze the results to measure the quality of the proposed approaches, propose new approaches for the studied problems, and contribute with novel ideas to the research field. These global objectives have been decomposed in more concrete partial objectives:

1. Identification of the most interesting open testing problems in the related literature.
2. Formal definition of the selected testing problems.
3. Description of the algorithms used to solve these optimization problems.
4. Application of search-based techniques to the selected problems.
5. Demonstration of the effectiveness through statistical evaluation.
6. Analysis of the results and conclusions extracted from them.

In order to reach this PhD thesis' goals we have followed the *Scientific Method* [79]. The first step is the *observation*, and so we have studied software testing problems and identified techniques that the scientific community is using to solve them. From this analysis, we have extracted the main shortcomings found in literature. The second step is making a *hypothesis*: here we propose new techniques to solve these shortcomings like different algorithms, operators, representations, and static measures of the source code. The *experimentation* is the third step, we design and perform fair experiments always comparing the techniques with the same number of evaluations. The fourth step is the *refute or support* of the claim, we analyze the obtained results applying statistical tests to know whether the proposed technique is better or worse than others. Then, we

extract *conclusions* after our research work and further research actions are outlined. In this PhD thesis we have confirmed our hypotheses made about several software testing problems. After this dissertation, the cross-fertilization between research fields have made it possible to solve problems with techniques that were not applied before. In addition, all scientific work must be reproducible by other scientists, this work is not an exception, so we provide all details of our proposals. We must highlight that we show the results of our studies in a clear structured simple way to be understandable by any other researcher of the scientific community.

## 1.3 PhD Thesis Contributions

The contributions of this PhD thesis are mainly related to the research in the software testing field, both in structural and functional testing. We have addressed interesting issues that improve the state-of-the-art techniques, meanwhile we provide new efficient solutions for open challenges. These contributions can be summarized as follows:

- Definition of a new distance measure to compute the branch distance in the presence of the `instanceof` operator in Object Oriented programs. Proposal of two mutation operators based on this previous definition.
- Definition of a new complexity measure based on a Markov model of a program, the *Branch Coverage Expectation*. This measure is aimed at providing some knowledge about the difficulty of testing programs.
- Theoretical prediction of the number of test cases needed to cover a concrete percentage of the program, computed from the control flow graph of the program.
- Proposal of a whole test suite approach for solving a multi-objective test data generation problem and comparison with a mono-objective approach followed by a test case selection.
- Comparison of different prioritization strategies to first test the most important functionalities in Software Product Lines and Classification Trees.
- Definition of the Extended Classification Tree Method to completely describe all aspects needed to generate sequences of tests for testing a software system.
- Exploration of the effect of different seeding strategies in the computation of the Pareto fronts considering test suite quality and oracle cost in Software Product Lines.
- Proposal of an exact technique for the computation of optimal Pareto fronts considering test suite quality and oracle cost in Software Product Lines.

In addition, a number of scientific articles have been published during the years in which this PhD thesis has been developed that support and validate the impact of these contributions on the scientific community and literature. These publications have appeared in impact journals and fora, summing up 21 research papers: 3 journal articles indexed by ISI JCR, 1 international journal article (not in JCR), 1 book chapter and 1 conference paper published in LNCS Series, 12 conference papers and 2 technical papers available in CoRR repository (references to these publications can be found in Appendix A).



## 1.4 PhD Thesis Organization

This PhD thesis document is structured in four parts and two appendices. The first part is devoted to present the fundamentals and basis for the work: what is software testing and the software testing problems tackled, an explanation of metaheuristics as resolution techniques, the algorithms used (throughout this PhD dissertation), and the methodology that we have employed for assessing and validate the numerical results. Second and third parts are devoted to the most important testing paradigms that are structural and functional testing, respectively. In the second part we address the automatic test data generation problem, we propose a new complexity measure for estimating the difficulty to test a piece of code, and we solve the bi-objective test data generation problem considering test suite quality and cost as equally important goals, in the context of white-box testing. The third part is devoted to explore the black-box version of the test data generation problem, that is without any information of the source code. In this part we deal with two representations of the system under testing, classification tree method and feature models. The fourth part recaps the main conclusions drawn throughout the work and outlines our future research lines. Finally, the appendices contains the set our related works that have been published during the years in which this PhD thesis has been carried out and a summary in Spanish language In the following we detail the content of each forthcoming chapter:

- **Part I. Fundamentals of Software Testing and Metaheuristics**
  - Chapter 2 introduces the main concepts of software testing, emphasizing structural and functional testing, the two main paradigms of software testing. After that, the problems addressed in this PhD thesis are formalized in order to provide the reader with the exact details of the problems we are solving.
  - Chapter 3 provides a generic description of the research field of Optimization and Metaheuristics, including a classification of the main techniques. The last part of this chapter concentrates on how to properly evaluate their results, including quality indicators and the standard statistical validation employed in all our experiments.
  - Chapter 4 introduces the most important versions of the algorithms used throughout this PhD thesis, including single-objective and multi-objective techniques. The specific implementation details (like the operators for mutation or crossover) which are problem-specific (or representation specific), are deferred to the corresponding chapters.
- **Part II. Structural Testing**
  - Chapter 5 deals with inheritance for generating test data in Object Oriented source code. This chapter proposes a distance measure to compute the branch distance in the presence of the `instanceof` operator and two mutation operators that change the candidate solutions based on the distance measure defined. In addition to the proposals we have performed a set of experiments to test our hypotheses comparing against uniform mutation.
  - Chapter 6 contains the definition of a new complexity measure based on a Markov model of a program, the "Branch Coverage Expectation". This measure is aimed at providing some knowledge about the difficulty of testing programs. After that, the theoretical and experimental validity of the measure is evaluated using the framework proposed by Kitchenham et al. [120].



  - Chapter 7 starts by describing two different approaches for tackling the multi-objective test data generation problem. It analyzes the performance of a direct multi-objective approach versus the application of mono-objective algorithms followed by a test case selection. Our approaches consider test suite quality and oracle cost as optimization goals. In contrast to previous results in the literature that have only focused on the coverage of a program, while the oracle cost is a significant cost that has been ignored in most of the previous studies.

- **Part III. Functional Testing**

  - Chapter 8 explores a series of issues related to the Classification Tree Method. We analyze the prioritization of test data to test first the most important functionalities. After that, we define an entire model (Extended Classification Tree Method) which both industry and academia could use to completely describe all aspects needed to generate sequences of tests for testing a program. Our proposals have been successfully implemented in the CTE XL professional tool, what adds value to our work.
  - Chapter 9 addresses the application of metaheuristic techniques to Software Product Lines testing. Throughout this chapter we fill several existent gaps in the SPL literature: we compare a parallel genetic solver with an state-of-the-art algorithm with pairwise coverage as adequacy criterion, explore the effect of different seeding strategies, propose classical multi-objective algorithms and a new exact algorithm to compute the true Pareto front considering test suite quality and oracle cost.

- **Part IV. Conclusions**

  - Chapter 10 contains a global review of the PhD dissertation, and revisits the main conclusions drawn. The research objectives and main contributions are then discussed in view of the results obtained. Lastly, the future lines of research are briefly sketched and discussed.

- **Appendices**

  - Appendix A presents the set of related works that have been published during the years in which this PhD thesis has been carried out.
  - Appendix B is a summary of this volume in Spanish.



# Part I

# Fundamentals of Software Testing and Metaheuristics



# Chapter 2

# Fundamentals of Software Testing

Software engineering is a computer science area whose focus is the cost-effective development of high-quality software systems [177]. Contrary to other elements studied in other engineering fields, software is abstract and intangible, which is, on the one hand positive because it is not constrained by physical laws. But, on the other hand, the lack of physical limitations on the potential of the software makes it extremely complex and hence very difficult to understand. Moreover, software is never used on its own but always as a part of some broader system including hardware, people and, often, organizations.

Software engineering focuses on the study and application of engineering to the design, development, and maintenance of software. However, nowadays there are many companies that still do not apply software engineering techniques effectively, too many projects produce software that is unreliable. After codification, software products require a test phase with the aim of finding errors and to ensure software correctness.

Software testing could be defined from two different points of view [215]. The first defines software testing as the process of evaluating a system or its components with the intent to find whether it satisfies the specified requirements or not - positive view. The second claims that software testing consists in executing a system in order to identify any gaps, errors, or missing requirements in contrast to the actual requirements - negative view. In practice, testing will combine elements of positive and negative testing, not only checking that a system meets the requirements, but also trying to find errors in the source code. Consequently, the testing phase consists in creating a set of test cases to execute the *software under test* (SUT), then the tester has to check whether the output of the execution is correct or not, follows the specification or not.

Testing is not a new concept at all, Romans had already used this concept when evaluating the quality of precious metals in the first century. This concept is still valid, though it is evolving from manual to automatic testing due to the software complexity. Let us draw an example of how difficult is testing even a simple function that classifies triangles[1]. Suppose the arguments of the function are three integers of 32 bits and the computation execution capacity is 1,000 tests per second, then the exhaustive testing of this function will last 2.5 quintillion years ($2.5 * 10^{18}$ years). Obviously, nobody would test such a function like that, but it is a good example to show that brute force approaches are useless dealing with this problem. The importance of the testing phase and the high economic impact of an inadequate testing infrastructure were detailed in a survey by

[1]This function uses the lengths of the triangle to classify triangles in types: equilateral, isosceles, scalene, and no-triangle.

Tassey [197]. In a software project it is estimated that roughly 60% of costs are development costs and 40% are testing costs. Moreover, for custom software, evolution costs often exceed development costs [192]. Therefore, the automation of test data generation is desirable and could reduce the cost of the whole project.

Automatic test data generation (*automatic software testing*) consists in proposing an adequate set of test data in an automatic way to test a program, thus preventing the engineer from the task of selecting an adequate set of test data to test the SUT. Therefore, this automation of the process requires the automatic selection of adequate test data. This is an optimization problem where the algorithm has to pick the best solution (test suite) among a number of feasible ones, and it can be formulated as a search problem [47] (see Section 3.1). In all types of engineering there are a large number of optimization problems, software engineering is not an exception. In fact, the term Search Based Software Engineering (SBSE) was coined by Harman and Jones [97] to refer to this research field that combines search algorithms with software engineering problems. The automatic generation of test data is the most widely studied topic [95, 147] among SBSE problems and Evolutionary Algorithms (EAs) are the most popular search-based algorithms to face this problem. Then, the research community use the term *evolutionary testing* to refer to this approach.

There are two complementary paradigms for testing a piece of code that rely on the knowledge the tester has of the internal structure of the software: structural testing (white box) and functional testing (black box). On the one hand, structural paradigm uses information about how the SUT is constructed for generating test cases [113]. This structural information generally comes from the control flow graph of the SUT, in particular from the control structures (decisions) that lead to the different branches of the program. This technique is typically used during the early stages of the testing process where the programmer is in charge of executing the test suite [215]. On the other hand, functional tests are designed without any information of the structure of the source code [38,39]. In this paradigm the design of the test cases must be based on the external behavior of the SUT. This technique is typically used when the SUT is implemented by 3rd-party developers, the source code is unavailable, or the entire system is tested. Throughout this PhD thesis we deal with the test data generation problem using structural and functional techniques, so in the rest of this chapter we make a deeper analysis of these two testing paradigms.

## 2.1 Structural Testing

The most relevant techniques for the automatic generation of test data in structural testing include symbolic execution [43], random testing [18], and search-based testing [99]. In symbolic execution the program's code is analyzed to automatically generate test data. Although this technique has been proven to be useful, it suffers problems that limit its effectiveness on real-world software [14]. Random testing is a competitive testing option only under particular circumstances [18], but it is definitely a good technique to compare with as a sanity check. The problem of automatically generating test inputs is NP-hard [147], so researchers seek to identify test suites that obtain near optimal coverage in reasonable time. For this reason, search-based testing is receiving more and more attention in the last few years [99]. Search-based techniques have revealed that they are the best option to automatically generate test data with a high trade-off between quality and cost. The most popular search-based algorithms for generating test data are EAs [77,147] such as genetic algorithms [8,12,89,168,195,222], evolutionary strategies [5,74], or genetic programming [1, 11, 40, 179, 214]. Other search-based techniques like Tabu Search [60–62], Scatter Search [34, 60, 184], Estimation of Distribution Algorithms [183], Particle Swarm Optimization [140], Simulated

Annealing [223], and Hybrid algorithms [213] have been also applied to solve this problem.

In structural testing the internal structure of the source code is essential, so different approaches have been presented depending on the particularities of the programming paradigm elements used to develop the SUT. In the related literature there are studies on procedural programming [62,98,128,140], object-oriented programming [19,73,78,179], and even aspect-oriented programming [9]. In particular, different elements of the structure of a program have been studied in detail. Researchers have analyzed different transformations on the source code to make it more testable, namely testability transformations [150]. Occasionally, there are source code elements that may cause inefficiency in the generation of test data, so they require some transformation, for example, the presence of flags in conditions [25], the coverage of loops [61], the existence of internal states [231], the presence of exceptions' treatment [201], the nesting problem [151], and the use of inheritance [73].

A key idea behind structural testing is the instrumentation of the SUT to measure coverage of some structural criteria. The most popular adequacy criterion for the quality of the generated test suite is branch coverage [44], though other structural criteria have been proposed. For example, loop coverage [61], control-flow coverage [209], data-flow coverage [83], statement coverage [170], or decision coverage [223]. In addition, several objective functions have been suggested [26, 185, 206, 224] including the use of penalties [180], normalization [16], approximation distance [216], and branch distance [125] in the so-called fitness function.

The solution of the test data generation problem is a set of test data whose execution is able to cover all possible software elements. Most works cited above follow this mono-objective approach. However, real-world engineers deal with the tedious and costly task of checking the system behavior for all the generated test cases. This significant and usually neglected cost is called the oracle cost [152]. Therefore, a reformulation of the problem as multi-objective have been proposed [98, 129] to take into account coverage and oracle cost as equally important objectives. When trying to cover all branches of the SUT, every single branch is a goal. Traditionally, most works focus on only one branch at a time, some others use an aggregated function to deal with all at the same time, namely whole suite approach [77], and more recently, all single branches could be considered as objectives at the same time [169], so it requires a multi-objective optimization technique.

This great research effort has led to the development of automatic tools for the generation of test data using a search-based engine. For example, CUTE [188] a concolic unit testing engine for C, DART [87] a directed automated random testing tool, AUSTIN [128] an open source tool for search based software testing of C programs, EvoSuite [80] an automatic test suite generator for Java, and many others. Some of the proposed tools have succeeded in testing safety-critical [202] and real time software [217].

## 2.2 Structural Testing Problems Addressed in this Thesis

In this section we formally define the structural testing problem addressed in this dissertation. We tackle the Test Data Generation Problem and the Multi-objective Test Data Generation Problem, the multi-objective version of the same problem.

### 2.2.1 Test Data Generation Problem

In *software testing* the engineer selects an initial set of configurations or inputs for the SUT, called test suite, and s/he checks the SUT behavior with them. In order to ensure the correctness of

a program with this technique, it would be necessary to execute the SUT with all the possible configurations, but in practice this is unfeasible. The alternative consists in testing the program with a representative set of test data. Let us define some important concepts related to the test data generation problem.

The *Control Flow Graph* (CFG) of a program P is a directed graph $G_p = (I, L)$ in which $I$ is the set of program's statements (nodes) and $L$ is the set of pairs such as $(i, j) \in L$ if after the execution of statement $i$, statement $j$ can be executed. We denote with $in(i)$ and $out(i)$ the in-degree of the node $i$ and the out-degree of the node $i$, respectively. In this graph there is only one node with entry degree zero which is denoted by $i_0$, $in(i_0) = 0$. This is the initial node of the graph, it represents the first statement of the program. There is also a special node $i_* \in I$ with out-degree zero $out(i_*) = 0$, it represents the end of the program. We denote with $V_P$ the set of variables of the program $P$. Each variable $v_i \in V_P$ can take values in a particular domain $D_i$ with $i = 1, \ldots, |V_P|$. There is a special variable $pc \in V_P$ that is the program counter, it takes values from the set $I$. A state $\rho$ of the program $P$ is an application that maps variables to domains:

$$\rho \quad : \quad V_P \rightarrow \bigcup_{i=1}^{|V_P|} D_i \tag{2.1}$$

$$v_i \quad \mapsto \quad \rho(v_i) \in D_i \tag{2.2}$$

We say that a state $\rho$ is an end state if $\rho(pc) = i_*$ and $\rho$ is an initial state if $\rho(pc) = i_0$. Let $\varepsilon_p$ denote the set of all possible states of a program $P$. We assume the existence of a state converter $S_p : \varepsilon_p \nrightarrow \varepsilon_p$. This function maps each state $\rho$ to the state $\rho'$ obtained after executing statement $\rho(pc)$.

An *execution* of a program $P$ with initial state $\rho$, denoted with $Exec_P(\rho)$, is a sequence of states $\{\rho_n\}$ where $\rho = \rho_1$ and for all $i \geq 1$, then $\rho_{i+1} = S_P(\rho_i)$. Each execution $\{\rho_n\}$ of the program $P$ determines a path in the control flow graph $G_p$. This path is formed by a succession of nodes $\{s_n\}$ where each element $s_i \in I$ is defined as $s_i = \rho_i(pc)$. Let $Proj_P(\rho_n)$ denote the sequence of nodes $\{s_n\}$ determined by the sequence of states $\{\rho_n\}$, that is the projection of the execution in the control flow graph. We denote with $Sta_P(\{s_n\})$ the set of elements in $I$ that are part of the sequence $\{s_n\}$. Finally, a *test datum* for a program $P$ is the initial state of the program and a *test suite* is a set of test data.

**Definition 2.2.1** (Statement Coverage)**.** *Given a test data set $T$ for a program $P$, we define the* statement coverage *of $T$, $covSta_P(T)$, as the ratio between the number of statements executed in the executions of the program with initial states given by $T$ and the total number of statements in the program, that is,*

$$covSta_P(T) = \frac{|\bigcup_{\rho \in T} Sta_P(Proj_P(Exec_P(\rho)))/\{i_*\}|}{|I/\{i_*\}|} \tag{2.3}$$

Now, we can define the *Statement Coverage* adequacy criterion, which establish that a set of test data $T$ for a program $P$ is adequate when $covSta_P(T) = 1$.

Conditional statements (if-then, switch, for, and while) are characterized in the control flow graph by having an out-degree greater than one. We call $CCF_P$ to the set of nodes of the control flow graph that represents this kind of statements, that is, $CCF_P = \{i \in I | out(i) > 1\}$. An arc of the control flow graph $(i, j)$ is a branch of the program $P$ if the tail of the arc belongs to the set $CCF_P$. We denote with $B_P$ the set of branches of the program P, i.e.,

$$B_P = \{(i,j) \in L | out(i) > 1\}, \tag{2.4}$$

and with $B_R(\{s_n\})$ the set of branches that appear implicitly in the sequence of nodes $\{s_n\}$:

$$B_R(\{s_n\}) = \{(i,j) \in B_P | \exists k \geq 1, i = s_k \wedge j = s_{k+1}\}. \tag{2.5}$$

**Definition 2.2.2** (Branch Coverage)**.** *Given a test data set $T$ for a program $P$, we define the* branch coverage *of $T$, $covBr_P(T)$, as the ratio between the number of branches taken in the executions of the program with initial states given by $T$ and the total number of branches in the program, that is,*

$$covBr_P(T) = \frac{|\bigcup_{\rho \in T} B_{RP}(Proj_P(Exec_P(\rho)))|}{|B_P|} \tag{2.6}$$

Therefore, the *Branch Coverage* adequacy criterion establish that a test data set $T$ for a program $P$ is adequate when $covBr(T) = 1$. Finally, we remark a well-known proposition in the test data generation domain, that is,

**Proposition 2.2.3.** *Let $P$ be a program and $T$ a test data set for $P$. If $covBr_P(T) = 1$ then $covSta_P(T) = 1$. A total branch coverage implies a total statement coverage*[2]*.*

Before we can enunciate the test data generation problem as an optimization problem, we need to formally define an optimization problem. Assuming, without loss of generality, a *minimization* case, the formal definition of an *optimization problem* is as follows:

**Definition 2.2.4** (Optimization problem)**.** *An* optimization problem *is defined as a pair $(S, f)$, where $S \neq \emptyset$ is called the solution space (or search space), and $f$ is a function named* objective function *or* fitness function*, defined as $f : S \to \mathbb{R}$ .*

*Solving an optimization problem consists in finding a solution $x^* \in S$ such that:*

$$f(x^*) \leq f(x), \qquad \forall\ x \in S\ . \tag{2.7}$$

Note that assuming either maximization or minimization does not restrict the generality of the results, since an equivalence can be made between the two cases in the following manner [22, 88]:

$$\max\{f(x) | x \in S\} \equiv \min\{-f(x) | x \in S\}\ . \tag{2.8}$$

Depending on the domain $S$ belongs to, we can speak of *binary* ($S \subseteq \mathbb{B}^*$), *integer* ($S \subseteq \mathbb{Z}^*$), *continuous* ($S \subseteq \mathbb{R}^*$), or *heterogeneous* optimization problems ($S \subseteq (\mathbb{B} \cup \mathbb{Z} \cup \mathbb{R})^*$).

After defining the most important concepts, we formalize the test data generation problem as follows.

**Definition 2.2.5** (Test data generation problem)**.** *Given a program $P$, the* test data generation problem *consist in finding a test data set $T$ which maximizes $covBr_P(T)$. Consequently, a solution*[3] *for this problem is the data set $T$.*

[2] The proof of this proposition is found in [45].

[3] In the testing field this solution is also called test suite.

### 2.2.2 Multi-Objective Test Data Generation Problem

Ideally we would generate and execute all possible tests, but this is practically impossible, as we said before. Since the size of the test suite is an engineer's decision, s/he can control the effort devoted to this task. The incurred cost of checking whether the output of a program's execution is correct or not is called oracle cost. Thus, another objective for a software engineer is the minimization of the oracle cost, which can be achieved if the test suite size is minimized. Consequently, a balance between coverage and cost to achieve such coverage is mandatory. Since the cost of the testing phase depends on the test suite size, minimizing the test suite size, denoted with $|T|$ (where $T$ is the test suite), must be another goal. Prior to the definition of the Multi-Objective Test Data Generation problem (MOTDGP), we need to define some important concepts related to multi-objective optimization in general.

Mono-objective techniques focus on trying to minimize (or maximize) the values obtained with one single fitness function $f$, so there is one single global optimum. But, most of the real-world optimization problems require the optimization of more than one objective functions which are usually in conflict with each other, i.e., if one objective is improved, some of the others will be worsened. In the absence of any further information, all the objectives of a Multi-objective Optimization Problem (MOP) are considered equally important.

Informally, a MOP can be defined as the problem consisting in finding a vector of decision variables which satisfies a set of constraints and optimizes a number of objective functions. Those functions define a set of performance criteria which are in conflict with each other. Thus, the term *optimization* refers to the search of such a vector, which has acceptable values for all the objective functions. The formulation of a MOP extends the classic definition of mono-objective optimization by considering the existence of two or more objective functions. Consequently, there is not a single solution, but a set of them. To choose the most accurate solutions we make use of the Pareto Optimality theory. In the following we formally define the most important concepts related to MOPs, assuming, without loss of generality, that all objectives have to be minimized.

**Definition 2.2.6** (Pareto Dominance)**.** *Given a vectorial function $f : S \to \mathbb{R}^k$, a solution $x^1 \in S$ is said to* dominate *a solution $x^2 \in S$, denoted with $x^1 \prec x^2$, if and only if $f_i(x^1) \leq f_i(x^2)$ for $i = 1, 2, ..., k$, and there exists at least one $j$ $(1 \leq j \leq k)$ such that $f_j(x^1) < f_j(x^2)$.*

**Definition 2.2.7** (Pareto Optimal Set)**.** *We say that a solution $x$ is* non-dominated *with respect to the solution space $S$, if there does not exist another solution $x^* \in S$ such that $f(x^*) \prec f(x)$. Then, the set of non-dominated solutions $X^*$ with respect to the solution space $S$ is called* Pareto Optimal Set*.*

Generating the *Pareto Optimal Set* of a problem is the main goal of multi-objective optimization techniques.

**Definition 2.2.8** (Pareto Optimal Front)**.** *The* Pareto Optimal Front PF *is the image by $f$ of the* Pareto Optimal Set *$X^*$ (in the objective space), that is, $PF = f(X^*)$.*

Figure 2.1 depicts some examples of dominated and non-dominated solutions. In this figure, $A$ dominates $C$ because $f_1(A) < f_1(C)$, and $f_2(A) < f_2(C)$. Meanwhile, $A$ and $B$ are non-dominated solutions because $A$ is better than $B$ in the first objective function ($f_1(A) < f_1(B)$), but $B$ is better than $A$ in the other objective function ($f_2(A) > f_2(B)$).

Taking into account this definition of the Pareto dominance, a MOP is defined as follows:

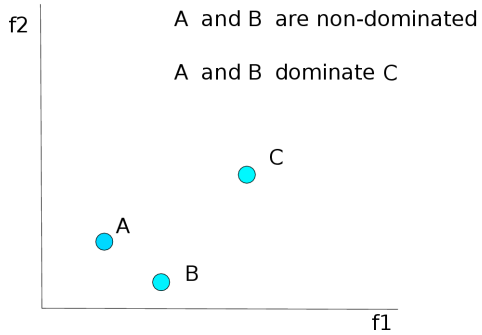



Figure 2.1: Examples of dominated and non-dominated solutions.

**Definition 2.2.9** (MOP)**.** *A* multi-objective optimization problem *is defined as a 2-tuple* $(S, f)$*, where* $S \neq \emptyset$ *is called the solution space (or search space), and* $f$ *is a vector function. A multi-objective optimization problem consists in finding the Pareto Optimal Set* $X^*$ *with respect to the solution space* $S$ *considering the vector function* $f$*.*

We formally define the Multi-Objective Test Data Generation problem (MOTDGP) with two conflicting objectives as follows:

**Definition 2.2.10** (MOTDGP)**.** *Given a program* $P$*, the* multi-objective test data generation problem *consists in finding a set of non-dominated solutions (test suites)* $X^*$ *from the set* $S$ *of all possible test suites and considering the maximization of* $f_1(x) = covBr_P(x)$ *and the minimization of* $f_2(x) = |x|$*,* $x \in S$*.*

## 2.3 Functional Testing

In the paradigm of structural testing a lot of research has been carried out using EAs, but the use of search-based techniques in functional testing is less frequent [210], the main cause being the implicit nature of the specification, which is generally written in natural language. Functional testing is used to confirm that the SUT meets its functional requirements. Typical functional approaches are decision table testing [32], equivalence partitioning [42], boundary value analysis [215] and category partition method [167]. Equivalent partitioning consists in performing a partition of the input domain of a function being tested, and then to select test data from each class of the partition. The idea is that all elements within an equivalence class are essentially the same for the purposes of testing. Boundary analysis and category partition method are considered a type of equivalent partitioning. In addition, there is no a standard representation of the system under test, so several approaches are available to represent the SUT such as the Classification Tree Method [91] or the Feature Models [115], a *de facto* standar for representing Software Product Lines (SPLs) [115].

Traditionally, the challenge has been to generate test suites to completely test the software. However, complete testing is not feasible for arbitrarily large projects [114], so a good subset of all possible test cases has to be selected. *Combinatorial Interaction Testing* (CIT) [53] is a black box sampling technique to complement traditional testing method that tries to address this problem. CIT is an effective testing approach for detecting failures caused by certain combinations of components or input values. Generally, this task consists of generating, at least, all possible combinations of the parameters' values. The most common coverage criterion is 2-wise (or pairwise)

testing, that is fulfilled if all possible pairs of values are covered by at least one test case in the result test set. Nearly all existing works investigate pairwise combination methods, but most of them can be extended to arbitrary $t$-combinations [82]. A large number of CIT approaches have been presented in the past [112, 173, 202], particularly a good overview and classification of approaches can be found in [127, 144], or more recently in [163]. A good survey that focuses on CIT with constraints is given in [51].

Project managers usually have a limited time for testing the whole system, therefore the most critical part of the software should be tested earlier than the non-critical. Prioritization of test cases is essential for discovering critical errors in early stages of the testing phase. But, in contrast the generation of minimal test suites that fulfill the demanded coverage criteria is an NP-hard problem [219]. For this reason, we tackle in this PhD thesis the generation of prioritized test suites in functional testing.

Nowadays software is present in critical applications as well as in practically all electronic devices present in our daily life. These two very different kind of programs have one feature in common, both could be represented by a state machine, defined by states and transitions. The actions performed by the system under test to change states are represented as transitions, therefore we generate sequences of tests using transitions. An isolated test datum does not describe which transitions were executed to get that state, thus, the test cases must be executed in a sequence. The benefits of using sequences of tests have been addressed by only a few works [33, 126, 165, 205]. In this PhD thesis we present an extension of the Classification Tree Method to deal with the generation of test sequences.

Although several search-based techniques have been applied to solve CIT problems, it remains largely unexplored. Garvin et al. applied simulated annealing to combinatorial interaction testing for computing n-wise coverage for SPLs. Ensan et al. [70] propose a genetic algorithm approach for test case generation for SPLs. Scatter search was also used to generate test cases for transition-pair coverage [33]. Recent work by Xu et al. uses a genetic algorithm for continuous test augmentation [225]. Their CONTESA tool incrementally generates test cases employing static analysis techniques for achieving coverage more effectively. None of these approaches use a multi-objective approach.

The work by Wang et al. presents an approach to minimize test suites using weights in the fitness function [212], that is, it uses a scalarizing function that transforms a multi-objective problem to a single-objective one [233]. Recent work by Henard et al. [103] presents an ad-hoc multi-objective algorithm whose fitness function is also scalarized. Incidentally we should point out there is an extensive body of work on the downsides of scalarization in multi-objective optimization [142]. Among the shortcomings are the fact that weights may show a preference of one objective over the other and, most importantly, the impossibility of reaching some parts of the Pareto front when dealing with convex fronts. In this PhD thesis we formalize the multi-objective version of the problem when the objectives are equally important (classical approach).

This great research effort has led to the development of automatic tools such as MoSo-PoLiTe [166], an approach that translates feature models associated to SPL and their constraints into binary constraint solver problems from which they compute pairwise covering arrays. Similarly, Hervieu et al. developed a tool called PACOGEN that also relies on constraint programming for computing pairwise coverage from feature models [104]. This tool has been recently included as part of a framework for practical pairwise testing in industrial settings [141]. Johansen et. al propose a greedy approach to generate n-wise test suites that adapts of Chvátal's algorithm to solve the set cover problem [110].

## 2.4 Functional Testing Problems Addressed in this Thesis

In this section we define the functional testing problems we address in this dissertation. We face the Prioritized Pairwise Test Data Generation Problem with Classification Tree Method, the Test Sequence Generation Problem with Extended Classification Tree Method, the Pairwise Test Data Generation Problem in SPL, and the Multi-Objective Test Data Generation Problem in SPL.

### 2.4.1 Prioritized Pairwise Test Data Generation Problem with Classification Tree Method

Combinatorial Interaction Testing is an effective testing approach for detecting failures caused by certain combinations of components or input values. Generally, this task consists of generating, at least, all possible combinations of the parameters' values (this task is NP-hard). There is not a common abstract nor concrete language to express combinatorial problems, so then, the formalization of this particular problem depending on how the combinatorial problem is expressed. In this case the problem and the solutions are represented using the Classification Tree Method [91] (CTM). This method aims at systematic and traceable test case identification for functional testing over all test levels (for example, component test or system test). It is based on the category partition method [167], which divides a test domain into disjoint classes representing important aspects of the test object. Applying the CTM involves two steps; designing the classification tree and defining test data.

**Design of the classification tree**: The classification tree is based on the functional specification of the test object. For each aspect of interest (called "*classification*"), the input domain is divided into disjoint subsets (called classes). In the classification tree method, classifications match parameters and classes match parameter values. Figure 2.2 shows a classification tree for a video game system. Two aspects of interest (*Game* and *Pause*) have been identified for the system under test. The classifications are partitioned into classes which represent the partitioning of the concrete input values. In our example the refinement aspect *Playing* is identified for the class *runningGame* and it is divided further into two classes *startup* and *controlling*. All classes have been assigned values of importance. As the figure shows, *running* is the most probable class in *Pause* classification. The weights of all classes at the same level in one classification sum 1 in the model. The class *runningGame* has an occurrence rate of 0.7 in *Game*. If the *Game* is *runningGame*, *statup* has an occurrence rate of 0.9 and *controlling* has an occurrence rate of 0.1. Refinements are interpreted as conditional probability in the occurrence model. The resulting occurrence probability for a *runningGame* with *statup* enabled is 0.63 ($= 0.7 \times 0.9$), for *controlling* it is 0.07, accordingly.

**Definition of test data**: Having composed the classification tree, test data can be defined by combining classes from different classifications. Since classifications only contain disjoint values, test data cannot contain several values of one classification. The length of the test data could vary if a class is refined into several classifications. In the following we formalize these essential concepts.

Throughout this section we formally define the CTM in order to describe all the aspects needed to generate combinatorial testing data for testing a software artifact. The CTM model can be totally defined by a tuple of three elements:

$$CTM = (C, V, w), \tag{2.9}$$

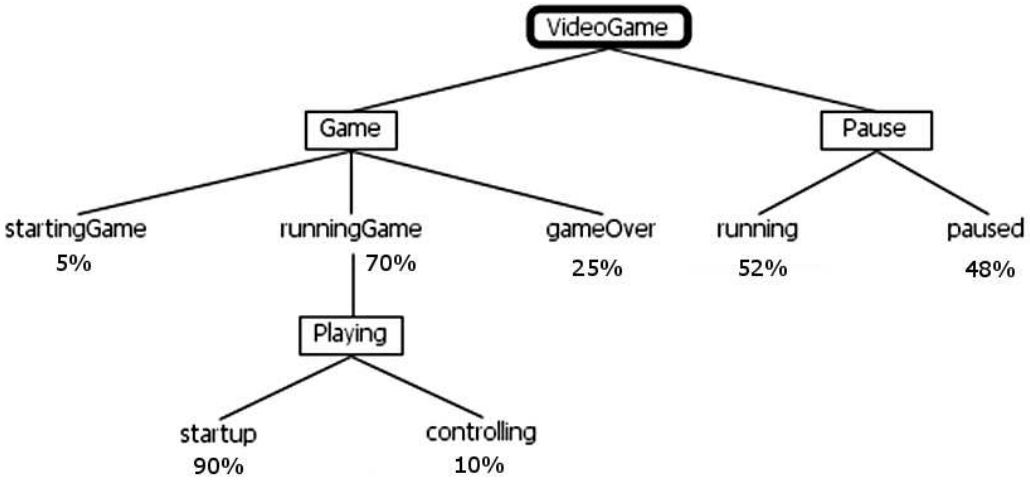



Figure 2.2: *Video Game System* test object.

where $C$ is the set of Classes, $V$ is the set of Classifications, and $w$ is a word of the language $L(G)$ generated by the grammar $G$ defined as follows:

$$G = (N, \Sigma, P, S) \tag{2.10}$$

where $N$ is the set of nonterminal symbols: $N = \{\text{Class, AtomicClass, RefinedClass, Classification}\}$. $\Sigma$ is the set of terminal symbols: $\Sigma = C \cup V \cup Punct$, where $Punct$ contains the squared brackets and comma. $P$ is the following set of production rules:

$$\begin{aligned}
S &\rightarrow Class \\
Class &\rightarrow AtomicClass | RefinedClass \\
AtomicClass &\rightarrow \beta \quad \forall \beta \in C \\
RefinedClass &\rightarrow \beta\ [Classification\ (, Classification)*] \quad \forall \beta \in C \\
Classification &\rightarrow [\alpha, \gamma, [Class\ (, Class)*]] \quad \forall \alpha \in V, \forall \gamma \in C
\end{aligned}$$

where $\gamma$ in the last rule represent the initial (default) class of classification $\alpha$. $S$ is the axiom of the grammar. The language $L(G)$ generated by the grammar represents all the possible trees that can be build using the same set of classes and classifications.

As an illustration, the CTM model shown in Figure 2.2 can be defined by the following triple:

$$\begin{aligned}
CTM_{Ex1} = (&\{VideoGame, startingGame, runningGame, startup, controlling, gameOver, \\
&running, paused\}, \\
&\{Game, Playing, Pause\}, \\
&w\})
\end{aligned}$$

where the word $w$ for this example is:

```
w:=VideoGame [
    [Game, startingGame,
        [startingGame,
         runningGame [Playing, startup, [startup, controlling]],
         gameOver]],
    [Pause, running, [running, paused]]]
```

Let us define some relations between the elements $e$ (classes and classifications) in the CTM model. An element $e_p$ is *parent* of $e_d$, if $e_d$ belongs to one of the classifications or classes defined by $e_p$. If $e_p$ is parent of $e_d$, then we say that $e_d$ is a child of $e_p$. The *ascendant* relation is the transitive closure of the parent relation and the *descendant* relation is the transitive closure of the children relation. In our running example, the class *runningGame* is the parent of the classification *Playing* and is also the ascendant of the classes *startup* and *controlling*. On the other hand, *Playing* is child of *runningGame*, meanwhile, the three elements (*startup*, *controlling* and *Playing*) are descendants of *runningGame*.

An element $e_s$ is *sibling* of another element $e_{s'}$ if they have the same parent. For example the class *startingGame* is sibling of *runningGame* and *gameOver*. In addition, the classification *Game* is sibling of the classification *Pause* and vice versa. The initial class $\gamma$ of a classification $v$ is defined in the word $w$. Finally, the *root* class is the first element that appears in $w$ and it does not have a parent in the tree (it only has descendants). From these relations we define the related functions that, given an element, return a set of elements: $Parent(e)$, $Ascendants(e)$, $Children(e)$, $Descendants(e)$, $Siblings(e)$ and $InitialClass(v)$.

**Definition 2.4.1** (Valid Test Datum)**.** *A* valid test datum *for a particular CTM model is a set of classes:*

$$Q := \{c_1, c_2, ..., c_n\}$$

*where the classes $c_i$ must fulfill the following rules:*

1. $\forall c \in Q \backslash root, Ascendants(c) \cap C \in Q$.
2. $\forall c \in Q, \forall s \in Children(c), \exists d \in Q, d \in Children(s)$.
3. $\forall c, b \in Q, c \neq b \implies b \notin Siblings(c)$.

Rule 1 says that if a class is in the test datum Q then all the classes in which it is included (ascendant classes) must also be in the test datum. Rule 2 requires that all the classifications under a class that is in $Q$ must have a class in $Q$. Finally, Rule 3 prevents from having two classes of the same classification in the test datum.

**Definition 2.4.2** (Valid Pair)**.** *Given a valid test datum $Q := \{c_1, c_2, ..., c_n\}$, a* pair $(c_i, c_j)$ *is* valid *if $c_i, c_j \in Q$*

Given an CTM model, the objective in this problem is the generation of a set of test data that maximizes a coverage criterion. In this study we use the pairwise coverage. This adequacy criterion consist in generating a test suite that uses every class pair from disjunctive classifications at least once in a test datum.

**Definition 2.4.3** (Pairwise coverage)**.** *Given a test data set T for a model M, we define the* pairwise coverage *of T, covPw(T), as the ratio between the number of valid pairs in T and the total number of posible valid pairs extracted from all test data generated from the model M.*

**Definition 2.4.4** (Pairwise Test Data Generation Problem)**.** *Given a CTM model M, the combinatorial* test data generation problem *consists in finding a test data set T which maximizes covPw(T).*

In order to define the Prioritized Pairwise Test Data Generation problem, we first detail how the priorities are assigned to classification tree elements. These priorities are also called *weights*. The higher the weight, the higher importance of the element. These weights can be used to guide the test data generation in order to cover first the most important values. Once we have assigned weights to each value (class), we need to define weights for the pair of classes.

**Definition 2.4.5** (Prioritized Valid Pair)**.** *Given a valid pair* $vp = (c_i, c_j)$ *with weighted values* $(c_i.w, c_j.w)$*, a* prioritized valid pair *is defined by a couple* $(vp, wp)$ *where the pair weight* $wp = c_i.w * c_j.w$*.*

We need to define a measure of the quality of a test suite in order to decide which test suite is the best one. We use here a coverage measure which is based on the weights of the class pairs covered.

**Definition 2.4.6** (Prioritized Test Datum)**.** *A* prioritized test datum *is a tuple* $(Q, w)$*, where Q represent a valid test datum, and w its weight. Let VT be the set of all valid pairs in Q,* $w = \sum_{v \in VT} v.w$*, that is the sum of all valid pairs of the test datum Q.*

**Definition 2.4.7** (Prioritized Test Suite)**.** *A* prioritized test suite *is a tuple* $(S, w)$*, where TS is a set of valid test data, and w its weight. Let VTS be the set of all test data in TS,* $w = \sum_{t \in VTS} t.w$*, that is the sum of the weight of all tests in the test suite.*

**Definition 2.4.8** (Prioritized Pairwise coverage)**.** *Given a prioritized test suite PTS for a model M, we define the* prioritized pairwise coverage *of PTS, covPPw(PTS), as the ratio between the weight value of the test suite PTS and the total weight value of all coverable pairs extracted from all test data generated from the model M.*

**Definition 2.4.9** (The Prioritized Pairwise Test Data Generation Problem)**.** *Given a CTM model M,* the prioritized combinatorial test data generation problem *consists in finding a prioritized test suite PTS which maximizes covPPw(PTS).*

### 2.4.2 Test Sequence Generation Problem with Extended Classification Tree Method

The Classification Tree defined above can be used to design test data in isolation. However, the test object can have operations related to transitions between classes in the classification tree and executing these transitions is the only way we can reach a given state (test datum) of the object. Let us continue with the video game example to illustrate new concepts. Imagine we need to execute some code when the user changes the state of the object from *starting game* to *running game*. These operations can be modeled by extending the classification tree method with transitions between classes (see Figure 2.3). In a real-world example, these transitions come from the semantics of the software object. We also assume that each classification has a *default class* that we highlight in the graphical representation by underlining its name. This extension of the classification tree can be seen as a hierarchical concurrent state machine (HCSM) or statechart [94] where classes match states, and classifications match orthogonal regions.



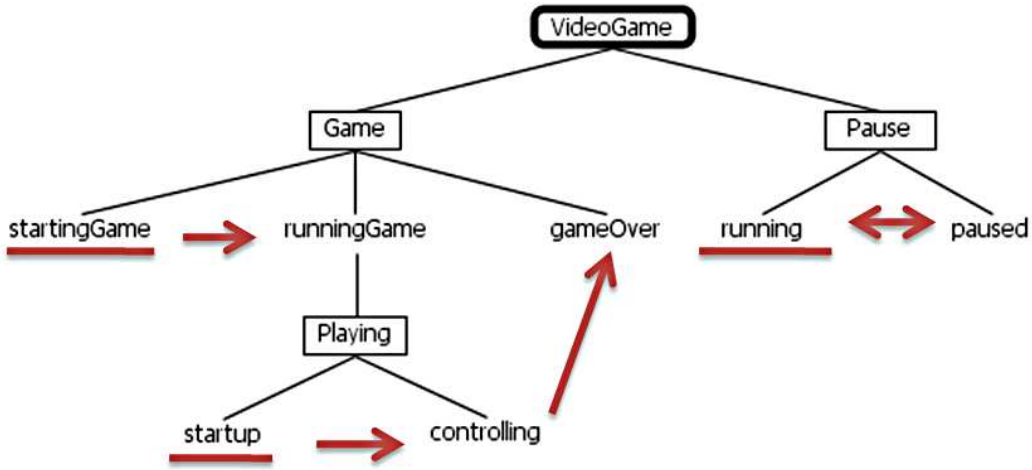



Figure 2.3: Video game ECTM example.

When the transition information is available we are also interested in covering all the possible transitions in the system. In this case, sequences of test data play a main role rather than the isolated test data. In effect, an isolated test datum does not describe which transitions were executed to get that test datum and, thus, does not determine the transitions executed. For this reason, our goal in this work is to provide test suites composed of sequences of test data that cover not only all the possible classes in the classification tree but also all the transitions by using the minimal number of total test data.

In addition to the constraints defined by the classification-classes hierarchy, in the Test Sequence Generation Problem (TSGP) we take dependency rules into account. These are constraints between single test steps i.e., restrictions on the transitions between classes. Within each test sequence, dependency rules must not be violated. Dealing with dependency rules is important since the testing of several states could be combined, resulting in shorter test sequences. In this way, we need fewer amount of resources to test all functionality. In the following we formally define the extension of the CTM (ECTM).

Our approach for test sequence generation is based on an idea proposed by Conrad [54], who suggests the interpretation of classification trees as parallel finite state machines. However, we need to extend Conrad's approach to interpret refined classes of the classification tree. This concept is similar to the refinements of states in UML statecharts. Our approach can be seen as a statechart, because we have concurrent states and we have added hierarchies to the model. The ECTM model can be totally defined by a tuple of four elements, extending the definition of CTM that we made previously (in Section 2.4.1) by adding a set of allowed transitions T:

$$ECTM = (C, V, w, T), \tag{2.11}$$

where $C$ is the set of Classes, $V$ is the set of Classifications, $w$ is a word of the language $L(G)$ generated by the grammar $G$ and $T$ is the set of allowed transitions between the classes $T \subseteq C \times C$. We will use either the notation $c_s \rightarrow c_d$ or $(c_s, c_d)$ to represent a transition between classes $c_s$ and $c_d$.

As an illustration, the ECTM model shown in Figure 2.3 can be defined by the following quadruple:

$$\begin{aligned} ECTM_{Ex1} = (&\{VideoGame, startingGame, runningGame, startup, controlling, gameOver,\\ &running, paused\},\\ &\{Game, Playing, Pause\},\\ &w,\\ &\{startingGame \rightarrow runningGame, startup \rightarrow controlling,\\ &controlling \rightarrow gameOver, running \rightarrow paused, paused \rightarrow running\}) \end{aligned}$$

The transition set in an ECTM model can contain any transition except those connecting classes of sibling classifications. Formally, any ECTM model must fulfill:

$$\begin{aligned} \forall v_1, v_2 \in V, v_1 \in Siblings(v_2) \implies &\forall c_1 \in Descendants(v_1) \cap C, \forall c_2 \in Descendants(v_2) \cap C,\\ &(c_1, c_2) \notin T. \end{aligned}$$

In order to build a sequence of test data we must define how to navigate from a source test datum $Q_1$ to a destination test datum $Q_2$. The initial test datum in a sequence, $Q_{ini}$, is composed by the initial classes of the children classifications under the root of the tree and all their ascendants.

Given a transition $t = (c_s, c_d) \in T$, the general rule to transit from $Q_1$ to $Q_2$ is as follows. We must find the deepest common classification of $c_s$ and $c_d$, say $v_a$. If there exists another common classification, then that classification must be an ascendant of the deepest one $v_a$. Once we have found $v_a$, we must remove from the source test datum $Q_1$ all the classes under $v_a$, in other words, any class that is descendant of $v_a$. Next, we have to add $c_d$ and its ascendants which are children of $v_a$, and add the initial classes of the classifications of these ascendants, except the siblings of $c_d$. If $c_d$ is a refined class, then the initial classes of all classifications of $c_d$ and their descendants are also added in order to build a valid test datum. Let us formally define all this procedure.

Let $Q_1$ be the source test datum, first we must remove from $Q_1$ the descendants of $v_a$:

$$Q' = Q_1 - Descendants(v_a)$$

where $\{c_s, c_d\} \subseteq Descendants(v_a)$ and does not exist $v_d \in Descendants(v_a)$ such that $\{c_s, c_d\} \subseteq Descendants(v_d)$. Then, we must add some classes to $Q'$ in order to transit to the new test datum $Q_2$. In order to do this, let us define the function $Incomplete(Q)$ as follows:

$$Incomplete(Q) = \{v \in V | Parent(v) \in Q \wedge \forall c' \in Children(v), c' \notin Q\}.$$

Then, we compute $Q_2$ iteratively using the next pseudocode:

$Q_2 = Q' \cup (Ascendants(c_d) \cap Descendants(v_a)) \cap C$
**while** $Incomplete(Q_2) \neq \emptyset$ **do**
  $Q_2 = Q_2 \cup InitialClass(Incomplete(Q_2))$
**end while**

We define a *test sequence* as a sequence of test data $TS = (Q_i)$ with $1 \leq i \leq n$, where the first test datum is the initial one, that is, $Q_1 = Q_{ini}$. In this work we have chosen two coverage criteria: class and transition coverage. The class coverage criterion consists of covering all the classes of

the classification tree with the generated test suite. The transition coverage requires covering all the transitions available between the classes of the ECTM. In our running example of Figure 2.3, we have to cover eight classes for total class coverage (*VideoGame, startingGame, runningGame, startup, controlling, gameOver, running, and paused*), and five transitions to obtain full transition coverage ($\{startingGame \rightarrow runningGame$, $startup \rightarrow controlling$, $controlling \rightarrow gameOver$, $running \rightarrow paused$, $paused \rightarrow running\}$)).

We formally define the coverage criteria used in this problem as follows:

$$ClassCoverage(sol) = \frac{\left|\bigcup_{i=1}^{n} Q_i\right|}{|C|} \tag{2.12}$$

$$TransitionCoverage(sol) = \frac{\left|\bigcup_{i=1}^{n-1} Transitions(Q_i, Q_{i+1})\right|}{|T|} \tag{2.13}$$

where $Transitions$ is defined as:

$$Transitions(Q_i, Q_{i+1}) = (Q_i \times Q_{i+1}) \cap T$$

Given an ECTM model, the objective of the TSGP is the generation of a set of test sequences that maximizes any of the coverage criterion (one each time) defined above (class or transition).

**Definition 2.4.10** (Test Sequence Generation Problem)**.** *Given an ECTM model M, the objective of the* TSGP *is the generation of a test sequence $TS$ that maximizes any of the coverage criterion (one each time) defined above ($ClassCoverage(TS)$ or $TransitionCoverage(TS)$).*

### 2.4.3 Pairwise Test Data Generation Problem in SPL

A *Software Product Line (SPL)* is a family of related software systems, which provide different feature combinations [175]. The effective management and realization of *variability* – the capacity of software artifacts to vary – is crucial to reap the benefits of SPLs such as increased software reuse, faster product customization, and reduced time to market. As we stated before, there is no a common representation for combinatorial testing problem, so in this work we represent SPL with feature models. Feature models have become the *de facto* standard for modelling the common and variable features of an SPL and their relationships collectively forming a tree-like structure. The nodes of the tree are the features which are depicted as labelled boxes, and the edges represent the relationships among them. Feature models denote the set of feature combinations that the products of an SPL can have [115].

Figure 2.4 shows the feature model of our running example for SPLs, the *Graph Product Line (GPL)* [134], a standard SPL of basic graph algorithms that has been widely used as a case study in the product line community. In GPL, a product is a collection of algorithms applied to directed or undirected graphs. In a feature model, each feature (except the root) has one parent feature and can have a set of child features. A child feature can only be included in a feature combination of a valid product if its parent is included as well. The root feature is always included. There are four kinds of feature relationships:

- *Mandatory features* are selected whenever their respective parent feature is selected. They are depicted with a filled circle. For example, features *Driver* and *Algorithms*,

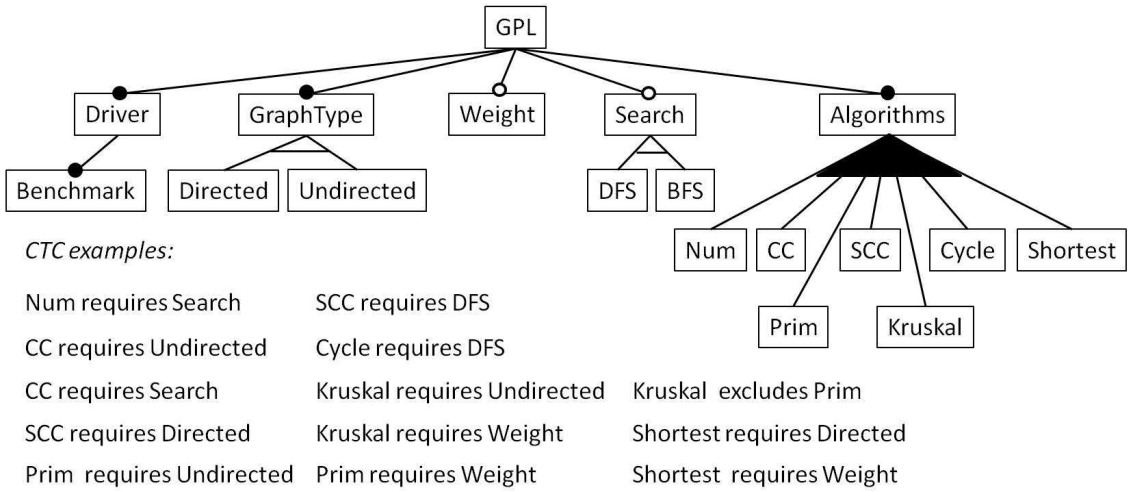



Figure 2.4: Graph Product Line feature model.

- *Optional features* may or may not be selected if their respective parent feature is selected. An example is feature *Search*,
- *Exclusive-or relations* indicate that exactly one of the features in the exclusive-or group must be selected whenever the parent feature is selected. They are depicted as empty arcs crossing over a set of lines connecting a parent feature with its child features. For instance, if feature *Search* is selected, then either feature *DFS* or feature *BFS* must be selected,
- *Inclusive-or relations* indicate that at least one of the features in the inclusive-or group must be selected if the parent is selected. They are depicted as filled arcs crossing over a set of lines connecting a parent feature with its child features. As an example, when feature *Algorithms* is selected then at least one of the features *Num*, *CC*, *SCC*, *Cycle*, *Shortest*, *Prim*, and *Kruskal* must be selected.

Besides the parent-child relations, features can also relate across different branches of the feature model with the so called *Cross-Tree Constraints (CTC)*. Figure 2.4 shows the CTCs of our feature model in textual form. For instance, *Num requires Search* means that whenever feature *Num* is selected, feature *Search* must also be selected. These constraints as well as those implied by the hierarchical relations between features are usually expressed and checked using propositional logic, for further details refer to [29].

*Combinatorial Interaction Testing (CIT)* is a testing approach that constructs samples to drive the systematic testing of software system configurations [52, 163]. When applied to SPL testing, the idea is to select a representative subset of products where interaction errors are more likely to occur rather than testing the complete product family [52]. In the following we provide the basic terminology of CIT for SPLs[4].

**Definition 2.4.11** (Feature list)**.** *A* feature list *FL is the list of features in a feature model.*

[4]Definitions based on [29] and [110].

Table 2.1: Sample feature sets of GPL feature model.

| Prod | G | D | B | GT | DI | U | W | S | BF | DF | A | N | C | SC | CY | P | K | SH |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| p0 | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | | | | ✓ | | | | | | ✓ | |
| p1 | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | | |
| p2 | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | | ✓ | | | ✓ | | | | ✓ |
| p3 | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | | | | | | | | | | | ✓ |

**Definition 2.4.12** (Feature set)**.** *A* feature set *fs is a pair* $(sel, \overline{sel})$ *where the first and second components are respectively the set of selected and not-selected features of a SPL product. Let FL be a feature list, thus* $sel, \overline{sel} \subseteq FL$*,* $sel \cap \overline{sel} = \emptyset$*, and* $sel \cup \overline{sel} = FL$*. Wherever unambiguous we use the term* product *as a synonym of feature set.*

**Definition 2.4.13** (Valid feature set)**.** *A* feature set *fs is* valid *with respect to a feature model fm iff fs.sel and* $fs.\overline{sel}$ *do not violate any constraints described by fm. The set of all valid feature sets represented by fm is denoted as* $FS^{fm}$*.*

An example of valid feature set *fs1* is one that computes the algorithm *Number*, on *Directed* graphs using *BFS* search. Thus, the selected features are *fs1.sel*={*GPL, Driver, Benchmark, GraphType, Directed, Search, BFS, Algorithms, Number*}[5]. Consider now another feature set *fs2* with selected features *BFS* and *Cycle*, meaning {*BFS,Cycle*} $\subseteq$ *fs2.sel*. This feature set is invalid because these two features violate the CTC that establishes that whenever *Cycle* feature is selected then feature *DFS* must be selected, i.e. *Cycle requires DFS*. In our running example GPL, the feature model denotes 73 distinct feature sets. Some of them are depicted in Table 2.1, where for any given feature set its selected features are ticked (✓) and its unselected features are empty. In this table, we use as column labels the shortest distinguishable prefix of the feature names (e.g. *G* for feature *GPL*).

The focus of our study is pairwise testing, thus our concern is on the combinations between two features. The coming definitions are consequently presented with that perspective; however, the generalization to combinations of any number of features is straightforward.

**Definition 2.4.14** (Pair[6])**.** *A* pair *ps is a 2-tuple* $(sel,\overline{sel})$ *involving two features from a feature list FL, that is,* $ps.sel \cup ps.\overline{sel} \subseteq FL \wedge ps.sel \cap ps.\overline{sel} = \emptyset \wedge |ps.sel \cup ps.\overline{sel}| = 2$*. We say pair ps is* covered *by feature set fs iff* $ps.sel \subseteq fs.sel \wedge ps.\overline{sel} \subseteq fs.\overline{sel}$*.*

**Definition 2.4.15** (Valid pair)**.** *A* pair *ps is* valid *in a feature model fm if there exists a valid feature set fs that covers ps. The set of all valid pairs of a feature model fm is denoted with* $VPS^{fm}$*.*

Let us illustrate pairwise testing with the GPL running example. Some samples of pairs are: *GPL* and *Search* selected, *Weight* and *Undirected* not selected, *CC* not selected and *Driver* selected. An example of invalid pair, i.e. not denoted by the feature model, is features *Directed* and *Undirected* both selected. Notice that this pair is not valid because they are part of an exclusive-or relation.

**Definition 2.4.16** (Pairwise test suite)**.** *A* pairwise test suite *pts for a feature model fm is a set of valid feature sets of fm. A pairwise test suite is complete if it covers all the valid pairs in* $VPS^{fm}$*, that is:* $\{fs | \forall ps \in VPS^{fm} \Rightarrow \exists fs \in FS^{fm}$ *such that fs covers ps*$\}$*.*

[5]Unselected features omitted for brevity.

[6]This definition of pair differs from the mathematical definition of the same term and is specific for SPLs. In particular, it adds more constraints to the traditional definition of pair.

In GPL there is a total of 418 valid pairs, so a complete pairwise test suite for GPL must have all these pairs covered by at least one feature set. Henceforth, because of our focus and for sake of brevity we will refer to pairwise test suites simply as *test suites*.

In the following we provide a formal definition of the priority scheme based on the sketched description provided in [111].

**Definition 2.4.17** (Prioritized product)**.** *A* prioritized product *pp is a 2-tuple (fs, w), where fs represents a valid feature set in feature model fm and* $w \in \mathbb{R}$ *represents its weight. Let* $pp_i$ *and* $pp_j$ *be two prioritized products. We say that* $pp_i$ *has higher priority than* $pp_j$ *for test-suite generation iff* $pp_i$*'s weight is greater than* $pp_j$*'s weight, that is* $pp_i.w > pp_j.w$.

As an example, let us say that we would like to prioritize product *p0* with a weight of 17. This would be denoted as *pp0=(p1,17)*.

**Definition 2.4.18** (Pairwise configuration)**.** *A* pairwise configuration *pc is a 2-tuple (sel,* $\overline{sel}$*) representing a partially configured product, defining the selection of 2 features of feature list FL, i.e.* $pc.sel \cup pc.\overline{sel} \subseteq FL \wedge pc.sel \cap pc.\overline{sel} = \emptyset \wedge |pc.sel \cup pc.\overline{sel}| = 2$. *We say a pairwise configuration pc is covered by feature set fs iff* $pc.sel \subseteq fs.sel \wedge pc.\overline{sel} \subseteq fs.\overline{sel}$.

Consider for example the pairwise configuration that indicates that feature *Driver* is selected while feature *Undirected* is deselected *pc1=({Driver},{Undirected})*. Notice that *pc1* is covered by products *p2* and *p3* of Table 2.1. Another example is pairwise configuration *pc2=({Directed, Undirected},{})* with features *Directed* and *Undirected* selected and no feature unselected. This configuration is covered by products *p2* and *p3* of Table 2.1.

**Definition 2.4.19** (Weighted pairwise configuration)**.** *A* weighted pairwise configuration *wpc is a 2-tuple (pc,w) where pc is a pairwise configuration and* $w \in \mathbb{R}$ *represents its weight computed as follows. Let PP be a set of prioritized products and* $PP_{pc}$ *be a subset,* $PP_{pc} \subseteq PP$*, such that* $PP_{pc}$ *contains all prioritized products in PP that cover pc of wpc, i.e.* $PP_{pc} = \{pp \in PP | pp.fs$ covers $wpc.pc\}$. *Then* $w = \sum_{p \in PP_{pc}} p.w$

Let us consider the following set of prioritized products from Table 2.1. Let *PP={(p0,17), (p1,15), (p2,5), (p3,3)}* with $pp_i = (fs_i, w_i)$, and assume that the remaining 69 products of our feature model in Figure 2.4 (i.e. 73 minus 4 shown in the table) have priority weight values of 0. The weight of pairwise configuration *pc1=({Directed},{Undirected})* is then $wpc1.w = pp2.w + pp3.w = 5 + 3 = 8$, that is, the summation of the weights of the products whose feature sets cover *pc1* with weight greater than zero, namely *p2* and *p3*. Similarly, the weight for *pc2* (*Directed* and *Undirected* selected) is $wpc2.w = pp0.w + pp1.w = 17 + 15 = 32$.

**Definition 2.4.20** (Prioritized pairwise covering array)**.** *A* prioritized pairwise covering array *ppCA for a feature model fm and a set of weighted pairwise configurations WPC is a set of valid feature sets FS that covers all weighted pairwise configurations in WPC whose weight is greater than zero:* $\forall wpc \in WPC$ $(wpc.w > 0 \Rightarrow \exists fs \in ppCA$ *such that fs covers* $wpc.pc)$.

The optimization problem we are interested in consists of finding a *prioritized pairwise covering array, ppCA*, with the minimum number of feature sets, that is: find $ppCA$ with minimum $|ppCA|$. What makes the problem far from trivial is the constraints imposed to $ppCA$ by Definition 2.4.20.

### 2.4.4 Multi-Objective Test Data Generation Problem in SPL

Most approaches for SPL pairwise testing have focused on achieving full coverage of all pairwise feature combinations with the minimum number of products to test. Though useful in many

contexts, this single-objective perspective does not reflect the prevailing scenario where software engineers do face trade-offs between the objectives of maximizing the coverage or minimizing the number of products to test. In contrast and to address this need, our work proposes a classical multi-objective formalisation where both objectives are equally important. In Section 2.2.2 we have defined the most important concepts related to multi-objective optimisation, then we only highlight the particularities of the Multi-Objective Test Data Generation Problem in SPL, that is, pairwise testing for two objectives.

The *decision space* or solution space is the set of possible solutions to an optimization problem, in our context, it corresponds to the set of all possible sets of valid feature sets represented by a feature model *fm*, denoted as $DS^{fm} = P(FS^{fm})$. A *decision vector* is an element of the decision space, that is $x \in DS^{fm}$. In our context we consider two objective functions:

- Coverage function. We want to maximize the number of pairs covered by a test suite. For simplicity, we use the alternative of minimizing the number of pairs not covered which is defined as follows:
  $f_1^{fm} : DS^{fm} \to \mathbb{N}, \;\; f_1^{fm}(x) = |VPS^{fm} \backslash covers(x)|,$
  where *covers* computes the pairs covered by the feature sets of test suite $x$.
- Test suite size function. We want to minimize the number of feature sets in the test suite. We define this function as follows:
  $f_2^{fm} : DS^{fm} \to \mathbb{N}, \;\; f_2^{fm}(x) = |x|.$

**Definition 2.4.21** (Multi-Objective SPL pairwise testing problem)**.** *Given a feature model $fm$, a* multi-objective pairwise SPL testing problem *consists in finding a set of non-dominated solutions $X^* \subset DS^{fm}$ such that it minimizes the objective functions $f_1^{fm}$ and $f_2^{fm}$.*

## 2.5 Conclusions

In this chapter we discuss the fundamentals of software testing from its beginning. Testing is not a new concept at all since it was already used by the Romans to assess the quality of metals. Nevertheless it is still valid since we want to assess the quality of software. In this chapter we formally defined actual testing problems that are addressed in this PhD dissertation. First, we defined the problems related to structural testing: Test Data Generation Problem and Multi-objective Test Data Generation Problem. We are aware that there is a negligible cost not considered in general, the oracle cost. For this reason we have taken into account the quality of the test suite and the oracle cost in the formulation of problems. Second, we define the problems related to functional testing. We can group the addressed problems in those related to Classification Tree Method, that are the Prioritized Pairwise Test Data Generation Problem and the Test Sequence Generation Problem with Extended Classification Tree Method. The other group are the problems related to Software Product Lines, that are the Pairwise Test Data Generation Problem with and without priorities and the Multi-Objective Test Data Generation Problem in SPL. We want to claim the usefulness of this chapter to provide the reader with the exact version of the problems we are going to solve throughout this PhD thesis.

# Chapter 3

# Fundamentals of Metaheuristics

A heuristic is a simple and intuitive technique that employs a practical methodology to produce close to optimal solutions for a given complex problem using specific information of the problem, not guaranteed to be optimal or perfect, but sufficient for the immediate goals [155]. When finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution. Heuristics can be seen as mental shortcuts that ease the cognitive load of making a decision. Heuristics are adapted to the problem at hand and they try to take full advantage of the particularities of the problem. However, they can get trapped in a local optimum and thus fail, in general, to obtain the global optimum solution. In addition, they have the drawback of being problem-specific techniques, thus a good heuristic for some given problem will help little when solving a different problem. Consequently, a more general-purpose technique was proposed: Metaheuristics [116].

Metaheuristics, in contrast, are more problem-independent techniques. They are generally conceived as high level heuristics able to employ heuristics methods by guiding them over the search space in order to exploit its best capabilities to achieve better solutions, especially with incomplete or imperfect information or limited computation capacity. The main advantage with respect to a simple heuristic is that they have mechanisms to avoid getting trapped in local optima.

This chapter serves as a presentation of metaheuristics, the main techniques used to solve the different variants of the automatic test data generation problem tackled in this PhD thesis. We formally define what is a metaheuristic, we classify the main metaheuristics and we present them for their latter detailed description in the next chapter. Then, we explain the multi-objective paradigm, used when we are interested in optimizing more than one objective at the same time. Finally, we describe the quality indicators and the statistical methods used to measure the quality of the solutions and the practical importance of the results of this research.

## 3.1 Formal Definition

In a first approach, the techniques can be classified into *Exact* and *Approximate*. Exact techniques, which are based on the mathematical finding of the optimal solution, or an exhaustive search until the optimum is found, guarantee the optimality of the obtained solution. However, these techniques present some drawbacks. The time they require, though bounded, may be very large, especially for NP-hard problems. Furthermore, it is not always possible to find such an exact technique for every problem. This makes exact techniques not to be a good choice in many occasions, since both their



time and memory requirements can become unreasonably high for large scale problems. For this reason, approximate techniques have been widely used by the international research community in the last few decades. These methods sacrifice the guarantee of finding the optimum in favor of providing some satisfactory solution within reasonable times [116].

Among approximate algorithms, we can find two types: *ad hoc* heuristics, and metaheuristics. We focus this chapter on the latter, although we mention before ad hoc heuristics, which can in turn be divided into *constructive heuristics* and *local search methods* [172]. Constructive heuristics are usually the swiftest methods. They construct a solution from scratch by iteratively incorporating components until a complete solution is obtained, which is returned as the algorithm output. Finding some constructive heuristic can be easy in many cases, but the obtained solutions are of low quality in general since they use simple rules for such construction. In fact, designing one such method that actually produces high quality solutions is a nontrivial task, since it mainly depends on the problem, and requires thorough understanding of it. For example, in problems with many constraints it could happen that many partial solutions do not lead to any feasible solution.

Local search or gradient descent methods start from a complete solution. They rely on the concept of *neighbourhood* to explore a part of the search space defined for the current solution until they find a *local optimum*. The neighbourhood of a given solution $s$, denoted as $N(s)$, is the set of solutions (neighbours) that can be reached from $s$ through the use of a specific modification operator (generally referred to as a *movement*). A local optimum is a solution having equal or better objective function value than any other solution in its own neighbourhood. The process of exploring the neighbourhood, finding and keeping the best neighbour, is repeated in a process until the local optimum is found. Complete exploration of a neighbourhood is often unapproachable, therefore some modification of this generic scheme has to be adopted. Depending on the movement operator, the neighbourhood varies and so does the manner of exploring the search space, simplifying or complicating the search process as a result. Out of the many descriptions of metaheuristics that can be found in the literature [35, 86], the following fundamental features can be highlighted:

- They are general strategies or templates that guide the search process.
- Their goal is to provide an efficient exploration of the search space to find (near) optimal solutions.
- They are not exact algorithms and their behavior is generally non deterministic (stochastic).
- They may incorporate mechanisms to avoid visiting non promising (or already visited) regions of the search space.
- Their basic scheme has a predefined structure.
- They may use specific problem knowledge for the problem at hand, by using some specific heuristic controlled by the high level strategy.

In other words, a metaheuristic is a general template for a stochastic process that has to be filled with specific data from the problem to be solved (solution representation, specific operators to manipulate them, etc.), and that can tackle problems with high dimensional search spaces. In these techniques, the success depends on the correct balance between *diversification* and *intensification*. The term diversification refers to the evaluation of solutions in distant regions of the search space (with some distance function previously defined for the solution space); it is also known as *exploration* of the search space. The term intensification refers to the evaluation of solutions in small bounded regions, or within a neighbourhood (*exploitation* of the search space). The balance between these two opposed aspects is of the utmost importance, since the algorithm has to



quickly find the most promising regions (exploration), but also those promising regions have to be thoroughly searched (exploitation).

We can distinguish two kinds of search strategy in metaheuristics. First, there are "intelligent" extensions of local search methods (trajectory-based metaheuristics in Figure 3.1). These techniques add some mechanism to escape from local optima to the basic local search method (which would otherwise stick to it). *Tabu Search* (TS) [84], *Iterated Local Search* (ILS) [86], *Variable Neighbourhood Search* (VNS) [157] or *Simulated Annealing* (SA) [119] are some techniques of this kind. These metaheuristics operate with a single solution at a time, and one (or more) neighbourhood structures. A different strategy is followed in *Ant Colony Optimization* (ACO) [64], *Particle Swarm Optimization* (PSO) [48] or *Evolutionary Algorithms* (EAs) [86]. These techniques operate with a set of solutions at any time (called colony, swarm or population, depending on the case), and use a learning factor as they, implicitly or explicitly, try to grasp the correlation between design variables in order to identify the regions of the search space with high-quality solutions (population-based techniques in Figure 3.1). In this sense, these methods perform a biased sampling of the search space.

A formal definition of metaheuristics can be found in [138], with an extension in [45]. A basic formulation of a metaheuristic is presented in the following definition:

**Definition 3.1.1** (Metaheuristic)**.** *A metaheuristic $\mathcal{M}$ is a tuple consisting of eight components as follows:*

$$\mathcal{M} = \langle \mathcal{T}, \Xi, \mu, \lambda, \Phi, \sigma, \mathcal{U}, \tau \rangle \ , \tag{3.1}$$

*where:*

- *$\mathcal{T}$ is the set of elements operated by the metaheuristic. This set contains the search space, and in many cases they both coincide.*
- *$\Xi = \{(\xi_1, D_1), (\xi_2, D_2), \ldots, (\xi_v, D_v)\}$ is a collection of $v$ pairs. Each pair is formed by a state variable of the metaheuristic and the domain of said variable.*
- *$\mu$ is the number of solutions operated by $\mathcal{M}$ in a single step.*
- *$\lambda$ is the number of new solutions generated in every iteration of $\mathcal{M}$.*
- *$\Phi : \mathcal{T}^\mu \times \prod_{i=1}^{v} D_i \times \mathcal{T}^\lambda \to [0,1]$ represents the operator that produces new solutions from the existing ones. The function must verify for all $x \in \mathcal{T}^\mu$ and for all $t \in \prod_{i=1}^{v} D_i$,*

$$\sum_{y \in \mathcal{T}^\lambda} \Phi(x,t,y) = 1 \ . \tag{3.2}$$

- *$\sigma : \mathcal{T}^\mu \times \mathcal{T}^\lambda \times \prod_{i=1}^{v} D_i \times \mathcal{T}^\mu \to [0,1]$ is a function that selects the solutions that will be manipulated in the next iteration of $\mathcal{M}$. This function must verify for all $x \in \mathcal{T}^\mu$, $z \in \mathcal{T}^\lambda$ and $t \in \prod_{i=1}^{v} D_i$,*

$$\sum_{y \in \mathcal{T}^\mu} \sigma(x,z,t,y) = 1 \ , \tag{3.3}$$

$$\forall y \in \mathcal{T}^\mu, \sigma(x,z,t,y) = 0 \vee \sigma(x,z,t,y) > 0 \wedge \tag{3.4}$$
$$(\forall i \in \{1,\ldots,\mu\}, (\exists j \in \{1,\ldots,\mu\}, y_i = x_j) \vee (\exists j \in \{1,\ldots,\lambda\}, y_i = z_j)) \ .$$

- $\mathcal{U} : \mathcal{T}^{\mu} \times \mathcal{T}^{\lambda} \times \prod_{i=1}^{v} D_i \times \prod_{i=1}^{v} D_i \rightarrow [0,1]$ *represents the updating process for the state variables of the metaheuristic. This function must verify for all* $x \in \mathcal{T}^{\mu}$, $z \in \mathcal{T}^{\lambda}$ *and* $t \in \prod_{i=1}^{v} D_i$,

$$\sum_{u \in \prod_{i=1}^{v} D_i} \mathcal{U}(x, z, t, u) = 1 \ . \tag{3.5}$$

- $\tau : \mathcal{T}^{\mu} \times \prod_{i=1}^{v} D_i \rightarrow \{false, true\}$ *is a function that decides the termination of the algorithm.*

The previous definition represents the typical stochastic behavior of most metaheuristics. In fact, the functions $\Phi$, $\sigma$ and $\mathcal{U}$ should be considered as conditional probabilities. For instance, the value of $\Phi(x, t, y)$ is the probability to generate the offspring vector $y \in \mathcal{T}^{\lambda}$, since the current set of individuals in the metaheuristic is $x \in \mathcal{T}^{\mu}$, and its internal state is given by the state variables $t \in \prod_{i=1}^{v} D_i$. One can notice that the constraints imposed over the functions $\Phi$, $\sigma$ and $\mathcal{U}$ enable them to be considered as functions that return the conditional probabilities.

## 3.2 Classification of Metaheuristics

There are many ways to classify metaheuristics [35]. Depending on the chosen features we can obtain different taxonomies: nature inspired vs. non nature inspired, memory-based vs. memory-less, one or several neighbourhood structures, etc. One of the most popular classifications distinguishes *trajectory based* metaheuristics from *population based* ones. Those of the first type handle a single solution of the search space at a time, while those of the latter work on a set of solutions (the population). This taxonomy is graphically represented in Figure 3.1, where the most representative techniques are also included.

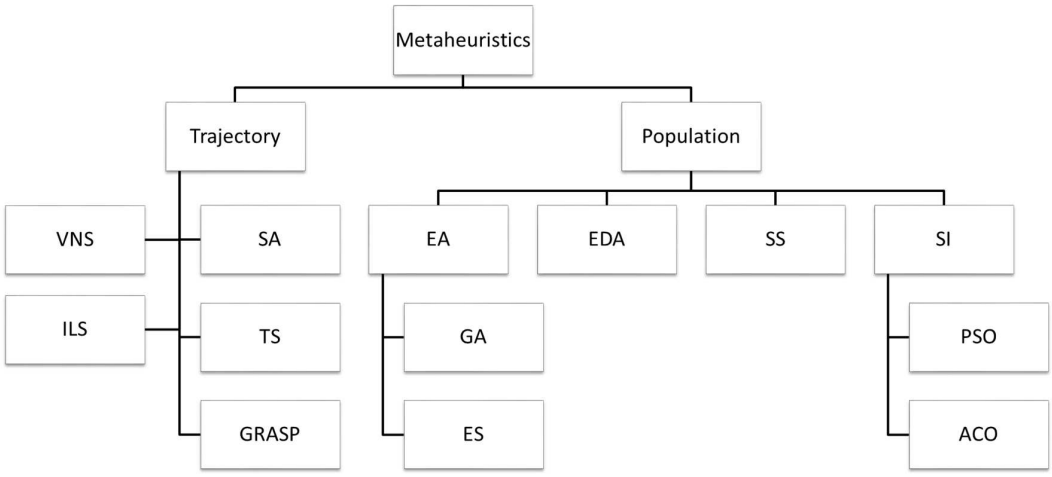



Figure 3.1: Classification of metaheuristics

### 3.2.1 Trajectory Based Metaheuristics

This section serves as a brief introduction to trajectory based metaheuristics. The main feature of these methods is the fact that they start from a single solution, and by successive neighbourhood explorations, update that solution, describing a trajectory through the search space. Most of these algorithms are extensions of the simple local search, which incorporate some additional mechanism for escaping local optima. This results in a more complex stopping condition than the simple detection of a local optimum. Widely used stopping criteria are completing some predefined number of iterations, or finding some acceptable solution. Although trajectory metaheuristics are not used in this PhD dissertation, we briefly draw up a generic description of the most representative techniques since they could be applied in future works extending our present studies.

**• Simulated Annealing**

*Simulated Annealing* was introduced in [119], and is one of the oldest techniques among metaheuristics. It was the first algorithm with an explicit strategy for escaping local optima. The main idea in SA is to simulate the annealing process of a metal or glass. To avoid getting stuck in a local optimum, the algorithm always allows the selection of a solution with worse *fitness* value than the current one with some probability. The mechanism works as follows: in each iteration a solution $s'$ is extracted from the neighbourhood $N(s)$ of current solution $s$; if $s'$ has better fitness value than $s$, then $s$ is discarded and $s'$ is kept instead, otherwise $s$ is replaced by $s'$ only with a given probability that depends on a dynamic parameter $T$ called temperature, and the difference between the fitness values of the two solutions, $f(s') - f(s)$.

**• Tabu Search**

*Tabu Search* [84] is one of the metaheuristics that has been most successfully used to solve combinatorial optimization problems. The main idea in TS is the use of an explicit search history (short term memory), that serves both for escaping from local optima and for enhancing the diversity of the search process. This short term memory is called the tabu list, and keeps record of the last visited solutions, preventing the algorithm from visiting them again. At the end of each iteration, the best solution among the allowed ones is included in the list.

**• Greedy Randomized Adaptive Search Procedure**

*Greedy Randomized Adaptive Search Procedure* (GRASP) [72] is a simple metaheuristic that combines constructive heuristics with local search. GRASP is an iterative procedure with two phases: first, a solution is constructed; second, the solution undergoes an improvement process. The improved solution is the final result of the search process. A randomized heuristic is used for the construction of the solution in the first phase. Step by step, different components $c$ are added to the partial solution, initially empty. Each added component is randomly selected from a restricted list of candidates. The components of the solution are sorted according to some problem dependent function in order to generate the list. The second phase of the algorithm consists in a local search method to improve the previously generated solutions. A simple local search heuristic can be employed, or some more complex technique like SA or TS.



**• Variable Neighbourhood Search**

*Variable Neighbourhood Search* is a metaheuristic proposed in [157], that uses an explicit strategy to switch among different neighbourhood structures during the search. It is a very generic algorithm with many degrees of freedom to design variations or particular instances.

The first step is to define the set of neighbourhood descriptions. There are many ways this can be done: from random selection up to complex mathematical equations deduced using problem knowledge. Each iteration contains three phases: selection of a candidate, improvement phase, and finally, the movement. During the first phase, a neighbour $s'$ is randomly chosen in the $k^{th}$ neighbourhood of $s$. This solution $s'$ acts then as the starting point for the second phase. Once the improvement process is over, the resulting solution $s''$ is compared with the original, $s$. If $s''$ is better then it becomes the current solution and the neighbourhood counter is reset ($k \leftarrow 1$); if it is not better, then the process is repeated for the next neighbourhood structure ($k \leftarrow k+1$). The local search can be considered as the intensity factor, whereas the switches among neighborhoods can be considered as the diversity factor.

**• Iterated Local Search**

The *Iterated Local Search* metaheuristic [86] is based on a simple yet effective concept. At each iteration, the current solution is perturbed and, to this new solution, a local search method is applied, to improve it. An acceptance test is applied to the local optimum obtained from the local search to determine whether it will be accepted or not. The perturbation method has an obvious importance: if it is not disruptive enough, the algorithm may still be unable to escape the local optimum; on the other side, if it is too disruptive, it can act as a random restarting mechanism. Therefore, the perturbation method should generate a new solution that serves as the starting point for the local search, but not so far away from the current solution as to be a random solution. The acceptance criterion determines the relative balance of intensification and diversification, since it filters new solutions to decide which can be accepted depending on the search history and the characteristics of the local optimum.

### 3.2.2 Population Based Metaheuristics

Population based methods are characterized by working with a set of solutions at a time, usually named *the population*, unlike trajectory based methods, which handle a single solution. We here make a brief survey to some relevant techniques for this study; the interested reader can get more information in [35, 86].

**• Evolutionary Algorithms**

*Evolutionary Algorithms* [23] are metaheuristic search techniques loosely based on the principles of natural evolution, namely, adaptation, and survival of the fittest. These techniques have been shown to be very effective in solving hard optimization tasks. They are based on a set of tentative solutions (individuals) called *population*. The problem knowledge is usually enclosed in an objective function, the so-called *fitness function*, which assigns a quality value to the individuals.

Initially, the algorithm creates, randomly or by using a seeding procedure, a population of $\mu$ individuals. At each step, the algorithm applies stochastic operators such as selection, recombination, and mutation operators in order to compute a set of $\lambda$ descendant individuals' *offspring*. The objective of the selection operator is to select some individuals from the population to which

the other operators will be applied. Different kinds of selection strategies exist: *roulette wheel*, *tournament*, *random*, $(\mu + \lambda)$, or $(\mu, \lambda)$, where $\mu$ represents the number of parent solutions and $\lambda$ the number of generated children. The recombination operator generates a new individual from several ones by combining their solution components. This operator is able to put together good solution components that are scattered in the population. On the other hand, the mutation operator modifies one single individual and is the source of new different solution components in the population. Recombination and mutation are the usual operations in genetic algorithms, having the rest of EAs other *variation operators* like local search or ad hoc techniques.

The individuals created are evaluated according to the fitness function. The last step of the loop is a replacement operation in which the individuals for the new population are selected from the offspring and the old one. This process is repeated until a stop criterion is fulfilled, such as reaching a pre-programmed number of iterations of the algorithm or finding an individual with a preset target quality. Depending on the individual representation and on how these phases are implemented, different instances of EAs arise such as *Evolution Strategy* (ES) and *Genetic Algorithm* (GA), which are used in this thesis.

**• Estimation of Distribution Algorithms**

*Estimation of Distribution Algorithms* (EDAs) [136] have a similar behavior with respect to the previously presented EAs, and many authors even consider EDAs as a special kind of EA. Like EAs, EDAs operate on a population of candidate solutions, but, unlike them, do not use recombination and mutation to generate the new solutions, but a probability distribution mechanism instead. They try to overcome the drawbacks of usual recombination operators, which are likely to break good building blocks. EDAs use probabilistic modelling of promising solutions to estimate a distribution over the search space, which is then used to produce the next generation by sampling the search space according to the estimated distribution. After every iteration the distribution is re-estimated. Graphic probabilistic models are commonly used tools to represent in an efficient manner the probability distributions when working with EDAs. Bayesian networks are usually to represent the probability distributions in discrete domains, while Gaussian networks are most often applied for continuous domains.

**• Scatter Search**

*Scatter Search* (SS) is a metaheuristic whose basic principles were presented in [85], and is currently receiving an increasing deal of attention from the research community. The algorithm's fundamental idea is to keep a relatively small set of candidate solutions (called the reference set), characterized by hosting diverse (distant in the search space) high-quality solutions. Five components are required for the complete definition of SS: initial population creation method, reference set generation method, subsets of solutions generation method, solution combination method, and improvement method. This algorithm is explicitly incorporating the exploration/exploitation idea in concrete steps and operations of its improvement loop.

**• Swarm Intelligence**

The collective and social behavior of living creatures motivated researchers to undertake the study of *Swarm Intelligence* [117]. Swarm intelligence techniques use the cooperation of large numbers of homogeneous agents. Such intelligence is decentralized, self-organizing and distributed throughout an environment. In nature, such systems are commonly used to solve problems such as effective

foraging for food, prey evading, or colony re-location. The information is typically stored throughout the participating homogeneous agents, or is stored or communicated in the environment itself such as through the use of pheromones in ants, dancing in bees, and proximity in fish and birds.

Swarm Intelligence systems are typically made up of a population of simple agents (an entity capable of performing/executing certain operations) interacting locally with one another and with their environment. Although there is normally no centralized control structure dictating how individual agents should behave, local interactions between such agents often lead to the emergence of global behavior. The paradigm consists of two dominant sub-fields: Ant Colony Optimization and Particle Swarm Optimization. We use ACO which will be detailed in next chapter.

## 3.3 A Methodology for Evaluating Results

Metaheuristics are non-deterministic techniques, hence different executions of the same algorithm over the same problem instance can produce different results. This can cause inconveniences to researchers at the time of evaluating and assessing those results, and when comparing different algorithms. Although there are works that tackle the theoretical analysis of many heuristic methods and problems [92], this kind of theoretical analysis still involves a great deal of complexity, therefore the most commonly adopted approach is to establish the comparisons on the basis of empirical data. For this, some indicators have to be defined that enable such comparisons. In a wide sense, there are two kinds of indicators. On the one hand, there are indicators that measure the quality of the obtained solutions. Since both mono-objective and multi-objective problems are solved in this PhD dissertation, specific indicators have to be defined for both approaches. On the other hand, there are indicators that measure the performance of the algorithms in terms of their required computation time or the amount of resources they use. Although the following discussion comments the two types of indicator separately, they are closely related and are often used together for the evaluation of metaheuristics, since the purpose of the latter is twofold: finding high quality solutions within a reasonable time.

Once the indicators have been established, a given number of unrelated or independent executions of the experimental configuration (algorithmic configuration and problem instance) are required to obtain statistically consistent results. A value of 30 executions is a commonly adopted and accepted minimum. The mere use of mean value and standard deviation, albeit quite frequent in the literature, is not sufficient and can lead to wrong conclusions. Thus, a global statistical analysis should be applied on the results before stating whether the observed differences are meaningful, and not just the result of the inherent randomness of the techniques. This section contains the discussion of the indicators used in the first place (for quality and performance), then the statistical tests that are used to assess the significance of the results.

### 3.3.1 Quality Indicators

Quality indicators or metrics are of paramount importance when evaluating a metaheuristic. They are defined in many ways depending on whether the optimal solution is known or unknown for the problem at hand (in a benchmark or a classic literature problem the optimum is often known, but for real problems this is hardly the case). As stated before, there are specific indicators for evaluating the solutions of mono-objective and multi-objective problems.



### Mono-objective Indicators

When the optimum is known beforehand, a simple and intuitive quality indicator for the metaheuristic is the expectancy of actually finding the optimum, or hit rate. This indicator is defined as the ratio or percentage of the number of executions in which the optimum is found over the total number of independent executions that have been performed. Unfortunately, knowing the optimum is not the common case for real problems or, even if they were known, sometimes they are so difficult to obtain that no execution of the experiment achieves it; in fact, experiments with metaheuristics are normally tailored to finish after a given computational effort has been spent (like visiting a maximum number of points of the search space, or running for a given time).

For these cases in which the optimum is not known in advance, or that the hit rate cannot be used, other indicators are used. The most popular are the mean and median of the best fitness value found in each independent execution. In general, other statistical data are required, such as the standard deviation, and a corresponding statistical analysis, in order to assess the statistical confidence on the observed results, should be performed.

In problems where the optimum is known, both metrics can be combined to offer a wider picture: for instance, a low hit rate with a low mean value speaks for the robustness of the method, and could be preferred over a higher hit rate but with higher median (assuming minimization).

### Multi-objective Indicators

In theory, a Pareto front could contain a large number of points. In practice, a usable approximate solution will only contain a limited number of them; thus, an important goal is that solutions should be as close as possible to the exact Pareto front and uniformly spread, otherwise, they would not be very useful to the decision maker. Besides, closeness to the Pareto front ensures that we are dealing with optimal solutions, while a uniform spread of the solutions means that we have made a good exploration of the objective space and no regions are left unexplored. There exist some well-known density estimators in the literature [50]: niching, adaptive grid, crowding, and the $k$-nearest neighbour distance.

Three different issues are normally considered for assessing the quality of the results computed by a multi-objective optimization algorithm [234]:

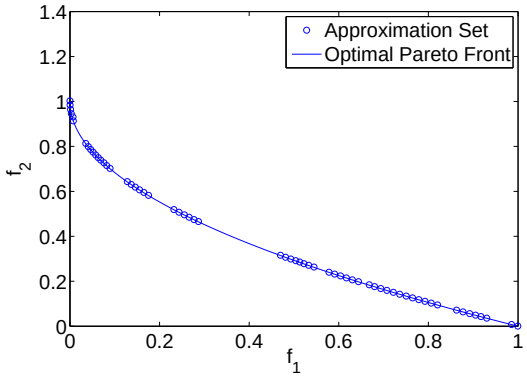
1. To minimize the distance of the computed solution set by the proposed algorithm to the optimal Pareto front (convergence towards the optimal Pareto front).

2. To maximize the spread of solutions found, so that we can have a distribution as smooth and uniform as possible (diversity).

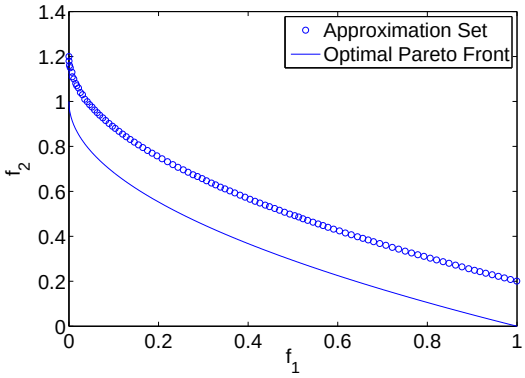
3. To maximize the number of elements of the Pareto optimal set found.

Figure 3.2 depicts these issues of convergence and diversity. The left front (a) depicts an example of good convergence and bad diversity: the approximation set contains Pareto optimal solutions but there are some unexplored regions of the objective space. The approximation set depicted on the right (b) illustrates poor convergence but good diversity: it has a diverse set of solutions but they are not Pareto optimal. Finally, the lowermost front (c) depicts an approximation front with both good convergence and diversity.

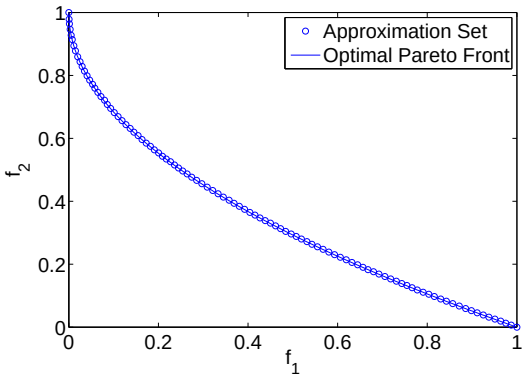
A number of quality indicators have been proposed in the literature trying to capture the three issues indicated above, but for the moment, there is not a single metric which captures all of them. Consequently, researchers should use more than one to measure different aspects of the




(a) good convergence and bad diversity.




(b) bad convergence and good diversity.




(c) good convergence and good diversity.

Figure 3.2: Examples of Pareto fronts with different behavior of convergence and diversity.

solutions generated by the multi-objective techniques. Among them, we can distinguish between *Pareto compliant* and *non Pareto compliant* indicators [123]. Given two Pareto fronts, A and B, if A dominates B, the value of a Pareto compliant quality indicator is higher for A than for B; meanwhile, this condition is not fulfilled by the non–compliant indicators. Thus, the use of Pareto compliant indicators should be preferable. To apply these quality indicators, it is usually necessary to know the optimal Pareto front. However, the location of the optimal front is usually unknown. Therefore, the front composed of all the non–dominated solutions computed by all analyzed approaches is used to obtain a reference Pareto front. Many quality indicators have proposed in the literature. Next, we highlight the advantages and disadvantages of some of the most common ones:

- **Number of Pareto optimal solutions**. This non–compliant indicator is very simple, it computes the number of solutions that are included in the optimal Pareto front. Its main advantage lies in the fact that it is very easy to compute. In contrast, the disadvantages are the lack of information about the diversity of solutions and the requirement of knowing the optimal Pareto front.

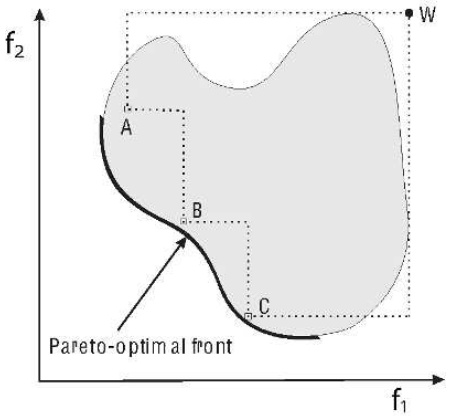



Figure 3.3: Example of hypervolume.

- **Hypervolume (HV) [236].** This is a very popular indicator that calculates the volume (in the objective space) covered by members of a non-dominated set of solutions $Q$ (the region enclosed into the discontinuous line in Figure 3.3, $Q = \{A, B, C\}$) for problems where all objectives are to be minimized. Mathematically, for each solution $i \in Q$, a hypercube $v_i$ is constructed with a reference point $W$ and the solution $i$ as the diagonal corners of the hypercube. The reference point can simply be found by constructing a vector of the worst objective function values. Thereafter, a union of all hypercubes is found and its hypervolume (HV) is calculated:

$$HV = \text{volume}\left(\bigcup_{i=1}^{|Q|} v_i\right). \tag{3.6}$$

  We apply this metric after a normalization of the objective function values to the range [0..1]. A Pareto front with a higher HV than another one could be due to: some solutions in the better front dominate solutions in the other, or, solutions in the better front are more widely distributed than in the other. Since both properties are considered to be good, algorithms with larger values of HV are considered to be desirable. To apply this quality indicator, it is usually necessary to know the optimal Pareto front (from normalization purposes). Of course, typically, we do not know the location of the optimal front. Therefore, we employ as a *reference Pareto optimal front* the front composed of all the non-dominated solutions out of all the executions carried out (i.e., the best front known until now).

  The main advantages of the hypervolume are that it considers the convergence as well as the diversity of the solutions, and it doesn't require the optimal Pareto front. A drawback is that it depends on the reference point selected. Different reference points produce different results. This could be critical to compare the results with existing approaches in the literature.

- **Spread (SD) [57].** It is a diversity non–compliant quality indicator that measures the distribution of individuals over the non-dominated region. This measure is based on the distance between solutions, so Pareto fronts with a smaller value of Spread are more desirable.

It takes a zero value for an ideal distribution, denoting a perfect spread of the solutions in the Pareto front. It is defined as follows:

$$\Delta = \frac{d_f + d_l + \sum_{i=1}^{N-1} |d_i - \overline{d}|}{d_f + d_l + (N-1)\overline{d}}, \tag{3.7}$$

where $d_i$ is the Euclidean distance between consecutive solutions, $\overline{d}$ is the mean of these distances, and $d_f$ and $d_l$ are the Euclidean distances of the extreme solutions of the optimal Pareto front in the objective space. The main advantage of this measure is that it summarizes the diversity of a Pareto front in one single scalar value. The main disadvantage is that it does not consider the other quality aspect, i.e., the solution set could be very well distributed, but the solutions could be far from the optimal Pareto front. Other quality indicators should be used to complement the Spread.

- **Generational Distance (GD) [207].** The generational distance is a non–compliant indicator. It measures how far the elements in the approximated Pareto front are from those in the optimal Pareto front. It considers the distance of the approximated Pareto front obtained to the reference front. Pareto fronts with a smaller value of GD are more desirable, so it takes a zero value for an ideal distribution where all the elements generated are in the reference front; It is defined as:

$$GD = \frac{\sqrt{\sum_{i=1}^{n} d_i^2}}{n}, \tag{3.8}$$

where $n$ is the number of solutions in the approximation front and $d_i$ is the Euclidean distance (measured in objective space) between each of these solutions and the nearest member in the optimal Pareto front.

The advantages of the generational distance are the ease of understanding and calculation, and the possibility to use different kinds of distance functions. In contrast, it does not take into account the diversity of the solutions found, i.e., a front with only one solution in the optimal Pareto front, will obtain an ideal value of generational distance.

- **Epsilon (Multiplicative) [237].** This Pareto–compliant indicator measures, in one single scalar value, how badly approximated the worst approximated solution of the Pareto front is. The approximation quality of solutions is the ratio between the optimal value and the best value found. This notion is extended to the multi-objective paradigm, resulting in a unary version of the indicator if the optimal Pareto front is known, and a binary version when a reference front is used instead. An approximated Pareto front is an $\epsilon$-approximation if for every point on the optimal Pareto front, the approximated Pareto front contains a point that is at least as good approximately (within a factor $\epsilon$) in all objectives. More formally, given $\vec{z^1} = (z_i^1, ..., z_n^1)$ and $\vec{z^2} = (z_i^2, ..., z_n^2)$, where $n$ is the number of objectives:

$$I_{\epsilon+}(A) = inf\left\{\epsilon \in \mathbb{R} | \forall \vec{z^2} \in \mathcal{PF}^* \exists \vec{z^1} \in A : \vec{z^1} \prec_\epsilon \vec{z^2}\right\}, \tag{3.9}$$

where, $\vec{z^1} \prec_\epsilon \vec{z^2}$ if and only if $\forall 1 \leq i \leq n : z_i^1 < \epsilon * z_i^2$

The main advantage of this quality indicator is that it allows us to compare the quality of solutions between different functions, different population sizes, and even different dimensions. In addition, it measures convergence of the algorithm, but it does not depend on a

chosen reference point like the hypervolume. In contrast, its main disadvantage is that it only considers part of the front, namely the worst solution.

The previous indicators have the advantage of summarizing an entire front into one single scalar value that allows the performance of different algorithms to be compared. However, from the point of view of a decision maker, knowing about a single number is not enough, because it gives no information about the shape of the front. In the related literature, the trade-off between the different objectives is usually presented by showing one of the approximated Pareto fronts obtained in one single run of a given algorithm. However, if the optimization algorithm used is stochastic there is no warranty that the same result is obtained after a new run of the algorithm. We need a way of representing the results of a multi-objective algorithm that allows us to observe the expected performance and its variability, in the same way as the average and the standard deviation are used in the single-objective case. For this reason, we use the concept of *Empirical Attainment Function* (EAF) [121].

Empirical Attainment Function: EAF is a function $\alpha$ from the objective space $\mathbb{R}^n$ to the interval $[0, 1]$ that estimates for each vector in the objective space the probability of being dominated by the approximated Pareto front of one single run of the multi-objective algorithm. Given the $r$ approximated Pareto fronts obtained in the different runs, the EAF is defined as:

$$\alpha(z) = \frac{1}{r} \sum_{i=1}^{r} I(A^i \preceq \{z\}) \tag{3.10}$$

where $A^i$ is the $i$-th approximated Pareto front obtained with the multi-objective algorithm and $I$ is an indicator function that takes value 1 when the predicate inside it is true, and 0 otherwise. The predicate $A^i \preceq \{z\}$ means $A^i$ dominates solution $z$. Thanks to the attainment function, it is possible to define the concept of $k$%-attainment surface [121]. The attainment function $\alpha$ is a scalar field in $\mathbb{R}^n$ and the $k$%-attainment surface is the level curve with value $k/100$ for $\alpha$. Informally, the 50%-attainment surface in the multi-objective domain is analogous to the median in the single-objective one. In a similar way, the 25%- and 75%-attainment surfaces can be used as the first and third "quartile fronts" and the region between them could be considered a kind of "interquartile region" (see Figure 3.4).

The attainment surfaces provide an engineer with a tool for evaluating the variability of an algorithm for the problem at hand. The variability in the results of one multi-objective algorithm is not reduced to a scalar (as in the single-objective case), so the main disadvantage is the visualization of solution with more than three objectives [204]. Using attainment surfaces the engineer can analyze and explore this range of possibilities. From a practical point of view, this tool helps the engineer to decide the more suitable multi-objective algorithm for her/his requirements.

### 3.3.2 Statistical Analysis Procedure

Metaheuristics are stochastic based algorithms, so we always need to perform a series of independent runs for each algorithm's configuration in order to obtain a distribution of results and quality indicators. Typically, a value of 30 independent executions for each algorithm's configuration is an accepted minimum. Once we have the results, we must compare the distributions by means of statistical tests, which are indispensable tools to validate and to provide confidence to our empirical analysis.

A representation of the statistical procedure carried out in this PhD thesis, which is recommended by the scientific community [81,189], can be seen in Figure 3.5. First, we choose the kind

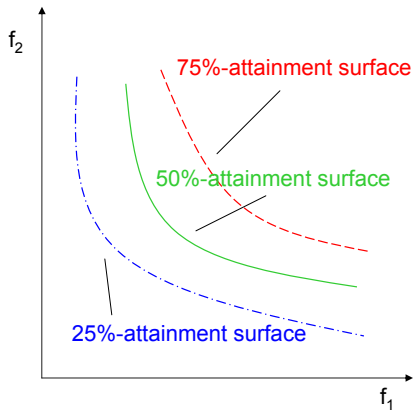



Figure 3.4: Examples of attainment surfaces.

of test to perform (non-parametric or parametric). To do so, we perform a *Kolmogorov-Smirnov* test to check whether the samples are distributed according to a normal distribution (Gaussian) or not. After this, the homoskedasticity (i.e., equality of variances) is then checked using the *Levene* test. If all distributions are normal and the Levene returns a positive value, then we use the parametric procedure. In this case, a *t-test* for comparing two distributions, and an *ANOVA* test for comparisons of three or more distributions.

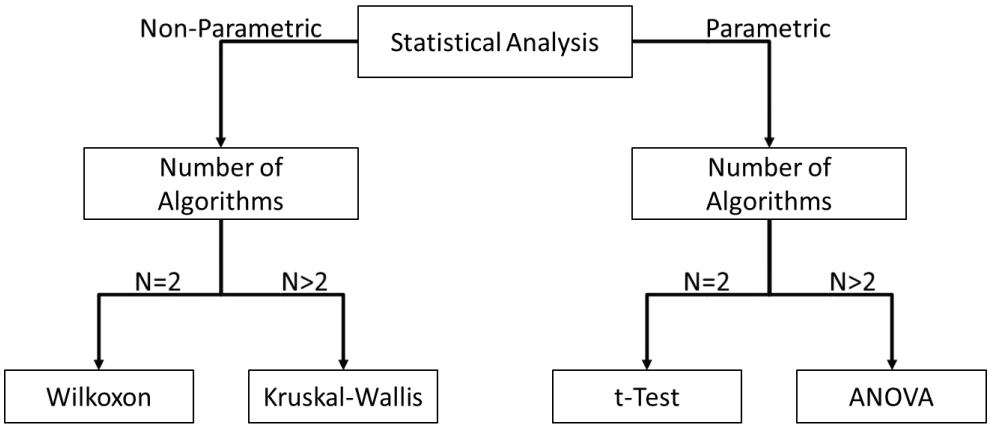



Figure 3.5: Statistical validation procedure for experimental results

For non-parametric procedure, we use the Wiloxon test to check the significance of the differences between two distributions, and the KruskalWallis test for multiple comparisons. Notice that we use the Bonferroni correction for each particular comparison. If there are no statistical differences, the procedure finishes without rejecting the null hypothesis (equality of distributions). In this study, the tests have been set with a confidence level of 95% or 99%, meaning that statistical differences can be found in distributions when resulted tests are with a $p\text{-}value < 0.05$ or $p\text{-}value < 0.001$, respectively.

In order to properly interpret the results of statistical tests, it is always advisable to report effect size measures. For that purpose, we have also used the non-parametric effect size measure $\widehat{A}_{12}$ statistic proposed by Vargha and Delaney [208]. It tells us how often, on average, one technique outperforms the other. It could be used to determine the probability of yielding higher performance by different algorithms. Given a performance measure $M$, $\hat{A}_{12}$ measures the probability that running algorithm $A$ yields higher $M$ values than running another algorithm $B$. If the two algorithms are equivalent, then $\hat{A}_{12} = 0.5$. If $\hat{A}_{12} = 0.3$ then one would obtain higher values for $M$ with algorithm $A$, 30% of the time.



# Chapter 4

# Algorithms

This chapter outlines the main metaheuristic techniques used throughout this PhD thesis. We focus in the particular description of algorithms and operators used to solve the different testing problems tackled here. The specific implementation details (like the operators for mutation or crossover) which are problem specific (or representation specific) are delayed to the corresponding chapters, where the application of the algorithms to solve the problems is discussed. The mono-objective techniques are described in Section 4.1, then the multi-objective techniques in Section 4.2.

## 4.1 Mono-objective Metaheuristics Used in this PhD Thesis

The specific mono-objective algorithms used in this PhD dissertation are described in this section: Genetic Algorithm (Section 4.1.1), Evolutionary Strategy (Section 4.1.2) and Ant Colony Optimization (Section 4.1.3).

### 4.1.1 Genetic Algorithm

Genetic Algorithms [88] belong to the family of EAs (Section 3.2.2). They appear for the first time as a widely recognized optimization method as a result of the work of John Holland in the early 70s, and particularly his 1975 book [106]. Our implementation of the genetic algorithm in this PhD thesis typically uses a ranking method for parent selection and elitist replacement for the next population, that is, the best individual of the current population is included in the next one. Should different operators or replacement policy is used, they will be explicitly described in the corresponding section. In Algorithm 1 we show the general scheme followed by an EA such as the Genetic Algorithm and the Evolutionary Strategy described in this chapter.

Initially, the algorithm creates a population of $\mu$ individuals randomly or by using a seeding algorithm (Line 2). At each step, the algorithm applies stochastic operators such as selection, recombination, and mutation in order to compute a set of $\lambda$ descendant individuals $Q$. The objective of the selection operator is to select some individuals from the population to which the other operators will be applied. In this work the *q-tournament* method is used. It randomly selects $q$ individuals from the population, then the best is selected as parent (Line 6). Notice that this method is applied twice to select both parents. The recombination operator generates two new individual from two parents by combining their solution components (Line 7). This operator is able to put together good solution components that are scattered in the population. On the other hand,

**Algorithm 1** Pseudocode of Evolutionary Algorithms.

```
1: proc Input:(ea)                    //Algorithm parameters in 'ea'
2: P ← initialize_population(μ) // P = population
3: while not Termination_Condition() do
4:    Q ← ∅                           // Q = auxiliary population
5:    for i ← 1 to (popSize / 2) do
6:       parents←selection(P)
7:       offspring←recombination(parents)
8:       offspring←mutation(offspring)
9:       evaluate_fitness(offspring)
10:      insert(offspring,Q)
11:   end for
12:   R ← P ∪ Q
13:   P' ← replacement(R)          // Select the best individuals
14: end while
15: end_proc
```

the mutation operator modifies one single individual and is the source of new different solution components in the population (Line 8). The new individuals created are evaluated according to the fitness function (Line 9). The last step of the loop is an elitist replacement operation in which the individuals for the new population $P'$ are selected from the offspring $Q$ and the old one $P$ (Line 13). This process is repeated until a stop criterion is fulfilled, such as reaching a pre-programmed number of iterations of the algorithm or finding an individual with a preset target quality.

Throughout this PhD dissertation we used two generic well-known recombination operators for GAs among other specific ones, which are *Single Point Crossover* and *Uniform Crossover*. In the following we detail how they work:

- Uniform Crossover (UX): each component of the new solution is randomly selected from the two parents. The formal definition is the following:

$$\mathbf{x}_i = \begin{cases} \mathbf{x}_i^1 & \text{if } U(0,1) < bias \\ \mathbf{x}_i^2 & \text{otherwise} \end{cases} \tag{4.1}$$

  where $U(0,1)$ denotes a random sample of a uniform distribution in the interval $[0,1)$ and $bias$ is a predefined probability. The UX operator is used in algorithms in Chapters 6 and 7.

- Single Point Crossover (SPX): a point is chosen between two consecutive positions at random in the solution encoding, then all genes beyond the chosen point are exchanged. The formal definition is the following:

$$\mathbf{x}_i = \begin{cases} \mathbf{x}_i^1 & \text{if } i < crossover\_point \\ \mathbf{x}_i^2 & \text{otherwise} \end{cases} \tag{4.2}$$

  where $crossover\_point$ is an integer value randomly selected in the interval $[0, length)$ and $length$ is the size of the individual. The SPX operator is used in algorithms in Chapters 5, 8 and 9.

As mutation operator, we have used the *Uniform Mutation*, when we deal with integer representation of the solution, the mutation operator adds a random value to the components of the vector. That is,

$$\mathbf{x}_i = \mathbf{x}_i^1 + U(-l, l)$$

where $U(-l, l)$ denotes a random sample of a uniform distribution in the interval $[-l, l]$. However, not all the components of the individual are perturbed, they have a $P_m$ probability of being altered. The UM is used in algorithms in Chapters 6, 7 and 8.

Genetic algorithms are one of the most popular metaheuristic techniques, so we have used this technique in most experiments performed throughout this PhD thesis. This technique is used in Chapters 5, 6, 7, 8, and 9.

### 4.1.2 Evolutionary Strategy

Evolutionary Strategies (ESs) [31] also belong to the family of EAs (Section 3.2.2), so they follow the same general scheme shown in Algorithm 1. It was created in the early 1960s and developed further in the 1970s by Ingo Rechenberg and Hans-Paul Schwefel [178]. In an ES each individual is composed of a vector of real numbers representing the problem variables ($\mathbf{x}$), a vector of standard deviations ($\sigma$) and a vector of angles ($\omega$). These two last vectors are used as parameters for the main operator of this technique: the Gaussian mutation. They are evolved together with the problem variables themselves, thus allowing the algorithm to self-adapt the search to the landscape. The mutation operator is governed by the three following equations:

$$\sigma_i' = \sigma_i \exp(\tau N(0,1) + \eta N_i(0,1)) \tag{4.3}$$
$$\omega_i' = \omega_i + \varphi N_i(0,1) \tag{4.4}$$
$$\mathbf{x}' = \mathbf{x} + \mathbf{N}(\mathbf{0}, C(\sigma', \omega')) \tag{4.5}$$

where $C(\sigma', \omega')$ is the covariance matrix associated to $\sigma'$ and $\omega'$, $N(0,1)$ is the standard univariate normal distribution, and $\mathbf{N}(\mathbf{0}, C)$ is the multivariate normal distribution with mean $\mathbf{0}$ and covariance matrix $C$. The subindex $i$ in the standard normal distribution indicates that a new random number is generated for each component of the vector. The notation $N(0,1)$ is used for indicating that the same random number is used for all the components. The parameters $\tau$, $\eta$, and $\varphi$ are set to $(2n)^{-1/2}$, $(4n)^{-1/4}$, and $5\pi/180$, respectively, as suggested in [181].

For the recombination operator of an ES there are many alternatives: each of the three real vectors of an individual can be recombined in a different way. In our particular implementation in this PhD dissertation, we use discrete uniform recombination for the solution vector $\mathbf{x}$, where each component is selected from the best parent with a predefined probability, called *bias*. For the vector of standard deviations and angles we use arithmetic recombination. The exact expressions for the components of the vectors are:

$$\mathbf{x}_i = \begin{cases} \mathbf{x}_i^1 & \text{if } U(0,1) < bias \\ \mathbf{x}_i^2 & \text{otherwise} \end{cases} \tag{4.6}$$
$$\sigma_i = (\sigma_i^1 + \sigma_i^2)/2 \tag{4.7}$$
$$\omega_i = (\omega_i^1 + \omega_i^2)/2 \tag{4.8}$$

where the subindices are used to denote the two parent solutions and $U(0,1)$ denotes a random sample of a uniform distribution in the interval $[0,1)$. With respect to the replacement operator, there is a special notation to indicate whether the old population is taken into account or not to

form the new population. When only the new individuals are used, we have a $(\mu, \lambda)$-ES; otherwise, we have a $(\mu + \lambda)$-ES. Regarding the representation, if each component of the vector solution $\mathbf{x}$ were an integer, they are rounded to the nearest integer. There is no limit in the input domain, thus allowing the ES to explore the whole solution space. This technique is used in Chapters 5, 6, and 7.

### 4.1.3 Ant Colony Optimization

Ant Colony Optimization [65] is a global optimization algorithm, which belongs to the family of Swarm Intelligence algorithms (Section 3.2.2). They are inspired by the foraging behavior of real ants in the search for food. The main idea consists of simulating the ants' behavior in a graph, called construction graph, in order to search for the shortest path from an initial set of nodes to the objective ones. The cooperation between the different simulated ants is a key factor in the search which is performed indirectly by means of pheromone trails, which is a model of the chemicals real ants use for their communication. The main procedures of an ACO algorithm are the construction phase and the pheromone update. These two procedures are scheduled during the execution of ACO until a given stopping criterion is fulfilled. In the construction phase, each artificial ant follows a path in the construction graph. In the pheromone update, the pheromone trails of the arcs are modified.

In what follows we describe the algorithm, but prior to that we clarify some issues related to the notation used in Algorithm 2. In the pseudocode, the path traversed by the $k$-th artificial ant is denoted with $a^k$. We use $|a^k|$ to refer to the length of the path, the $j$-th node of the path is denoted with $a^k_j$, and $a^k_*$ is the last node of the path.

**Algorithm 2** Pseudocode of ACO.

```
1: proc Input:(ACO)                                   // Algorithm parameters in ACO
2: τ ← initialize_pheromone();                        // Pheromone trails are initialize by default
3: while not Termination_Condition() do
4:    for k = 1 to colsize do
5:       while |a^k| ≤ λ_ant ∧ T(a^k_*) − a^k ≠ ∅ do
6:          node ← select_successor(a^k_*, T(a^k_*), τ, η);   // Selects the best succesor
7:          a^k ← a^k + node;
8:       end while
9:    end for
10:   τ ← pheromone_evaporation(τ, ρ);
11:   τ ← pheromone_update(τ, a^best);
12: end while
13: end_proc
```

The algorithm works as follows. First, the pheromone trails are initialized in Line 2 with defaults values. After the initialization, the algorithm enters a loop that is executed until a termination condition is reached, typically a given maximum number of steps (Line 3). In Line 5, we use the expression $T(a^k_*) - a^k$ to refer to the elements of $T(a^k_*)$ that are not in the sequence $a^k$. That is, in that expression we interpret $a^k$ as a set of nodes. In the loop each ant $k$ stochastically selects the next node (Line 6) according to the pheromone ($\tau_{ij}$) and the heuristic value ($\eta_{ij}$) associated with each arc $(i, j)$. Then, the next node is selected with probability

$$p^k_{ij} = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{s \in T(i)} [\tau_{is}]^\alpha [\eta_{is}]^\beta} \ , \text{ for } j \in T(i) \ , \tag{4.9}$$

where $\alpha$ and $\beta$ are two parameters of the algorithm determining the relative influence of the pheromone trail and the heuristic value on the path construction, and $T$ is the set of available transitions from $i$.

The whole construction phase is iterated until the ant reaches the maximum length $\lambda_{ant}$, or it fulfills the coverage criterion. When all the ants have built their paths, a pheromone update phase is performed. In Line 10, all the pheromone trails are reduced, simulating the real world evaporation of pheromone trails, according to the expression $\tau_{ij} \leftarrow (1-\rho)\tau_{ij}$, where $\rho$ is the pheromone evaporation rate and it holds that $0 < \rho \leq 1$. Then, the pheromone trails associated with the arcs traversed by the best-so-far ant ($a^{best}$) are increased (Line 11). Once the termination condition has been fulfilled, the algorithm return the best solution. This technique is used in Chapter 8.

## 4.2 Multi-objective Metaheuristics Used in this PhD Thesis

In this thesis work multi-objective algorithms are also used to deal with the bi-objective version of different testing problems. The following algorithms are described: Non-dominated Sorting Genetic Algorithm-II (NSGA-II, Section 4.2.1), Strength Pareto Evolutionary Algorithm 2 (SPEA2, Section 4.2.2), Multi-Objective Cellular Algorithm (MOCell, Section 4.2.3), Pareto Archived Evolution Strategy (PAES, Section 4.2.4), and Random Multi-Objective Algorithm (RNDMulti, Section 4.2.5).

### 4.2.1 Non-dominated Sorting Genetic Algorithm II

*Non-dominated Sorting Genetic Algorithm-II* (NSGA-II), proposed by K. Deb *et al.* [58], is a genetic algorithm which is the reference algorithm in multi-objective optimization (with over 14,494 citations at the time of writing[1]). Its pseudocode is presented in Algorithm 3. NSGA-II makes use of a population (P) of candidate solutions (known as individuals). In each generation, it works by creating new individuals after applying the genetic operators to P, in order to create a new population Q (lines 5 to 8). Then, both the current (P) and the new population (Q) are joined; the resulting population (R) is ordered according to a ranking procedure and a density estimator known as crowding distance (Line 13). The crowding-distance computation requires sorting the population according to each objective function value in ascending order of magnitude to get a normalized distance among solutions. Then, the overall crowding-distance value is calculated as the sum of individual distance values corresponding to each objective (for further details, please see [58]). Finally, the population $P$ is updated with the best individuals in $R$ (Line 14). These steps are repeated until the termination condition is fulfilled. This technique is used in Chapters 7 and 9.

### 4.2.2 Strength Pareto Evolutionary Algorithm 2

*Strength Pareto Evolutionary Algorithm* (SPEA2) is a multi-objective evolutionary algorithm proposed by Zitler *et al.* in [235]. We show the algorithm's pseudocode in Algorithm 4. SPEA2 uses a population and an archive simultaneously in its operation. In it, each individual is assigned a fitness value that is the sum of its strength raw fitness and a density estimation. The strength value of a solution $i$ represents the number of solutions (in either the population or the archive) that are dominated by that solution, that is $S(i) = |\{j | j \in P_t \cup \overline{P_t} \wedge i > j\}|$. The strength raw

[1]Data from Google Scholar: 14,494 citations on July $1^{th}$ 2015.

**Algorithm 3** Pseudocode of NSGA-II.

```
1: proc Input:(nsga-II)                    //Algorithm parameters in 'nsga-II'
2: P ← initialize_population()             // P = population
3: while not termination_condition() do
4:    Q ← ∅                                // Q = auxiliary population
5:    for i ← 1 to (popSize / 2) do
6:       parents←selection(P)
7:       offspring←recombination(parents)
8:       offspring←mutation(offspring)
9:       evaluate_fitness(offspring)
10:      insert(offspring,Q)
11:   end for
12:   R ← P ∪ Q
13:   ranking_and_crowding(nsga-II, R)
14:   P ←select_best_individuals(nsga-II, R)
15: end while
16: end_proc
```

fitness value of a given solution $i$, on the contrary, is the sum of strengths of all the solutions that dominate it, and is subject to minimization, that is, $R(i) = \sum_{j \in P_t \cup \overline{P_t}, j>i} S(j)$. The algorithm applies the selection, crossover, and mutation operators to fill an archive of individuals; then, the non-dominated individuals of both the original population and the archive are copied into a new population. If the number of non-dominated individuals is greater than the population size, a truncation operator based on calculating the distances to the $k$-th nearest neighbour is used (a typical value is $k = 1$), $D(i) = \frac{1}{\sigma_i^k + 2}$, where $\sigma_i^k$ is the distance from solution $i$ to its $k$-th nearest neighbour. This way, the individuals having the minimum distance to any other individual are chosen. This technique is used in Chapters 7 and 9.

**Algorithm 4** Pseudocode of SPEA2.

```
1: proc
2: t ← 0
3: Initialize(P_0, P̄_0)
4: while not termination_condition() do
5:    fitness_assignment(P_t, P̄_t)
6:    P̄_{t+1} ← non_dominated(P_t ∪ P̄_{t+1})
7:    if |P̄_{t+1}| > N̄ then
8:       P̄_{t+1} ← truncate(P̄_{t+1})
9:    else
10:      P̄_{t+1} ← fill_with_dominated(P̄_t)
11:   end if
12:   parents ← selection(P̄_{t+1})
13:   offspring ← crossover(parents)
14:   P̄_{t+1} ← mutation(offspring)
15:   t ← t + 1
16: end while
17: end_proc
```

### 4.2.3 Multi-Objective Cellular Algorithm

Multi-Objective Cellular Genetic Algorithm (MOCell), introduced by Nebro *et al.* [162], is a cellular genetic algorithm (cGA). In cGAs, the concept of (small) *neighbourhood* is paramount. This

means that an individual may only cooperate with its nearby neighbours in the breeding loop. Overlapped small neighbourhoods of cGAs help in exploring the search space because they induce a slow diffusion of solutions through the population, providing a kind of exploration (diversification). Exploitation (intensification) takes place inside each neighbourhood by applying the typical genetic operations (crossover, mutation, and replacement).

MOCell includes an external archive to store the non-dominated solutions found as the algorithm progresses. This archive is limited in size and uses the crowding distance of NSGA-II to maintain diversity. The pseudocode of MOCell is presented in Algorithm 5, which corresponds with the version called aMOCell4, described in [161].

**Algorithm 5** Pseudocode of MOCell.

```
 1: proc Input:(MOCell)       //Algorithm parameters in 'MOCell'
 2: archive ← ∅ //Creates an empty archive
 3: while not termination_condition() do
 4:    for individual ← 1 to MOCell.popSize do
 5:       n_list←get_neighbourhood(MOCell,position(individual))
 6:       parent1←selection(n_list)
 7:       parent2←selection(archive)
 8:       offspring←recombination(MOCell.Pc,parent1, parent2)
 9:       offspring←mutation(MOCell.Pm,offspring)
10:       evaluate_fitness(offspring)
11:       replacement(position(individual),offspring,MOCell)
12:       insertParetoFront(offspring, archive)
13:    end for
14: end while
15: end_proc
```

We can observe that, in this version, for each individual we select one parent from its neighbourhood and one from the archive, in order to guide the search towards the best solutions found (Lines 5 to 8). Then a new solution is created by applying the genetic operators to these parents. The new solution is used to replace the current solution (Line 11) and is considered for inclusion in the archive (Line 12). This constitutes a single iteration of the algorithm. The overall algorithm iterates until a termination condition is fulfilled. This technique is used in Chapters 7 and 9.

### 4.2.4 Pareto Archived Evolution Strategy

*Pareto Archived Evolution Strategy* (PAES) is a metaheuristic proposed by Knowles and Corne [122]. The algorithm is based on a simple (1+1) evolution strategy. To find diverse solutions in the Pareto optimal set, PAES uses an external archive of non-dominated solutions, which is also used to make decisions about new candidate solutions. An adaptive grid is used as a density estimator in the archive. The most remarkable characteristic of PAES is that it does not make use of any recombination operators (crossover). New solutions are generated only by modifying the current solution. The pseudocode of PAES is presented in Algorithm 6. It starts with a random solution (Line 3). In each iteration, a new solution is produced by modifying the current solution (Line 5). This new solution is included in the archive and it is considered as a potential replacement for the current solution (lines 7 to 14). These steps are repeated until the maximum number of evaluations is reached.

We have included PAES in this thesis work because of its simplicity. PAES does not use any recombination operator, and its only parameter is the number of partitions of the adaptive grid of the archive. Its relative simplicity makes it attractive since there is only one parameter that

**Algorithm 6** Pseudocode of PAES.

```
1: proc Input:(paes)        //Algorithm parameters in 'paes'
2: archive ← ∅
3: currentSolution ← createSolution(paes) // Creates an initial solution
4: while not termination_condition() do
5:    mutatedSolution←mutation(currentSolution)
6:    evaluate_fitness(mutatedSolution)
7:    if isDominated(currentSolution, mutatedSolution) then
8:       currentSolution ← mutatedSolution
9:    else
10:      if solutions_are_nondominated(currentSolution, mutatedSolution) then
11:         insert(archive, mutatedSolution)
12:         currentSolution ← select(paes, archive)
13:      end if
14:   end if
15: end while
16: end_proc
```

require tuning in order to know that the algorithm is being applied properly (e.g., population size, crossover probability, mutation probability). This technique is used in Chapters 7 and 9.

### 4.2.5 Random Multi-Objective Algorithm

In this thesis dissertation we also use a random search multi-objective algorithm (RNDMulti). We want to check whether the metaheuristic algorithms are capable of outperform random search or not in our proposed testing problems. The pseudocode of the RNDMulti is presented in Algorithm 7. The final result of this random search is the set of all the non-dominated solutions found. This technique is used in Chapters 7 and 9.

**Algorithm 7** Pseudocode of RNDMulti.

```
1: proc
2: archive ← ∅
3: currentSolution ← create_solution() // Creates an initial solution
4: while not termination_condition() do
5:    newSolution ← create_solution()
6:    insert(archive, newSolution)
7: end while
8: extractParetoFront(archive)
9: end_proc
```



# Part II

# Structural Testing



# Chapter 5

# Test Data Generation in Object-Oriented Software

## 5.1 Introduction

Most of the work found in the literature related to automatic software testing focuses on procedural languages and programs. However, most of the software developed nowadays is *object-oriented*. Object-Oriented (OO) languages are considered an evolution of procedural languages and they allow developers to design and implement really large software applications more easily than using procedural languages, thanks to their ability for abstracting and modularizing software components. Most of the ideas used for testing procedural programs can be used with no change in the OO software. However, we can find new features in OO languages that do not exist in procedural ones. Some examples of these features are inheritance, polymorphism, overloading, generics, and so on. No doubt these new features solve some common errors found in procedural languages but they also trigger new kinds of errors. Thus, when dealing with OO programs, automatic software testing techniques must include new ideas that are not present in procedural testing.

One of the first works in the literature related to OO software was the work by Tonella [199], which used genetic algorithms. Later, Liu et al. [133] proposed the use of ant colony optimization. The work by Wappler and Wegener [213, 214] using hybrid algorithms and strongly-typed genetic programming focuses on container classes. More recently, the work by Arcuri and Yao [19] optimizes test cases for covering all the branches at the same time and compares different search strategies. None of the previous work explicitly deals with inheritance for generating test data. The objects used as parameters are generated by calling the constructor and some methods of one class in order to reach a given state of the object. However, no attention is paid to the position of the class in the hierarchy, thus we wish to address this issue in this thesis chapter. We propose new approaches to take into account the position in the class hierarchy of the classes used in the parameters of methods being tested. In particular, this thesis focuses on the Java `instanceof` operator, which is related to inheritance. It determines if the specified object 'o' is assignment-compatible with the object represented by the class 'c' (`o instanceof c`). The operator returns true if the specified object is non-null and can be cast to the reference type represented by the class 'c' without raising a `ClassCastException`. It returns false otherwise.

The main and motivating research question addressed here is **RQ1**: what is the guidance for an automatic test data generator if there is a method containing the code in Figure 5.1 and it needs

to make the branch condition true? We can use an objective function for this branch that takes value 1 if the condition is true and 0 otherwise. However, this objective function gives no clue on how near a test datum is from making the condition true and an algorithm using this function can behaves like a blind search. This is called the flag problem [96]. Consequently, the fitness landscape consists of two plateaus, corresponding to the two possible values (true or false). One of these plateaus will be super-fit and the other super-unfit. A search-based approach will not be able to locate the super-fit plateau any better than a random search, because the fitness landscape provides no guide to direct the search from unfit to fit regions of the landscape. Where the fit plateau may be very small relative to the unfit plateau, this makes the program hard to test. This issue has been studied by many researchers, however, previous work does not address the issue of a special type of the flag problem that often occurs in the context of object-orientation related to inheritance that occurs when `instanceof` operators are used.

```
void function (Collection c)
{
     if (c instanceof Set)
     {
     ...
     }
}
```

Figure 5.1: Instanceof expression in a sentence.

The specific contributions in this thesis chapter are a new branch distance defined for the `instanceof` operator, and two mutation operators taking into account the class hierarchy. The proposal made in this thesis allows the algorithms to better guide the search for test data in the presence of the `instanceof` operator. This operator appears in 2,700 of the 13,000 classes of the JDK 1.6 class hierarchy and more than 850 classes include this operator in between 1% and 12% of their source code lines. This means that this operator is used in real software and any contribution that facilitates the testing process when it is present will have an important impact on the OO software testing field.

## 5.2 Test Data Generator

Automatic generation of test data needs to be carried out using a tool to generate test data in an intelligent way, following the global objective of covering all branches of the source code. This problem is formalized in Section 2.2.1 to give the reader precise details of the problem we are coping with. In that section we define the adequacy criterion used to formalize the objective of the generator. In this thesis we use the branch coverage which is the most popular criterion in the literature and was formalized in Eq. (2.6). In the following we describe our test data generator and the complete process of the generation of test data.

Our test data generator breaks down the global objective (to cover all the branches) into several partial objectives consisting of dealing with only one branch of the program. Then, each partial objective can be treated as a separate optimization problem in which a solution to the problem is a test datum and the function to be minimized is a distance between the current test datum and one satisfying the partial objective. In order to solve such minimization problem optimization algorithms such EAs are used. The main loop of the test data generator is shown in Figure 5.2.

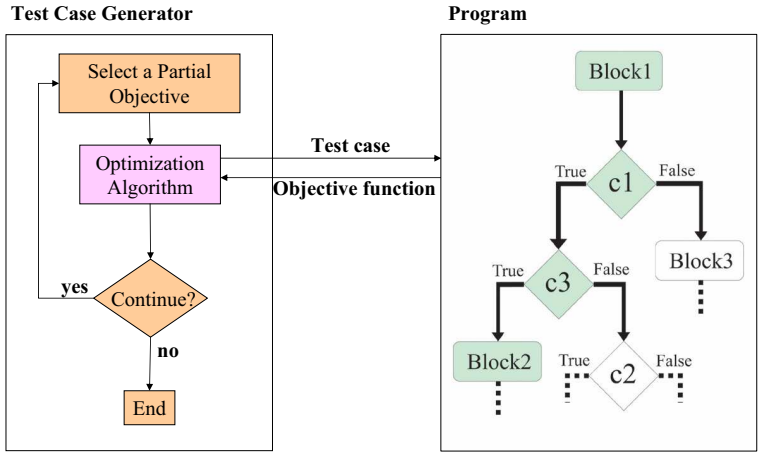



Figure 5.2: The test data generation process.

In a loop, the test data generator selects a partial objective (a branch) and uses the optimization algorithm to search for test data taking that branch. When a test datum covers a branch, the test datum is stored in a set associated to that branch. The structure composed of the sets associated to all the branches is called *coverage table*. After the optimization algorithm stops, the main loop starts again and the test data generator selects a different branch. This scheme is repeated until total branch coverage is obtained or a maximum number of evaluations or consecutive failures of the optimization algorithm is reached. When this happens the test data generator exits the main loop and returns the sets of test data associated to all the branches. In the following two sections we describe two important issues related to the test data generator: the objective function to minimize and the instrumentator used to generate an instrumented equivalent version of the program under test.

### 5.2.1 Objective Function

We have to solve several minimization problems: one for each branch. Now we need to define an objective function (for each branch) to be minimized. This function will be used for evaluating each test datum, and its definition depends on the desired branch and whether the program flow reaches the branching condition associated to the target branch or not. If the condition is reached we can define the objective function on the basis of the logical expression of the branching condition and the values of the program variables when the condition is reached. The resulting expression is called *branch distance* and can be recursively defined on the structure of the logical expression. That is, for an expression composed of other expressions joined by logical operators the branch distance is computed as an aggregation of the branch distance applied to the component logical expressions.

For the Java logical operators `&&` and `||` we define the branch distance as:

$$bd(a\&\&b) = bd(a) + bd(b) \tag{5.1}$$

$$bd(a||b) = \min(bd(a), bd(b)) \tag{5.2}$$

where $a$ and $b$ are logical expressions.

In order to completely specify the branch distance we need to define its value in the base case of the recursion, that is, for atomic conditions. The particular expression used for the branch distance in this case depends on the operator of the atomic condition. The operands of the condition appear in the expression. A lot of research has been devoted in the past to the study of appropriate branch distances in software testing. An accurate branch distance considering the value of each atomic condition and the value of its operands can better guide the search. In procedural software testing these accurate functions are well-known and popular in the literature. They are based on distance measures defined for relational operators like $<$, $>$, and so on [154]. We use here these distance measures described in the literature.

When a test datum does not reach the branching condition of the target branch we cannot use the branch distance as objective function. In this case, we identify the branching condition $c$ whose value must first change in order to cover the target branch (critical branching condition) and we define the objective function as the branch distance of this branching condition plus the *approximation level*. The approximation level, denoted here with $ap(c, b)$, is defined as the number of branching nodes lying between the critical one ($c$) and the target branch ($b$) [216].

We also add a real valued penalty in the objective function to those test data that do not reach the branching condition of the target branch. With this penalty, denoted by $p$, the objective value of any test datum that does not reach the target branching condition is higher than the one of any test datum that reaches the target branching condition. The exact value of the penalty depends on the target branching condition and it is always an upper bound of the target branch distance. Finally, the expression for the objective function is as follows:

$$f_b(x) = \begin{cases} bd_b(x) & \text{if } b \text{ is reached by } x \\ bd_c(x) + ap(c, b) * p & \text{otherwise} \end{cases} \tag{5.3}$$

where $c$ is the critical branching condition, and $bd_b$, $bd_c$ are the branch distances of branching conditions $b$ and $c$. The use of the penalty $p$ could be avoided by normalizing the branch distance to the interval $[0, 1)$ (see [16] for example).

Nested branches pose a great challenge for the search. For example, if the condition associated to a branch is nested within three conditional statements, all the conditions of these statements must be true in order for the program flow to proceed onto the next one. Therefore, for the purpose of computing the objective function, it is not possible to compute the branch distance for the second and third nested conditions until the first one is true. This gradual release of information might cause efficiency problems for the search (what McMinn calls the *nesting problem* [149]), which forces us to concentrate on satisfying each predicate sequentially. In order to alleviate the nesting problem, our test data generator selects as objective in each loop one branch whose associated condition has been previously reached by other test data stored in the coverage table. Some of these test data are inserted in the initial population of the algorithm used for solving the optimization problem. The percentage of individuals introduced in this way in the population is called the *replacement factor* and is denoted by $Rf$. At the beginning of the generation process some random test data are generated in order to reach some branching conditions.

### 5.2.2 Instrumentation Tool

Instrumentation is the implementation of code instructions that monitors specific components of a software system without modifying the behavior of the original program. In this thesis we are interested in code tracing, that is, receiving information about the execution of an application. Instrumentation approaches can be of two types: source instrumentation and binary instrumenta-

tion. Since we have access to the source code of the programs that we want to test, we decided to use source instrumentation in this dissertation.

We have developed an instrumentator for Java source code using a Java CC grammar of the Java language structures. The instrumentator adds instrumentation code and transforms the source code to receive information of the program execution. For this purpose, we have implemented the visitor design pattern, which is the best way to analyze and transform the source code related to general structures, control structures, and assignation expressions. Particularly, our instrumentator redefines conditions and decisions associated to control structures (If-then, while, switch, ...). We add some own instrumentation statements that give us the required information to be able to know which branches were traversed in an execution of the program under test. After the execution of our instrumentator over a source code, it returns a new version of the program with the same functionality but annotated with some statements which provide us useful information for computing the branch distance which guide the search to the optimal test data set.

## 5.3 Distance for instanceof operator

In this section we try to answer the main research question of this chapter (**RQ1**) by presenting our proposal of distance measure for the `instanceof` operator. This distance will be used in the base case of the branch distance definition (see Section 5.2.1) when the `instanceof` operator is found. Since our distance is based on the class and interface hierarchy, first we need to present the notation for referring to the classes, interfaces, and their relationship. In Java there is no multiple inheritance among classes, that is, a class can only *extend* one class. However, this does not hold for interfaces: one interface can *extend* several interfaces and one class can *implement* several interfaces. The natural way of representing the class and interface hierarchy is by means of a graph. Following the terminology proposed in the Java language specification [90], we use the term *reference type* to refer to a class or an interface.

Let us denote with $G_R = (R, E_R)$ the graph representing a hierarchy of reference types, where $R$ is the set of reference types considered and $E_R \subseteq R \times R$ is the set of arcs. We call this graph the *hierarchical graph* of $R$. The set $R$ can be composed of all the classes and interfaces accessible from a virtual machine or a subset of them. We will use the notation $C_R$ and $I_R$ to refer to the set of classes and interfaces in $R$, respectively. The set of arcs $E_R$ is determined by the relationship between classes and interfaces in $R$ in the following way. Let $r1, r2 \in R$ be two reference types, then $(r1, r2) \in E_R$ if and only if:

- $r1, r2 \in C_R$ and class $r1$ is a *direct superclass* of $r2$
- $r1 \in I_R$ and interface $r1$ is a *direct superinterface* of reference type $r2$

We must recall here that there is only one class in Java with no superclass: `Object`. This fact, together with the lack of multiple inheritance for classes, implies that the subgraph of $G_R$ composed only by the set of classes $C_R$ is a directed tree with the `Object` class in the root. We denote this subgraph $G_{C_R}$. At this point we can define the value returned by the `instanceof` operator based on our definition of hierarchical graph. Let "$o$ `instanceof` $r$" be an `instanceof` expression where $o$ is an object of class $c$ and $r$ a reference type. This expression evaluates to true if and only if a walk exists in $G_R$ from $r$ to $c$.

Once we have defined the hierarchical graph for a set of reference types $R$ we present now the definition of the distance $d$ between one class $c$ and a reference type $r$. This distance is the one used for comparing the two arguments of an `instanceof` operator. In this operator, the first argument

is an object from which only its class $c$ is used for the comparison. The second argument can be any reference type $r$. We distinguish two cases: when $r$ is a class and when $r$ is an interface.

If $r$ is a class then $c$ and $r$ belong to the directed tree $G_{C_R}$. Let us call $c'$ the deepest class in the directed tree that is ascendant of both $c$ and $r$ at the same time. The class $c'$ could also be $c$ or $r$. Since $G_{C_R}$ is a directed tree there exists a unique walk from $c'$ to $c$, denoted by $w_{c'\to c}$, and a unique walk from $c'$ to $r$ , $w_{c'\to r}$. We call the first walk *hierarchical walk* and the second one *approximation walk*. The distance between $c$ and $r$ is defined as:

$$d(c,r) = h|w_{c'\to c}| + a|w_{c'\to r}| \quad \text{, if } r \in C_R \quad , \tag{5.4}$$

where $h$ and $a$ are the *hierarchical* and *approximation* constants that weight the length of the hierarchical and approximation walks, respectively. In order to satisfy an `instanceof` expression the length of the approximation walk must be zero. However, in this last case, the length of the hierarchical walk is irrelevant to the satisfaction of the `instanceof` expression. For this reason, in order to reflect the real impact of each walk in the distance we should weight the approximation walk with a higher value than the hierarchical walk. We will empirically analyze the weights in the experimental section.

When $r$ is an interface we consider the distance from $c$ to the concrete classes of $C_R$ that implement interface $r$. Let $S_r \subseteq C_R$ be the set of concrete (not abstract) classes implementing $r$ (a walk exists in $G_R$ from $r$ to any class of $S_r$). Then the distance from $c$ to $r$ is defined as:

$$d(c,r) = \min_{t \in S_r} d(c,t) \quad , \tag{5.5}$$

where $d(c,t)$ is the distance between two classes defined above in Eq. (5.4).

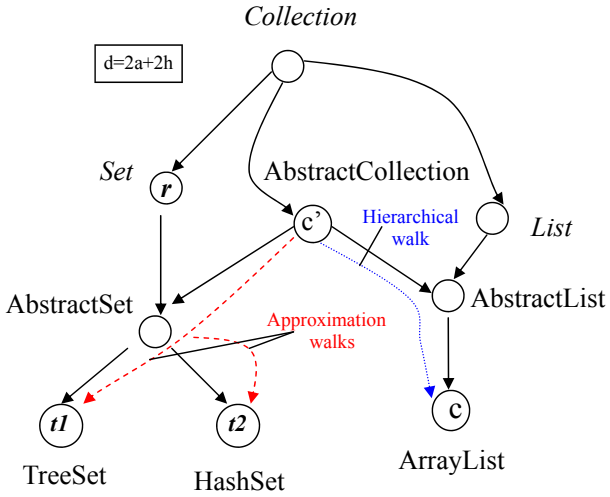



Figure 5.3: Example of distance between a class and an interface.

Figure 5.3 illustrates the computation of the distance between class `ArrayList` and interface `Set`. There are two concrete classes that implement the `Set` interface in our example: `TreeSet` and `HashSet`. The distance between class `ArrayList` and them is in both cases the same: $2a+2h$. Thus, the distance between interface `Set` and class `ArrayList` is $d = 2a + 2h$. If we set $a = 50$ and $h = 10$, then $d = 120$.

The computation of the distance defined in this section has complexity $O(a + h)$ in the case of two classes and $O((a + h) * i)$ in the case of a class and an interface, where $a$ and $h$ are the approximation and hierarchical distances (maximum values in the case of the class and the interface) and $i$ is the size of the subset $S_r$. However, in order to reduce the cost of computing these distances, these computations can be made prior to the test data generation and stored in a table.

## 5.4 Experimental Setup

In this section, we describe the details of the EA used in this chapter. Then, we use the distance presented throughout the chapter to design a mutation operator. We also present a benchmark set of object-oriented test programs that are used in the experiments.

### 5.4.1 Algorithm Details

In section 4.1.1 we draw a general description of the GA used for solving the test data generation problem for OO source code.

Regarding the representation, one solution is a vector $\vec{o}$ of objects (see Figure 5.4). These objects are used in the order determined by the vector as actual parameters of the method under test.

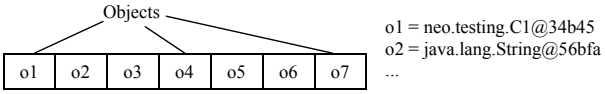



Figure 5.4: Representation of one solution vector $\vec{o}$.

For the selection operator we use $q$-tournament selection and the recombination operator used is a single point crossover. We defer the description of the mutation operator to Section 5.4.2 because it is one of the contributions of this thesis. The replacement operator is an elitist procedure, where any individual can potentially leave the pool and be replaced by a new one if the new one has a better fitness value. The fitness function used in this dissertation is the one presented in Eq. (5.3). Regarding the parameters of the EA, we use the values 0.1, 0.2, and 0.3 for the probability of mutation $P_M$, and the values 0.25, 0.50, 0.75 for the replacement factor $Rf$ presented in Section 5.2.1.

Since we are working with stochastic algorithms, we perform in all the cases a minimum of 30 independent runs of the algorithms (increased up to 200 in some cases) and Kruskal-Wallis's statistical tests for multiple comparison with a confidence of 95%. In order to obtain well-founded conclusions we base all our claims on statistically significant differences.

### 5.4.2 Mutation Operator

The mutation operator decides whether or not to change a component of an individual according to the probability of mutation $P_M$. The mutation operator is usually designed to introduce small variations in the tentative solutions. In this chapter we propose the use of the distance defined in Section 5.3 for computing the probability to change one object of the solution to an instance of a different class. We call this mutation operator *distance-based mutation*, denoted by MDn, and it

works as follows. For each component in the solution it decides whether to change it or not with probability $P_M$. If the object is changed, it selects the class of the new object from a universe $U$ of concrete classes. If the old object had class $c$, the probability of changing to an object of class $c'$ is given by the following expression:

$$p(c,c') = \begin{cases} \frac{\frac{1}{d(c,c')}}{\sum_{r \in U, r \neq c} \frac{1}{d(c,r)}} & \text{if } c \neq c', \\ 0 & \text{if } c = c'. \end{cases} \tag{5.6}$$

where $d$ is the distance between classes defined in Eq. (5.4). This way, it is more probable to mutate the object to a new one whose class is near the old one in the class hierarchy. This is how a "small change" is interpreted (and implemented) in the mutation operator. In Section 5.5.3, we will justify the use of an adaptive mutation MD$\alpha$ that evolves during the search. We defer its definition to the next section because its mathematical expression can be better understood after some results are shown.

Finally, the parameters for the proposed distance measure $d$ are the following: a length for the approximation walk greater than zero implies that the `instanceof` expression is false. Thus, the approximation constant $a$ must be large. The values used for $a$ are 50, 100 and 200. The hierarchical constant $h$ appears in the distance expression even when the `instanceof` expression is true. This distance helps the algorithm to search for test data that are in the boundary of the satisfaction region of the logical expressions. It seems reasonable to think that this constant should be smaller than $a$ and, for this reason, we use the values 1, 25, 50, and 100 for it.

### 5.4.3 Benchmark of Test Programs

We use a benchmark set of nine test programs with different features[1]. Since we are interested in studying our proposals for dealing with the `instanceof` operator, all the atomic conditions in the test programs are `instanceof` expressions. This way we analyze the `instanceof` operator in isolation, avoiding any influence on the results of the distance expressions used for other relational operators. All the programs have the same number of branches and we use the name `obj`$i$`_`$j$ to refer to the program with $i$ atomic conditions per logical expression (varying from 2 to 4) and nesting degree $j$ (varying from 1 to 3). Each program consists of one method with six decisions, varying the number of atomic conditions that appear in each one. In addition to this benchmark set, we use another program with only one condition composed of a conjunction of four `instanceof` expressions that will be used in an experiment discussed in Section 5.5.2.

## 5.5 Experimental Analysis

In the next section we show some preliminary results and analyze the influence of some parameters in order to set the best values for the parameters. Our main purpose is to study the measure of distance defined in Section 5.3 for the `instanceof` operator and the specific mutation operator designed. So, we compare the distance-based mutation MDn against a simpler mutation operator. Then, motivated with the obtained results we propose an adaptive mutation operator in Section 5.5.3.

[1] They are available at http://neo.lcc.uma.es/staff/javi/resources.html

### 5.5.1 Preliminary Results

In this section we analyze the influence on the average coverage of the replacement factor $Rf$, the mutation probability $P_M$, the approximation constant $a$, and the hierarchical constant $h$. The objective of this first study is to discover the best values for these parameters. The experiments performed have a factorial design. That is, for each of the nine test programs and each of the 108 combinations of the four above mentioned factors we have performed 30 independent runs. This means a total number of almost $30,000$ independent runs of the test data generator.

In this section we only show the average results of the most complex programs (`obj4_3`, `obj4_2`, and `obj3_3`) because the others achieve 100% coverage in most cases and could inflate coverage percentage. For each combination of $Rf$ and $P_M$ we have computed the average value of the coverage when the other parameters ($a$ and $h$) change. In this way we have obtained Table 5.1. If we focus on the parameter $P_M$ we can observe that the coverage is higher when $P_M$ is small. The statistical tests confirm that the differences between the results obtained with $P_M = 0.1$ and the other two values are significant. Thus, we conclude that a small probability of mutation must be used for programs with a high degree of nesting and a large number of atomic conditions in each logical expression. Regarding the replacement factor $Rf$, we can observe that the coverage increases with $Rf$ when $P_M = 0.2$ and $P_M = 0.3$. However, the differences in the results are not statistically significant.

Table 5.1: Average coverage obtained changing $P_M$ and $Rf$ in the most complex programs.

| | $P_M = 0.1$ | $P_M = 0.2$ | $P_M = 0.3$ |
|---|---|---|---|
| $Rf = 0.75$ | 83.08 | 76.10 | 67.29 |
| $Rf = 0.50$ | 83.43 | 75.43 | 67.20 |
| $Rf = 0.25$ | 82.10 | 73.81 | 66.74 |

We have also studied the influence on the average coverage of the hierarchical and approximation constants $h$ and $a$. As in the previous tables, for each program and each combination of $a$ and $h$ we have computed the average value of the coverage percentage when the other parameters ($P_M$ and $Rf$) change. The results are shown in Table 5.2. On the basis of our previous intuition on the behavior of the distance proposed for the `instanceof` operator we expected no significant differences except in the case $a < h$. The results confirm our expectations; as we can observe in Table 5.2, when $a < h$ the average coverage is minimum (with statistically significant differences). However, we cannot find a configuration that is better than all the rest. We can only conclude that $a$ should not be less than $h$.

Table 5.2: Average coverage obtained changing $h$ and $a$ in the most complex programs.

| | $h = 1$ | $h = 25$ | $h = 50$ | $h = 100$ |
|---|---|---|---|---|
| $a = 200$ | 75.45 | 75.33 | 74.93 | 75.68 |
| $a = 100$ | 75.53 | 74.74 | 75.10 | 74.79 |
| $a = 50$ | 74.86 | 75.81 | 74.44 | 73.57 |

According to the results shown in this section we fix the values of the four parameters studied in the following experiments. The values chosen are $P_M = 0.1, Rf = 50\%, a = 50$ and $h = 25$. In the rest of the experimental section we analyze the proposed mutation operators.

### 5.5.2 Uniform vs. Distance-based Mutation

In this section we compare the distance-based mutation MDn against a simpler mutation operator: one that selects the class using a uniform distribution of probabilities. We will call this operator *uniform mutation* (MU) in the following. The only difference between the distance-based and the uniform mutation is the probability distribution used for selecting the new class $c'$. In MDn is given by Eq. (5.6) and in MU is given by:

$$p(c, c') = \begin{cases} \frac{1}{|U|-1} & \text{if } c \neq c' \\ 0 & \text{if } c = c' \end{cases} \tag{5.7}$$

where $U$ is the universe of classes. One of the first questions we want to answer in this experimental section is: which mutation operator is better? Before the empirical evaluation we present a theoretical analysis that allows us to make some speculations. These speculations are based on the probability of obtaining an objective vector $\vec{t}$ from a given vector $\vec{o}$ by one application of the mutation operator. In the case of the uniform mutation this probability is

$$p_u(\vec{o}, \vec{t}) = (1 - M)^l M^{n-l} \left(\frac{1}{|U| - 1}\right)^{n-l} \tag{5.8}$$

where $M$ is the mutation probability, $n$ is the length of the vector $\vec{o}$, and $l$ is the number of objects of $\vec{o}$ that are instances of the correct class (the class required by the objective vector $\vec{t}$). In the case of the distance-based mutation the expression is more complex:

$$p_d(\vec{o}, \vec{t}) = (1 - M)^l M^{n-l} \prod_{\{i|o_i \neq t_i\}} p(o_i, t_i) \tag{5.9}$$

where $p$ is the probability distribution defined in Eq. (5.6) and we have used $o_i$ and $t_i$ to denote the classes of the corresponding objects.

At this point we must notice the following fact. If the classes of two objects $o_i$ and $t_i$ are near in the hierarchy of classes we have $p(o_i, t_i) > 1/(|U| - 1)$. Thus, $p_d(\vec{o}, \vec{t}) > p_u(\vec{o}, \vec{t})$ if solution $\vec{o}$ is near the objective $\vec{t}$. On the other hand, $p_d(\vec{o}, \vec{t}) < p_u(\vec{o}, \vec{t})$ if solution $\vec{o}$ is far from $\vec{t}$.

In order to check this speculation we have performed an experiment in which we used one program with one branch. The EA is used to search for a solution satisfying the condition. We have performed 200 independent runs of the EA and we have registered the best fitness of the population at each step of the EA in order to analyze the evolution of the search. In Figure 5.5 we show the average of the 200 independent runs at each step of the execution for MDn and MU.

We can observe in Figure 5.5 that, although the behavior is quite similar using the two mutation operators, the MU curve is lower than the other one at the beginning. The advance produced by MU to the objective solution is faster, since the initial population is far from this objective solution, so using MDn provides no advantage. However, as the search progresses we can observe that MDn is able to reach the objective solution before MU. When any individual of the population is near the objective solution, MDn guides the individual to the objective solution better than MU. This is the result expected from the speculations we made at the beginning of this section.

In order to confirm this behavior we perform a new experiment in which the initial population is randomly generated using individuals that are near the objective solution. We want to check if MDn is really faster than MU in this situation. In Figure 5.6 we show the average evolution over 200 independent runs and we can observe that MDn has a clear advantage over MU when the

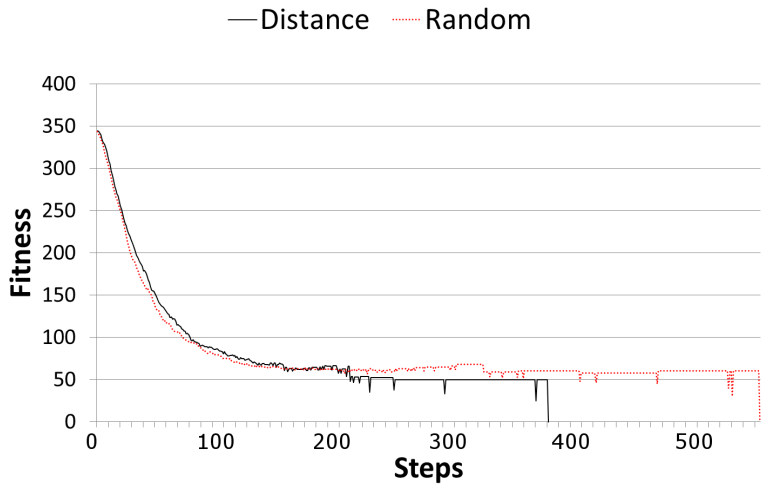



Figure 5.5: Fitness evolution with a uniformly initialized population.

population is near the optimum. Thus, our main conclusion in this section is that, in general, MDn is better than MU. However, MU can advance faster to the objective than MDn at the beginning of the search. Ideally, we would like to obtain a combination of the variability provided by the behavior of MU at the beginning and the behavior of MDn at the middle stage of the search. One way to achieve this is by means of an adaptive mutation operator. This is what we analyze in the following section.

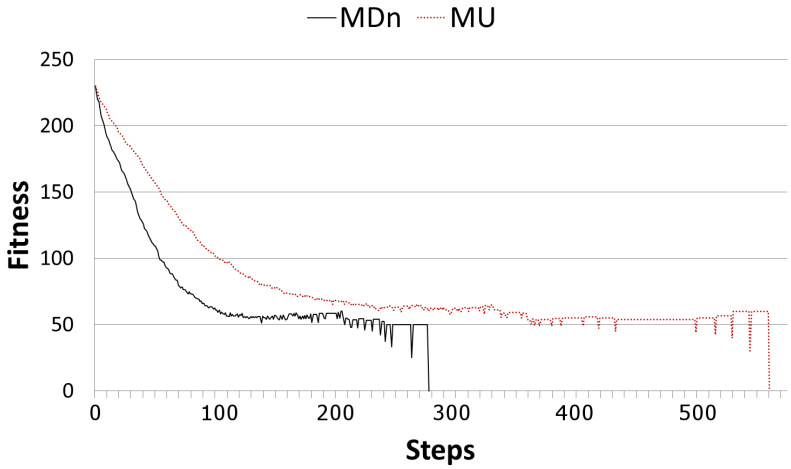



Figure 5.6: Fitness evolution with a population near the objective solution.

### 5.5.3 Adaptive Mutation

Motivated by the results of the previous section we present in this section a new mutation operator that changes its behavior throughout the search. The difference between this adaptive operator, denoted by MD$\alpha$, and the ones studied in the previous section is the probability distribution used for selecting a class. In MD$\alpha$ the probability distribution is:

$$p(c,c') = \begin{cases} \frac{\left(\frac{1}{d(c,c')}\right)^{\alpha}}{\sum_{r \in U, r \neq c} \left(\frac{1}{d(c,r)}\right)^{\alpha}} & \text{if } c \neq c' \\ 0 & \text{if } c = c' \end{cases} \tag{5.10}$$

In this expression, if $\alpha = 0$, we have the uniform mutation MU and if $\alpha = 1$, we have the distance-based mutation MDn. We can use values higher than 1 for $\alpha$. If the value of $\alpha$ is high, then the mutation only selects classes that are close to the ones in the individual. We can see $\alpha$ as an exploitation-exploration parameter. A low value for $\alpha$ leads to an explorative search. A high value leads to an exploitative search. In order to make it adaptive we must change the value of $\alpha$ throughout the search. We use a linear increase for $\alpha$, that is:

$$\alpha = \lambda \cdot step \tag{5.11}$$

where $\lambda$ is a parameter called *adaptive speed*. With this expression for $\alpha$, the behavior of the adaptive mutation is the same as the behavior of MU at the beginning and it switches to the behavior of MDn as the search progresses. The higher the value of $\lambda$, the higher the speed of this change in the behavior. If $\lambda = 0$ we have the uniform mutation, MU. On the other hand, if $\lambda = 1/T$, then MD$\alpha$ behaves like MDn in $T$ steps.

The adaptive speed $\lambda$ is a new parameter and we must analyze the behavior of the algorithm for different values of $\lambda$ in order to give some guidelines for selecting its value. A low value for $\lambda$ means a very explorative search. A high value for $\lambda$ makes the algorithm change very fast from the explorative phase to a very exploitative one. It is well-known in the metaheuristic field that one of the key points in the design of an algorithm is to select the exact balance between exploration and exploitation. Thus, we expect the best value for $\lambda$ to be not too high and not too low: it should be something in between. In order to support this hypothesis we have applied our test data generator using the adaptive mutation to the nine programs presented in Section 5.4.3. We used nine different values for $\lambda$ and performed 100 independent runs for each program and configuration. In all the cases the generator was executed until 100% branch coverage was obtained and we use the number of evaluations for comparison purposes. In Figure 5.7 we show the average number of evaluations for all the programs and the nine values of $\lambda$. We have also included the results of MU ($\lambda = 0$).

As expected, when extreme values for $\lambda$ are used the effort required to reach the total coverage is higher. In particular, when random mutation is used ($\lambda = 0$) the effort is higher than for intermediate values of $\lambda$ (there are statistically significant differences that confirm this observation). On the other hand, when $\lambda = 1/60$, the higher value of $\lambda$, the effort required is again increased. The reason is that the search reaches a very exploitative stage in a few steps, in which newly generated solutions are similar to the parent solutions. In this situation it is difficult for the algorithm to reach the objective. The best values for $\lambda$ are between 1/100 and 1/200.

We have also compared our proposals against a random search. The random search proposes random classes for the vector of objects that is used as test case. The random search is able to reach 100% branch coverage only in `obj2_1` with an average of 1302 evaluations. For the other

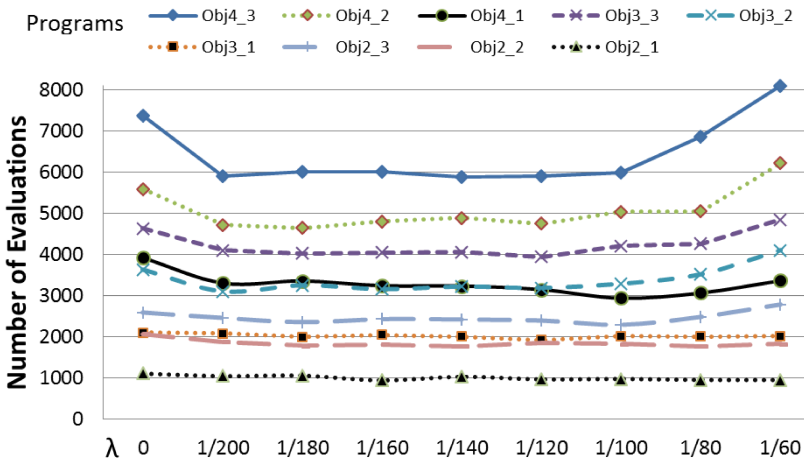



Figure 5.7: Average number of evaluations required for 100% branch coverage in all the test programs for different values of $\lambda$.

(more complex) programs we stopped the random search after 50,000 evaluations and the average coverage obtained varies from 21% in `obj4_3` to 99% in `obj3_1`. In the results shown in Figure 5.7 the maximum number of average evaluations for a 100% branch coverage is around 8,000. Thus, we conclude that our proposals are much better than a simple random search.

The results of Figure 5.7 also show the "difficulty" of the programs for the test data generator. From the results we can sort the programs according to the effort required to reach 100% branch coverage. We can observe that, except in a few cases, this ranking is independent of the value of $\lambda$ and is correlated with the value $i + j$ where $i$ is the number of atomic conditions per logical expression, and $j$ is the nesting degree. Furthermore, we can observe that the influence of $\lambda$ on the results is higher in the "most difficult" programs, as we could expect.

## 5.6 Conclusions

In this chapter we have studied one aspect of OO Software, inheritance, to propose some approaches that can help to better guide the search of test data in the context of OO evolutionary testing. In particular, we have answered the **RQ1** providing a distance measure to compute the branch distance in the presence of the `instanceof` operator in Java programs. We have also proposed two mutation operators that change the solutions based on the distance measure defined. In addition to the proposals we have performed a set of experiments to test our hypothesis. First, we have analyzed the most important parameters of the algorithm in order to select the best configuration. After that, we have analyzed and compared one of the proposed mutation operators against a uniform mutation. Finally, we have proposed an adaptive mutation operator that is able to make a better exploration and we have studied its main parameter.

One of the main conclusions of this work is that the difficulty to test a program depends on the number of atomic conditions per logical expression, and the nesting degree. Since we are interested in measuring the complexity of testing a program, in the next chapter we analyze the most common static measures and define a testing complexity measure to estimate the effort required to test software.

# Chapter 6

# Estimating Software Testing Complexity

## 6.1 Introduction

Since the birth of Software Industry, there has been a high interest in measuring the effort in terms of time and cost required by a task. Nowadays, software applications are essential for Industry, thus software developers need to measure all sort of elements. Tom DeMarco stated [59]: "You can not control what you cannot measure. Measurement is the prerequisite to management control". The importance of metrics have also been highlighted by the famous physicist Lord Kelvin [198]: "When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the state of science". For these reasons, in this chapter we focus on complexity measures, which quantify the effort required to complete any kind of task.

First, it is needed to define what program complexity means. Basili [27] defines complexity as a measure of the resources used by a system while interacting with a piece of software to perform a given task. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation described by the program. If the interacting system is a programmer then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing or modifying the software. There exist metrics introduced as all-purpose measures of software complexity, however these measures seem to be ineffective in order to measure the testing complexity [107]. The absence of a metric to properly measure the difficulty to test a piece of code encourage us to characterize the testing complexity, and drive us to the following research question **RQ1**: how difficult is the automated testing of a piece of code?

Analyzing the testing complexity, it can be seen as the difficulty for a computer to create a test suite for finding errors in the developed code. Finding errors in early stages of the development is an important task that saves costs of the project. It is estimated that half the time spent on the software project development and more than half its cost, is devoted to testing the product [158]. To this end, in recent years researchers have attempted to predict fault-prone software modules using complexity metrics [228]. In addition, the overall experimental results show that complexity metrics are able to predict fault-prone source code [232].



In most previous works they defined the testing complexity as the number of test cases required [130, 227]. Some works try to compute the lower bound [30] of the test cases required, and other works try to provide better understanding on the testing criterion used to generate those test cases [139]. However, they do not focus on the effort to generate these test cases. In a recent work, Nogueira focuses on the correlation between the complexity of the SUT and the complexity of the test cases [164], but the work did not propose any estimation measure.

We propose in this thesis dissertation a new complexity measure with the aim of helping the tester to find errors in the code. This measure will predict in a better way the behavior of an automatic test data generator depending on the SUT. This original complexity measure, called "Branch Coverage Expectation" (BCE), is the main contribution of this chapter. The definition of the new measure lies on a Markov model that represents the program. Based on the model of a program, we can also provide an estimation of the number of random test cases that must be generated to obtain a concrete coverage. From these estimations, we can create a theoretical prediction of the evolution of the coverage depending on the number of generated test cases. This second contribution will help the testers to obtain some knowledge about the possible evolution of the testing phase.

The validation of the proposed measure is also addressed in this work. For the theoretical validation of the BCE complexity measure we have used the validation framework proposed by Kitchenham et al. [120]. For the experimental validation we have used Evolutionary and Random Testing techniques, which are the most popular search algorithms for automatically generating test cases [10, 14, 74, 128], to compare our estimation with the real value obtained by several test data generators.

Finally, we also analyze software complexity measures at program level and we discuss a number of issues associated with these known measures. In addition, we have performed an experimental study of correlations with the aim of highlighting the existing relationships among some static measures. We are especially interested in the existing relationships between the static measures and the branch coverage. In this experimental study we have used two large groups of automatically generated programs and one group of real-world ones to serve as a benchmark.

## 6.2 Static Measures

Quantitative models are frequently used in different engineering disciplines for predicting situations, due dates, required cost, and so on. These quantitative models are based on some kind of measure made on project data or items. Software Engineering is not an exception. A lot of measures are defined in Software Engineering in order to predict software quality [187], task effort [36], etc. We are interested here in measures made on source code pieces. We distinguish two kinds of measures: *dynamic*, which require the execution of the program, and *static*, which do not.

Some time ago, project managers began to worry about concepts like productivity and quality, then the lines of code (LOC) metric was proposed. Nowadays, the LOC metric is still the primary quantitative measure in use. An examination of the main metrics reveals that most of them confuse the complexity of a program with its size. The underlying idea of these measures are that a program will be much more difficult to work with than a second one if, for example, it is twice the size, has twice as many control paths leading through it, or contains twice as many logical decisions. Unfortunately, these various ways in which a program may increase in complexity tend to move in unison, making it difficult to identify the multiple dimensions of complexity.

In this section we present the measures used in this study. In a first group we select the main measures that we selected from the literature:



- Lines of Code ($LOC$)
- Source Lines of Code ($SLOC$)
- Lines of Code Equivalent ($LOCE$)
- Total Number of Disjunctions ($TNDj$)
- Total Number of Conjunctions ($TNCj$)
- Total Number of Equalities ($TNE$)
- Total Number of Inequalities ($TNI$)
- Total Number of Decisions ($TND$)
- Number of Atomic Conditions per Decision ($CpD$)
- Nesting Degree ($N$)
- Halstead's Complexity ($HD$)
- McCabe's Cyclomatic Complexity ($MC$)

Let's have a look at the measures that are directly based on source lines of code (in C-based languages). The $LOC$ measure is a count of the number of semicolons in a method, excluding those within comments and string literals. The $SLOC$ measure counts the source lines that contain executable statements, declarations, and/or compiler directives. However, comments, and blank lines are excluded. The $LOCE$ measure [191] is based on the idea of weighing each source line of code depending on how nested it is. The previous three measures based on the lines of code have several disadvantages:

- Depend on the print length
- Depend of the programmer's style for writing source code
- Depend on how many statements does one put in one line

We have analyzed several measures as the total number of disjunctions (OR operator) and conjunctions (AND operator) that appear in the source code, these operators join atomic conditions. The number of (in)equalities is the number of times that the operator $(!=)$ $==$ is found in atomic conditions of a program. The total number of decisions and the number of atomic conditions per decision do not require any comment. The nesting degree is the maximum number of control flow statements that are nested one inside another. In the following paragraphs we describe the McCabe's cyclomatic complexity and the Halstead complexity measures in detail.

Halstead complexity measures are software metrics [93] introduced by Maurice Howard Halstead in 1977. Halstead's Metrics are based on arguments derived from common sense, information theory and psychology. The metrics are based on four easily measurable properties of the program, which are:

- $n1$ = the number of distinct operators
- $n2$ = the number of distinct operands
- $N1$ = the total number of operators
- $N2$ = the total number of operands

From these values, six measures can be defined:

- Halstead Length (HL): $N = N1 + N2$
- Halstead Vocabulary (HV): $n = n1 + n2$
- Halstead Volume (HVL): $V = N * \log_2 n$
- Halstead Difficulty (HD): $HD = \frac{n_1}{2} * \frac{N_2}{n_2}$
- Halstead Level (HLV): $L = \frac{1}{HD}$
- Halstead Effort (HE): $E = HD * V$
- Halstead Time (HT): $T = \frac{E}{\beta}$
- Halstead Bugs (HB): $B = \frac{E^{2/3}}{3000}$

The most basic one is the Halstead Length, which simply totals the number of operators and operands. A small number of statements with a high Halstead Volume would suggest that the individual statements are quite complex. The Halstead Vocabulary gives a clue on the complexity of the statements. For example, it highlights if a small number of operators are used repeatedly (less complex) or if a large number of different operators are used, which will inevitably be more complex. The Halstead Volume uses the length and the vocabulary to give a measure of the amount of code written. The Halstead Difficulty uses a formula to assess the complexity based on the number of unique operators and operands. It suggests how difficult the code is to write and maintain. The Halstead Level is the inverse of the Halstead Difficulty: a low value means the program is prone to errors. The Halstead Effort attempts to estimate the amount of work that it would take to recode a particular method. The Halstead Time is the time to implement or understand a program and it is proportional to the effort. The experiments were used for calibrating this quantity ($\beta = 18$). Finally, the Halstead Bugs attempts to estimate the number of bugs that exist in a particular piece of code.

McCabe's cyclomatic complexity is a complexity measure related to the number of ways there exists to traverse a piece of code. This measure determines the minimum number of test cases needed to test all the paths using linearly independent circuits [146]. Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of sentences of a program (basic blocks), and a directed edge connects two nodes if the second group of sentences might be executed immediately after the first one. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program, and is formally defined as follows:

$$v(G) = E - N + 2P; \tag{6.1}$$

where $E$ is the number of edges of the graph, $N$ is the number of nodes of the graph and $P$ is the number of connected components.

In Figure 6.1, we show an example of control flow graph (G). It is assumed that each node can be reached by the entry node and each node can reach the exit node. The maximum number of linearly independent circuits in G is $9 - 6 + 2 * 1 = 5$, so 5 is the cyclomatic complexity of the example.

The correlation between the cyclomatic complexity and the number of software faults has been studied in some research articles [28,118]. Most such studies find a strong positive correlation between the cyclomatic complexity and the errors: the higher the complexity the larger the number

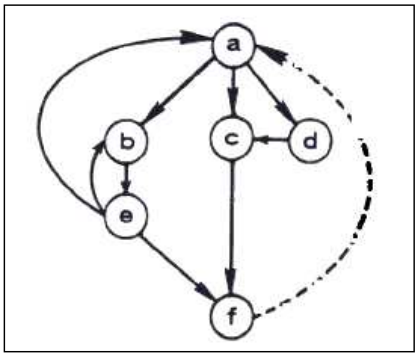



Figure 6.1: The original graph of the McCabe's article.

of faults. For example, a 2008 study by metric-monitoring software supplier Energy [63] analyzed classes of open-source Java applications and divided them into two sets based on how common mistakes were found in them. They found a strong correlation between the cyclomatic complexity and their faultiness, with classes with a combined complexity of 11 having a probability of being fault-prone of just 0.28, rising to 0.98 for classes with a complexity of 74.

In addition to this correlation between complexity and errors, a connection has been found between complexity and difficulty to understand software. Nowadays, the subjective reliability of software is expressed in statements such as "I understand this program well enough to know that the tests I have executed are adequate to provide my desired level of confidence on it". For that reason, we make a close link between complexity and difficulty of discovering errors. Software complexity metrics developed by Halstead and McCabe are related to the difficulty programmers experience in locating errors in code [55]. They can be used in providing feedback to programmers about the complexity of the code they have developed and to managers about the resources that will be necessary to maintain particular sections of code.

Since McCabe proposed the cyclomatic complexity, it has received several criticisms. Weyuker concluded that one of the obvious intuitive weaknesses of the cyclomatic complexity is that it makes no provision for distinguishing between programs which perform very little computation and those which perform massive amounts of computation, provided that they have the same decision structure [218]. Piwarski noticed that cyclomatic complexity is the same for $N$ nested `if` statements and $N$ sequential `if` statements [174]. Moreover, we find the same weaknesses in the group of Halstead's metrics. No notice is made for the nesting degree, which may increase the effort required by the program severely. The solution of both McCabe's and Halstead's weakness is a factor to consider that a nested statement is more complex. For example, we have also studied the LOCE measure that takes into account whether a statement is nested or not.

The proposed existing measures of decision complexity tend to be based upon a graph theoretical analysis of a program control structure like McCabe's complexity. Such measures are meaningful at the program and subprogram level, but metrics computed at those levels will depend on program or subprogram size. However, the values of these metrics primarily depend upon the number of decision points within a program. This suggests that we can compute a size-independent measure of decision complexity by measuring the density of decisions within a program. In addition we have considered making the LOCE measure size-independent. The resulting expression takes into account the nesting degree and the density of the sentences. Following this assumption, we consider in this work two measures derived from some of the first group:

- Density of Decisions (DD) = $TND/LOC$.
- Density of LOCE (DLOCE) = $LOCE/LOC$.

Finally, we use a dynamic measure called Branch Coverage, which is the percentage of branches of the program that are traversed. This coverage measure is used in most of the related articles in the literature and was formally defined in Section 2.2.1 (Definition 2.6).

## 6.3 Branch Coverage Expectation

This section is aimed at presenting our main contribution in this chapter, a new complexity measure that might help testers to estimate the difficulty of testing a piece of code. The definition of the new measure lies on a Markov chain that represents the program. In this section we briefly explain the characteristics of a Markov chain and the way we generate a model of a given program. The Markov model of the program can be used not only to compute the BCE, but also to estimate the number of random test cases that must be generated to achieve a concrete value of branch coverage. We first introduce the required concepts of Markov chains [124].

### 6.3.1 Markov Chain

A *first order Markov chain* is a random sequence of states $X_t$ where each state depends only on the previous one. That is, $P(X_{t+1} = j|X_k; 0 \leq k < t) = P(X_{t+1} = j|X_t)$ for all $t \in \mathbb{N}$. We consider here that the set of possible states is finite and, without loss of generality, we label the states using elements of the set $[n] = \{1, ..., n\}$. The conditional probabilities of a first order Markov chain $P(X_{t+1} = j|X_t = i) = P_{ij}(t)$ are called *one-step transition probabilities* and the matrix $P(t) = [P_{ij}(t)]$ is the so-called *transition probability matrix*. We will assume here that these probabilities do not depend on the step $t$, and thus, $P_{ij}(t) = P_{ij}$ for all $t$. The Markov chains fulfilling this property are called *homogeneous*. Two properties of the transition probability matrices are:

$$P_{ij} \geq 0, \tag{6.2}$$

$$\sum_{j=1}^{n} P_{ij} = 1. \tag{6.3}$$

Matrices fulfilling the above equations are called *stochastic*. Let us denote with the column vector $q(t)$ the probability distribution of the states at step $t$. The component $q_i(t)$ is the probability of having state $i$ at step $t$. A state which is reached infinitely often in a finite Markov chain is called *positive-recurrent*. If every state in a Markov chain can be reached from every other state, then we say that the Markov chain is *irreducible*. For irreducible Markov chains having only positive-recurrent states the probability distribution of the states $q(t)$ tends to a given probability distribution $\pi$ as the time tends to infinite. This probability distribution $\pi$ is called the *stationary distribution* and can be computed solving the following linear equations:

$$\pi^T P = \pi^T, \tag{6.4}$$

$$\pi^T 1 = 1. \tag{6.5}$$

### 6.3.2 Definition of the Branch Coverage Expectation

In our case the Markov model is built from the control flow graph of the program, where the states of the Markov chain are the basic blocks of the program. A *Basic Block* (BB) is a portion of the code that is executed sequentially with no interruption. It has one entry point and one exit point, meaning that only the last instruction can be a jump. Whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order. In order to completely characterize a Markov chain we must assign a value to the edges among vertices. The transition probabilities of all branches are computed according to the logical expressions that appear in each condition. We recursively define this probability as follows:

$$P(c1\&\&c2) = P(c1) * P(c2), \tag{6.6}$$

$$P(c1||c2) = P(c1) + P(c2) - P(c1) * P(c2), \tag{6.7}$$

$$P(\neg c1) = 1 - P(c1), \tag{6.8}$$

$$P(a < b) = \frac{1}{2}, \tag{6.9}$$

$$P(a \leq b) = \frac{1}{2}, \tag{6.10}$$

$$P(a > b) = \frac{1}{2}, \tag{6.11}$$

$$P(a \geq b) = \frac{1}{2}, \tag{6.12}$$

$$P(a == b) = q, \tag{6.13}$$

$$P(a! = b) = 1 - q, \tag{6.14}$$

where $c1$, $c2$ are conditions and $a$, $b$ are integers.

We establish a $1/2$ probability when the operators are ordering relational operators ($<, \leq, >, \geq$). Despite that the actual probability in a random situation is not always $1/2$, we have selected the value with the lowest error rate. In the case of equalities and inequalities the probabilities are $q$ and $1 - q$, respectively, where $q$ is a parameter of the measure and its value should be adjusted based on the experience. Satisfying an equality is, in general, a hard task and, thus, $q$ should be close to zero. This parameter could be highly dependent on the data dependencies of the program. The quality of the complexity measure depends on a good election for $q$. Based on a previous phase for setting parameters, we use $q = 1/16$ for the experimental analysis.

Then, once we have the CFG completed with the transition probabilities, the generation of the transition matrix is automatic. This matrix relates the states and the probability to move from one to another. We assume, without loss of generality, that there is only one entry and exit basic block in the code. Then, in order to obtain a positive-recurrent irreducible Markov chain we add a fictional link from the exit to the entry basic block (labelled as $BB_1$) having probability 1. We then compute the stationary probability $\pi$ and the frequency of appearance of each basic block in one single execution of the program ($E[BB_i]$). The stationary probability of a basic block is the probability of appearance in infinite program executions starting in any state. On the other hand, the frequency of appearance of a basic block is the mathematical expectation of traversing the basic block in one single execution and is computed as:

$$E[BB_i] = \frac{\pi_i}{\pi_1}, \tag{6.15}$$

where $\pi_1$ is the stationary probability of the entry basic block, $BB_1$.

Thus, the expectation of traversing a branch $(i, j)$ is computed from the frequency of appearance of the previous basic block and the probability to take the concrete branch from the previous basic block as:

$$E[BB_i, BB_j] = E[BB_i] * P_{ij} \tag{6.16}$$

Finally, we define the *Branch Coverage Expectation* as the average of the values $E[BB_i, BB_j]$ with a value lower than $1/2$. If a program has a low value of BCE then a random test data generator is supposed to require a large number of test cases to obtain full branch coverage. The BCE is bounded in the interval (0, 1/2]. Formally, let $A$ be the set of edges with $E[BB_i, BB_j] < 1/2$:

$$A = \left\{ (i,j) \middle| E[BB_i, BB_j] < \frac{1}{2} \right\}. \tag{6.17}$$

Then, the BCE is defined as:

$$BCE = \frac{1}{|A|} \sum_{(i,j)\in A} E[BB_i, BB_j]. \tag{6.18}$$

In the experimental section we analyze the new complexity measure over program artifacts and we shed light on the **RQ1** by providing a testing complexity measure to quantify the difficulty of testing a piece of code. But first, we illustrate here its computation based on the piece of code shown in Figure 6.2. First, we compute the Control Flow Graph (CFG) of the piece of code, which can be seen in Figure 6.3. This CFG is composed of BBs and transitions among the BBs. Interpreted as a Markov chain, the basic blocks are the states, and the transitions are defined by the probabilities to move from one basic block to another. These probabilities depend on the condition associated to a concrete branch. For example, to move from $BB_1$ to $BB_2$ in our example, the condition $(x < 0)||(y < 2)$ must be true, then according to Equations (6.7) to (6.14) the probability of this transition is:

$$P((x < 0)||(y < 2)) = P(x < 0) + P(y < 2) - P(x < 0) * P(y < 2) = \tfrac{1}{2} + \tfrac{1}{2} - \tfrac{1}{2} * \tfrac{1}{2} = \tfrac{3}{4} = 0.75.$$

Once we have computed all the transition probabilities, we build the transition matrix that represents the Markov chain.

$$P = \begin{pmatrix} 0.0 & 0.75 & 0.25 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1 \\ 0.0 & 0.0 & 0.0 & 0.75 & 0.25 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.75 & 0.25 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1 \\ 1 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

We can now compute the stationary probabilities $\pi$ and the frequency of appearance $E[BB_i]$ of the basic blocks in one execution of the program (see Table 6.1). It is sure that the control flow of the program traverses exactly once the $BB_1$ and $BB_6$ in one run. In this way, the start and the end of the program always have a $E[BB_i] = 1$. An example of the computation of the mathematical expectation is:

$$E(BB_2) = \frac{\pi_2}{\pi_1} = \frac{0.1875}{0.2500} = 0.75.$$

```
/* BB1 */
if (x < 0) || (y < 2)
{
    /* BB2 */
    y=5;
}
else
{
    /* BB3 */
    x=y-3;
    while (y > 5) || (x > 5)
    {
           /* BB4 */
           y=x-5;
    }
    /* BB5 */
    x=x-3;
}
/* BB6 */
```

Figure 6.2: A piece of code to illustrate the computation of Branch Coverage Expectation.

Table 6.1: Stationary probabilities and the frequency of appearance of the basic blocks of the piece of code shown In Figure 6.2.

| | Stationary Probabilities $\pi_i$ | Frequency of Appearance $E[BB_i]$ |
|---|---|---|
| BB1 | 0.2500 | 1.00 |
| BB2 | 0.1875 | 0.75 |
| BB3 | 0.0625 | 0.25 |
| BB4 | 0.1875 | 0.75 |
| BB5 | 0.0625 | 0.25 |
| BB6 | 0.2500 | 1.00 |

Now, we are able to compute the probability of appearance of a branch in one single run. For example the expectation of traversing the branch $(BB_3, BB_4)$ is:

$E[BB_3, BB_4] = E(BB_3) * P_{34} = \frac{1}{4} * \frac{3}{4} = \frac{3}{16} = 0.1875.$

In Figure 6.4 we show the mathematical expectations of traversing all the branches of the CFG of our example in one single execution. So, finally we can compute the BCE by averaging the expectations of traversing the branches which have a value lower than 1/2. We have excluded those values equals to 1/2 because both branches have the same value. In case all branches have the expectation of 1/2, then the BCE is 1/2. In addition, a program with a Branch Coverage Expectation value of 1/2 would be the easiest one to be tested. In this example the value of Branch

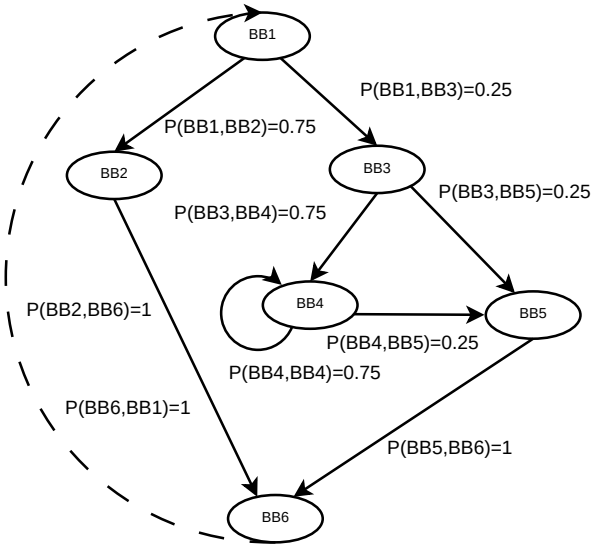



Figure 6.3: The CFG and the probabilities used to build a Markov Chain of the piece of code of Figure 6.2.

Coverage Expectation is:

$$\text{BCE} = \frac{E[BB_1,BB_3]+E[BB_3,BB_4]+E[BB_3,BB_5]+E[BB_4,BB_5]+E[BB_5,BB_6]}{5} = \frac{\frac{1}{4}+\frac{3}{16}+\frac{1}{16}+\frac{3}{16}+\frac{1}{4}}{5} = 0.1875.$$

Based on the model of a program, we can also provide an estimation of the number of random test cases that must be generated to obtain a concrete coverage. Following with the example, executing the branch $(BB_3, BB_5)$ is more difficult according to the expectations than the others. The inverse of this expectation, is the expected number of random test cases that must be generated to execute the branch. In this example, the number of expected test cases needed to traverse the branch between BB3 and BB5 is:

Number of test cases (BB3,BB5)=$\frac{1}{\frac{1}{16}} = 16$.

From these estimations, we can create a theoretical prediction of the evolution of the coverage depending on the number of generated test cases. This contribution could help the testers to obtain some knowledge about the possible evolution of the testing phase. In Section 6.6.3 we compare our theoretical prediction with the results obtained by a test data generator.

## 6.4 Validation of the Branch Coverage Expectation

Software applications are essential for Industry and software measurement is a key factor in understanding and controlling software development practices. Consequently, measures must represent accurately those attributes which they quantify. Thus, validation is critical when a new measure is introduced. The software measurement validation implies two basic methods, theoretical and

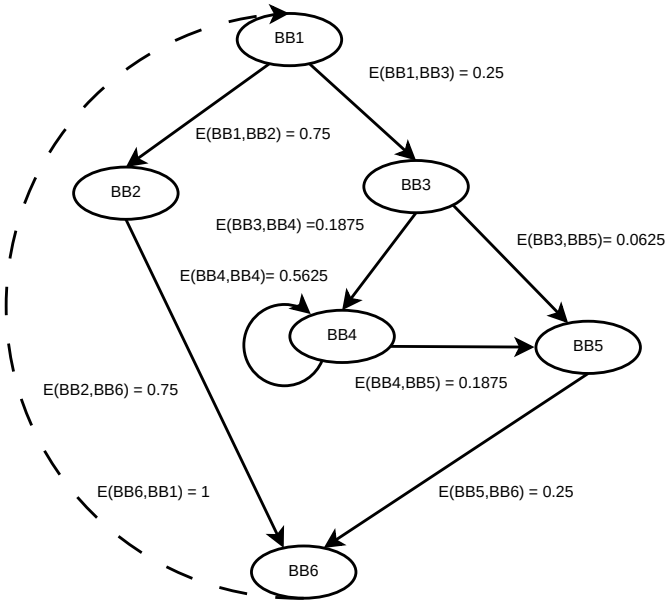



Figure 6.4: The CFG and the expectations of traversing each branch in the piece of code of Figure 6.2.

empirical validation. Theoretical methods allow us to say that a measure is valid with respect to certain criteria, meanwhile empirical methods only provide evidence of validity or invalidity. In the experimental section we obtain evidences of the validity of the proposed measure, but first, in this section we focus on theoretical validity using the framework proposed by Kitchenham et al. [120]. The requirements defined when validating a measure are attribute validity, unit validity, instrument validity, and protocol validity:

- **Attribute validity**: Attributes are the properties that an entity possesses. For a given attribute, there is a relationship of interest in the empirical world that we want to capture formally in the mathematical world. The attribute we consider for our measure is the testing complexity of a piece of software. Two measures are defined to estimate testing complexity: "branch coverage expectation"(BCE) and "number of expected test cases"(related to the inverse of BCE). And both are estimated by capturing transition probabilities at each edge among vertices (decisions and conditions of branches). The measure is able to satisfy the proposed criteria. There could be two different programs for which the measure results in different values. Our measure also obeys the *Representation Condition*. The BCE value of two programs is the same, when they are the same except they have different labels. So, different programs can have the same BCE value.
- **Unit validity**: A measure maps an empirical attribute to the formal, mathematical world. A measurement unit determines how we measure an attribute. We define the unit of BCE by reference to a wider theory explained in the previous Subsection 6.3.1 (Markov Chain). We must highlight that the inverse of the BCE measure is related to the number of test cases needed to achieve full coverage by a random test data generator. Then, the measurement unit used by the BCE is $number\ of\ test\ cases^{-1}$. In conclusion, we use an alternative unit

that is valid because is an admissible transformation from an original unit (number of test cases).

- **Instrument validity**: Our instrument model defines how to capture the data, and it is also theory based. The validity again depends on the validity of the underlying theory. It can be defined by reference to properties of the control flow graph. Our instrument model is valid because the underlying theory-based model is valid.
- **Protocol validity**: Measurement protocols let us measure a specific attribute on a specific entity consistently and repeatedly. We can measure a specific attribute of a program consistently, repeatable, and the measurement is independent of the measurer. Our measurement protocols are unambiguous, self-consistent, and prevent problems such as double counting. The same measurement could be done with a different measurer obtaining the same results. A protocol that does not violate these criteria is usually validated by peer acceptance rather than logical or empirical studies.

Empirical validation of our proposed measure is also required, so we are using a tool for generating test data described in Section 5.2, a tool for generating synthetic programs that we introduce in Section 6.5.2, and a benchmark of real and synthetic programs also described in next Section 6.5.3, which will help us to compare our estimation of BCE with the result of branch coverage obtained by the execution of our testing tool. Branch Coverage is the dynamic measure used in this work to measure the difficulty of testing a program. The test data generator goal is generating a test suite that covers all the source code, with the aim of helping the tester to find errors in the code.

Finally, we claim that our measure is valid because we are unable to invalidate it using the Kitchenham et al. framework.

## 6.5 Empirical Validation Setup

In this section we outline the configuration of the optimization algorithms used as core of the test data generation tool (Section 5.2) used for empirical validation of our proposed measure. Then, we outline the main characteristics of the used benchmark of test programs.

### 6.5.1 Algorithms Details

In the following we focus on the details of the specific EAs used in this work to perform the test data generation. In this work we applied a GA described in Section 4.1.1 and an ES introduced in Section 4.1.2.

In our GA the individuals are vectors of integer values. As the recombination operator we use the uniform crossover (UX), in which each component of the new solution is randomly selected from the two parents. The formal definition is the same as equation (4.6) with $bias = 0.5$. The mutation operator adds a random value to the components of the vector. That is,

$$\mathbf{x}_i = \mathbf{x}_i + U(-500, 500)$$

where the probability distribution of these random values is a uniform distribution in the range $[-500, 500]$. However, not all the components of the individual are perturbed, only half of them are, randomly selected.

As we said when we explained the test data generator (Section 5.2) the generator breaks down the global objective (to cover all the branches) into several partial objectives consisting of dealing

with only one branch of the program. Thus, two stopping conditions exist: one for partial objectives and the other one for the whole test data generation process. The search for one partial objective stops when 1,000 evaluations are performed while the test data generation process ends after 150,000 evaluations. For comparison purposes we also use a random algorithm (RND) for the generation of test inputs. We use the same evaluations than the evolutionary approaches (150,000 evaluations in total). To finish this section, we show in Table 6.2 a summary of the parameters used by the two EAs in the experimental section. Note that the ES algorithm uses the standard operators described previously in Section 4.1.2.

Table 6.2: Parameters of the two EAs used in the experimental section.

| | ES | GA |
|---|---|---|
| Population | 25 indivs. | 25 indivs. |
| Selection | Random, 5 indivs. | Random, 5 indivs. |
| Mutation | Gaussian | Add $U(-500, 500)$ |
| Crossover | discrete (bias = 0.6) + arith. + arith. | Uniform |
| Replacement | Elitist | Elitist |
| Stopping cond. | 1,000 evals. | 1,000 evals. |
| Total Evals. | 150,000 evals. | 150,000 evals. |

### 6.5.2 Program Generator Tool

We have designed an automatic program generator that generates programs with values for the "static measures" like nesting degree, number of atomic conditions, number of (in)equalities, that are similar to the ones of the real-world software, but the generated programs do not solve any concrete problem. Our program generator is able to create programs for which total branch coverage is possible. We propose this generator with the aim of generating a big benchmark of programs with certain characteristics chosen by the user.

The automatic program generation raises a non-trivial research question: are the generated programs "realistic"? That is, could them be found in real-world? Using automatic program generation it is not likely to find programs that are similar to the ones who a programmer would make. This is especially true if the program generation is not driven by a specification. However, this is not a drawback for us, since we are only interested in some static measures of the programs and branch coverage. In this situation, "realistic programs" means programs that have similar values for the considered static measures as the ones found in real-world; and we can easily fulfil this requirement.

In a first approximation we could create a program using a representation based on a general tree and a table of variables. The tree stores the sentences that are generated and the table of variables stores basic information about the variables declared and their possible use. With these structures, we are able to generate programs, but we can not ensure that all the branches of the generated programs are reachable. The unreachability of all the branches is a quite common feature of real-world programs, so we could stop the design for the generator at this stage. However, another objective of the program generator is to be able of creating programs that can be used to compare the performance of different algorithms, programs for which total coverage is reachable are desirable. With this goal in mind we introduce logic predicates in the program generation process.

The program generator is parameterizable, the user can set several parameters of the program under construction ($PUC$). Thus, we can assign through several probability distributions the number of sentences of the $PUC$, the number of variables, the maximum number of atomic conditions in a decision, and the maximum nesting degree by setting these parameters. The user can define the structure of the $PUC$ and, thus, its complexity. Another parameter the user can tune is the percentage of control structures or assignment sentences that will appear in the code. By tuning this parameter the program will contain the desired density of decisions.

Once the parameters are set, the program generator builds the general scheme of the PUC. It stores in the used data structure (a general tree) the program structure, the visibility, the modifiers of the program, and creates a main method where the local variables are first declared. Then, the program is built through a sequence of basic blocks of sentences where, according to a probability, the program generator decides which sentence will be added to the program. The creation of the entire program is done in a recursive way. The user can decide whether all the branches of the generated program are reachable (using logic predicates).

If total reachability is desired, logic predicates are used to represent the set of possible values that the variables can take at a given point of the PUC. Using these predicates the range of values that a variable can take is known. This range of values is useful to build a new condition that can be true or false. For example, if at a given point of the program we have the predicate $x \leq 100$ we know that a forthcoming condition $x \leq 3$ will be always true and if this condition appears in an `if statement`, the `else` branch will not be reachable. Thus, the predicates are used to guide the program construction to obtain a 100% coverable program.

In general, at each point of the program the predicate is different. During the program construction, when a sentence is added to the program, we need to compute the predicate at the point after the new sentence. For this computation we distinguish two cases. First, if the new sentence is an assignment then the new predicate $CP'$ is computed after the previous one $CP$ by updating the values that the assigned variable can take. For example, if the new sentence is $x = x + 7$ and $CP \equiv x \leq 3$, then we have $CP' \equiv x \leq 10$.

Second, if the new sentence is a control statement, an `if` statement for example, then the program generator creates two new predicates called True-predicate ($TP$) and False-predicate ($FP$). The $TP$ is obtained as the result of the AND operation between $CP$ and the generated condition related to the control statement. The $FP$ is obtained as the result of the AND operation between the $CP$ and the negated condition. In order to ensure that all the branches can be traversed, we check that both, $TP$ and $FP$ are not equivalent to $false$. If any of them were false, this new predicate is not valid and a new control structure would be generated.

Once these predicates are checked, the last control statement is correct and new sentences are generated for the two branches, the predicates are computed inside the branches in the same way. After the control structure is completed, the last predicates of the two branches are combined using the OR operator and the result is the predicate after the control structure. In Figure 6.5 we illustrate the previous explanation with one example. At a certain point of the program's execution our current predicate ($CP_1$) is $x \leq 3$. The new sentence is an `if` statement with an associated decision $x < 0$. Then, the program generator creates two new predicates. The first one ($CP_2$) is $CP_1 \wedge x < 0 \equiv x < 0$. The second one is the $AND$ operation between $CP_1$ and the negation of $x < 0$, which is $x \geq 0$. The resulting expression is $CP_4 \equiv 0 \leq x \leq 3$. Next, the program generator modifies the predicates according to the assignment sentences. Finally, the resulting expression after the execution of the `if` statement is $CP_6 \equiv x < 0 \wedge y = 5 \vee -3 \leq x \leq 0$. This expression is the $OR$ operation between $CP_3$ (true branch) and $CP_5$ (false branch). Those values that satisfy the logic predicate may participate in the following generated sentences.



```
/* CP1 ≡ x ≤ 3 */
if (x < 0)
{
    /* CP2 ≡ TP1 ≡ x ≤ 3 ∧ x < 0 ≡ x < 0 */
    y=5;
    /* CP3 ≡ x < 0 ∧ y = 5 */
}
else
{
    /* CP4 ≡ FP1 ≡ x ≤ 3 ∧ x ≥ 0 ≡ 0 ≤ x ≤ 3 */
    x=x-3;
    /* CP5 ≡ −3 ≤ x ≤ 0 */
}
/* CP6 ≡ x < 0 ∧ y = 5 ∨ −3 ≤ x ≤ 0 */
```

Figure 6.5: Illustration of the predicates transformation.

### 6.5.3 Benchmark of Test Programs

The empirical validation is carried out using synthetic programs and real programs. In the following we describe the main characteristics of the programs generated by our program generator and a set of real programs extracted from the literature.

#### Synthetic Programs

The program generator can create programs having the same value for the static measures, as well as programs having different values for the measures. The programs we generated for this work can be separated in two groups. One group is characterized by being 100% coverable (called 100%CP), thus all the branches are reachable. The main advantage of these programs is that algorithms can be tested and analyzed on fair way. This kind of programs is not easy to find in the literature. On the other hand, the other group of programs does not guarantee a 100% branch coverage, called ¬100%CP, this fact makes them similar to the real world programs.

The methodology applied for the program generation was the following. First, we analyzed a set of Java source files from the JDK 1.6 (java.util.*, java.io.*, java.sql.*, etc.) and we computed the static measures on these files. Next, we used the ranges of the most interesting measures, obtained in this previous analysis as a guide to generate Java source files having values in the same range for the static measures. These values are realistic with respect to the static measures, making the following study meaningful. Our program generator takes into account the desired values for the number of atomic conditions, the nesting degree, the number of sentences and the number of variables. With these parameters our program generator creates a program with a defined control flow graph containing several decisions. The main features of the generated programs are: they deal with integer input parameters, their conditions are joined by whichever logical operator and they are randomly generated. This way, we generated programs with the values in the ranges shown in Table 6.3.

Table 6.3: Range of values for some static measures from the two benchmarks of programs.

| | 100% CP | $\neg 100\% CP$ |
|---|---|---|
| SLOC | 33-150 | 33-235 |
| Nesting | 1-4 | 1-7 |
| Conditions | 1-4 | 1-7 |
| Decisions | 3-50 | 1-37 |
| McCabe | 5-125 | 3-127 |

Finally, we generated a total of 2600 (800 in 100%CP and 1800 in ¬100%CP) Java programs[1] using our program generator. With the aim of studying the BCE, we applied our test data generator using a GA and an ES as optimization algorithms and a random test data generator (RND). The test data generators proceed by generating test data until total coverage is obtained or a maximum of 150,000 test cases are generated. Since we are working with stochastic algorithms, we perform in all the cases 30 independent runs of the algorithms to obtain a very stable average of the branch coverage. The experimental study requires a total of $2600 \times 30 \times 3 = 234,000$ independent runs of the test data generators. We have followed the statistical analysis procedure described in Section 3.3.2 to analyze the obtained results to compare them with a certain level of confidence. The statistical test that we have carried out is the non-parametric Kruskal-Wallis test used to compare the average of the algorithms. We always consider in this work a confidence level of 95% (i.e., $p$-value under 0.05) in the statistical tests, which means that the differences are unlikely to have occurred by chance with a confidence of 95%.

### Real Programs

In order to improve the interest of our work we propose an additional benchmark of real programs. It is composed of 10 real programs extracted from the literature [15, 113, 145]. Some of them have been extracted from the book *C Numerical Recipes*, available on-line at `http://www.nr.com/`. They deal with real and integer input values and some of them also contain loops. The programs are listed in Table 6.4, where we inform on the maximum nesting degree, the lines of code (LOC), the number of branches, and the number and type of input arguments.

Table 6.4: Characteristics of the real programs.

| Name | ND | LOC | Branches | Arguments | Description |
|---|---|---|---|---|---|
| calday | 2 | 47 | 22 | 3 Integer | Calculate the day of the week |
| gcd | 2 | 28 | 8 | 2 Integer | Greatest common denominator |
| line | 8 | 92 | 36 | 8 Integer | Check if two rectangles overlap |
| numbers | 3 | 71 | 28 | 1 Integer | Parse a big number from integer to string |
| qformula | 2 | 24 | 4 | 3 Double | Solve Real Equations |
| qformulas | 2 | 22 | 6 | 3 Integer | Solve Integer Equations |
| tmichael | 5 | 69 | 20 | 3 Integer | Classify triangles in 4 types: Michael |
| triangle | 4 | 53 | 28 | 3 Integer | Classify triangles in 4 types: Our implementation |
| tsthamer | 3 | 76 | 26 | 3 Integer | Classify triangles in 5 types: Sthamer |
| twegener | 3 | 46 | 26 | 3 Double | Classify triangles in 5 types: Wegener |

[1] They are available at http://neo.lcc.uma.es/staff/javi/resources.html

## 6.6 Empirical Results

In this section we describe the experimental analysis performed and we interpret the relationship of the studied measures. We divide the main study in four subsections. In the first one, we analyze the static measures in order to reveal their correlations. In the second subsection, we highlight the existing relationship between the static measures and the code coverage. We analyze which of them are more appropriate for estimating the difficulty for a computer to generate an adequate test suite. In the third subsection, we use the Markov model of programs to predict the relationship between coverage and the number of required test cases. We analyze if our theoretical prediction is similar to a real execution of an automatic test data generator. Finally, we perform the study on the real-world programs. For computing correlations among measures we use the Spearman's correlation coefficient $\rho$. This coefficient takes into account the rank of the values of the samples instead of the samples themselves.

### 6.6.1 Analysis of the Correlation Between the Static Measures

In this section we analyze the existing relationship among the static measures of the generated programs. With this previous study we want to clarify the possible similarities and differences of the analyzed static measures in this work.

In this study there are three different measures that try to rate the complexity of a program: McCabe's complexity, the Halstead Difficulty and LOCE. In Tables 6.7 and 6.8 we show a comparison of correlation between these measures, the nesting degree and the group of measures derived from some of the first group (density of decisions and density of LOCE). We show these measures because they are higher correlated with the branch coverage among a number of them. In Tables 6.5 and 6.6 the reader could check all the correlations coefficients for the 100%CP benchmark and the ¬100%CP benchmark, respectively. In addition, we include the BCE measure in order to compare it with the other studied measures.

As we can see, the three measures that rate the complexity (MC, HD and LOCE) are highly correlated among them in the 100%CP benchmark and less correlated in the ¬100%CP one. This gives us a clue about the similarities that the complexity measures have among them. On the other hand, the nesting degree is lowly correlated with McCabe's complexity and Halstead Difficulty. In the case of LOCE, the correlations with the nesting degree are 0.344 in 100%CP and 0.692 in ¬100%CP. This was expected because the LOCE measure weighs the nested statements. In addition, we can remark that the Halstead Difficulty does not correlate with the density of decisions in 100%CP and ¬100%CP (0.052 and 0.023, respectively).

We can also observe that the static measures that are highly correlated in one benchmark are highly correlated in the other one too. This is the case of the relationship between nesting, density of decisions and density of LOCE that must be emphasized. All of them are highly correlated, being the correlation between nesting degree and density of LOCE 0.877 in 100%CP and 0.870 in ¬100%CP, nesting degree and density of decisions 0.765 in 100%CP and 0.708 in ¬100%CP and density of decisions and density of LOCE 0.912 in 100%CP and 0.774 in ¬100%CP.

Once we have analyzed the other static measures, we report the correlation coefficients of our proposal and the most important static measures studied in this work. The nesting degree must be emphasized because it is the most correlated static measure with the BCE, -0.540 in 100%CP and -0.575 in ¬100%CP, what means that the nesting degree is the most similar measure. In addition, we can see that our proposal is not correlated with McCabe's and Halstead's complexities.

This analysis of the three rates of complexity is not complete if we do not highlight the static

Table 6.5: The correlation coefficients among all the measures analyzed in the benchmark 100%CP

| | HD | MC | LOCE | N | DD | DLOCE | BCE | LOC | SLOC | TNDj | TNCj | TNE | TNI | TND | CpD | HL | HV | HVL | HLV | HE | HT | HB | ES | GA | RND |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| HD | - | 0.796 | 0.786 | -0.108 | 0.052 | -0.035 | 0.285 | 0.932 | 0.853 | 0.742 | 0.731 | 0.644 | 0.639 | 0.799 | 0.454 | 0.870 | 0.842 | 0.864 | 1.0 | 0.920 | 0.920 | 0.864 | 0.070 | -0.101 | 0.077 |
| MC | 0.796 | - | 0.965 | 0.266 | 0.519 | 0.408 | 0.025 | 0.805 | 0.962 | 0.925 | 0.934 | 0.829 | 0.811 | 0.985 | 0.524 | 0.976 | 0.969 | 0.977 | -0.796 | 0.954 | 0.954 | 0.977 | -0.150 | -0.226 | -0.074 |
| LOCE | 0.786 | 0.965 | - | 0.344 | 0.515 | 0.474 | -0.038 | 0.796 | 0.974 | 0.884 | 0.882 | 0.822 | 0.789 | 0.976 | 0.501 | 0.945 | 0.938 | 0.945 | -0.786 | 0.921 | 0.921 | 0.945 | -0.186 | -0.251 | -0.133 |
| N | -0.108 | 0.266 | 0.344 | - | 0.765 | 0.877 | -0.540 | -0.207 | 0.180 | 0.235 | 0.240 | 0.311 | 0.234 | 0.276 | 0.136 | 0.138 | 0.127 | 0.139 | 0.108 | 0.089 | 0.089 | 0.139 | -0.543 | -0.381 | -0.434 |
| DD | 0.052 | 0.519 | 0.515 | 0.765 | - | 0.912 | -0.377 | -0.043 | 0.405 | 0.449 | 0.489 | 0.485 | 0.437 | 0.538 | 0.283 | 0.368 | 0.367 | 0.372 | -0.052 | 0.302 | 0.302 | 0.372 | -0.439 | -0.304 | -0.311 |
| DLOCE | -0.035 | 0.408 | 0.474 | 0.877 | 0.912 | - | -0.485 | -0.132 | 0.336 | 0.352 | 0.380 | 0.410 | 0.353 | 0.418 | 0.217 | 0.270 | 0.258 | 0.271 | 0.035 | 0.208 | 0.208 | 0.271 | -0.504 | -0.345 | -0.397 |
| BCE | 0.285 | 0.025 | -0.038 | -0.540 | -0.377 | -0.485 | - | 0.307 | 0.081 | 0.065 | 0.008 | -0.124 | 0.009 | 0.017 | 0.078 | 0.121 | 0.129 | 0.120 | -0.285 | 0.159 | 0.159 | 0.120 | 0.510 | 0.375 | 0.534 |
| LOC | 0.932 | 0.805 | 0.796 | -0.207 | -0.043 | -0.132 | 0.307 | - | 0.879 | 0.753 | 0.730 | 0.634 | 0.646 | 0.810 | 0.419 | 0.891 | 0.892 | 0.890 | -0.932 | 0.910 | 0.910 | 0.890 | 0.136 | -0.053 | 0.120 |
| SLOC | 0.853 | 0.962 | 0.974 | 0.180 | 0.405 | 0.336 | 0.081 | 0.879 | - | 0.884 | 0.878 | 0.794 | 0.778 | 0.973 | 0.492 | 0.975 | 0.970 | 0.975 | -0.853 | 0.960 | 0.960 | 0.975 | -0.091 | -0.194 | -0.050 |
| TNDj | 0.742 | 0.925 | 0.884 | 0.235 | 0.449 | 0.352 | 0.065 | 0.753 | 0.884 | - | 0.773 | 0.813 | 0.719 | 0.897 | 0.515 | 0.919 | 0.908 | 0.919 | -0.742 | 0.900 | 0.900 | 0.919 | -0.119 | -0.175 | -0.036 |
| TNCj | 0.731 | 0.934 | 0.882 | 0.240 | 0.489 | 0.380 | 0.008 | 0.730 | 0.878 | 0.773 | - | 0.734 | 0.806 | 0.905 | 0.497 | 0.913 | 0.901 | 0.913 | -0.731 | 0.895 | 0.895 | 0.913 | -0.158 | -0.235 | -0.072 |
| TNE | 0.644 | 0.829 | 0.822 | 0.311 | 0.485 | 0.410 | -0.124 | 0.634 | 0.794 | 0.813 | 0.734 | - | 0.618 | 0.822 | 0.435 | 0.798 | 0.785 | 0.797 | -0.644 | 0.779 | 0.779 | 0.797 | -0.272 | -0.279 | -0.207 |
| TNI | 0.639 | 0.811 | 0.789 | 0.234 | 0.437 | 0.353 | 0.009 | 0.646 | 0.778 | 0.719 | 0.806 | 0.618 | - | 0.799 | 0.439 | 0.794 | 0.791 | 0.795 | -0.639 | 0.774 | 0.774 | 0.795 | -0.121 | -0.201 | -0.095 |
| TND | 0.799 | 0.985 | 0.976 | 0.276 | 0.538 | 0.418 | 0.017 | 0.810 | 0.973 | 0.897 | 0.905 | 0.822 | 0.799 | - | 0.503 | 0.961 | 0.959 | 0.962 | -0.799 | 0.935 | 0.935 | 0.962 | -0.147 | -0.226 | -0.082 |
| CpD | 0.454 | 0.524 | 0.501 | 0.136 | 0.283 | 0.217 | 0.078 | 0.419 | 0.492 | 0.515 | 0.497 | 0.435 | 0.439 | 0.503 | - | 0.524 | 0.518 | 0.523 | -0.454 | 0.514 | 0.514 | 0.523 | -0.089 | -0.132 | 0.035 |
| HL | 0.870 | 0.976 | 0.945 | 0.138 | 0.368 | 0.270 | 0.121 | 0.891 | 0.975 | 0.919 | 0.913 | 0.798 | 0.794 | 0.961 | 0.524 | - | 0.991 | 1.0 | -0.870 | 0.989 | 0.989 | 1.0 | -0.071 | -0.180 | -0.012 |
| HV | 0.842 | 0.969 | 0.938 | 0.127 | 0.367 | 0.258 | 0.129 | 0.892 | 0.970 | 0.908 | 0.901 | 0.785 | 0.791 | 0.959 | 0.518 | 0.991 | - | 0.994 | -0.842 | 0.971 | 0.971 | 0.994 | -0.061 | -0.172 | -0.003 |
| HVL | 0.864 | 0.977 | 0.945 | 0.139 | 0.372 | 0.271 | 0.120 | 0.890 | 0.975 | 0.919 | 0.913 | 0.797 | 0.795 | 0.962 | 0.523 | 1.0 | 0.994 | - | -0.864 | 0.987 | 0.987 | 1.0 | -0.072 | -0.181 | -0.011 |
| HLV | -1.0 | -0.796 | -0.786 | 0.108 | -0.052 | 0.035 | -0.285 | -0.932 | -0.853 | -0.742 | -0.731 | -0.644 | -0.639 | -0.799 | -0.454 | -0.870 | -0.842 | -0.864 | - | -0.920 | -0.920 | -0.864 | -0.070 | 0.101 | -0.077 |
| HE | 0.920 | 0.954 | 0.921 | 0.089 | 0.302 | 0.208 | 0.159 | 0.910 | 0.960 | 0.900 | 0.895 | 0.779 | 0.774 | 0.935 | 0.514 | 0.989 | 0.971 | 0.987 | -0.920 | - | 1.0 | 0.987 | -0.046 | -0.168 | 0.006 |
| HT | 0.920 | 0.954 | 0.921 | 0.089 | 0.302 | 0.208 | 0.159 | 0.910 | 0.960 | 0.900 | 0.895 | 0.779 | 0.774 | 0.935 | 0.514 | 0.989 | 0.971 | 0.987 | -0.920 | 1.0 | - | 0.987 | -0.046 | -0.168 | 0.006 |
| HB | 0.864 | 0.977 | 0.945 | 0.139 | 0.372 | 0.271 | 0.120 | 0.890 | 0.975 | 0.919 | 0.913 | 0.797 | 0.795 | 0.962 | 0.523 | 1.0 | 0.994 | 1.0 | -0.864 | 0.987 | 0.987 | - | -0.072 | -0.181 | -0.011 |
| ES | 0.070 | -0.150 | -0.186 | -0.543 | -0.439 | -0.504 | 0.510 | 0.136 | -0.091 | -0.119 | -0.158 | -0.272 | -0.121 | -0.147 | -0.089 | -0.071 | -0.061 | -0.072 | -0.070 | -0.046 | -0.046 | -0.072 | - | 0.365 | 0.445 |
| GA | -0.101 | -0.226 | -0.251 | -0.381 | -0.304 | -0.345 | 0.375 | -0.053 | -0.194 | -0.175 | -0.235 | -0.279 | -0.201 | -0.226 | -0.132 | -0.180 | -0.172 | -0.181 | 0.101 | -0.168 | -0.168 | -0.181 | 0.365 | - | 0.403 |
| RND | 0.077 | -0.074 | -0.133 | -0.434 | -0.311 | -0.397 | 0.534 | 0.120 | -0.050 | -0.036 | -0.072 | -0.207 | -0.095 | -0.082 | 0.035 | -0.012 | -0.003 | -0.011 | -0.077 | 0.006 | 0.006 | -0.011 | 0.445 | 0.403 | - |

Table 6.6: The correlation coefficients among all the measures analyzed in the benchmark ¬100%CP

| | HD | MC | LOCE | N | DD | DLOCE | BCE | LOC | SLOC | TNDj | TNCj | TNE | TNI | TND | CpD | HL | HV | HVL | HLV | HE | HT | HB | ES | GA | RND |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| HD | - | 0.698 | 0.359 | -0.062 | 0.023 | 0.014 | 0.051 | 0.664 | 0.648 | 0.653 | 0.651 | 0.557 | 0.569 | 0.463 | 0.441 | 0.764 | 0.576 | 0.747 | -1.0 | 0.872 | 0.872 | 0.747 | 0.069 | 0.067 | 0.079 |
| MC | 0.698 | - | 0.571 | 0.257 | 0.432 | 0.351 | -0.142 | 0.472 | 0.667 | 0.936 | 0.937 | 0.803 | 0.827 | 0.718 | 0.671 | 0.782 | 0.762 | 0.786 | -0.698 | 0.803 | 0.803 | 0.786 | -0.177 | -0.168 | -0.173 |
| LOCE | 0.359 | 0.571 | - | 0.692 | 0.590 | 0.833 | -0.461 | 0.414 | 0.717 | 0.435 | 0.432 | 0.479 | 0.485 | 0.814 | 0.086 | 0.564 | 0.503 | 0.560 | -0.359 | 0.524 | 0.524 | 0.560 | -0.461 | -0.452 | -0.476 |
| N | -0.062 | 0.257 | 0.692 | - | 0.708 | 0.870 | -0.575 | -0.160 | 0.190 | 0.163 | 0.161 | 0.229 | 0.220 | 0.502 | -0.031 | 0.020 | 0.009 | 0.019 | 0.062 | -0.007 | -0.007 | 0.019 | -0.563 | -0.554 | -0.589 |
| DD | 0.023 | 0.432 | 0.590 | 0.708 | - | 0.774 | -0.426 | -0.178 | 0.280 | 0.306 | 0.304 | 0.385 | 0.372 | 0.723 | 0.026 | 0.089 | 0.056 | 0.087 | -0.023 | 0.070 | 0.070 | 0.087 | -0.476 | -0.473 | -0.497 |
| DLOCE | 0.014 | 0.351 | 0.833 | 0.870 | 0.774 | - | -0.556 | -0.113 | 0.284 | 0.247 | 0.243 | 0.308 | 0.291 | 0.593 | 0.013 | 0.096 | 0.076 | 0.095 | -0.014 | 0.073 | 0.073 | 0.095 | -0.577 | -0.564 | -0.602 |
| BCE | 0.051 | -0.142 | -0.461 | -0.575 | -0.426 | -0.556 | - | 0.075 | -0.143 | -0.078 | -0.079 | -0.200 | -0.138 | -0.318 | 0.080 | -0.021 | -0.006 | -0.020 | -0.051 | 0.001 | 0.001 | -0.020 | 0.714 | 0.698 | 0.732 |
| LOC | 0.664 | 0.472 | 0.414 | -0.160 | -0.178 | -0.113 | 0.075 | - | 0.857 | 0.398 | 0.397 | 0.386 | 0.406 | 0.494 | 0.144 | 0.906 | 0.821 | 0.901 | -0.664 | 0.874 | 0.874 | 0.901 | 0.102 | 0.099 | 0.116 |
| SLOC | 0.648 | 0.667 | 0.717 | 0.190 | 0.280 | 0.284 | -0.143 | 0.857 | - | 0.533 | 0.532 | 0.549 | 0.572 | 0.834 | 0.152 | 0.916 | 0.813 | 0.910 | -0.648 | 0.875 | 0.875 | 0.910 | -0.137 | -0.137 | -0.137 |
| TNDj | 0.653 | 0.936 | 0.435 | 0.163 | 0.306 | 0.247 | -0.078 | 0.398 | 0.533 | - | 0.849 | 0.753 | 0.781 | 0.555 | 0.747 | 0.702 | 0.697 | 0.707 | -0.653 | 0.731 | 0.731 | 0.707 | -0.110 | -0.101 | -0.102 |
| TNCj | 0.651 | 0.937 | 0.432 | 0.161 | 0.304 | 0.243 | -0.079 | 0.397 | 0.532 | 0.849 | - | 0.753 | 0.771 | 0.551 | 0.746 | 0.702 | 0.697 | 0.707 | -0.651 | 0.731 | 0.731 | 0.707 | -0.116 | -0.107 | -0.111 |
| TNE | 0.557 | 0.803 | 0.479 | 0.229 | 0.385 | 0.308 | -0.200 | 0.386 | 0.549 | 0.753 | 0.753 | - | 0.623 | 0.600 | 0.544 | 0.633 | 0.619 | 0.636 | -0.557 | 0.646 | 0.646 | 0.636 | -0.278 | -0.270 | -0.270 |
| TNI | 0.569 | 0.827 | 0.485 | 0.220 | 0.372 | 0.291 | -0.138 | 0.406 | 0.572 | 0.781 | 0.771 | 0.623 | - | 0.619 | 0.559 | 0.658 | 0.645 | 0.662 | -0.569 | 0.671 | 0.671 | 0.662 | -0.207 | -0.198 | -0.204 |
| TND | 0.463 | 0.718 | 0.814 | 0.502 | 0.723 | 0.593 | -0.318 | 0.494 | 0.834 | 0.555 | 0.551 | 0.600 | 0.619 | - | 0.132 | 0.688 | 0.605 | 0.683 | -0.463 | 0.648 | 0.648 | 0.683 | -0.338 | -0.336 | -0.348 |
| CpD | 0.441 | 0.671 | 0.086 | -0.031 | 0.026 | 0.013 | 0.080 | 0.144 | 0.152 | 0.747 | 0.746 | 0.544 | 0.559 | 0.132 | - | 0.394 | 0.436 | 0.402 | -0.441 | 0.437 | 0.437 | 0.402 | 0.026 | 0.026 | 0.031 |
| HL | 0.764 | 0.782 | 0.564 | 0.020 | 0.089 | 0.096 | -0.021 | 0.906 | 0.916 | 0.702 | 0.702 | 0.633 | 0.658 | 0.688 | 0.394 | - | 0.932 | 0.999 | -0.764 | 0.980 | 0.980 | 0.999 | -0.021 | -0.018 | -0.010 |
| HV | 0.576 | 0.762 | 0.503 | 0.009 | 0.056 | 0.076 | -0.006 | 0.821 | 0.813 | 0.697 | 0.697 | 0.619 | 0.645 | 0.605 | 0.436 | 0.932 | - | 0.946 | -0.576 | 0.874 | 0.874 | 0.946 | -0.040 | -0.030 | -0.022 |
| HVL | 0.747 | 0.786 | 0.560 | 0.019 | 0.087 | 0.095 | -0.020 | 0.901 | 0.910 | 0.707 | 0.707 | 0.636 | 0.662 | 0.683 | 0.402 | 0.999 | 0.946 | - | -0.747 | 0.974 | 0.974 | 1.0 | -0.023 | -0.020 | -0.011 |
| HLV | -1.0 | -0.698 | -0.359 | 0.062 | -0.023 | -0.014 | -0.051 | -0.664 | -0.648 | -0.653 | -0.651 | -0.557 | -0.569 | -0.463 | -0.441 | -0.764 | -0.576 | -0.747 | - | -0.872 | -0.872 | -0.747 | -0.069 | -0.067 | -0.079 |
| HE | 0.872 | 0.803 | 0.524 | -0.007 | 0.070 | 0.073 | 0.001 | 0.874 | 0.875 | 0.731 | 0.731 | 0.646 | 0.671 | 0.648 | 0.437 | 0.980 | 0.874 | 0.974 | -0.872 | - | 1.0 | 0.974 | 0.004 | 0.005 | 0.016 |
| HT | 0.872 | 0.803 | 0.524 | -0.007 | 0.070 | 0.073 | 0.001 | 0.874 | 0.875 | 0.731 | 0.731 | 0.646 | 0.671 | 0.648 | 0.437 | 0.980 | 0.874 | 0.974 | -0.872 | 1.0 | - | 0.974 | 0.004 | 0.005 | 0.016 |
| HB | 0.747 | 0.786 | 0.560 | 0.019 | 0.087 | 0.095 | -0.020 | 0.901 | 0.910 | 0.707 | 0.707 | 0.636 | 0.662 | 0.683 | 0.402 | 0.999 | 0.946 | 1.0 | -0.747 | 0.974 | 0.974 | 1.0 | -0.023 | -0.020 | -0.011 |
| ES | 0.069 | -0.177 | -0.461 | -0.563 | -0.476 | -0.577 | 0.714 | 0.102 | -0.137 | -0.110 | -0.116 | -0.278 | -0.207 | -0.338 | 0.026 | -0.021 | -0.040 | -0.023 | -0.069 | 0.004 | 0.004 | -0.023 | - | 0.954 | 0.940 |
| GA | 0.067 | -0.168 | -0.452 | -0.554 | -0.473 | -0.564 | 0.698 | 0.099 | -0.137 | -0.101 | -0.107 | -0.270 | -0.198 | -0.336 | 0.026 | -0.018 | -0.030 | -0.020 | -0.067 | 0.005 | 0.005 | -0.020 | 0.954 | - | 0.950 |
| RND | 0.079 | -0.173 | -0.476 | -0.589 | -0.497 | -0.602 | 0.732 | 0.116 | -0.137 | -0.102 | -0.111 | -0.270 | -0.204 | -0.348 | 0.031 | -0.010 | -0.022 | -0.011 | -0.079 | 0.016 | 0.016 | -0.011 | 0.940 | 0.950 | - |

Table 6.7: Correlation coefficient of the most interesting static measures in the 100%CP benchmark. We highlight the highest value per row.

| | 100%CP | | | | | | |
|---|---|---|---|---|---|---|---|
| | MC | HD | LOCE | N | DD | DLOCE | BCE |
| MC | • | 0.796 | 0.965 | 0.266 | 0.519 | 0.408 | 0.025 |
| HD | 0.796 | • | 0.786 | −0.108 | 0.052 | −0.035 | 0.284 |
| LOCE | 0.965 | 0.786 | • | 0.344 | 0.515 | 0.474 | -0.038 |
| N | 0.266 | −0.108 | 0.344 | • | 0.765 | 0.877 | -0.540 |
| DD | 0.519 | 0.052 | 0.515 | 0.765 | • | 0.912 | -0.377 |
| DLOCE | 0.408 | −0.035 | 0.474 | 0.877 | 0.912 | • | -0.485 |
| BCE | 0.025 | 0.284 | −0.038 | −0.540 | −0.377 | −0.485 | • |

Table 6.8: Correlation coefficient of the most interesting static measures in the ¬100%CP benchmark. We highlight the highest value per row.

| | ¬100%CP | | | | | | |
|---|---|---|---|---|---|---|---|
| | MC | H | LOCE | N | DD | DLOCE | BCE |
| MC | • | 0.698 | 0.571 | 0.257 | 0.432 | 0.351 | -0.142 |
| HD | 0.698 | • | 0.359 | 0.062 | 0.023 | 0.014 | 0.051 |
| LOCE | 0.571 | 0.359 | • | 0.692 | 0.590 | 0.833 | -0.461 |
| N | 0.257 | 0.062 | 0.692 | • | 0.708 | 0.870 | -0.575 |
| DD | 0.432 | 0.023 | 0.590 | 0.708 | • | 0.774 | -0.426 |
| DLOCE | 0.351 | 0.014 | 0.833 | 0.870 | 0.774 | • | −0.556 |
| BCE | −0.142 | 0.051 | −0.461 | −0.575 | −0.426 | −0.556 | • |

measures that are more correlated with these complexity measures (remember that all correlation coefficients can be seen in Tables 6.5 and 6.6. McCabe's complexity is highly correlated with the number of conjunctions, disjunctions, equalities and inequalities (0.934, 0.925, 0.829 and 0.811 in 100%CP and 0.937, 0.936, 0.803 and 0.827 in ¬100%CP, respectively). These high coefficients were expected because McCabe's complexity depends on the CFG of the program. Halstead Difficulty is highly correlated with the other Halstead measures. In addition, it is highly correlated with McCabe's complexity (0.796 in 100%CP and 0.698 in ¬100%CP). LOCE is highly correlated with the total number of decisions and SLOC (0.976 and 0.974) in 100%CP and in ¬100%CP (0.814 and 0.717). These results were expected because SLOC and LOCE are very similar measures and the total number of decisions gives us an idea of the length of the code. The Halstead Length is highly correlated with LOC and SLOC, with a minimum value of correlation of 0.906. Moreover, the other Halstead measures are highly correlated too, except Halstead Difficulty and Level. This indicates that several Halstead measures are similar to a simple count of lines of code.

In this subsection we have provided an overview of static measures that are part of our study. Now, we know the measures that are similar and those that are different. In the next section we show the measures that are more correlated with the branch coverage, which is the way we measure the difficulty of testing a program.

### 6.6.2 Correlation Between Coverage and Static Measures

In the previous section we showed the basic relationship among the static measures, in this section we include the branch coverage in the study. The existing correlations between the branch coverage and the static measures studied give us an idea of which static measures are useful to determine

*a priori* the complexity of the automatic test data generation task. In this study we have applied three different test data generators, two based on evolutionary techniques (ES, GA) and one based on random testing (RND).

Table 6.9: Relationship between the most important static measures and the average branch coverage for all the algorithms. We highlight the highest value of correlation for each algorithm and benchmark.

| | 100%CP | | | ¬100%CP | | |
|---|---|---|---|---|---|---|
| | ES | GA | RND | ES | GA | RND |
| MC | -0.150 | -0.226 | -0.074 | -0.177 | -0.168 | -0.173 |
| HD | 0.070 | -0.101 | 0.077 | 0.069 | 0.067 | 0.079 |
| LOCE | -0.186 | -0.251 | -0.133 | -0.461 | -0.452 | -0.476 |
| N | -0.543 | -0.381 | -0.434 | -0.563 | -0.554 | -0.589 |
| DD | -0.439 | -0.304 | -0.311 | -0.476 | -0.473 | -0.497 |
| DLOCE | -0.504 | -0.345 | -0.397 | -0.577 | -0.564 | -0.602 |
| BCE | 0.510 | 0.375 | 0.534 | 0.714 | 0.698 | 0.732 |

The first question we should answer is if there exists a link between the coverage and the traditional measures of code complexity: McCabe's, Halstead's, and LOCE. In Table 6.9 we show the correlation coefficients for the most important static measures and the branch coverage obtained with three automatic test data generators. The correlations between Halstead's Difficulty and the coverage are very low, so the answer is no in this case. The correlation coefficients of McCabe's complexity are higher than Halstead Difficulty but too low. This result was expected because, as we showed in the previous section, Halstead Difficulty is highly correlated with McCabe's complexity. Finally, the correlation coefficients of LOCE indicate that it is more correlated with the branch coverage because this measure takes into account the nested statements. After analyzing these results, we realise that the traditional complexity measures (MC, HD, and LOCE) are not useful to measure the difficulty of testing a program.

In the second group of measures, there exist higher correlations with branch coverage. The nesting degree is the static measure with the highest correlation coefficient with branch coverage in the 100%CP benchmark for the evolutionary test case generators. On the other hand, DLOCE is more correlated than the nesting degree in the ¬100%CP benchmark. Despite that the total number of decisions is not correlated with coverage, as can be seen in Tables 6.5 and 6.6 in the Appendix, the density of decisions correlates with the obtained coverage, as we show in Table 6.9. Moreover, the density of decisions is also more correlated than the traditional complexity measures. In Figure 6.6 the trend indicates that the programs with a high density of decisions are more difficult to test because a lower coverage is obtained.

After analyzing the LOCE measure, we supposed that if the influence of the LOC were removed by dividing LOCE by LOC, it could be obtained a measure with a high influence of the nested level (DLOCE) (recall that that the LOCE measure weighs those nested statements). As the nesting degree is highly correlated with the branch coverage, the DLOCE would have high correlation too. After doing the correlation test, our expectations were true, as one can see in Table 6.9. These results are similar to the results obtained with the nesting degree. In the case of the benchmark ¬100%CP, DLOCE has more influence than nesting (N) in general. In Figure 6.7, we can see that the coverage clearly increases as the DLOCE decreases with the exception of the programs with DLOCE between 7 and 8.

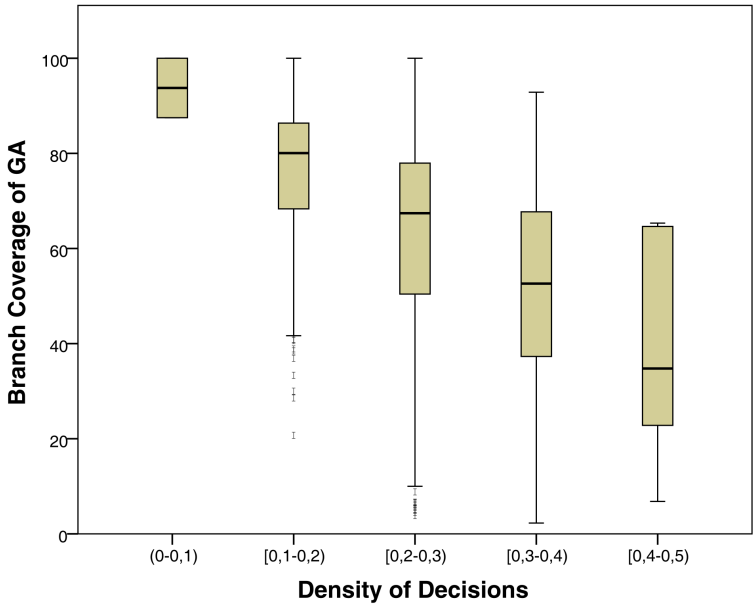



Figure 6.6: Boxplots showing the branch coverage against the Density of Decisions for GA in ¬100%CP.

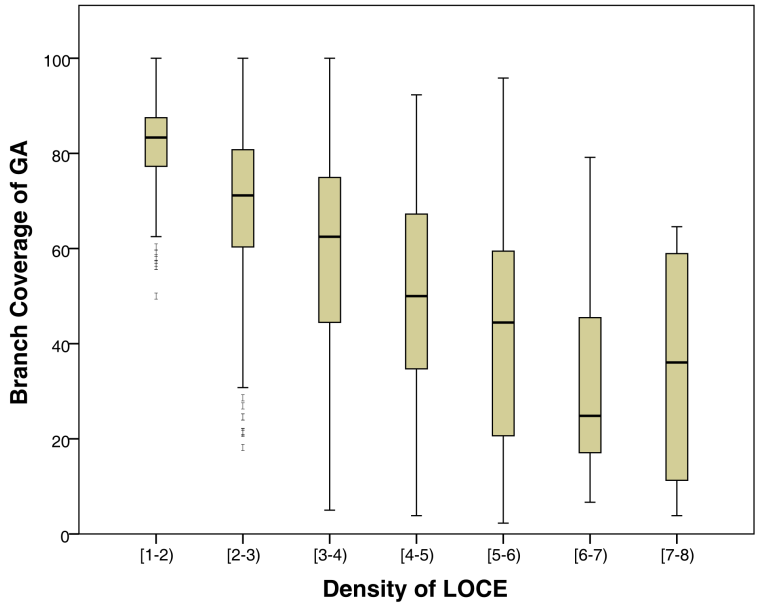



Figure 6.7: Boxplots showing the branch coverage against the DLOCE for GA in ¬100%CP.

Let us analyze the nesting degree. In Table 6.10, we summarize the obtained coverage in programs with different nesting degree in the two benchmarks of programs. If the nesting degree is increased, the branch coverage decreases and vice versa. It is clear that there is an inverse correlation between these variables. These correlation values are the highest ones obtained in the study of the different static measures, so we can say that the nesting degree is the feature with the highest influence on the coverage that evolutionary and random testing techniques can achieve. Nested branches pose a great challenge for the search. The high correlation value of the nesting degree supports that claim.

Table 6.10: Relationship between the nesting degree and the average coverage for all the algorithms. The standard deviation is shown in subscript. We highlight the highest values of branch coverage for each algorithm and benchmark.

| | 100%CP | | | ¬100%CP | | |
|---|---|---|---|---|---|---|
| Nesting degree | ES | GA | RND | ES | GA | RND |
| 1 | $96.30_{4.83}$ | $96.67_{5.78}$ | $86.13_{10.81}$ | $82.32_{8.16}$ | $82.51_{7.89}$ | $81.36_{7.97}$ |
| 2 | $92.28_{7.66}$ | $95.33_{7.11}$ | $79.87_{13.06}$ | $73.43_{11.70}$ | $73.92_{11.58}$ | $71.86_{11.66}$ |
| 3 | $83.92_{12.06}$ | $92.68_{10.28}$ | $73.46_{13.99}$ | $69.85_{15.35}$ | $70.33_{15.46}$ | $68.20_{15.23}$ |
| 4 | $81.44_{14.32}$ | $85.41_{13.96}$ | $68.67_{16.03}$ | $62.55_{17.93}$ | $62.37_{18.07}$ | $59.83_{17.82}$ |
| 5 | - | - | - | $53.81_{21.09}$ | $54.48_{21.57}$ | $51.83_{20.80}$ |
| 6 | - | - | - | $50.32_{21.14}$ | $50.78_{21.90}$ | $46.33_{20.93}$ |
| 7 | - | - | - | $44.31_{20.57}$ | $45.33_{20.77}$ | $42.77_{19.68}$ |
| $\rho$ | -0.543 | -0.381 | -0.434 | -0.563 | -0.554 | -0.589 |

Finally, we analyze the BCE measure, the new measure proposed to estimate the difficulty to generate an adequate test suite. In the 100%CP benchmark the correlation between this new measure and the coverage was 0.510 for ES, 0.375 for GA and 0.534 for RND, as we can see in Table 6.9. The obtained correlation coefficients when an RND generator is used are higher because the Markov model is inspired on it. In addition, in the ¬100%CP the correlations are even higher: 0.714 for ES, 0.698 for GA and 0.732 for RND. This promising measure is more correlated with the coverage (especially in the RND generator) than the nesting degree and the other static measures. This suggests that it is the best static complexity measure for measuring the difficulty of testing a program by an automatic test data generator.

In Figure 6.8 we show the obtained average branch coverage with the random test data generator against the BCE measure. The trend is clear: the lower the value of Branch Coverage Expectation, the lower the coverage. We have opened a way to estimate the difficulty to test a program that is better than using the existing complexity measures or other known static measures like the nesting degree.

### 6.6.3 Another use of the Branch Coverage Expectation

As we detailed in Section 6.3 for each branch $(BB_i, BB_j)$ the expected number of test cases required to traverse it is $1/E[BB_i, BB_j]$. Then, given a number of test cases $x$, we can compute the number of branches that would be theoretically traversed if the tester execute $x$ random test cases, according to this equation:

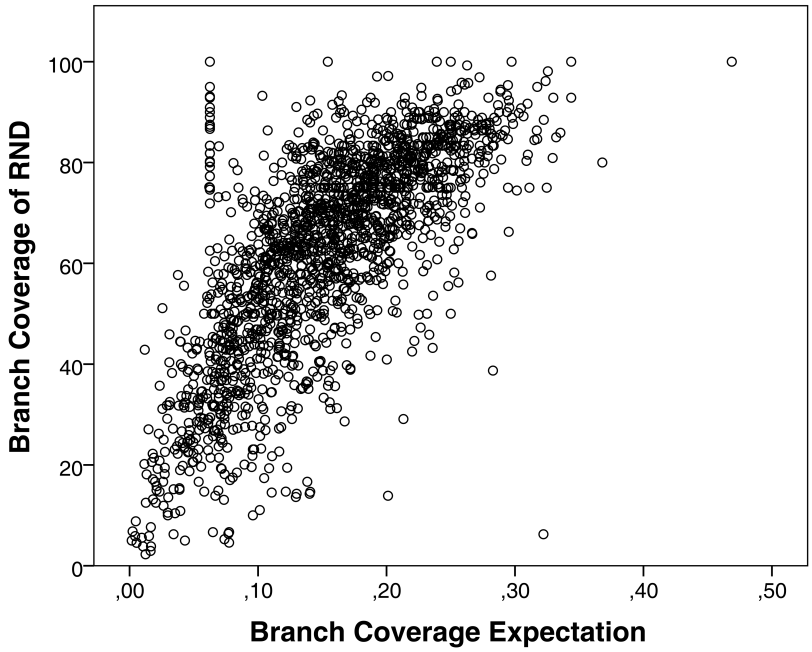



Figure 6.8: Average Branch Coverage of RND against the BCE measure.

$$f(x) = \left| \left\{ (i,j) \,\middle|\, \frac{1}{E[BB_i, BB_j]} < x \right\} \right|. \tag{6.19}$$

Thanks to this estimation, we propose a theoretical prediction about the behavior of an automatic test data generator based on random testing.

In Figure 6.9 we show a plot for a particular program with the expected theoretical behavior together with the experimental data obtained using the average branch coverage of the 30 independent executions of an RND generator for that program. The features of this test program are shown in Table 6.11. The resulting curves show that our theoretical prediction and the experimental data are very similar. The theoretical prediction is more optimistic because it does not take into account data dependencies. At the first steps of the algorithm, the experimental behavior is better than the theoretical prediction, but in the region of high coverage (close to 90%), the behavior of the RND test data generator is worse than expected. One explanation for this behavior could be the presence of data dependencies in the program, which is not considered in the theoretical approach in order to keep it simple.

This new proposal is useful to decide which is the best way of generating a test suite for a piece of code. It could be useful to decide the parameters of an evolutionary test data generator prior to its execution, for example, the stopping condition. It is also a way to simulate the Multi-objective approach, so you know the expected branch coverage using a particular number of test cases, for any number of them.

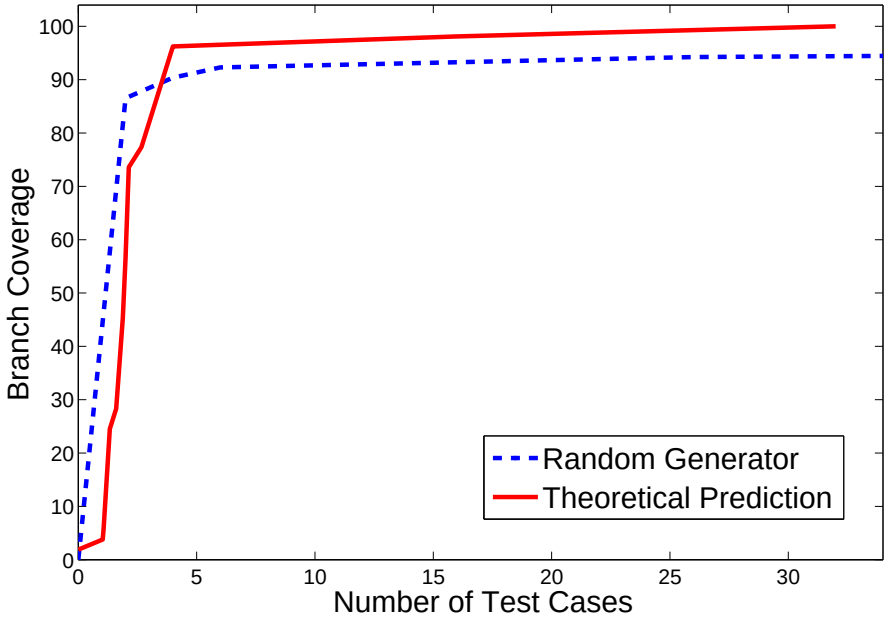



Figure 6.9: Coverage against the number of test cases of the random generator and the theoretical model.

### 6.6.4 Validation on Real Programs

In this section we want to make some validation of our proposed measure on real programs. We study 10 real programs extracted from the literature with characteristics similar to the artificial programs used in the previous sections. The reader must take into account that the number of programs used in the previous sections gives us the chance to average among 2,600 programs and extract statistically more reliable results. Despite the fact that in this section we only analyze the proposed testing measure over 10 programs, most of the conclusions are similar to the ones we have obtained with the synthetic programs.

In Figure 6.10 we show the average coverage obtained with the GA against the BCE. Once again we can see that the higher the Branch Coverage Expectation the higher the coverage. Relying on this figure we can state that there is a strong correlation between the obtained coverage and the BCE. Besides showing the figure, we have computed the Spearman's correlation coefficient. The coefficients are 0.770 and 0.758 for GA and RND, respectively. These values of correlation are even higher than the values obtained with the synthetic programs. Thanks to the experiments on real programs we can state that the proposed measure (BCE) is useful in order to measure the difficulty to automatically generate an adequate test suite.

When we have analyzed the behavior of the ES algorithm, we obtained that the correlation does not exist (-0.013). We can try to justify this unexpected result because the ES has problems when it deals with a few complex branches. This algorithm achieves high coverage in most programs, but in a few, it obtains less coverage than expected. It is not able to cover some complex branches. This statement is supported by the value of correlation between the coverage obtained with ES and the estimation of test cases needed introduced in the previous section. They are correlated

Table 6.11: Static measures for a representative program.

| Features | Value |
|---|---|
| Nesting | 1 |
| Atomic Conditions | 4 |
| Total Decisions | 26 |
| Equalities | 3 |
| Inequalities | 8 |
| McCabe | 62 |
| Halstead Difficulty | 32.53 |
| LOCE | 107 |
| Density of Decisions | 0.37 |
| Density of LOCE | 1.51 |

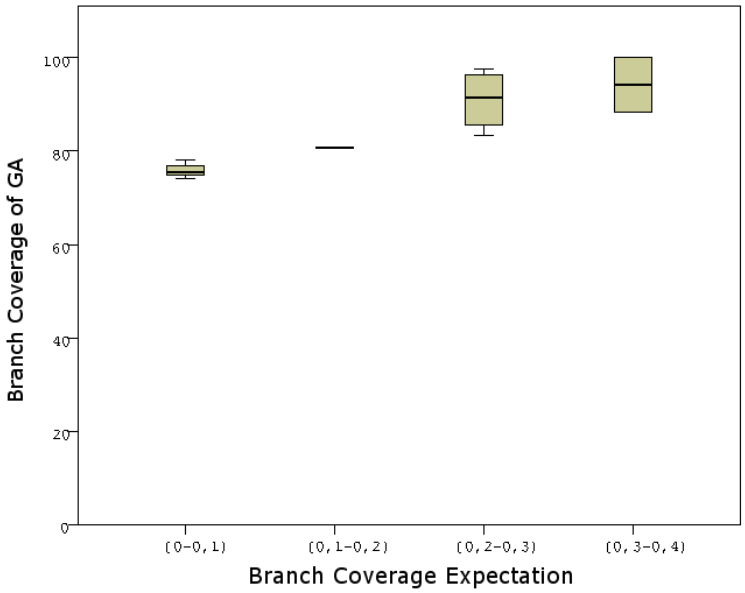



Figure 6.10: Average Branch Coverage of GA against the Branch Coverage Expectation for the real programs.

with a value of -0.582. This means that the most complex branch to cover in a program has high influence in the computation of the coverage.

## 6.7 Conclusions

In this chapter we dealt with the testing complexity from an original point of view: a program is more complex if it is more difficult to be automatically tested. With the intention of answering the **RQ1**, we defined the "Branch Coverage Expectation" in order to provide some knowledge about the difficulty of testing programs. The foundation of this measure is based on a Markov model of the program. The Markov model provides a theoretical background. The analysis of this measure indicates that it is more correlated with branch coverage than the other studied static measures. This means that this is a good way of estimating the difficulty of testing a program. We think, supported by the results, that this measure is useful for predicting the behavior of an automatic test data generator.

In this study, we also analyzed the static features and the most common complexity measures in Software Engineering. This analysis was performed in two automatically generated benchmarks of programs. We studied the correlations among static measures of a program and we determined which of them can be useful to estimate the complexity of a program. Summing up, the studied complexity measures like McCabe's and Halstead's seemed to be useless for this task. Instead, the nesting degree, the density of decisions and the density of LOCE were the static measures more correlated with branch coverage, although none of them is so correlated as the "Branch Coverage Expectation".

The Markov model of the program can also be used to provide an estimation of the number of test cases needed to cover a concrete percentage of the program. We have compared our theoretical prediction with an average of real executions of a test data generator. The results show that our prediction is very similar to the evolution of a real execution of the test data generator. This model can help to predict the evolution of the testing phase, which consequently can save time and cost of the entire project. This theoretical prediction could be also very useful to determine the coverage percentage using a particular number of test cases, despite it could be a good estimation, the multi-objective approach could be better helping testers to pick an adequate test suite when the resources are limited. In the next chapter we compare mono and multi-objective approaches solving the test data generation problem.

# Chapter 7

# Multi-Objective Test Data Generation

## 7.1 Introduction

Traditionally, the solution of the Test Data Generation Problem (TDGP) is a set of test cases whose execution is able to cover all the software elements. Indeed, branch coverage is usually the most popular goal. Despite most previous work have only considered coverage, real-world engineers deal with the tedious and costly task of checking the system behavior for all the generated test cases. This significant and usually neglected cost is called the oracle cost [98]. Thus, a reformulation of the TDGP to deal with real-world problems is a need, taking into account the oracle cost as another important objective to minimize. The oracle cost can be reduced by minimizing the test suite size. The ideal scenario is to reduce the test suite size without any loss of coverage. However, in certain situations the two objectives are in conflict: minimizing the oracle cost implies minimizing the coverage. When there are multiple conflicting objectives the optimization literature recommends the consideration of a Pareto optimal optimization approach that is able to take into account the need to balance the conflicting objectives. Hence, the TDGP has been reformulated into a multi-objective problem (MOTDGP) in the work by Lakhotia *et al.* [129] and more recently, in 2010, in a work by Harman *et al.* [98].

Our main goal in this chapter is the comparison between two approaches to deal with the MOTDGP: a direct multi-objective approach (*MM*) and a combination of a mono-objective algorithm followed by a multi-objective test case selection optimization (*mM*). The general scheme of the proposed approaches can be seen in Figure 7.1. On the one hand, the *MM* approach considers the conflicting objectives during the entire test data generation process, thus *a priori* it focuses both on the test suite size minimization and the coverage maximization. On the other hand, the *mM* approach only considers the branch coverage during the test data generation process, thus *a priori* it focuses only on the branch coverage maximization. In order to deal with the optimization of the test suite size, in this second approach an additional second phase of multi-objective test case selection is performed. As nobody has previously compared these approaches yet, we can raise the following research questions and try to answer them in an extensive experimental study.

- **RQ1**: How does *MM* perform for MOTDGP?
- **RQ2**: Is the *MM* approach good enough in maximizing the coverage?

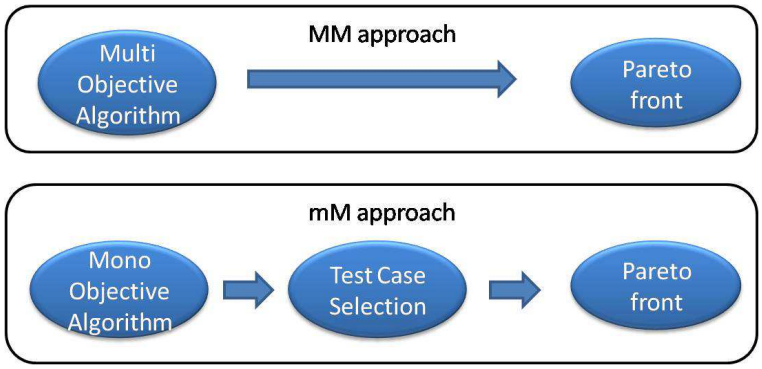



Figure 7.1: The general scheme of the two proposed approaches.

- **RQ3**: How good is the *mM* approach performance in optimizing the coverage and the test suite size?
- **RQ4**: Which approach is the best?

In order to completely answer the questions we should use all the possible automatic test data generators both in multi and mono-objective or, at least, a large number of them. We can also focus on some test data generators and answer the previous questions on them, taking into account that in this case the results will be valid for the test data generators considered. This is what we do in this thesis. In particular, we study the MOTDGP with two objectives, maximizing the branch coverage and minimizing the oracle cost. Among our contributions, we generate the test data and we also minimize the number of tests needed to achieve different values of coverage of the program. The solutions are provided as Pareto fronts. For the *MM* approach, we use five test data generators: four of them based on evolutionary testing and an additional one based on random search. In the *mM* approach we use three mono-objective test data generators with a second phase of multi-objective test data selection.

## 7.2 Experimental Methodology

This section is aimed at presenting the two approaches proposed to solve the MOTDGP that we have already defined formally in Section 2.2.2. It outlines the indicators used to measure the quality of the obtained results and the benchmark programs we have used. In this section we also describe how the solutions of the problem have been encoded and the genetic operators employed, the configuration of the algorithms, and the methodology we have followed.

### 7.2.1 The *MM* Approach

The *MM* approach considers the conflicting objectives during all the test data generation process, thus *a priori*, it focuses on both objectives during all the process. In this approach a solution to the problem is a test suite, that is, a set of test data. These test suites are evaluated according to both objectives. The evaluation of the first objective (coverage) requires, in general, the execution of the test suite over the SUT. The evaluation of the second objective is a simple count of the number of test data in the set.

**Details of the Multi-objective algorithms**

Here we detail the configuration of the operators and the encoding of the solutions used in the multi-objective algorithms. In the multi-objective approach, each individual is encoded as a set of test data. Table 7.1 shows the parameters of the multi-objective EAs used in the experimental section (all of them were previously described in Section 4.2). As genetic operators, we have used 2-tournament (*binary tournament*) as the selection scheme. This operator works by randomly choosing two individuals from the population and the one dominating the other is selected; if both solutions are non-dominated one of them is randomly selected.

Table 7.1: Parameters of the multi-objective EAs used in the experimental section.

| | NSGA-II | MOCell | SPEA2 | PAES |
|---|---|---|---|---|
| Population | 20 indivs. | 20 indivs. | 20 indivs. | 20 indivs. |
| Selection | BT, 2 indivs. | BT, 2 indivs. | BT, 2 indivs. | BT, 2 indivs. |
| Mutation | Adaptive Mutation | Adaptive Mutation | Adaptive Mutation | Adaptive Mutation |
| Crossover | Union Crossover | Union Crossover | Union Crossover | - |
| Replacement | Elitist | Elitist | Elitist | Elitist |
| Total Evals. | 150000 evals. | 150000 evals. | 150000 evals. | 150000 evals. |

The best results were obtained with the *union crossover* that takes two solutions, $C_1$ and $C_2$, and creates a new one $C$ that is the union of both, that is: $C = C_1 \cup C_2$. If the resulting solution $C$ has more coverage than $C_1$ and $C_2$ then $C$ is the new offspring. Otherwise, the solution with more coverage ($C_1$ or $C_2$) is the new child.

Finally, the mutation operator adds new test data to the solution with probability 0.6, deletes one test datum with probability 0.2 and keeps the individual unchanged with probability 0.2. In the case of adding test data, the number of new test data is 30% of the test data present in the solution. If the resulting individual has the same coverage and more test data, at the end of the iteration, the algorithm deletes it from the population because this solution is dominated.

All the multi-objective algorithms have been implemented using jMetal [66], a Java framework aimed at the development, experimentation, and study of metaheuristics for solving multi-objective optimization problems.

### 7.2.2 The *mM* Approach

In this section we present the second approach. In this approach we use a mono-objective test data generator to obtain a set of test data with the highest coverage. The mono-objective test data generator deals with only one branch of the program at the same time. This is a beneficial feature to obtain high coverage because the search can focus on covering the most complex branches of the program. However, the resulting test suite is usually large, redundant and inefficient because these algorithms do not try to minimize the test suite size. One way to reduce the number of test cases in a test suite, and still test the same functionality, is by solving a Multi Objective Test Case Selection Problem (MOTCSP) on the given test suite. This problem was recently formalized by Yoo and Harman in [226] as follows: Given a test suite $T$ and several objective functions $F_i$, we must find a subset $T' \subseteq T$ such that $T'$ is a Pareto optimal set with respect to the objective functions. The resulting subset of the test suite, $T'$, is composed of the non-dominated solutions considering the objectives as equally important.

In order to solve the MOTCSP we always use in these experiments the multi-objective algorithm NSGA-II. Our implementation is able to generate a Pareto front from thousands of test cases

previously generated by the mono-objective algorithms. But first, we delete repeated test cases from the obtained test suite in order to reduce from thousands of test cases to hundreds of them. Two test cases are repeated when both of them traverse the same branches. We have compared the results obtained with and without this reduction phase, and the results are better when this reduction is applied. Finally, for the mono-objective algorithm involved in the first phase of test data generation, we use three different algorithms: a genetic algorithm, an evolutionary strategy and a random search, described in Section 4.1.

**Details of the Mono-objective algorithms**

In this work, each solution is encoded as an integer/real vector of length $n$ (the number of arguments). As we said when we explained the test data generator (Section 5.2) the generator breaks down the global objective (to cover all the branches) into several partial objectives consisting of dealing with only one branch of the program. Thus, two stopping conditions exist: one for partial objectives and the other one for the whole test data generation process. The search for one partial objective stops when 1,000 evaluations are performed while the test data generation process ends after 150,000 evaluations.

In our GA we use as recombination operator the uniform crossover (UX), in which each component of the new solution is randomly selected from the two parents. The formal definition is the same as equation (4.1) with $bias = 0.5$. The mutation operator adds a random value to the components of the vector. That is,

$$x_i = x_i + U(-500, 500) \tag{7.1}$$

where the probability distribution of these random values is a uniform distribution in the range $[-500, 500]$. However, not all the components of the individual are perturbed, only half of them are. In our ES, we use a discrete crossover operator and a Gaussian mutation. We show in Table 7.2 a summary of the parameters used by the two EAs in the experimental section.

Table 7.2: Parameters of the two mono-objective EAs used in the experimental section.

| | ES | GA |
|---|---|---|
| Population | 25 indivs. | 25 indivs. |
| Selection | Random, 5 indivs. | Random, 5 indivs. |
| Mutation | Gaussian | Add $U(-500, 500)$ |
| Crossover | discrete (bias = 0.6) + arith. + arith. | Uniform |
| Replacement | Elitist | Elitist |
| Stopping cond. | 1,000 evals. | 1,000 evals. |
| Total Evals. | 150,000 evals. | 150,000 evals. |

After the execution of the test data generator, we obtain a huge table of coverage where the test data that satisfy a concrete branch during the execution are saved. Then, a test data selection is performed over this set using a standard NSGA-II.

### 7.2.3 Benchmark of Test Programs

In the experimental section we use two benchmarks. The first one is composed of 800 synthetic programs and the second one is composed of 13 real-world programs[1].

#### Synthetic Programs

Our program generator (Section 6.5.2) can create programs having the same value for the static measures, as well as programs having different values for the measures. In addition, the generated programs are characterized by having a 100% coverage, thus all possible branches are reachable.

Our program generator takes into account the desired values of some static measures. The static measures selected are: the number of atomic conditions, the nesting degree, the number of sentences and the number of variables. The main features of the generated programs are: they deal with integer input parameters, their conditions are joined by whichever logical operator, they are randomly generated and all their branches are reachable.

The methodology applied for the program generation was the following. First, we analyzed a set of Java source files from the JDK 1.6 (java.util.*, java.io.*, java.sql.*, etc.) and we computed the static measures on these files. Next, we used the ranges of the most interesting values, obtained in this previous analysis as a guide to generate Java source files having values in the same range for the static measures. This way, we generated programs with the values in these ranges, e.g., nesting degree in 1-4 (25% for each value), atomic conditions per condition in 1-4 (68.43% with 4 conditions per decision), and statements in 25, 50, 75 or 100 (25% for each value). The percentage of control flow statements is 32.23% (in this work we use IF statements), this means that the test case generator should cover around 64 different branches (32 true and 32 false) in programs with 100 statements. The previous values are realistic with respect to the static measures, making our study meaningful. Besides, we generated 50 programs for each size and nesting degree (50 x 4 sizes x 4 nesting degrees = 800), which is a total of 800 Java programs.

#### Real Programs

In order to improve the interest of our work we propose an additional benchmark of real programs. It is composed of the ten real programs used in the previous chapter (Table 6.4) and three additional programs extracted from the literature. The main features of the new programs are listed below.

Table 7.3: Characteristics of the real programs.

| Name | ND | LOC | Branches | Arguments | Description |
|---|---|---|---|---|---|
| complex | 3 | 74 | 24 | 6 Integer | Calculate complex arithmetic functions |
| remainder | 6 | 49 | 18 | 2 Integer | Calculate the remainder of an integer division |
| tmyers | 6 | 54 | 12 | 3 Integer | Classify triangles in 4 types: Myers |

## 7.3 Experimental Analysis

In this section we present the results of the two proposed approaches. In the first subsection we analyze the *MM* approach and we compare the performance of the multi-objective algorithms. In the second subsection we study the *mM* approach and we compare the performance of the mono-objective algorithms used as the base for the approach. Then, in a third subsection we compare

[1]They are available at http://neo.lcc.uma.es/staff/javi/resources.html

the two proposed approaches for the academic benchmark, and finally, in the last subsection, we compare both approaches with a benchmark of real programs.

The benchmarks used were described in Section 7.2.3. We performed 30 independent runs of each algorithm and program, in order to obtain a very stable average of the measures. All test data generators used in this work proceed by generating test data until a maximum of 150,000 test data are generated. We also perform a multiple comparison statistical test for each program on the obtained results to compare the algorithms among them. We set a confidence level of 95% ($p$-value under 0.05) for the whole comparison (all the algorithms acting on a program) and we used the Bonferroni correction for each particular comparison.

### 7.3.1 Evaluation of the *MM* approach

In this section, we analyze the behavior of the multi-objective algorithms with the aim of be able to answer **RQ1**. We have analyzed 800 programs, so we cannot represent all HV values for all the programs. For this reason, we summarize in Table 7.4 the times one algorithm has better median HV than the others. We have classified the results according to the nesting degree and the size of the SUT. For this indicator, the higher the value, the better the quality of the obtained results. Thus, by looking at the tables, we can see that MOCell was usually the algorithm computing clearly the best results regarding HV. However, when the programs are small (25-50 statements) and complex (nesting degree four), the NSGA-II algorithm has a better behavior. We must highlight the big difference between MOCell (443), NSGA-II (198) and the others altogether (43).

Then, we compare the HV values of all the programs and independent executions with the Kruskal-Wallis test. In each cell of a table of statistics we have a pair (number, triangle). The number indicates how many programs are significantly different, and the triangle indicates that the program in the row is significantly better (▲) or worse (▽) than the program in the column. The results are summarized in Table 7.5. Although the previous values set a clear tendency, the absence of significant differences between MOCell, NSGA-II and SPEA2, does not allow us to say that MOCell is better than the other two. However, we can mention that RNDMulti is the worst algorithm in all the programs (800) and PAES is worse than MOCell in 18 programs, NSGA-II in 9 programs, and SPEA2 in only 2 programs.

With the aim of showing an example of the computed fronts for the instances, we selected one program for each nesting degree, which can represent the typical behavior of the different algorithms in this kind of instance. In Figure 7.2 are depicted the 50%-attainment surfaces of these selected programs. In the instance with low nesting degree, MOCell dominates the others and has a good performance because it reaches almost the same or better coverage with the same test data. NSGA-II has a similar behavior except in the right extreme of the figure where it is not able to reach the same maximum coverage as MOCell. On the other hand, in the program with nesting degree 4, NSGA-II is the algorithm that is able to reach the best coverage and dominates all the other fronts. The other two multi-objective algorithms (SPEA2 and PAES) have problems finding the solutions with high coverage, in the upper-right bound of the figure, and are worse than MOCell and NSGA-II. RNDMulti is always the worst. MOCell has been able to find non-dominated solutions in the right area where SPEA2, PAES and RNDMulti have not found any of them (solutions in the extremes of the front). This is related to a better exploration of the search space by MOCell. Specifically, this is one of the properties of the cellular GA model, on which MOCell is based. This fact has been reported in many studies on single-objective optimization (see [6]). There is only one exception, when a program has nesting degree 4 and it is more difficult to obtain high coverage, NSGA-II has the best performance.



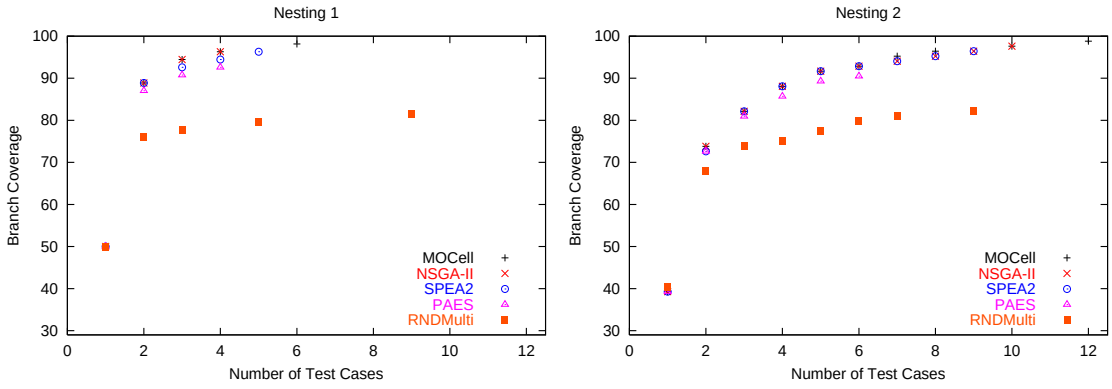
Table 7.4: Programs in which the median hypervolume of one algorithm is better than the others.

| Nesting degree | Statements | MOCell | NSGA-II | SPEA2 | PAES | RNDMulti |
|---|---|---|---|---|---|---|
| 1 | 25 | 10 | 1 | 0 | 0 | 0 |
| | 50 | 24 | 9 | 0 | 2 | 2 |
| | 75 | 34 | 6 | 1 | 1 | 0 |
| | 100 | 38 | 4 | 0 | 1 | 0 |
| | Total | 106 | 20 | 1 | 4 | 2 |
| 2 | 25 | 13 | 5 | 1 | 2 | 3 |
| | 50 | 35 | 13 | 0 | 0 | 0 |
| | 75 | 37 | 12 | 0 | 0 | 0 |
| | 100 | 40 | 10 | 0 | 0 | 0 |
| | Total | 125 | 40 | 1 | 2 | 3 |
| 3 | 25 | 18 | 11 | 3 | 1 | 2 |
| | 50 | 33 | 15 | 0 | 0 | 0 |
| | 75 | 32 | 16 | 1 | 0 | 0 |
| | 100 | 30 | 19 | 0 | 0 | 0 |
| | Total | 116 | 61 | 4 | 1 | 2 |
| 4 | 25 | 17 | 20 | 3 | 2 | 2 |
| | 50 | 23 | 25 | 2 | 1 | 0 |
| | 75 | 27 | 19 | 2 | 0 | 1 |
| | 100 | 29 | 13 | 10 | 0 | 0 |
| | Total | 96 | 77 | 17 | 3 | 3 |
| Total | | 443 | 198 | 23 | 10 | 10 |

Table 7.5: Number of programs where there exists significant difference among the HV obtained.

| | RNDMulti | PAES | SPEA2 | NSGA-II | MOCell |
|---|---|---|---|---|---|
| MOCell | 800▲ | 18▲ | 0 | 0 | – |
| NSGA-II | 800▲ | 9▲ | 0 | – | 0 |
| SPEA2 | 798▲ | 2▲ | – | 0 | 0 |
| PAES | 750▲ | – | 2▽ | 9▽ | 18▽ |
| RNDMulti | – | 750▽ | 798▽ | 800▽ | 800▽ |

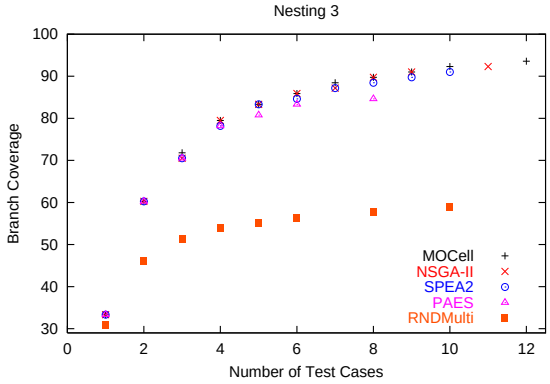
We have also analyzed the reduction obtained in the number of test cases, since one of our goals is to minimize the number of test cases. We analyze the reduction experienced using our approaches compared with the use of all the generated test cases. It is very difficult to analyze this reduction because not all the algorithms achieve a 100% coverage in all the programs. For this reason, we cannot simply average the number of test cases, but we must take into account the maximum obtained coverage in order to give the real reduction made by the multi-objective algorithm. The total reduction is from thousands of test cases generated to around ten, but this reduction could also be easily computed based in the *table of coverage* of the algorithms by choosing one test case per branch. The drawback of the latter approach is that the minimization of the test suite would be far from optimal. For this reason, we establish a theoretical upper bound of



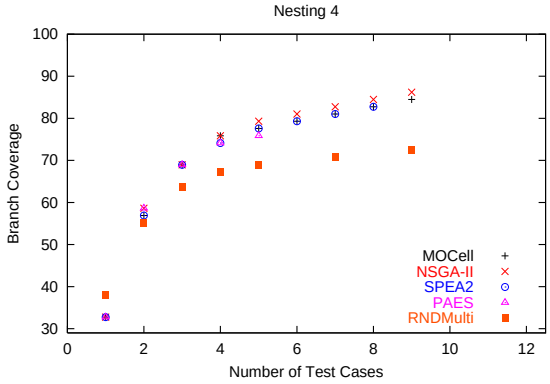

(a) Program with nesting degree 1




(b) Program with nesting degree 2




(c) Program with nesting degree 3




(d) Program with nesting degree 4

Figure 7.2: 50%-attainment surfaces: coverage against the number of test cases.

the required number of test cases needed. This upper bound is the number of branches that were achieved by the algorithm. We compute the real oracle cost of the test suites generated by any algorithm according to the next expression:

$$\begin{aligned} upper\_bound(P, A) &= B_P * MaxCov(P, A) \\ oracle\_cost(P, A) &= \frac{tc(P,A)}{upper\_bound(P,A)} \end{aligned} \tag{7.2}$$

where $P$ is a program, $A$ is an algorithm, $B_P$ is the number of branches of the program $P$, $MaxCov(P, A)$ is the maximum coverage obtained by the algorithm $A$ in the program $P$, and $tc(P, A)$ is the number of test cases needed by the algorithm $A$ to obtain the maximum coverage in program $P$.

We can state that the oracle cost of the test suite generated by all the multi-objectives algorithms can be reduced by our approach, only 15.12% of the test cases are needed in comparison with the computed upper bound. This reduction is computed in the case of the maximum coverage, and hence the largest number of computed test cases. But we must bear in mind that our solution is a complete Pareto front offered to the expert to make a decision about the test suite that best fits his/her needs, therefore a similar percentage of reduction is carried out for each couple *coverage-number of test cases* that appears in the Pareto front.

In the TDGP, it is particularly hard to achieve a 100% branch coverage, especially if one uses a multi-objective algorithm because its execution is not entirely guided to obtain a total coverage. The multi-objective approach deals with all the branches at the same time, this provokes a lack of information. In addition, the search does not spend most of its effort to cover the most complex branches. In Table 7.6 we show the average of maximum coverage (among the solutions in the front) obtained with the solutions for all the programs with different nesting degree in order to answer **RQ2**. We highlight the maximum values in the table for each nesting degree. As we expected, MOCell's performance is the best on nesting degree 1, 2 and 3. On the other hand, NSGA-II obtains the best coverage with nesting degree 4. Since the differences are low, we compared the coverage values of all the programs and independent executions with the Kruskal-Wallis test. The results are summarized in Table 7.7. As we expected, MOCell obtains significant differences in more programs with respect to PAES and RNDMulti, than NSGA-II and SPEA2.

Table 7.6: Relationship between the nesting degree and the average maximum coverage for the multi-objective algorithms. The standard deviation is shown in subscript.

| Nesting degree | MOCell | NSGA-II | SPEAII | PAES | RNDMulti |
|---|---|---|---|---|---|
| 1 | $98.10_{2.08}$ | $97.90_{2.22}$ | $97.53_{2.34}$ | $93.08_{5.30}$ | $81.36_{12.74}$ |
| 2 | $94.77_{3.44}$ | $94.42_{3.49}$ | $93.56_{3.75}$ | $87.59_{6.31}$ | $75.04_{14.00}$ |
| 3 | $90.66_{5.83}$ | $90.41_{5.46}$ | $89.29_{5.65}$ | $81.55_{7.68}$ | $69.77_{13.87}$ |
| 4 | $85.50_{9.45}$ | $85.77_{8.18}$ | $84.61_{8.12}$ | $75.87_{9.22}$ | $63.87_{15.95}$ |
| Total | $92.26_{7.54}$ | $92.12_{6.99}$ | $91.24_{7.24}$ | $84.52_{9.72}$ | $72.51_{15.57}$ |

Table 7.7: Number of programs where there exists a significant difference among the coverage values obtained.

| | RNDMulti | PAES | SPEA2 | NSGA-II | MOCell |
|---|---|---|---|---|---|
| MOCell | 800▲ | 800▲ | 2▲ | 0 | – |
| NSGA-II | 800▲ | 799▲ | 0 | – | 0 |
| SPEA2 | 800▲ | 782▲ | – | 0 | 2▽ |
| PAES | 711▲ | – | 782▽ | 799▽ | 800▽ |
| RNDMulti | – | 711▽ | 800▽ | 800▽ | 800▽ |

If we consider the HV obtained (Table 7.4), the significant HV differences (Table 7.5), the attainment surfaces and the average maximum coverage achieved showed in Table 7.6, it is clear that the ranking of the performance of the algorithms is: MOCell is the best, second NSGA-II, third SPEA2, fourth PAES, and finally RNDMulti, the worst one, as expected.

### 7.3.2 Evaluation of the *mM* approach

In this section we analyze the *mM* approach and try to answer **RQ3**. First of all, we study the values of HV. We show in Table 7.8 the programs in which one algorithm has a better value of HV.

It is noteworthy that when the nesting degree is the smallest (1) the ES obtains better results and when the nesting degree is large (3 and 4) the GA is better than the others. In other words, when the program is more complex, the GA is clearly the best. The ES is better in large programs (100 statements) except when the program has nesting degree four. Then, we compared the HV

Table 7.8: Programs in which the median hypervolume of one algorithm is better than the others.

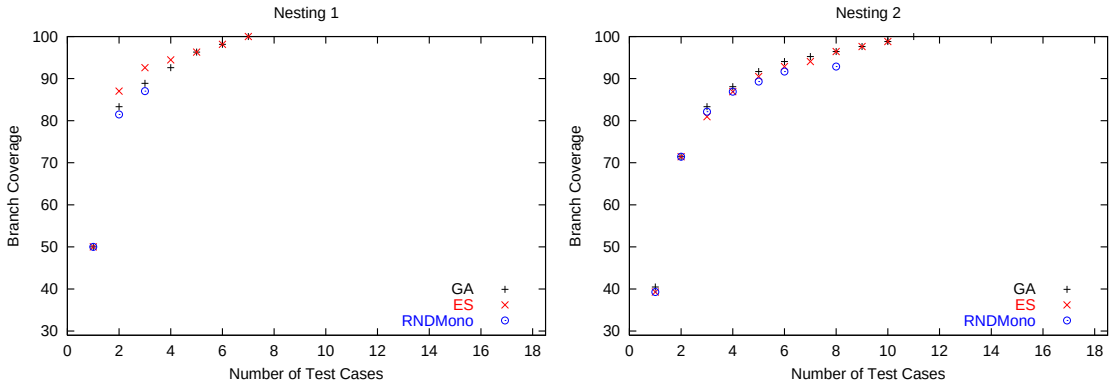
| Nesting degree | Statements | GA | ES | RNDMono |
|---|---|---|---|---|
| 1 | 25 | 3 | 3 | 0 |
| | 50 | 7 | 18 | 1 |
| | 75 | 7 | 26 | 3 |
| | 100 | 9 | 33 | 3 |
| | Total | 26 | 80 | 7 |
| 2 | 25 | 13 | 8 | 1 |
| | 50 | 23 | 17 | 0 |
| | 75 | 23 | 22 | 1 |
| | 100 | 18 | 29 | 1 |
| | Total | 77 | 76 | 3 |
| 3 | 25 | 23 | 6 | 0 |
| | 50 | 31 | 16 | 0 |
| | 75 | 30 | 16 | 0 |
| | 100 | 21 | 29 | 0 |
| | Total | 105 | 67 | 0 |
| 4 | 25 | 37 | 3 | 0 |
| | 50 | 41 | 6 | 0 |
| | 75 | 39 | 11 | 0 |
| | 100 | 34 | 14 | 0 |
| | Total | 151 | 34 | 0 |
| Total | | 359 | 257 | 10 |

values of all the programs and independent executions with the Kruskal-Wallis test. The results indicate that there is no significant difference between GA and ES (Table 7.9). As we expected, the results of RNDMono are worse than ES in 786 programs and GA in 765 programs.

Table 7.9: Programs where a significant difference exists among the HV obtained.

| | RNDMono | ES | GA |
|---|---|---|---|
| GA | 765▲ | 0 | – |
| ES | 786▲ | – | 0 |
| RNDMono | – | 786▽ | 765▽ |

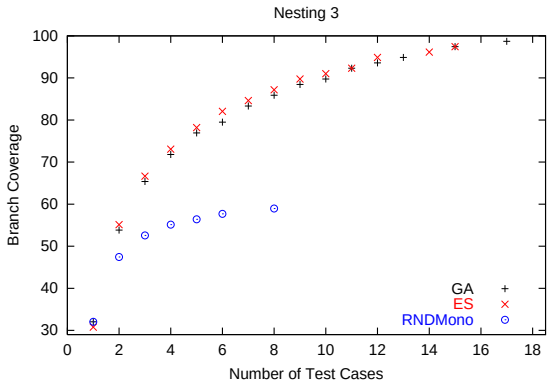
Second, we show the 50%-attainment surfaces of four representative programs with different nesting degree in Figure 7.3. In the instance with nesting degree 1, the attainment surfaces are very similar between GA and ES. RNDMono is far from the behavior of the others. In the instance with nesting degree 2, the three algorithms obtain similar results. The instances with nesting degree 3 and 4, represent the general behavior of the algorithms in most of the programs. The RNDMono is far from the others, the ES obtains similar values of coverage to the GA with the same number of test cases, but GA can achieve the best value of coverage. The GA is the best algorithm in maximum obtained coverage. This is related to a better exploitation of the search space by GA.



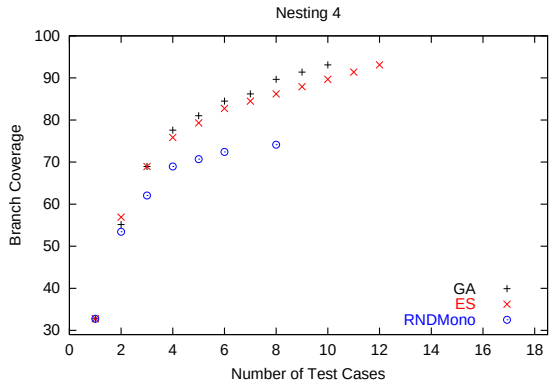

(a) Program with nesting degree 1




(b) Program with nesting degree 2




(c) Program with nesting degree 3




(d) Program with nesting degree 4

Figure 7.3: 50%-attainment surfaces: coverage against the number of test cases.

In order to highlight the reduction of the test cases needed to achieve the maximum coverage, we have applied Equation (7.2). We can state that the oracle cost of the test suite generated by the three studied mono-objective algorithms can be reduced by our approach, only 19.32% of the test cases are needed in comparison with the computed upper bound. This percentage of test cases needed to achieve a concrete coverage is larger than the one obtained with the *MM* approach (15.12%).

Now, let us analyze the best value of coverage obtained with the three algorithms. In Table 7.10 we show the average of maximum coverage of the three algorithms. As is known, achieving a total coverage is a great challenge for the search, for this reason, we consider that an algorithm must focus on obtaining a high value of coverage. In this sense, the GA and ES obtain very good values of coverage, both above 90% in all the cases. However, the average coverage obtained by the GA is always the best. This advantage of the GA increases in programs with higher nesting degree where high values of coverage are very difficult to obtain. We performed a statistical test (Table 7.11), but, although the GA obtains the best results, significant differences only exist in 23 programs between GA and ES. Thus, it seems that GA is the best in obtaining a high value of coverage, specifically in more complex programs.

Table 7.10: Relationship between the nesting degree and the average maximum coverage for the mono-objective algorithms. The standard deviation is shown in subscript.

| Nesting degree | GA | ES | RNDMono |
|---|---|---|---|
| 1 | $99.19_{2.20}$ | $98.70_{2.63}$ | $85.33_{10.51}$ |
| 2 | $98.85_{2.02}$ | $97.87_{2.44}$ | $79.20_{12.14}$ |
| 3 | $98.52_{2.09}$ | $95.66_{4.54}$ | $71.94_{13.36}$ |
| 4 | $96.89_{4.80}$ | $93.19_{6.66}$ | $66.42_{14.80}$ |
| Total | $98.36_{3.13}$ | $96.36_{4.90}$ | $75.72_{14.65}$ |

Table 7.11: Number of programs where there exists a significant difference between the coverage obtained.

| | RNDMono | ES | GA |
|---|---|---|---|
| GA | 800▲ | 23▲ | – |
| ES | 800▲ | – | 23▽ |
| RNDMono | – | 800▽ | 800▽ |

### 7.3.3 *MM* vs. *mM* approaches

In the previous sections we have performed a comparison between the algorithms used in each of the approaches. In the *mM* approach, GA seems to be the best algorithm in most of the programs and ES is the best algorithm in programs with the lowest nesting degree. Regarding the *MM* approach, MOCell was the best in most of the programs, except in a few programs with high nesting degree. In this section we compare all the algorithms together, with the aim of showing what technique is the overall best (**RQ4**).

First of all, we analyze the HV quality indicator. In Table 7.12 we summarize the number of times where the median HV value of an algorithm is better than the rest. The results show that, on the one hand, MOCell is better for programs with low nesting degree (1-2). On the other hand, the GA is better for programs with high nesting degree (3-4). The performance of the MOCell algorithm and the GA is similar but they work better in different kind of programs. This performance depends on the maximum nesting degree of the program. NSGA-II and the ES have similar performances among them; however they are clearly worse than MOCell and GA. Finally, the performance of SPEA2, PAES, RNDMono, and RNDMulti is clearly worse than the previous algorithms (MOCell, GA, NSGA-II and ES).

In order to clarify the obtained results, we have performed the statistical test to check if there exist significant differences among the HV values. In Table 7.13, we can see that there is just a small significant difference among the main evolutionary algorithms. However, there are significant differences between the worst algorithms (the two random algorithms and PAES), and the rest. In Table 7.13 we show that the HV values of GA are significantly better than the others, except the ES. The same observation can be made on ES: it is significantly better than the others (except the GA). NSGA-II, MOCell and SPEA2 are worse than GA and ES, but for most of the programs their HV values are better than the random algorithms. In some programs, there are significant differences between MOCell and PAES and also between NSGA-II and PAES.

Table 7.12: Programs in which the median hypervolume of one algorithm is better than the others.

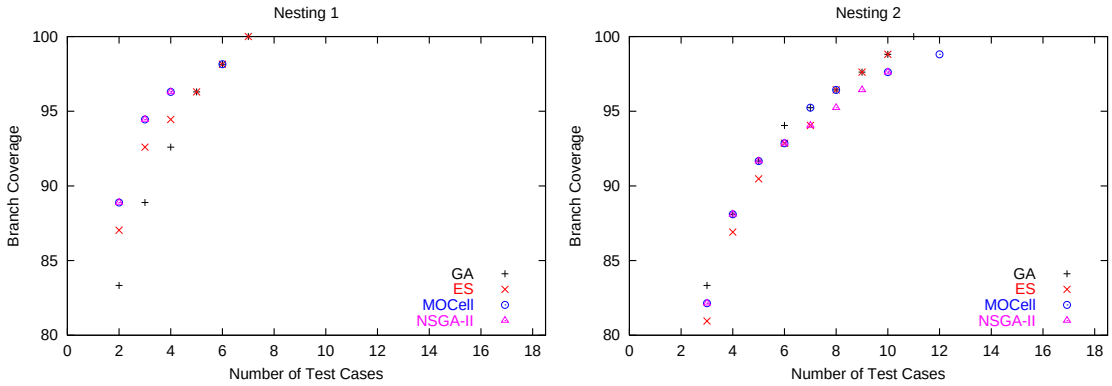
| ND | Statements | MOCell | NSGA-II | SPEA2 | PAES | RNDMulti | GA | ES | RNDMono |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 25 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 50 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 75 | 16 | 2 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 100 | 14 | 2 | 0 | 0 | 0 | 0 | 2 | 0 |
| | Total | 36 | 4 | 0 | 1 | 0 | 0 | 2 | 0 |
| 2 | 25 | 4 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 50 | 16 | 4 | 0 | 0 | 0 | 4 | 2 | 0 |
| | 75 | 24 | 6 | 0 | 0 | 0 | 4 | 1 | 0 |
| | 100 | 26 | 8 | 0 | 0 | 0 | 6 | 4 | 0 |
| | Total | 70 | 19 | 0 | 0 | 0 | 14 | 7 | 0 |
| 3 | 25 | 4 | 0 | 1 | 0 | 0 | 6 | 1 | 0 |
| | 50 | 10 | 1 | 0 | 0 | 0 | 18 | 4 | 0 |
| | 75 | 14 | 4 | 0 | 0 | 0 | 20 | 9 | 0 |
| | 100 | 13 | 10 | 0 | 0 | 0 | 13 | 11 | 0 |
| | Total | 41 | 15 | 1 | 0 | 0 | 57 | 25 | 0 |
| 4 | 25 | 2 | 1 | 1 | 0 | 0 | 19 | 1 | 0 |
| | 50 | 6 | 5 | 0 | 0 | 0 | 34 | 0 | 0 |
| | 75 | 7 | 2 | 0 | 0 | 0 | 33 | 6 | 0 |
| | 100 | 3 | 6 | 4 | 0 | 0 | 27 | 7 | 0 |
| | Total | 18 | 14 | 5 | 0 | 0 | 113 | 14 | 0 |
| Total | | 165 | 52 | 6 | 1 | 0 | 184 | 48 | 0 |

In summary, the *mM* approach using the evolutionary algorithms (GA and ES) always achieves good HV values. We observed that MOCell, NSGA-II and SPEA2 are significantly better than PAES in more programs than GA and ES. In this case, the HV values of the *mM* approach are worse, concretely they do not get a good diversity because the Pareto fronts are computed from a finite subset of test cases obtained by the mono-objective algorithms. However, the *MM* approach takes better care of the convergence as well as the diversity of the Pareto front, consequently their HV values will be better. For the purpose of illustrating this issue we plot in Figure 7.4 the 50%-attainment surfaces for the best algorithms: MOCell, NSGA-II, GA, and ES.

Table 7.13: Programs where a significant difference exists among the HV obtained.

| | RNDMulti | PAES | SPEA2 | NSGA-II | MOCell | RNDMono | GA | ES |
|---|---|---|---|---|---|---|---|---|
| MOCell | 800▲ | 235▲ | 0 | 0 | – | 800▲ | 39▽ | 7▽ |
| NSGA-II | 800▲ | 197▲ | 0 | – | 0 | 800▲ | 29▽ | 5▽ |
| SPEA2 | 800▲ | 61▲ | – | 0 | 0 | 800▲ | 13▽ | 2▽ |
| PAES | 799▲ | – | 61▽ | 197▽ | 235▽ | 645▲ | 36▽ | 18▽ |
| RNDMulti | – | 799▽ | 800▽ | 800▽ | 800▽ | 24▽ | 782▽ | 795▽ |
| ES | 795▲ | 18▲ | 2▲ | 5▲ | 7▲ | 737▲ | 0 | – |
| GA | 782▲ | 36▲ | 13▲ | 29▲ | 39▲ | 689▲ | – | 0 |
| RNDMono | 24▲ | 645▽ | 800▽ | 800▽ | 800▽ | – | 689▽ | 737▽ |

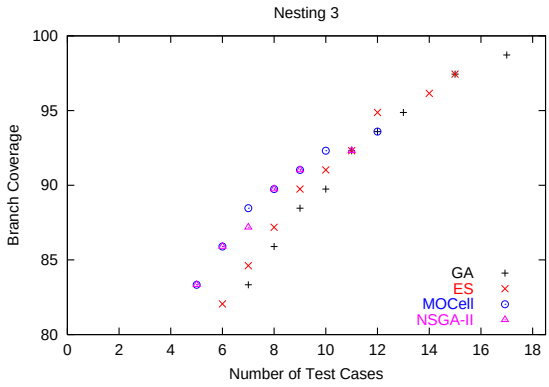
We focus on the most interesting area (80%-100% coverage) of the plots in Figure 7.4. In all the pictures, we appreciate that MOCell and NSGA-II have the best fronts in the programs with nesting degree 1, 2 and 3, although they do not obtain the best coverage in all cases. In addition, we must highlight that the fronts of GA and ES are dominated in this case. We find the exception when the program has nesting degree four, where the GA is the best algorithm because its solutions dominate the others. The second in performance is the ES; close to the values of GA. The other algorithms only find solutions with middle values of coverage and more test cases.



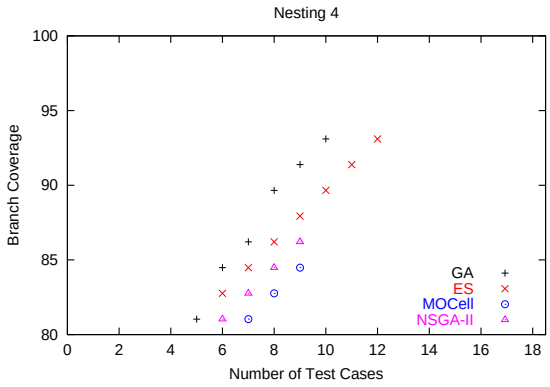

(a) Program with nesting degree 1




(b) Program with nesting degree 2




(c) Program with nesting degree 3




(d) Program with nesting degree 4

Figure 7.4: 50%-attainment surfaces: coverage against the number of test cases of all the algorithms.

At this stage of the study, we know that the *MM* approach provides more diversity in the solutions. In other words, it is able to find a test suite with few test cases, but the obtained coverage is not very large. On the other hand, the *mM* approach is able to better explore the search space to find solutions with a high coverage, but it needs more test cases than the *MM* approach. The *MM* approach obtains worse average coverage because nested statements pose a great challenge for the search. We think that the main reason for this fact is that the multi-objective algorithms deal with all the branches at the same time and less information is obtained to guide the search.

As we previously said, automatically generating a test suite that covers the entire program is a hard task. When a program has high nesting degree and the decisions are very complex, the task of covering all the program code requires a lot of effort. It is important for an algorithm to be able to find test cases to cover all the program's branches. We show in Table 7.14 a comparison of the average maximum coverage obtained for all the algorithms and all the programs. It is clear that the best algorithm, if coverage is the main objective, is GA. It obtains the best results in all the groups of programs with different nesting degree, and therefore in the complete benchmark. The

performance of ES is also very good because it is always better than the multi-objective algorithms. If the nesting degree increases, the distance between the average coverage of GA and ES increases. In other words, the ES has a similar performance to GA in low complexity programs and it is worse than GA in complex programs. On the other hand, MOCell and NSGA-II have almost the same coverage; it only varies at the decimal level. SPEA2 is only 1% worse in the entire benchmark with respect to MOCell and NSGA-II. The results of the PAES algorithm are in the middle between the best (GA) and the worst (RNDMulti).

Table 7.14: Relationship between the nesting degree and the average coverage for all the algorithms. The standard deviation is shown in subscript.

| ND | GA | ES | RNDMono | MOCell | NSGA-II | SPEA2 | PAES | RNDMulti |
|---|---|---|---|---|---|---|---|---|
| 1 | $99.19_{2.20}$ | $98.70_{2.63}$ | $85.33_{10.51}$ | $98.10_{2.08}$ | $97.90_{2.22}$ | $97.53_{2.34}$ | $93.08_{5.30}$ | $81.36_{12.74}$ |
| 2 | $98.85_{2.02}$ | $97.87_{2.44}$ | $79.20_{12.14}$ | $94.77_{3.44}$ | $94.42_{3.49}$ | $93.56_{3.75}$ | $87.59_{6.31}$ | $75.04_{14.00}$ |
| 3 | $98.52_{2.09}$ | $95.66_{4.54}$ | $71.94_{13.36}$ | $90.66_{5.83}$ | $90.41_{5.46}$ | $89.29_{5.65}$ | $81.55_{7.68}$ | $69.77_{13.87}$ |
| 4 | $96.89_{4.80}$ | $93.19_{6.66}$ | $66.42_{14.80}$ | $85.50_{9.45}$ | $85.77_{8.18}$ | $84.61_{8.12}$ | $75.87_{9.22}$ | $63.87_{15.95}$ |
| Total | $98.36_{3.13}$ | $96.36_{4.90}$ | $75.72_{14.65}$ | $92.26_{7.54}$ | $92.12_{6.99}$ | $91.24_{7.24}$ | $84.52_{9.72}$ | $72.51_{15.57}$ |

In order to provide a high level of confidence to these results, we have performed statistical tests. The results are shown in Table 7.15. There are some differences among the best algorithms; we can take as a reference the column of the GA values. This column can be seen as a ranking of performance of all algorithms. The GA has the best results and outperforms the rest of the algorithms in average maximum coverage. ES is the second in average maximum coverage (with significant difference), next MOCell, then NSGA-II, and finally SPEA2. As we expected, the statistical test does not show significant differences among MOCell, NSGA-II and SPEA2, but if the number of independent runs were higher, the significant differences would appear. We should highlight that PAES is not much better than the random algorithms. The differences in average maximum coverage shown in Table 7.14 have been confirmed by the statistical tests: using a GA is the best way to obtain high branch coverage.

Table 7.15: Number of programs where a significant difference exists among the coverage obtained.

| | RNDMulti | PAES | SPEA2 | NSGA-II | MOCell | RNDMono | GA | ES |
|---|---|---|---|---|---|---|---|---|
| MOCell | 800▲ | 797▲ | 0 | 0 | – | 800▲ | 350▽ | 30▽ |
| NSGA-II | 800▲ | 786▲ | 0 | – | 0 | 800▲ | 438▽ | 87▽ |
| SPEA2 | 800▲ | 691▲ | – | 0 | 0 | 800▲ | 613▽ | 322▽ |
| PAES | 503▲ | – | 691▽ | 786▽ | 797▽ | 85▲ | 800▽ | 800▽ |
| RNDMulti | – | 503▽ | 800▽ | 800▽ | 800▽ | 7▽ | 800▽ | 800▽ |
| ES | 800▲ | 800▲ | 322▲ | 87▲ | 30▲ | 800▲ | 1▽ | – |
| GA | 800▲ | 800▲ | 613▲ | 438▲ | 350▲ | 800▲ | – | 1▲ |
| RNDMono | 7▲ | 85▽ | 800▽ | 800▽ | 800▽ | – | 800▽ | 800▽ |

Finally, we have considered in this experimental study the obtained HV, the significant differences, the attainment surfaces and the average maximum coverage achieved with all the algorithms and the benchmark of 800 programs. After analyzing the experimental results we can state that the GA is the best mono-objective algorithm and MOCell is the best multi-objective algorithm. We expected that an algorithm like MOCell would be clearly superior to all the mono-objective ones in the MOTDGP, but in fact this is not true. In addition, the GA is clearly superior in HV and average maximum coverage when we are testing programs with high nesting degree. This fact

is due to a better exploration of the search space because this algorithm is able to find solutions for the most complex branches that appear in the code. It uses most of its evaluations in most complex branches, in order to achieve a high coverage. However, the multi-objective algorithms deal with all the branches at the same time, for this reason they do not use most of its evaluations trying to cover a concrete complex branch. This fact suggests that if there exist hard requirements of coverage and the program has high nesting degree, we should use the GA as search engine of an automatic test data generator. Nevertheless, a second phase of multi-objective test case selection must be performed in order to minimize the oracle cost. On the other hand, if there are cost requirements, we highly recommend the use of MOCell algorithm.

### 7.3.4 Validation on Real Programs

In this section we analyze the two proposed approaches using some real-world programs. We study 13 real programs extracted from the literature and with characteristics similar to the artificial programs used in the previous sections. The reader must take into account that the number of programs used in the previous sections gives us the chance to average among 800 programs and extract statistically more reliable results. Despite the fact that in this section we only analyze the performance of the proposed approaches and algorithms over 13 programs, most of the conclusions are similar to the ones we have been obtained with the synthetic programs.

Once again we start the analysis with the HV indicator. In Table 7.16 we summarize the number of programs where the HV value of an algorithm is better than the others. There are six programs where an algorithm is the best. The GA outperforms the other algorithms in four programs, then the ES in two programs, and the MOCell in only one program. In the previous results these three algorithms also obtain the best results.

Table 7.16: Real programs in which the median hypervolume of one algorithm is better than the others and average maximum coverage of all the real programs.

| - | MOCell | NSGA-II | SPEA2 | PAES | RNDMulti | GA | ES | RNDMono |
|---|---|---|---|---|---|---|---|---|
| HV Better | 1 | 0 | 0 | 0 | 0 | 4 | 2 | 0 |
| Avg.Max.Cov. | 87.26 | 91.35 | 89.31 | 72.43 | 76.84 | 94.14 | 92.27 | 80.09 |

In order to validate these previous results we compared the HV values of all the real programs using the multiple comparison statistical test. In Table 7.17 we show the existing differences among the HV value of all the algorithms. We can observe that the GA outperforms the other algorithms in at least one program. Then, there is a group of algorithms composed by ES, NSGA-II, SPEA2, MOCell and RNDMono that are better than PAES and RNDMulti, but the statistical test does not show significant differences among them. In addition, we can analyze the PAES column in order to obtain an informal ranking of algorithms according to the HV indicator.

Let us analyze the average maximum coverage obtained by the algorithms when are applied to real programs (Table 7.16). We must highlight that the GA and the ES are the best algorithms in coverage for the real programs. In contrast, PAES has obtained the worst results, even worse than the random algorithms. In Table 7.18, we show the results of a statistical test to compare the maximum coverage. Once again the GA is the best algorithm: it obtains significant differences in 40 comparisons. NSGA-II obtains significant differences in 30 comparisons. Next, SPEA2 and ES are better than the others in 18 comparisons. Most of these significant differences are obtained in

Table 7.17: Real programs where a significant difference exists among the HV obtained.

| | RNDMulti | PAES | SPEA2 | NSGA-II | MOCell | RNDMono | GA | ES |
|---|---|---|---|---|---|---|---|---|
| MOCell | 1▲ | 1▲ | 0 | 0 | – | 0 | 2▽ | 0 |
| NSGA-II | 3▲ | 3▲ | 0 | – | 0 | 0 | 1▽ | 0 |
| SPEA2 | 3▲ | 2▲ | – | 0 | 0 | 0 | 1▽ | 0 |
| PAES | 0 | – | 2▽ | 3▽ | 1▽ | 1▽ | 7▽ | 6▽ |
| RNDMulti | – | 0 | 3▽ | 3▽ | 1▽ | 1▽ | 7▽ | 4▽ |
| ES | 4▲ | 6▲ | 0 | 0 | 0 | 0 | 1▽ | – |
| GA | 7▲ | 7▲ | 1▲ | 1▲ | 2▲ | 3▲ | – | 1▲ |
| RNDMono | 1▲ | 1▲ | 0 | 0 | 0 | – | 3▽ | 0 |

comparison with PAES or the random algorithms. Only a few differences exist between the best algorithms. Nevertheless, the performance of GA seems to be better than the other algorithms. On the other hand, PAES has obtained the worst results. We think that these results are due to the absence of crossover operator and the nature of the selected programs (i.e., number of equalities in the code).

Table 7.18: Number of real programs where a significant difference exists among the coverage obtained.

| | RNDMulti | PAES | SPEA2 | NSGA-II | MOCell | RNDMono | GA | ES |
|---|---|---|---|---|---|---|---|---|
| MOCell | 6▲ | 12▲ | 0 | 0 | – | 0 | 2▽ | 0 |
| NSGA-II | 12▲ | 13▲ | 1▲ | – | 0 | 4▲ | 0 | 0 |
| SPEA2 | 7▲ | 13▲ | – | 0 | 1▲ | 0 | 3▽ | 0 |
| PAES | 0 | – | 13▽ | 13▽ | 12▽ | 4▲ | 13▽ | 12▽ |
| RNDMulti | – | 0 | 7▽ | 12▽ | 6▽ | 3▽ | 12▽ | 9▽ |
| ES | 9▲ | 12▲ | 0 | 0 | 0 | 0 | 3▽ | – |
| GA | 12▲ | 13▲ | 3▲ | 0 | 2▲ | 7▲ | – | 3▲ |
| RNDMono | 3▲ | 4▽ | 0 | 4▽ | 0 | – | 7▽ | 0 |

Finally, with the aim of showing an example of the computed fronts for the instances, we selected the *line* program. This program can represent the typical behavior of the different algorithms in this kind of instance. In Figure 7.5 the 50%-attainment surfaces of the best algorithms are depicted. In this instance, GA dominates the others and has a good performance because it always reaches the best coverage with the same test data. MOCell is the only algorithm able to obtain all the points of the front. This is a desirable property for a solution of a multi-objective problem.

## 7.4 Conclusions

In this chapter we have studied the Multi-Objective Test Data Generation Problem with the aim of analyzing the performance of a direct multi-objective approach ($MM$) versus the application of mono-objective algorithms followed by a test case selection ($mM$). Previous results in the literature have only focused on the coverage of a program while the oracle cost is a significant cost that has

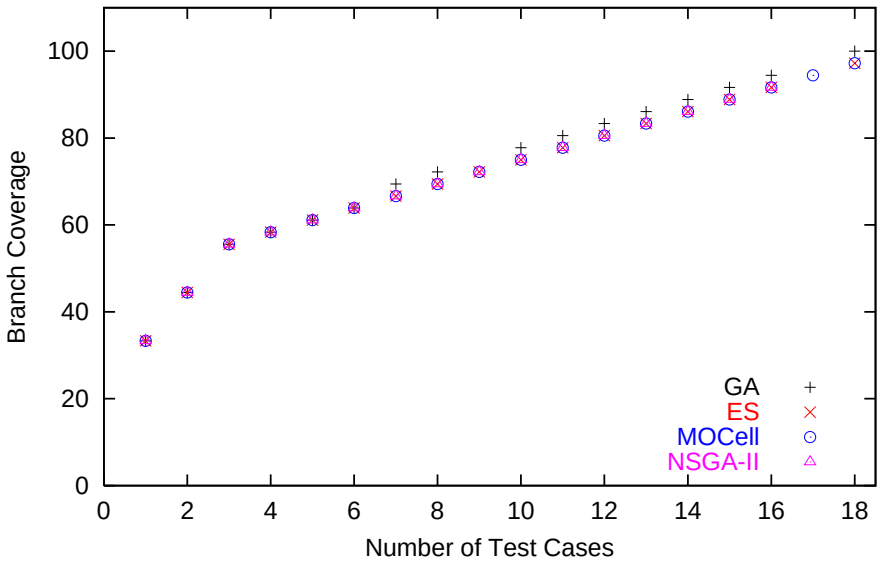



Figure 7.5: 50%-attainment surfaces: coverage against the number of test cases for the program *line*.

been ignored in most of the previous studies. For this reason, in this work we have dealt with the coverage and the oracle cost as equally important targets.

Our study has been performed on 800 synthetic programs. We used our program generator able to produce programs ensuring a 100% of branch coverage. This kind of programs is very useful because all the branches are reachable and we can compare the algorithms in a fair way using coverage. In addition, we have also analyzed the two proposed approaches with a benchmark of 13 real and popular programs in the literature. We have evaluated four state-of-the-art multi-objective optimization algorithms: MOCell, NSGA-II, SPEA2, and PAES, two mono-objective algorithms GA, ES, and two random algorithms as merely a 'sanity check'. This comparison has been done on the basis of three quality indicators: the hypervolume, the 50%-empirical attainment surface, and the average maximum coverage obtained by those algorithms. We can see a final ranking of algorithms in Table B.1.

In terms of convergence towards the optimal Pareto front, GA and MOCell have been the best solvers in our comparison. On the one hand, MOCell has obtained the best fronts in programs with nesting degree 1 and 2, values commonly found in practice. On the other hand, GA is the best algorithm for facing programs with high nesting degree and it is the algorithm which is significantly better in most of the programs, attending to the HV indicator and to the average maximum coverage. This fact indicates that GA is the best alternative if the tested program has a high nesting degree or we need a high coverage. But, if we have time restrictions, we highly recommend the use of MOCell as a search engine for an automatic test data generator. Although the multi-objective approach is working very well in most of the programs, we realized that dealing with only one branch at the same time (mono-objective approach) can be more effective when the program under test has high nesting degree. In addition, we must highlight that both approaches ($MM$ and $mM$) are quite good at reducing the number of test cases needed to obtain a given

Table 7.19: Ranking of algorithms according to maximum coverage and hypervolume grouped by nesting degree.

| Coverage | | | | | |
|---|---|---|---|---|---|
| Rank | ND 1 | ND 2 | ND 3 | ND 4 | All |
| 1 | GA | GA | GA | GA | GA |
| 2 | ES | ES | ES | ES | ES |
| 3 | MOCell | MOCell | MOCell | NSGA-II | MOCell |
| 4 | NSGA-II | NSGA-II | NSGA-II | MOCell | NSGA-II |
| 5 | SPEA2 | SPEA2 | SPEA2 | SPEA2 | SPEA2 |
| 6 | PAES | PAES | PAES | PAES | PAES |
| 7 | RNDMono | RNDMono | RNDMono | RNDMono | RNDMono |
| 8 | RNDMulti | RNDMulti | RNDMulti | RNDMulti | RNDMulti |
| Hypervolume | | | | | |
| Rank | ND 1 | ND 2 | ND 3 | ND 4 | All |
| 1 | MOCell | MOCell | GA | GA | GA |
| 2 | NSGA-II | GA | MOCell | MOCell | MOCell |
| 3 | ES | NSGA-II | NSGA-II | ES | ES |
| 4 | GA | ES | ES | NSGA-II | NSGA-II |
| 5 | PAES | SPEA2 | SPEA2 | SPEA2 | SPEA2 |
| 6 | SPEA2 | PAES | PAES | PAES | PAES |
| 7 | RNDMono | RNDMono | RNDMono | RNDMono | RNDMono |
| 8 | RNDMulti | RNDMulti | RNDMulti | RNDMulti | RNDMulti |

coverage. The oracle cost can be greatly reduced because the $mM$ approach only needs 19.32% of the upper bound of test cases needed for obtaining the maximum value of coverage, and the $MM$ approach is even better, only needing a 15.12% of the test cases. This improvement justifies the use of our approaches to deal with the MOTDGP.



# Part III

# Functional Testing

# Chapter 8

# Combinatorial Interaction Testing using Classification Tree Method

## 8.1 Introduction

Software product specifications provide fundamental information of the software, meanwhile structural information is omitted. Software product specifications can be used as a guide for designing functional tests for the software product. However, there is no standard representation of the SUT. The Classification Tree Method [91] is a particular representation of the SUT, which is based on the category partition method [167]. It decomposes a test domain into disjoint classes representing important aspects of the test object.

Frequently, the systems under test tend to be large and exhaustive testing is generally impractical. Therefore, we should use an adequacy criterion less exhaustive but with certain level of confidence, one widely used technique is Combinatorial Interaction Testing. CIT is a black box sampling technique derived from the statistical field of design of experiments. It is used to determine the smallest possible subset of tests that covers all combinations of values specified by a coverage criterion with at least one test case. A coverage criterion is defined by its strength $t$ that determines the degree of parameter interaction and assumes that all parameters are considered. The most common coverage criterion is 2-wise (or pairwise, $t = 2$) testing, that is fulfilled if all possible pairs of values are covered by at least one test case in the result test set. Hence, this criterion is the chosen one in this thesis dissertation.

In addition, when the resources are limited in time or cost, important test cases should be executed earlier, then the prioritization of test cases is a must. The prioritization of test cases may reveal faults in early stages of the testing phase. Prioritization can be computed in two ways: (1) reorder an existing test suite based on a prioritization criteria; and (2) generate an ordered test suite, taking into account the importance of combinations. Consequently, we study here the *Prioritized Pairwise Test Data Generation Problem* (PPTDP) previously defined in Section 2.4.1. We evaluate the performance of metaheuristic techniques and compare the two different prioritization approaches. In order to achieve this objective we perform a comparison among five algorithms on a set of benchmarks found in the literature.

Although combinatorial testing has been widely studied, we still find two main issues that have not been addressed by the traditional generation of test suites: the dependencies between individual test cases and the state of the SUT. Sometimes software is required to be in a particular

state to test a given functionality. This is the case of most software artifacts. Actually, in very large software systems, the cost incurred to place the system in a certain state can be an issue. For example, testing the anti-lock braking system (ABS) of a car requires that the car reaches a certain speed before the system can be tested. So it makes sense to consider the generation of test sequences that allow us to test a particular functionality (acceleration of the car) while we change the state of the SUT (considering the dependency rules in the test cases) to test the next functionality (ABS). The implicit cost savings of using this technique is the reason why the generation of test sequences is relevant and deserves more research effort.

In this sense, we have extended the CTM (ECTM) in Section 2.4.2 to be able to automatically generate test sequences, since it cannot be done automatically by means of the CTE XL professional tool[1]. We study the Test Sequence Generation Problem (TSGP), previously formulated in Section 2.4.2. We have compared the behavior of two metaheuristic techniques with an existing greedy algorithm [126]. The first proposed approach is a GA called *Genetic Test Sequence Generator* (GTSG). We have improved a GTSG with the addition of a Memory Operator (MemO), which is based on the operator proposed by Alba et al. [4]. It is used to reduce the amount of resources needed to compute a solution. The other proposed algorithm is based on the ACO, described in Section 4.1.3. Specifically, we propose a new technique based on ACO that is able to deal with large construction graphs. It is able to find near-optimal solutions in separated areas of the search space for the *Test Sequence Generation Problem* (TSGP). It is called ACO for test sequence generation (ACOts). Both proposed metaheuristics are used to generate test sequences to obtain full class and transition coverage of 12 different programs extracted from the literature.

Overall, we can raise the following research questions and try to answer them in an extensive experimental study.

- RQ1: Is our evolutionary approach effective when dealing with prioritization?
- RQ2: Do the test sequences generated by evolutionary approaches save resources?

The *Classification Tree Editor* (CTE) [132] is a commercial tool supporting the CTM (Figure 8.1). This tool is able to automatically generate test data, therefore we have followed its specification to integrate our proposals in this tool.

## 8.2 Prioritized Pairwise Test Data Generation using CTM

Software testers are faced with situations in which there is not enough time for testing, since the software under test must be finished on time for the release date not to be delayed. Hence, software testers have to deal with limited resources, unfinished systems, and not much time to test the software. Although a tester aims at executing as many test cases as possible, often a test case selection has to be done. The prioritization of test cases is a re-ordering of tests to find faults in early stages. But, if the time run-out, this technique also allows the tester to specify the desired level of coverage and failure-detection. The result of the prioritization is then a schedule of test cases so that those with the highest priority, according to some criterion, are executed earlier.

In this study, priorities are assigned to the classification tree elements (see Section 2.4.1) in order to indicate the importance of the element. These weights can be used to guide the test case generation in order to cover first the most important values.

There exist different prioritization techniques. Elbaum et al. provide good overviews of existing approaches [67, 68]. The following three models were selected to provide a basis for prioritization:

[1]This is a commercial tool developed by Berner & Mattner company. It was recently renamed as TESTONA

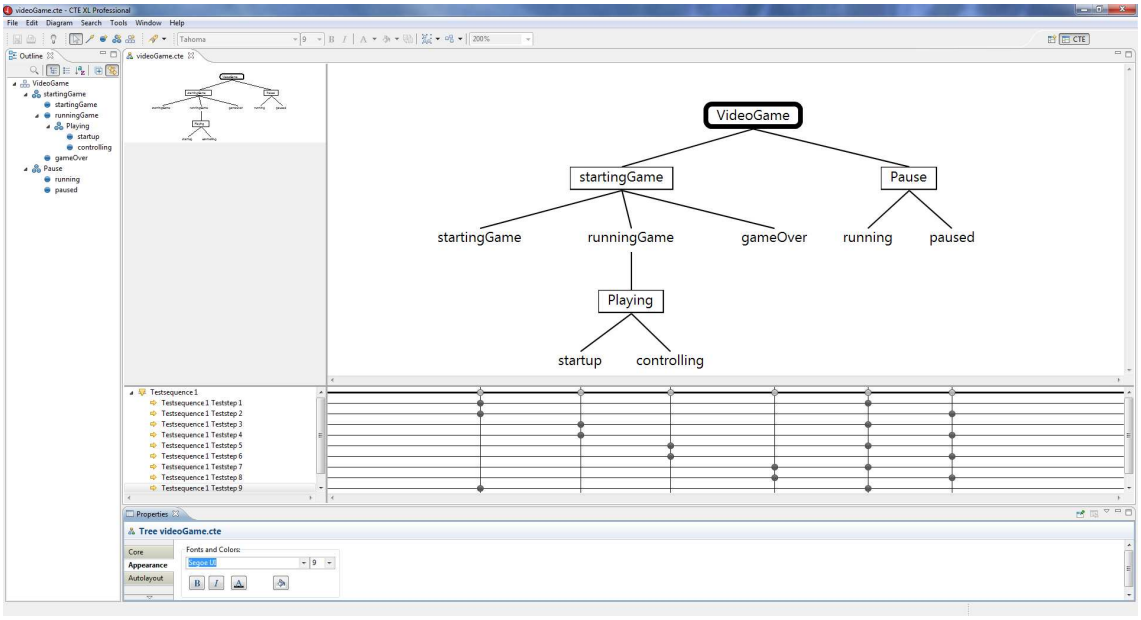



Figure 8.1: CTE XL professional tool.

- Prioritization based on a usage model [211] tries to reflect usage distribution of all classes in terms of usage scenarios. Classes with a high occurrence have higher weights than classes with a low occurrence.
- Prioritization based on an error model [67] aims to reflect distribution of error probabilities of all classes. Classes with a high probability of revealing an error have higher weights than classes with a low probability.
- Prioritization based on a risk model [13] is similar to prioritization based on an error model but also takes error costs into account. Classes with a high risk have higher weights than classes with a low risk.

Since we are facing here the Prioritized Pairwise Test Data Generation problem, we need to define weights for each pair of classes. This is done by multiplying the weight of each class involved in the pair. Following the Video Game example (Figure 2.2) the pair weight for (Startup (Playing), running) is 0.63 * 0.52 = 0.3276 .

### 8.2.1 Solution Approaches

This study aimed to evaluate the performance of metaheuristic techniques for dealing with the PPTDGP. In order to achieve this objective we perform a comparison among five algorithms on a set of benchmarks found in the literature. We first introduce the details of our evolutionary approach, a genetic algorithm. To the best of our knowledge the author of this thesis was the first to apply an evolutionary approach to the target problem [75]. Then, we briefly describe two deterministic algorithms that we have implemented for comparison purposes, the *Prioritized*

*Pairwise Combination algorithm* (PPC) and the *Plain Pairwise Sorting* (PPS). We finally present the *Deterministic Density Algorithm* (DDA) developed in [37] and an algorithm based on *Binary Decision Diagrams* (BDD) introduced by Lee [131].

#### Prioritized Genetic Solver

The *Prioritized Genetic Solver* (GS) is an evolutionary approach that constructs an entire test suite taking into account priorities in the generation. It is a constructive algorithm that adds one new test datum to the partial solution in each iteration until all pairwise combinations are covered. In each iteration the algorithm tries to find the test datum that adds more coverage to the partial solution. Overall, our algorithm aims at generating an entire test suite (to cover all pairwise combinations) by generating the best test datum at a time until all pairwise combinations are covered. The best test datum is the one that most reduce the weighted value of the set of remaining pairs to cover.

In this particular implementation we have used a single point crossover with probability 1 of recombining the two selected individuals. This operator is able to put together good solution components that are scattered in the small population used of 4 individuals. Regarding the mutation operator, it iterates over all the components in the solution vector changing its value by a random one of the same classification with probability 0.05. The maximum number of evaluations used as stopping criterion in the internal loop is $5,000$ while the stopping condition of the external loop is to achieve full pairwise coverage.

The computation of the fitness value for each solution is done through the following process: The algorithm computes the combined class pairs of the partial solution (the next test datum). After that, it removes these pairs from the set of remaining pairs. Finally, the fitness value of a solution is computed as the sum of the weights of the remaining pairs. That is, the objective value of a proposed test datum is the sum of the weights of the class pairs that are not covered yet after adding the test datum to the suite. This objective function must be minimized in order to take first the test datum covering the class pairs with higher weights. As the search progresses the computational cost of computing the fitness function is reduced, since less class pairs remain uncovered.

#### Prioritized Pairwise Combination

In this algorithm, the class pair with the highest weight from the set of uncovered class pairs is chosen for the new test datum. We determine all candidate test data containing this class pair and calculate the index values for these candidates. This index value includes the weights and the number of newly covered class pairs. PPC then selects the test datum with the highest assigned index value. This way, we can guarantee that the $n$ first test data cover the $n$ more important class pairs. The generation process using PPC is deterministic: the same test suite is generated for identical classification trees.

#### Plain Pairwise Sorting

This algorithm first applies a plain pairwise algorithm (the one integrated in CTE XL professional tool), which computes a set of test data covering all the class pairs. Then it sorts the test data taking into account their absolute weight at first. Then, it applies as many discriminatory reorderings as test data. Note that this approach does not guarantee coverage of any $n$ most important class pairs by the $n$ first test data. However, the generated test suite will have exactly the same size

as the plain pairwise combination, as the suite does not grow by sorting. The generation process using sorting is deterministic, however its results differ from the PPC results.

#### Deterministic Density Algorithm

In the Deterministic Density Algorithm [37] one test datum is constructed at a time and new test data are generated until all $t$-tuples are covered. Each classification is assigned a class value one-at-a-time. A classification that has been assigned a class value is referred to as fixed; one that has not, as free. For each classification, the class value that covers the largest density is selected. Then, a density formula calculates the likelihood of covering future tuples. To modify DDA to account for prioritization, the density formula is modified. Instead of computing the ratio of uncovered pairs to be covered, the amount of weight to be covered is computed.

#### Binary Decision Diagrams Algorithm

Binary Decision Diagrams [131] are acyclic directed graphs used to represent propositional logical formulas. In [186] the authors introduced an approach based on the modeling of the combinatorial interaction test problem as a single propositional logic formula. They constructed the formula such that the set of satisfying interpretations of the formula corresponds to the set of valid test cases and such that a one-to-one relation between a satisfying interpretation of the formula and a valid test case of the CIT problem exists. The formula is the conjunction of a subformula representing the set of all test cases and a subformula representing the set of constraints. They have used this formula in a greedy algorithm, in the following BDD, to select test cases until the desired coverage criterion is fulfilled.

### 8.2.2 Experimental Benchmark

The experimental benchmark used for the comparison among algorithms was proposed in [37]. The scenarios $S1 - S8$ are given in Table 8.1[2]. The number of classes of each scenario are given in a shorthand notation, where for example *S5* with $8^2 7^2 6^2 2^4$ consists of 2 classifications with 8 classes, 2 classifications with 7 classes, 2 classifications with 6 classes, and 4 classifications with 2 classes.

Table 8.1: Scenarios and number of factors.

| Scenarios | #Classes |
|---|---|
| *S1* | $3^4$ |
| *S2* | $10^{20}$ |
| *S3* | $3^{100}$ |
| *S4* | $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1$ |
| *S5* | $8^2 7^2 6^2 2^4$ |
| *S6* | $15^1 10^5 5^1 4^1$ |
| *S7* | $3^{50} 2^{50}$ |
| *S8* | $20^2 10^2 3^{100}$ |

The given benchmark uses four different weight distributions applied to the eight scenarios. The distributions are:

[2]They are available at http://neo.lcc.uma.es/staff/javi/resources.html

- *d1* (equal weights): All classes have the same weight,
- *d2* (50/50 split): Half of the weights for each classification are set to 0.9 the other half to 0.1,
- *d3* ($(1/vmax)^2$ split): All weights of classes for a classification are equal to $(1/vmax)^2$, where $vmax$ is the number of classes associated with the classification,
- *d4* (random): Weights are randomly distributed.

Table 8.2: Number of test cases needed for the GA, PPC, and PPS algorithms in eight scenarios and for four distributions. When significant differences exist between the GS and other algorithm we add an asterisk.

| Scenario | Coverage | *d1* | | | *d2* | | | *d3* | | | *d4* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | GS | PPC | PPS | GS | PPC | PPS | GS | PPC | PPS | GS | PPC | PPS |
| *S1* | 25% | 3 | 3 | 3 | 1 | 1 | 1 | 3 | 3 | 3 | 2 | 2 | 2 |
| | 50% | 5 | 5 | 5 | 1 | 1 | 2 | 5 | 5 | 5 | 3 | 3 | 3 |
| | 66% | 6.29 | 7* | 6* | 1 | 1 | 3* | 6.29 | 7* | 6* | 5 | 5 | 5 |
| | 75% | 7.48 | 8* | 7* | 3 | 3 | 4* | 7.35 | 8* | 7* | 6 | 6 | 5* |
| | 90% | 9.3 | 9* | 9* | 6.29 | 7* | 5* | 9.18 | 9* | 9* | 8 | 8 | 7* |
| | 95% | 9.93 | 10 | 9* | 8 | 8 | 7* | 9.88 | 10 | 9* | 9 | 10* | 8* |
| | 99% | 10.42 | 10 | 9* | 10.28 | 11* | 8* | 10.19 | 10 | 9* | 11 | 11 | 9* |
| *S2* | 25% | 26 | 27* | 27* | 8.23 | 9* | 12* | 26 | 27* | 27* | 12 | 12 | 19* |
| | 50% | 56.1 | 56 | 60* | 19 | 18* | 36* | 56.11 | 56 | 60* | 31 | 31 | 47* |
| | 66% | 80.05 | 79* | 89* | 29 | 27* | 60* | 80.04 | 79* | 89* | 49.83 | 50 | 74* |
| | 75% | 96,75 | 95* | 110* | 38.13 | 36* | 79* | 96.78 | 95* | 110* | 64.02 | 65* | 95* |
| | 90% | 134.48 | 132* | 162* | 89.69 | 87* | 131* | 134.48 | 132* | 162* | 102.33 | 104* | 150* |
| | 95% | 154.84 | 152* | 190* | 122.77 | 121* | 163* | 154.76 | 152* | 190* | 125.59 | 129* | 181* |
| | 99% | 182.94 | 180* | 228* | 169.9 | 169* | 212* | 182.93 | 180* | 228* | 163.14 | 169* | 223* |
| *S3* | 25% | 3 | 3 | 3 | 1 | 1 | 2* | 3 | 3 | 3 | 1 | 1 | 3* |
| | 50% | 6 | 6 | 6 | 1 | 1 | 4* | 6 | 6 | 6 | 3 | 3 | 5* |
| | 66% | 8 | 8 | 8 | 1.76 | 1* | 7* | 8 | 8 | 8 | 5 | 5 | 8* |
| | 75% | 9.02 | 9 | 10* | 3.64 | 4* | 8* | 9.03 | 9 | 10* | 7 | 7 | 10* |
| | 90% | 14 | 15* | 16* | 8 | 9* | 13* | 14 | 15* | 16* | 12 | 12 | 15* |
| | 95% | 17.9 | 20* | 20* | 11.84 | 12 | 16* | 17.91 | 20* | 20* | 15 | 15 | 19* |
| | 99% | 24.13 | 37* | 26* | 19 | 19 | 23* | 24.07 | 37* | 26* | 21 | 21 | 26* |
| *S4* | 25% | 9.41 | 11* | 10* | 3 | 3 | 4* | 5 | 4* | 4* | 7 | 6* | 7 |
| | 50% | 21.02 | 24* | 22* | 7 | 8* | 11* | 9 | 8* | 8* | 15.09 | 15 | 18* |
| | 66% | 31.95 | 36* | 34* | 11.62 | 12* | 17* | 13.03 | 12* | 14* | 23.12 | 23 | 28* |
| | 75% | 39.67 | 45* | 42* | 15.46 | 16* | 23* | 17 | 16* | 19* | 29.09 | 29 | 36* |
| | 90% | 59.12 | 64* | 63* | 32.7 | 36* | 38* | 30.84 | 30* | 35* | 45.02 | 46* | 56* |
| | 95% | 70.57 | 74* | 74* | 49.4 | 53* | 51* | 43.28 | 42* | 46* | 55.37 | 59* | 67* |
| | 99% | 88.24 | 86* | 88 | 77.41 | 76* | 77* | 72.17 | 72 | 70* | 75.76 | 82* | 82* |
| *S5* | 25% | 8 | 8 | 7* | 2.46 | 2* | 3* | 2 | 2 | 2 | 3 | 3 | 4* |
| | 50% | 16.97 | 18* | 17 | 5.46 | 6* | 8* | 4 | 4 | 4 | 8 | 8 | 10* |
| | 66% | 27.09 | 28* | 27 | 9.46 | 10* | 15* | 7 | 7 | 7 | 13.12 | 14* | 17* |
| | 75% | 34.65 | 35* | 34* | 12.52 | 13* | 21* | 9 | 9 | 10* | 18.05 | 19* | 24* |
| | 90% | 50.04 | 50* | 51* | 26.89 | 28* | 37* | 14.01 | 15* | 19* | 32.13 | 33* | 41* |
| | 95% | 58 | 57* | 59* | 42.06 | 43* | 46* | 25.22 | 23* | 25* | 41.32 | 44* | 50* |
| | 99% | 69.5 | 66* | 68* | 60.86 | 61 | 62* | 52.54 | 52* | 54* | 56.61 | 60* | 65* |
| *S6* | 25% | 22 | 23* | 22 | 7 | 7 | 9* | 12 | 12 | 12 | 10.04 | 11* | 14* |
| | 50% | 45,98 | 52* | 49* | 16 | 17* | 24* | 25 | 26* | 26* | 25.98 | 27* | 36* |
| | 66% | 67.17 | 74* | 74* | 24.53 | 25* | 39* | 36 | 38* | 41* | 40.51 | 43* | 57* |
| | 75% | 82.48 | 89* | 92* | 31.33 | 31 | 53* | 44.1 | 46* | 53* | 52.1 | 56* | 73* |
| | 90% | 117.65 | 123* | 131* | 71.71 | 73* | 92* | 81.15 | 83* | 92* | 84.32 | 90* | 111* |
| | 95% | 136.98 | 139* | 149* | 105.23 | 106* | 114* | 105.4 | 107* | 120* | 104.83 | 112* | 131* |
| | 99% | 158.19 | 159* | 169* | 146.44 | 148* | 153* | 146.43 | 149* | 158* | 139.87 | 148* | 160* |
| *S7* | 25% | 2 | 2 | 2 | 1 | 1 | 2* | 2 | 2 | 2 | 1 | 1 | 2* |
| | 50% | 4 | 4 | 4 | 1 | 1 | 3* | 3 | 3 | 4* | 2 | 2 | 4* |
| | 66% | 6 | 6 | 6 | 1 | 1 | 4* | 5 | 5 | 5 | 4 | 4 | 6* |
| | 75% | 7 | 7 | 8* | 2 | 2 | 5* | 6 | 6 | 7* | 5 | 5 | 7* |
| | 90% | 11 | 11 | 12* | 6 | 6 | 9* | 9 | 10* | 10* | 8.81 | 9 | 11* |
| | 95% | 14 | 14 | 15* | 8.06 | 9* | 12* | 12 | 13* | 13* | 11 | 11 | 14* |
| | 99% | 20.99 | 20* | 21 | 14.65 | 14* | 18* | 19 | 21* | 19 | 16.99 | 17 | 21* |
| *S8* | 25% | 3 | 3 | - | 1 | 1 | - | 2.82 | 3 | - | 2 | 3* | - |
| | 50% | 8 | 8 | - | 3 | 3 | - | 5.64 | 6 | - | 5 | 6* | - |
| | 66% | 12 | 13* | - | 6 | 7* | - | 7.52 | 8 | - | 9 | 8* | - |
| | 75% | 16.38 | 18* | - | 9 | 12* | - | 9.39 | 9* | - | 14 | 9* | - |
| | 90% | 37.86 | 64* | - | 20.37 | 30* | - | 14.1 | 15 | - | 31.66 | 15* | - |
| | 95% | 52.97 | 92* | - | 35.06 | 56* | - | 17.71 | 19* | - | 46.62 | 19* | - |
| | 99% | 130.87 | 145* | - | 97.32 | 114* | - | 25.94 | 28* | - | 118.22 | 28* | - |
| | Times | 22 | 12 | 7 | 26 | 11 | 3 | 19 | 14 | 5 | 24 | 8 | 4 |

### 8.2.3 Comparison between GS, PPC and PPS

In this comparison we are evaluating first three algorithms (GS, PPC and PPS) in order to analyze their behavior for the computation of minimal test suites when we use weight coverage as adequacy criterion (Definition 2.4.8). In general the GS performs better than the PPC and the PPS algorithm in most cases, nevertheless we want to highlight the weaknesses and strengths of each algorithm. The detailed results for all the scenarios, distributions, and weight coverage are given in Table 8.2. We should take into account that the observations of the number of test cases needed to achieve the different values of coverage are taken in the same execution of the algorithm. We execute the GS 100 times for a particular scenario-distribution combination, then we have done $3,200$ executions of the GS algorithm. In order to validate the experimental results we compared the 100 samples of the GS with the values of the deterministic algorithms using the Wilcoxon rank-sum test with 95% of statistical confidence. In Table 8.2 we marked with an asterisk ($*$) the values that are statistically different with respect to the GS's value.

Let us first analyze the results obtained by distribution. If all the weights of the interactions are the same (distribution *d1*), the GS performs better than the other algorithms. In particular, the GS is the best 22 times, while the PPC algorithm is the best 12 times, and the PPS algorithm only 7 times. Although the GS is the best algorithm, it is worse than the PPC algorithm when 99% of coverage is required. Besides, the PPS algorithm obtains its better results with this distribution. This behavior was expected because solving the problem with *d1* is the same as the pairwise combination problem without priorities as we commented in Section 8.2.2. Particularly, the sorting in PPS is carried out after executing of a plain pairwise algorithm, which do not use the weight, since it only tries to cover as many class pairs as possible in each test datum. This is the reason why the PPS algorithm works well with the distribution *d1*.

In *d2*, when extreme values are used, GS obtains the best results. It is the best algorithm in 26 scenarios, while the PPC is the best in 11 and the PPS in 3. The GS performs quite well for all target weights and it is specially good with 95% of weight coverage, where it is the best in 6 out of 8 scenarios (with statistically significant differences in most of the cases). In *d3*, the GS is slightly better than the PPC algorithm. The GS obtains the minimum number of test cases in 19 scenarios while the PPC algorithm is the best in 14. Thus, the GS and the PPC algorithm obtain similar results. When the target weight is 75%, the PPC algorithm is better in 5 out of the 8 scenarios, but when high coverage is required (90%) the GS is better in the same proportion (5 out of 8). When random weights are used (distribution *d4*), the PPC and PPS algorithms obtain the worse results of all the distributions, whilst the GS behaves very well. The GS is the best in 24 scenarios, the PPC algorithm in 8 and the PPS only in 4. At the beginning of the search, when the target weights are less than 75%, the differences among the algorithms are not very large, but for weights larger than 75% the GS is much better than the others.

The GS is the best algorithm for all the studied distributions as we have commented in the previous paragraphs. However, if we analyze the results by considering each scenario independently, some weaknesses of the GS appear. Let us analyze the influence of the benchmark scenarios in the obtained results. In Table 8.3 we summarize the number of scenarios in which one algorithm is better than the others.

In *S1*, the GS is not the best in any combination distribution/scenario. In this small scenario, the PPS algorithm is the best (18 times out of 28), while the other algorithms cannot outperform PPS. Since it is a small scenario, there is no reason to use a prioritized test case generation, but at least these bad results should be taken into account as a disadvantage of the prioritized generation. In *S2*, the PPC algorithm is the best. It is the best in 18 observations, the GS is the best in 8 and the PPS algorithm never outperforms the previous algorithms in this scenario. In *d1*, *d2* and *d3*

Table 8.3: Number of times that one algorithm is better than the other two for each instance.

| Scenario | GS | PPC | PPS |
|---|---|---|---|
| *S1* | 0 | 0 | 18 |
| *S2* | 8 | 18 | 0 |
| *S3* | 9 | 3 | 0 |
| *S4* | 14 | 9 | 1 |
| *S5* | 13 | 6 | 3 |
| *S6* | 24 | 1 | 0 |
| *S7* | 5 | 2 | 0 |
| *S8* | 19 | 6 | - |

distributions the GS is the best for 25% target weight, which means that the algorithm is able to combine high weight pairs in early test cases. For the rest of target weights, the PPC algorithm is the best. In the other six scenarios we can observe that GS is usually the algorithm computing the best results. In particular, in *S6* it is the best in 24 observations out of 28, while the PPC algorithm is the best only in 1 observation, and the PPS algorithm is never the best.

We also show in Table 8.4 the number of observations where there exists significant difference between the GS, the PPC and PPS algorithms. The number in front of a triangle up (▲) is the number of times that the results of GS are better than the ones of the other algorithms with statistically significant difference. The number in front of a triangle down (▽) is the number of times GS is worse with statistically significant difference. According to the results of Table 8.4, the GS is better than the other two algorithms in all the distributions. In the comparison between GS and PPC, the best distributions for the GS are *d1* and *d2* where the GS outperforms the PPC algorithm in 28 and 26 times, respectively. If we compare the GS and the PPS algorithm, the differences between the algorithms are even larger. In *d2* and *d4* the GS outperforms in 42 and 41 times, respectively. Thus, based on the statistical tests, we can state that the GS is clearly the best algorithm.

Table 8.4: Number of observations where there exists significant difference among the GS, the PPC and PPS algorithms.

| Algorithm-Distribution | PPC | PPS |
|---|---|---|
| GS-d1 | 28▲10▽ | 29▲8▽ |
| GS-d2 | 26▲9▽ | 42▲3▽ |
| GS-d3 | 19▲10▽ | 29▲8▽ |
| GS-d4 | 22▲6▽ | 41▲4▽ |

In conclusion, we can partially answer **RQ1**: the GS obtains better test suite size for all the distributions compared with PPC and PPS. There is not much difference among the distributions, thus the good performance of the GS does not depend on the distribution itself. According to the target weights, the GS is always the best except when the target weight is 75%. Under that circumstances the PPC algorithm and the GS obtain similar results. PPC is better in 12 observations while the GS is better in 11. Thus, we consider that GS is better in most target weights as well. Despite the optimization solver is the best in most cases, there are two scenarios, *S1* and *S2*, where the GS is not the best. In the first one, *S1* (the smallest), sorting is clearly the best option. In the second one, *S2*, GS is not the best but for the firsts weight coverage it obtains the best results. This behavior is desirable when we are computing prioritized test data.



### 8.2.4 Comparison between Genetic Solver and other existing algorithms

In this section we compare the results of our evolutionary algorithm with the ones of other approaches found in the literature. We show in Table 8.5 the results of three approaches (GS, DDA and BDD), with eight scenarios, four distributions and three values of weight coverage 50%, 75% and 100%. We have chosen these values according to the information available on these approaches extracted from the literature.

Table 8.5: Number of test cases needed for the GA, DDA, and BDD algorithms in eight scenarios and for four distributions. When significant differences exist between the GS and other algorithm we add an asterisk.

| | | 50% | | | 75% | | | 100% | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | GS | DDA | BDD | GS | DDA | BDD | GS | DDA | BDD |
| *d1* | *S1* | 5 | 5 | 5 | 7,48 | 7* | 7* | 10,42 | 9* | 10 |
| | *S2* | 56,1 | 57* | 62* | 96,76 | 97 | 112* | 203,29 | 220* | 251* |
| | *S3* | 6 | 6 | 6 | 9,02 | 9 | 11* | 31,91 | 32 | 33* |
| | *S4* | 21,02 | 22* | 26* | 39,67 | 40* | 47* | 97,53 | 95* | 94* |
| | *S5* | 15,97 | 16 | 19* | 33,65 | 33* | 37* | 75,16 | 72* | 74* |
| | *S6* | 45,98 | 47* | 62* | 82,48 | 84* | 105* | 171,38 | 175* | 184* |
| | *S7* | 4 | 4 | 4 | 7 | 7 | 8* | 29,47 | 29* | 28* |
| | *S8* | 8 | 8 | 13* | 16,38 | 16* | 35* | 437,81 | 400* | 400* |
| *d2* | *S1* | 1 | 1 | 1 | 3 | 3 | 3 | 11,98 | 14* | 14* |
| | *S2* | 19 | 19 | 21* | 38,13 | 39* | 49* | 221,53 | 256* | 278* |
| | *S3* | 1 | 1 | 1 | 3,64 | 4* | 4* | 31,45 | 35* | 35* |
| | *S4* | 7 | 7 | 10* | 15,46 | 16* | 22* | 103,39 | 203* | 100* |
| | *S5* | 5 | 5 | 7* | 12,06 | 12 | 15* | 79,21 | 83* | 83* |
| | *S6* | 16 | 16 | 28* | 31,33 | 32* | 55* | 185,94 | 192* | 207* |
| | *S7* | 1 | 1 | 1 | 2 | 2 | 2 | 28,81 | 31* | 29 |
| | *S8* | 3 | 3 | 3 | 9 | 9 | 16* | 438,4 | 400* | 400* |
| *d3* | *S1* | 5 | 5 | 5 | 7,35 | 8* | 7* | 10,19 | 13* | 10 |
| | *S2* | 56,09 | 60* | 62* | 96,78 | 112* | 112* | 203,2 | 325* | 249* |
| | *S3* | 6 | 6 | 6 | 9,03 | 10* | 11* | 31,9 | 37* | 33* |
| | *S4* | 8 | 8 | 8 | 16 | 19* | 18* | 113,24 | 115* | 137* |
| | *S5* | 4 | 4 | 5* | 9 | 10* | 10* | 84,66 | 94* | 101* |
| | *S6* | 25 | 26* | 27* | 44,1 | 51* | 53* | 179,63 | 208* | 222* |
| | *S7* | 3 | 3 | 4* | 6 | 6 | 7* | 30,81 | 33* | 33* |
| | *S8* | 5,76 | 6 | 6 | 9,59 | 10 | 11* | 407,47 | 417* | 463* |
| *d4* | *S1* | 3 | 3 | 4* | 6 | 5* | 6 | 12 | 13* | 13* |
| | *S2* | 31 | 32* | 44* | 64,02 | 67* | 94* | 220,59 | 265* | 286* |
| | *S3* | 3 | 3 | 5* | 7 | 7 | 11* | 31,28 | 34* | 47* |
| | *S4* | 14,09 | 25* | 17* | 28,09 | 29* | 33* | 99,95 | 100 | 108* |
| | *S5* | 8 | 9* | 10* | 18,05 | 19* | 24* | 76,96 | 82* | 88* |
| | *S6* | 25,98 | 27* | 35* | 52,1 | 56* | 70* | 178,44 | 192* | 208* |
| | *S7* | 2 | 2 | 3* | 5 | 5 | 7* | 28,05 | 32* | 42* |
| | *S8* | 5 | 5 | 8* | 14 | 15* | 26* | 418,77 | 406* | 406* |
| | | 11 | 0 | 0 | 18 | 5 | 1 | 23 | 2 | 4 |

Let us analyze the obtained results by weight coverage. If we focus on 50% of weight coverage, we can see that the GS is the unique algorithm that is able to outperform the others. The GS is the best in 11 observations. The GS is even better in the first measures of coverage, when it is more difficult to generate differences. For 50% of weight coverage the DDA algorithm needs 5.24% more test cases and the BBD algorithm needs 23.34% more test cases than the GS. Thus, we can state that our algorithm is clearly the best for 50% of weight coverage.

For 75% of weight coverage, the GS is the best again. Our algorithm is better than the other algorithms in 18 observations, while the DDA algorithm is the best in 5 observations and the BDD algorithm is the best in only one. For this value of weight coverage, the DDA and BDD algorithms need 4.98% and 28.63% more test cases, respectively. Once again the GS obtains better results than the DDA and BDD algorithms.

For total coverage, the GS is better than the other algorithms in 23 observations, the DDA in 2 and the BDD in 4. The difference between the GS and the others here is the largest one. This is a very interesting property of our GS algorithm, since it is not usual that an algorithm which is good for low/medium values of weight coverage is also good for total coverage. It is noteworthy that we do not configure the algorithm to obtain good test suite size for a particular value of coverage, but we just try to achieve all values of coverage with the minimum number of test data.

Regarding the different distributions, the GS maintains a good behavior in all the distributions. The GS algorithm is the best in 10 observations with distributions *d1* and *d2*. In addition, the GS is even better in the distributions *d3* and *d4*, since it is the best in 16 observations. Besides, if we focus on the other algorithms, the differences appear. The *d1* is clearly the best distribution for them. We should highlight that *d1* (equal weights) is the distribution where the priority is not used, the same weight is used for all the classes.

In order to provide a high level of confidence to these results, we have performed statistical tests. The results are shown in Table 8.6. There are some differences among the algorithms; we again take as a reference the GS values. Despite the GS and the DDA are both statistically better in 7 times for the *d1* distribution, in the rest of values of the table we can see that the GS is clearly the best algorithm. In the comparison between GS and DDA, the GS is significantly better in 49 observations while the DDA is only significantly better in 10 observations. In the comparison between GS and BDD, the GS is significantly better in 71 observations while the BDD is only significantly better in 9 observations. Answering **RQ1**, we can state again that the GS is the best overall algorithm for the Prioritized Pairwise Test Data Generation problem.

Table 8.6: Number of observations where there exists significant difference among the GS, the DDA, and BDD algorithms.

| Algorithm-Distribution | DDA | BDD |
|---|---|---|
| GS-d1 | 7▲7▽ | 15▲5▽ |
| GS-d2 | 10▲1▽ | 16▲2▽ |
| GS-d3 | 16▲0▽ | 18▲1▽ |
| GS-d4 | 16▲2▽ | 22▲1▽ |

## 8.3 Test Sequence Generation using ECTM

The problem of generating test sequences has received little attention in the existing literature, much less than the traditional generation of test data. In addition to the constraints defined by the classification-classes hierarchy, in the Test Sequence Generation Problem (TSGP) we take dependency rules associated to the transitions among classes. In addition, optimal sequences of test could reduce the cost to test all functionality. For example, testing a car at high speed implies using the accelerator pedal, but it is not possible to use the brake at the same time, so after it is necessary to test the brake. We could plan a sequence of test cases to check several functionalities instead of one. We could reduce the cost by testing the functionalities in a sequence. Since it would be more costly to test one functionality, then putting the system into an initial state to test the next functionality, than testing all the functionalities sequentially. In addition, it is desirable that the set of generated test sequences as a whole fulfills predefined coverage levels. So, it could be useful to generate a test suite with test sequences covering all possible classes or transitions between classes of the classification tree. We now describe in plain text the Test Sequence Generation Problem.

One *test case* for an ECTM is a set of classes that fulfills some rules. In particular, it is not possible to have two classes that belong to the same classification and if a refined class is in the test case then there must be one class for each classification in which the parent class is refined. In addition, if a class is in the test case, all the ascendant classes in the ECTM must be also included in the test case. For example, the set $Q = \{startingGame, running\}$ is a test case, but the set $Q = \{runningGame\}$ is not a test case because there is no class of the *Pause* and *Playing* classifications.

We can *transit* from one test case to another one by taking one of the transitions between classes. The test case we reach excludes the source class of the transition and includes the destination class of the transition. In order to fulfill the rules described for the test cases, some classes in the starting test case could also go out of the set and additional classes could enter the new test case. For example, if we take transition $startingGame \rightarrow runningGame$ from test case $Q_1 = \{startingGame, running\}$ in our video game example, we reach the test case $Q_2 = \{runningGame, startup, running\}$. We observe that class *startingGame* was removed from $Q_1$ and class *runningGame* was added to $Q_2$, but we also need to add class *startup* because *runningGame* is a refined class. A *test sequence* is a list of test cases in which all except the first one are obtained by applying a transition from the previous one. A sequence of length three for our running example could be composed of test cases ($Q_1 = \{startingGame, running\}$, $Q_2 = \{runningGame, startup, running\}$ and $Q_3 = \{runningGame, startup, paused\}$).

If two different transitions can be used to transit in a given state and they affect different sets of classes it is possible to group them and consider one single step transition with the joint effect of both. In our example the transitions $startingGame \rightarrow runningGame$ and $running \rightarrow paused$ affect different sets of states since they belong to sibling classifications. Then, we can compose the transitions and build the test sequence of length two ($Q_1$,$Q_3$). This sequence covers the same transitions as the sequence ($Q_1$, $Q_2$, $Q_3$).

Given a test sequence we define the *class coverage* as the number of classes appearing in the test cases of the sequence divided by the total number of classes in the ECTM. We define the *transition coverage* as the number of transitions covered by the sequence divided by the total number of transitions. The problem we are interested in solving consists in finding a set of test sequences such that the coverage (class or transition, one each time) is maximized. For a more precise and formal definition of the concepts presented in this section the reader should refer to Section 2.4.2.

### 8.3.1 Algorithms Details

In this section we describe three different approaches used to solve the TSGP. We first introduce an evolutionary approach, a Genetic Algorithm. Second, we describe our algorithmic proposal based on ACO. Finally, we briefly describe a state of the art technique from the literature for comparison purposes. We would like to highlight that the size of the test cases that compose a test sequence can vary from one to another. This fact is due to the hierarchical structure of the model. One class could be refined in several sub-classes, then the length of the test cases would be different. Consequently, we have to deal with the dynamic size of test cases in the ECTM.

#### Genetic Test Sequence Generator

The Genetic Test Sequence Generator (GTSG) constructs an entire test suite taking into account the dependencies between test data in the generation of the sequence. GTSG is an algorithm that evolves a population of solutions in each iteration until a given coverage criterion is fulfilled. The algorithm tries to find the tests that maximize the coverage, then it sequentially adds them to the solution (test sequence).

In this particular algorithm the representation of a solution *sol* (test sequence) is a vector of integers of length *l*. We determine the length of the chromosome as a parameter of the memory operator (see next subsection).

$$sol = [I_1, I_2, I_3, ..., I_l].$$

The outgoing transitions from a class of the current test case can be enumerated, thus each number ($I_i$) can be seen as the next transition chosen from the actual class to the next one.

The evaluation of a solution is done by sequentially taking every single transition (class to class) of the solution and generating a sequence of test data with a particular coverage. The evaluation function selects one leaf class (from left to right) and one gene in the solution is consumed to select the next transition $t_i$. Then, $t_i$ is added to the set of selected transitions, $T'$. In order to transit from one test case to another, the evaluation function consumes, at most, as many genes as the number of leaf classes present in the source test case. We may need to consume a variable number of integer numbers of the solution to transit to the next test case. It depends on the source test case. We use the following expression to select the next transition:

$$t_i = I_i \mod |Transitions(c)|. \tag{8.1}$$

where $I_i$ is the $i-$th component of the Solution and $Transitions(c)$ is the list of possible outgoing transitions from class $c$.

For example, if the evaluation function is considering class $c_i$ and that class has 4 outgoing transitions, we consume the next gene (integer), e.g. 6, in the solution to determine the next transition. In this example, we take the second possible transition ($t_i = 6 \mod 4 = 2$).

The fitness value of a solution is the class or transition coverage, Equations (2.12) and (2.13) in Section 2.4.2, obtained by the solution when all genes have been consumed in the evaluation. In this algorithm we wish to maximize the fitness function given by Equation (2.12) for Class Coverage and Equation (2.13) for Transition Coverage.

The recombination operator is not used because the exchange of genes between two individuals could generate sequences of meaningless transitions. Since we interpret each gene in the chromosome as the transition to take from among all those possible, the interpretation of each number depends on the previously consumed numbers. Let us explain this issue in detail.

Let us say we obtain, after using the single point crossover and from the initial individuals $I_1 = \{1, 1, 1, 2, 1, 1\}$ and $I_2 = \{1, 2, 1, 2, 1, 2\}$, two mixed individuals $I_{1'} = \{1, 1, 1, 2, 1, 2\}$ and $I_{2'} = \{1, 2, 1, 2, 1, 1\}$. Since the interpretation of each number depends on the state where they are consumed, the number indicates which transition must be taken from the source state. When we evaluate $I_{1'}$, the first part of the individual is properly optimized. However, when the second part of the individual is being consumed, the actual test case (source state) is not the same as in the $I_2$ individual, because we have taken other transitions regarding to the first part of the individual. If we focus on the position 2 of the vector, in the $I_2$ individual it indicates to take the second transition, but in $I_1$ individual it indicates to take the first one. Consequently, now the rest of the transitions of the second part of the individual $I_{1'}$ are not properly optimized so as to cover all states and transitions, because the source class of the next transition is different.

Regarding the mutation operator, it iterates over all the components in the solution vector uniformly changing their value by $\pm 1$. It linearly increases the probability to mutate a component in order to give a low probability to the first components of the chromosome, and a larger probability to the genes at the end of the chromosome. We aim to maintain the first part of the individual with fewer changes because a change in a gene could affect the rest of the sequence. We increase the probability from $p_{m1}$ to $p_{m2}$. So here, $p_{m1} = 0.05$ and $p_{m2} = 0.25$.

In this particular implementation, the best individuals of the population are kept for the next generation using a memory operator. As the population evolves, the first transitions in the individual tend to stabilize, but the algorithm still has to evaluate them at each generation. We propose saving the resulting stable first transitions in a memory slot to use them as the starting point for following optimization steps. We use the memory operator (MemO) to allow the algorithm to search in stages. This operator was first proposed by Alba et al. [4] in the context of software verification. The algorithm can optimize the whole sequence of numbers (transitions) in stages, step by step, at the same time saving the memory required to evaluate complete individuals. Instead, we only have to evaluate a shorter sequence in each individual evaluation. This operator is based on the so-called missionary technique used in [3] for reaching deep graph regions in an ACO.

We use the memory operator as follows: the GTSG is executed using a relatively small chromosome length (in this approach we use a chromosome length of 20 integers). After a predetermined number of evaluations (100, 000), the memory operator selects the best individual and stores its transitions to use them as the starting points for the next optimization steps. The MemO could store more than one individual as the starting point for the next generation, but in accordance with previous experimentation performed in the early stages of this study and the small population we used, the best choice is to select only the best individual. All the other transitions are removed from the memory.

The internal loop of the algorithm is executed until a maximum number of evaluations is reached. Then, the best individual (partial sequence) found is added to the test suite list and the *Coverage* set is updated by removing the classes or transitions which are going to be covered by the new best partial sequence. Then, the memory operator stores the last test case of the best sequence to be the initial test case for the next generation. Finally, the external loop starts again with a new population until there is no class or transition left in the *Coverage* set or the algorithm reach a predefined number of evaluations.

The advantages of using the memory operator are obvious: less memory and time are required to evaluate an individual, and thus the path can maintain a constant growth without requiring more time and memory. There are, of course, disadvantages. In particular, part of the search space is discarded and that part might in fact contain a good solution, but this is common in any non-exhaustive search algorithm.



### ACO Test Sequence

Our *ACO Test Sequence* (ACOts) algorithm is an adaptation of the ACOhg algorithm proposed by Alba and Chicano [3] that can deal with the construction of huge graphs of unknown size. This new model was proposed for applying an ACO-like algorithm to the problem of searching for counterexamples of safety properties in very large concurrent models. We have adapted the algorithm with the intention of solving the TSGP.

In short, two main differences between ACOts and the original ACO [65] model are as follows. First, the traditional ACO searches for the shortest path from an initial set of nodes to the objective ones. Since our objective in TSGP is to cover all classes or transitions, so we are also interested in visiting all classes and using all possible transitions between the first test case and the final test case. Second, ACOts cannot define final classes or test cases, the algorithm adds new test cases until the coverage criterion is fulfilled.

In this particular implementation, the heuristic function $\eta$ depends on each arc of the construction graph and is defined in the context of ACO algorithms. It is a non-negative function used by ACO algorithms for guiding the search. The higher the value of $\eta_{ij}$, the higher the probability of selecting arc $(i, j)$ during the construction phase of the ants. We use the same heuristic rate algorithm based on coverage of the greedy deterministic algorithm (Section 8.3.1) that will be presented in the following section.

Another particularity of our ACO-based approach is that the algorithm applies a compact function in order to minimize the steps of the best solution ($a^{best}$), resulting in the minimum number of different test cases. The compaction is as follows: since we only apply single transitions between classes, we can apply several transitions at the same time provided that the source class of the transitions is not the same or it is not an ascendant or descendant of the source class of any already selected transition. Then, we compact some of the single transitions in a complete transition that save some test cases in the resulting test suite. Continuing with the Video Game example shown in Figure 2.2, if the actual test case is $Q_1 = \{controlling, running\}$, the following selected transitions are $controlling \rightarrow gameOver$ and $running \rightarrow paused$. Then, we can compact the two transitions into a complete transition to obtain directly $Q_2 = \{gameOver, paused\}$ in only one test step.

### Greedy Deterministic Approach

This approach was first introduced in [126], it will be used here for validation of our results. This algorithm uses a multi-agent system with two kinds of agents to traverse the classification tree: the walker agent and the coverage agent. Both agents will cooperatively traverse the ECTM. Travelling is done in such a way that only valid paths are taken and that all traversed paths together result in the desired coverage. A full description of the algorithm has been given in [126], so we will only outline it here.

For any classification in the classification tree, a walker agent is introduced at the initial class. The initial test case is interpreted as a test step and taken into account for coverage calculation (e.g. class coverage, transition coverage). All walker agents are then moved one after another. The path of movement is calculated by coverage agents. When all agents have been considered once, the actual position of all agents is again interpreted as a test step, and is taken into account for coverage calculation, then added to the resulting test sequence. This is repeated until the desired coverage level has been reached. When there are no more valid paths to take, walker agents are stuck. In this case, the whole ECTM is reset to its initial state and a new, additional sequence, is created. When the algorithm has finished a test suite with all test steps is returned.



**Algorithm 8** Pseudocode of the Heuristic Rate algorithm.

```
1: proc Input:candidate (Class or Transition)
2: if classcoverage && selfTransition then
3:     return 0
4: end if
5: weight=1.0 ; rating=0
6: queue ← φ
7: queue += (candidate, weight)
8: while !queue.empty() do
9:     (item, weight)= queue.poll()
10:    if (item==candidate && rating> 0) then
11:        rating+=100 ; continue
12:    end if
13:    if (ratedItems contains item) then
14:        continue
15:    end if
16:    ratedItems+=item
17:    if (targetNodes contains item) then
18:        rating+=10*weight
19:    end if
20:    if (item has (outgoing transitions || childnodes || classifications)) then
21:        weight= weight*0.95;
22:    end if
23:    for all (item has (outgoing transitions && childnodes && classifications) of item) do
24:        queue+=(item, weight)
25:    end for
26: end while
27: return rating
28: end_proc
```

The Heuristic Rate algorithm is run by the aforementioned coverage agent. This agent guides the main algorithm to achieve full class and/or transition coverage. The Heuristic Rate algorithm is outlined in Algorithm 8. Its main goal is to rate the candidate transitions or classes in order to decide which is going to add more coverage to the current solution. The heuristic algorithm gets a candidate class or transition as input. For class coverage, self transitions are ignored and then zero is returned. A self-transition does not increase class coverage because the origin and the end of the transition is the same class. Otherwise, it then adds this candidate to a queue together with a weight factor, with an initial weight factor of one. A weight factor is needed to give more weight to the closest uncovered classes than those farthest away. The initial rating is set to zero. The candidate is added to the list of rated items. Then, while the queue is not empty the algorithm polls (FIFO) the next class and weight factor from the queue. If the polled node is the original candidate and if the rating is larger than zero, the algorithm has found a loop path with new items. This loop path is weighed by adding the value of 100 to the rating because we found a promising candidate. In this case or when the current item is on the list of rated items, the while loop passes to its next cycle. Otherwise this node is added to the list of rated items. If the node is on the list of target classes (it has not been used in any test step before), the algorithm adds 10 times the weight factor to the result rating. Then, if there are outgoing transitions, child nodes or classifications, the weight factor is multiplied with a punishment value. Target classes of outgoing transitions and child classes are then added to the queue together with the new weight factor. When the queue is empty the rating is returned.

### 8.3.2 Experimental Setup

ACOts and GTSG are non-deterministic algorithms, so we performed 30 independent runs per program/coverage criterion for a meaningful statistical analysis. All the executions were run in a

cluster of 16 machines with Intel Core2 Quad processors Q9400 (4 cores per processor) at 2.66 GHz and 4 GB memory running Ubuntu 12.04.1 LTS and managed by the HT Condor 7.8.4 cluster manager. In order to check whether the differences between the algorithms are statistically significant or just a matter of chance, we applied the Wilcoxon rank-sum [189] test and highlight in the tables, the differences that are statistically significant. We set a confidence level of 99.9% ($p$-value under 0.001) for the entire comparison (both metaheuristics acting on a program/coverage).

We have marked a result in dark grey when it is the best and in light grey when it is the second best in performance. When the result of one algorithm is significantly better than the result of another algorithm (typically the one whose results is farthest), we have added an asterisk. Two asterisks are added if the algorithm is significantly better than the other two algorithms. In addition, with the aim of properly interpreting the results of statistical tests, it is always advisable to report effect size measures. For that purpose, we have also used the non-parametric effect size measure $\hat{A}_{12}$ statistic proposed by Vargha and Delaney [208]. Effect size provides information about the magnitude of an effect, which can be useful in determining whether it is of practical significance or not.

For the experiments we use a benchmark with 12 different models of programs/artifacts listed in Table 8.7[3]. We use a Keyboard instance [156], a Microwave [137], an Autoradio [101], and Harel's Citizen watch [94] which is relevant in the literature. From the IBM Rhapsody instances, we took the Coffee Machine, the Communication example, the Elevator, and the Tetris game [108]. In Matlab Simulink Stateflow, we found Mealy Moore, Fuel Control, Transmission, and Aircraft [143]. In this table the second and third columns list some statistics of the resulting artifacts. Both the number of classes and number of transitions are given. The fourth and fifth column list the results for conventional test case generation computed by the CTE tool with the greedy algorithm for minimal and complete combination. Numbers indicate the size of the generated test suite. Even though the details of the case studies are given in Table 8.7, we highlight here that most instances are hierarchical and concurrent. This means that we are going to deal with test cases of different lengths. In other words, there are test cases of different lengths in the same sequence.

Let us explain the notation used in the Tables 8.9 and 8.11. A single number $n$ indicates the size of the unique test sequence: it is the number of generated test steps $n$ needed for 100% coverage. In the case of the metaheuristic algorithms, we provide the mean over the 30 executions. A number $n$ followed by a percentage value ($p$%) indicates the number of generated test steps $n$ together with a coverage level $p$% below 100%. When the number $n$ is followed by another number ($m$), the first number $n$ indicates the total number of test steps while the second number $m$ in parentheses indicates the number of sequences needed. We have implemented a re-boot mechanism in all the algorithms in case they reach a class with no exit transition.

### Parameter Settings

A possible threat to internal validity is that we have experimented with only one set of algorithms' parameters. Nevertheless, we have performed a previous experiment in order to select the best parameters for the GTSG and ACOts algorithms. We have tried all combinations of values shown in Table 8.8. Note that the parameters used in the final experimentation are the ones highlighted in bold.

[3]They are available at http://neo.lcc.uma.es/staff/javi/resources.html

Table 8.7: General characteristics of the benchmark of programs.

| Name | Classes | Transitions | Minimal | Complete |
|---|---|---|---|---|
| Keyboard [156] | 5 | 8 | 2 | 4 |
| Microwave [137] | 19 | 23 | 7 | 56 |
| Autoradio [101] | 20 | 35 | 11 | 66 |
| Citizen [94] | 62 | 74 | 31 | 3121 |
| Coffee Machine | 21 | 28 | 9 | 81 |
| Communication | 10 | 12 | 7 | 7 |
| Elevator | 13 | 18 | 5 | 80 |
| Tetris | 11 | 18 | 10 | 10 |
| Mealy Moore | 5 | 11 | 5 | 5 |
| Fuel Control | 5 | 27 | 5 | 600 |
| Transmission | 7 | 12 | 4 | 12 |
| Aircraft | 24 | 20 | 5 | 625 |

Table 8.8: Parameters setting for our proposals. The parameter's values used in the experimentation are highlighted in bold.

| **Algorithm** | **Parameter** | **Value** |
|---|---|---|
| GTSG | Population Size | **4**, 8, 10 |
| | Crossover | **No**, Yes (1.0 , 0.9, 0.8) |
| | Mutation Prob. | 0.05, 0.1, 0.2, **Dynamic (0.05-0.25)** |
| | Memory Operator | No , **Yes** |
| | Memory Slots | **1**, 2, 5 |
| | Chromosome length | 10, **20**, 50, 100 |
| ACOts | $\alpha$ | **1**, 2, 5 |
| | $\beta$ | 1, **2**, 5 |
| | $\rho$ | 0.1, **0.5**, 0.9 |
| | $maxsteps$ | 10, 20, 50, **100** |
| | $colSize$ | 2, 5, **10** |
| | $\lambda_{ant}$ | **400** |

### 8.3.3 Test Sequences' Quality

In this section, we analyze the behavior of the proposed approaches with the aim of analyzing the computed best solutions and highlighting the algorithm that behaves the best. The main results of the executions of the algorithms for class coverage and transition coverage can be seen in Table 8.9 and Table 8.11, respectively.

For *class coverage*, 100% coverage was reached for 11 out of 12 programs. Achieving full coverage is the main objective for test case generation. The Aircraft program was the only one resulting in below 100%, having an 86.2% coverage, as there are unreachable or orphaned classes in the model. In all 12 programs, the highest possible class coverage was reached in a single test sequence, which is a desirable result. Regarding class coverage, differences appear in four programs (Microwave, Autoradio, Citizen, and Tetris). The greedy approach obtains better results

Table 8.9: Results for test sequence generation for class coverage.

| Name | GTSG | ACOts | Greedy |
|---|---|---|---|
| Keyboard [156] | 2 | 2 | 2 |
| Microwave [137] | 8* | 8* | 9 |
| Autoradio [101] | 13.30* | 14 | 13* |
| Citizen [94] | 39.47* | 36** | 47 |
| Coffee Machine | 9 | 9 | 9 |
| Communication | 7 | 7 | 7 |
| Elevator | 6 | 6 | 6 |
| Tetris | 12* | 12* | 15 |
| Mealy Moore | 5 | 5 | 5 |
| Fuel Control | 5 | 5 | 5 |
| Transmission | 4 | 4 | 4 |
| Aircraft | 4 *(86.2%)* | 4 *(86.2%)* | 4 *(86.2%)* |

in the Autoradio program, where the difference with GTSG is not significant. For the other three programs the metaheuristic algorithms achieve total coverage using fewer test steps. For instance, both metaheuristic algorithms reduced the test suite size by more than 20% for the Tetris program.

Let us analyze the Citizen program for class coverage. The analysis of this program is especially interesting because this is the most complex program. Furthermore, the differences between the algorithms are the largest. ACOts obtains the best results in this program. ACOts reduces the test suite size by more than 23% with respect to the Greedy algorithm, moreover it is 9% better than GTSG. In addition, GTSG is 15% better than the Greedy algorithm. The ACOts approach is more effective and accurate for the largest model used in this study. Regarding **RQ1**, it is true that the evolutionary algorithms save costs because they generate shorter test sequences than the greedy algorithm.

Table 8.10: Vargha and Delaney's statistical test results ($\hat{A}_{12}$) for class coverage. $A$ represents algorithms in rows and $B$ represents algorithms in columns.

| | GTSG | ACOts | Greedy |
|---|---|---|---|
| GTSG | - | 0.5139 | 0.3889 |
| ACOts | 0.4861 | - | 0.4167 |
| Greedy | 0.6111 | 0.5833 | - |

In light of these results and with the intention of determining whether the results are of practical significance or not, we analyze the $\hat{A}_{12}$ statistic. In Table 8.10 we summarize the average of the $\hat{A}_{12}$ statistic values for class coverage and all programs. The differences between algorithms are not very large due to we have selected small, medium, and large programs. Consequently, it is very difficult to obtain large differences in small and medium models. Numerically, the results of the ACOts are going to be better than the ones provided by GTSG and Greedy in 51.39% and 58.33% times, respectively. In addition, the results of GTSG are going to be better than the Greedy ones in 61.11% times, which is a big difference.

For *transition coverage* (Table 8.11), only ACOts is able to obtain 100% coverage in all the programs. The other two algorithms fail to obtain total coverage in the one program (Citizen). In 11 of the 12 programs, the result only consisted of one test sequence, while in the Aircraft program two sequences were generated. The differences appear in five programs (Autoradio, Citizen, Coffee,

Table 8.11: Results for test sequence generation for transition coverage

| Name | GTSG | ACOts | Greedy |
|---|---|---|---|
| Keyboard [156] | 5 | 5 | 5 |
| Microwave [137] | 17 | 17 | 17 |
| Autoradio [101] | 36.30 | 36 | 36 |
| Citizen [94] | 75.27* *(99.9%)* | 64.17** | 51 *(92.7%)* |
| Coffee Machine | 19 | 19 | 18** |
| Communication | 16* | 16* | 17 |
| Elevator | 9 | 9 | 9 |
| Tetris | 31 | 31 | 31 |
| Mealy Moore | 24 | 24 | 24 |
| Fuel Control | 11* | 11* | 12 |
| Transmission | 9 | 9 | 9 |
| Aircraft | 7 (2) | 7 (2) | 7 (2) |

Communication, and Fuel Control). In this case the Greedy algorithm is only better than the others in the *Coffee Machine* program, the Greedy algorithm reduces the test suite size, in this program, in one test case compared to the metaheuristic approaches. The existing differences are low in most cases except in the Citizen program where ACOts is clearly the best. It is the only algorithm that always achieves 100% transition coverage for all the programs. In the Citizen program the Greedy algorithm does not achieve full transition coverage while GTSG obtains total coverage in most executions. ACOts is better than GTSG in coverage and test suite size. ACOts is able to reduce the test suite size by 14.7% (with respect to GTSG).

Table 8.12: Vargha and Delaney's statistical test results ($\hat{A}_{12}$) for transition coverage. $A$ represents algorithms in rows and $B$ represents algorithms in columns.

| | GTSG | ACOts | Greedy |
|---|---|---|---|
| GTSG | - | 0.5125 | 0.4670 |
| ACOts | 0.4875 | - | 0.4545 |
| Greedy | 0.5329 | 0.5455 | - |

Table 8.12 shows the $\hat{A}_{12}$ statistical results for measuring the effect size for transition coverage. We have considered all programs, with the exception of the Citizen program where the results are not comparable. This fact is because neither GTSG nor the Greedy algorithm are able to reach full transition coverage, so the test suite is shorter but it is quite worse in quality (coverage). Although we have not included the Citizen results, where the ACOts algorithm is clearly superior, ACOts is still better than GTSG and Greedy by 51.25% and 54.55%, respectively. Furthermore, GTSG obtains smaller test suites than Greedy by 53.29%. Regarding the solution quality (coverage level), the metaheuristic approaches (ACOts and GTSG) seem to be competitive. They are both capable of generating test sequences with maximum levels of coverage, and obtain better results than the Greedy algorithm with a high probability.

### 8.3.4 Test Suite Coverage versus Test Suite Size

Another aspect that we must take into account is the increase in the test suite size with the coverage in order to obtain total coverage. This behavior requires a further analysis to evaluate the tradeoff

between coverage and test suite size because this is a key aspect when you are generating test suites [159]. We illustrate this tradeoff for the Citizen program in Figure 8.2 and 8.3 for class and transition coverage. In the figures, we show the deterministic solution of the Greedy algorithm and the median and interquartile range of the 30 executions of the metaheuristic algorithms in order to capture the average behavior of the approaches. We would like to stress that this analysis is performed on the solutions, already computed.

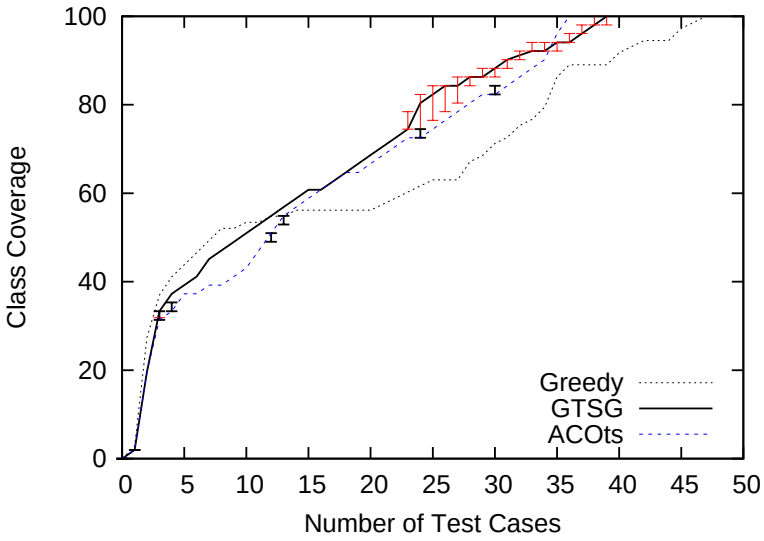



Figure 8.2: Median solutions and interquartile range of ACOts, GTSG and Greedy algorithms for the Citizen example for Class Coverage. Coverage versus number of test cases in the solution.

Let us start with the analysis of the solution where we want to cover all the classes (class coverage) of the Citizen program. In Figure 8.2 we show that the obtained coverage is similar for the first test steps. The Greedy algorithm is slightly better with up to 54% coverage. Then, both metaheuristic algorithms continue adding coverage with the same ratio in contrast with the Greedy algorithm, which is worse in the middle stage of the sequence. GTSG obtains its maximum advantage when it achieves 80% coverage, while ACOts only achieves 72% with the same test steps (24). When only a few classes remain unvisited, ACOts is able to visit them in fewer test steps. Thus, it achieves full class coverage in only 36 test cases, three test cases less than GTSG and 11 test cases less than the Greedy algorithm. ACOts obtains total coverage with only 64 test cases in median, meanwhile GTSG has achieved 94.12% and the Greedy algorithm has achieved only 89.04% coverage with the same number of test cases.

In certain regions of the graph (Figure 8.2) we observe that the same coverage is repeated in consecutive test steps. The reason is that not every class can be reached from any other class, but requires additional traversal of other covered classes and, therefore, additional test steps. We see this behavior, in particular, in the solution of the Greedy algorithm for the class coverage. This implies that we are not adding any coverage in these traversal test steps, so our algorithm should minimize them.

In Figure 8.3 we show the median transition coverage and the interquartile range of the proposed

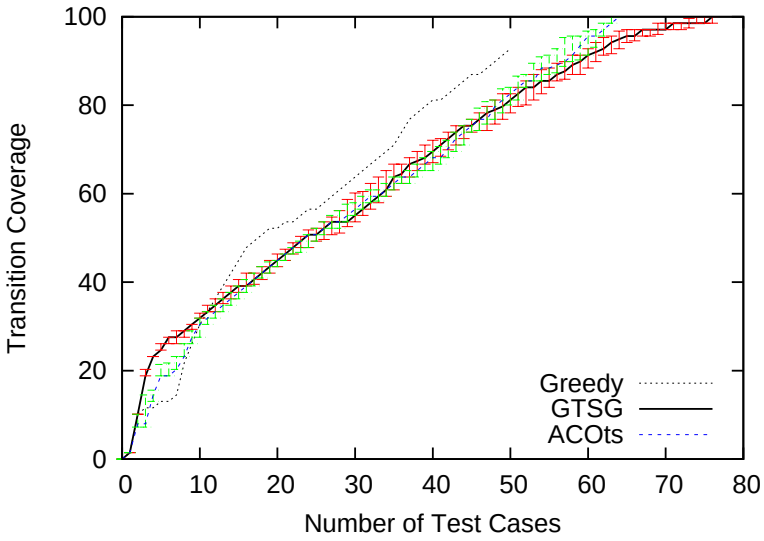



Figure 8.3: Median solutions and interquartile range of ACOts, GTSG and Greedy algorithms for the Citizen example for Transition Coverage. Coverage versus number of test cases in the solution.

algorithms achieved with each test case of all test sequences (average of 30 executions of non-deterministic algorithms). In this case, GTSG is better at the beginning because it first explores an area with a higher density of transitions (i.e. the algorithm does not have to visit an already visited node to reach a non-visited node). Besides, the Greedy algorithm obtains better coverage using the same number of test cases from 12 test cases onwards, but it is not able to achieve more than 92.7% coverage. Although the Greedy algorithm has achieved 11.59% more coverage than GTSG and 10.14% more than ACOts with 51 test cases, both metaheuristic algorithms are able to reach full coverage. In this case, ACOts is better because it achieves full transition coverage in fewer test cases and it adds coverage in each test case, progressively. This great effort in reducing traversal test steps makes the algorithm reasonably predictable. This behavior is desirable because the obtained coverage is proportional to the test cases needed to reach certain levels of coverage.

For the goal of test suite minimization we have tried to optimize the test suite sizes while still achieving high levels of coverage. We have used GTSG and ACOts to search for the optimal solution but we need to evaluate the minimal test suite size using an exact approach, allowing us to know if we have reach the optimal one. Regarding computation times, we can say that the generation times for GTSG is less than 10 minutes in average, for ACOts is less than a minute in average, while the deterministic algorithm takes around 10 seconds in average. If we take into account the performance and the quality of the obtained results, it seems that ACOts is the best option, at least for the largest instances.

## 8.4 Conclusions

In this chapter we have studied the prioritized pairwise test data generation problem with the aim of analyzing the performance of several approaches. We have compared five different approaches, three

of them proposed by us. We have performed some experiments on 32 different scenario/distribution combinations and for different values of weight coverage, which makes our study meaningful. The genetic algorithm outperforms the other algorithms in most scenarios and distributions, it is the best choice when one has some time restrictions or the execution of a test case is quite costly.

After analyzing the results obtained by all the algorithms we can draw some advices about which technique should be used. If the results of a particular technique like PPS are good for equal weight distribution (*d1*) but are not good enough for the other distributions, then the technique is designed to be used without priorities. If one really needs some values of weight coverage for different scale scenarios and non-uniform distributions, the genetic solver is the best choice. But, if the GS does not achieve a satisfactory result for a particular configuration, one should use the PPC algorithm. Finally, if the test suites have already been computed, the PPS algorithm should be used in order to give a better ordering of the test data.

Our three approaches for the PPTDGP have been successfully implemented in the CTE XL tool using CTM, but for the generation of test sequences we needed to define an extension of the CTM. We have defined an entire model (ECTM) which both industry and academia could use to completely describe all aspects needed to generate sequences of tests for testing a program. Its benefits are clear, we can save costs and time executing all test steps sequentially because the previous test step puts the software in the adequate state to test the next functionality.

We have presented two different metaheuristic approaches to optimize the automatic generation of test sequences for the CTM. The first is a genetic algorithm with memory operator (GTSG), which is able to preserve the memory required to evaluate individuals, while also allowing the algorithm to compute a solution faster than without the operator. The second is an ACO algorithm, concretely, we propose ACOts, a variation of the ACOhg implementation that is able to obtain good quality solutions, using little memory.

We have also compared our results with the ones of an existing greedy deterministic algorithm. We have used the algorithms to find test sequences for 12 different programs extracted from the literature. After analyzing the solutions obtained by the three approaches, we can conclude that the metaheuristic approaches are significantly better than the greedy deterministic approach for the largest model of program, specially the ACOts algorithm. The Greedy algorithm is only better than GTSG and ACOts in, respectively, 1 and 2 out of 8 scenarios where statistical differences exist. GSTG is statistically better than the Greedy algorithm in 4 out of 8 scenarios. Finally, ACOts is better than the Greedy algorithm in 6 out of 8 scenarios where statistical differences exist. Therefore ACOts is the best algorithm in the comparison. It has a good tradeoff between test suite size and coverage.



# Chapter 9

# Pairwise Testing in Software Product Lines

## 9.1 Introduction

Software Product Lines are families of related software products, where each product provides a unique combination of *features* – increments in program functionality [229]. Some of the proven benefits of SPLs are increased software reuse, faster product customization, and reduced time to market [175]. *Feature Models (FMs)* are the *de facto* standard to represent all the valid feature combinations of an SPL [115]. The typically large number of feature combinations in a SPL poses a unique set of challenges for software testing because testing each individual product may not be technically or economically feasible.

Recent surveys and mapping studies on SPL testing [56, 69], attest the increasing relevance of the topic within the research and industrial SPL communities. Salient among the existing approaches are those based on CIT, whose premise is to select a group of products where faults due to feature interactions are more likely to occur [163]. In the SPL context, most of the focus has been on *pairwise* testing, that is, on the four possible combinations between any two features[1]. The pairwise feature combinations of a product determine its *coverage*. Thus, pairwise SPL testing aims to select a set of products, referred to as *test suite*, such that their combined coverage contains *all* the possible pairwise feature combinations of the SPL.

Because of the large number of feature combinations in typical SPLs, variability modelling poses a unique set of challenging problems for software testing. In recent years many verification and testing SPL approaches, which rely on different techniques, have been proposed (e.g. [46,111]). However, and despite the extensive and successful use of evolutionary computation techniques for software testing [99, 148], their potential application to SPL testing remains largely unexplored, in particular regarding test prioritization. For this reason, first, we present a parallel genetic algorithm for the generation of prioritized pairwise testing suites for SPLs. The experimentation carried out will help us with our first research question (**RQ1**) in this chapter: Is our evolutionary proposal competitive in the computation of prioritized test suite in SPL?

Many SPL pairwise testing approaches have been proposed (e.g. [70, 82, 102, 104, 110, 166, 173, 225]) that tackle SPL pairwise testing as an optimization problem where either coverage (maximize)

[1] For `A` and `B` features: both selected, both not selected, `A` selected and `B` not, `A` not selected and `B` selected.

or test suite size (minimize) are considered the main optimization objective. While this single-objective perspective might be enough for certain limited contexts, it does not capture the more prevailing scenario where, for instance, it might not be feasible or desirable to test all the products of a test suite, or for example when both coverage and test suite size are of equal importance. Thus, the multi-objective perspective enables software developers to analyse the trade-offs between their objectives (sometimes conflicting) and select the test suites that best match their technical and economical constraints. Unfortunately, those pieces of work that have explored a multi-objective perspective in the realm of SPL testing (i.e. [103] and [212]) do use a scalarization technique that flattens all objectives into a single objective by assigning them weights; an approach with proven limitations [142]. Therefore, a second contribution is the application to SPL pairwise testing of four classical multi-objective evolutionary algorithms. This fact leads us to the second research question **RQ2**: What is the best algorithm among these four for multi-objective SPL pairwise testing? Additionally, we want to explore the impact of seeding techniques in MO techniques and we will answer this research question (**RQ3**) in this chapter.

Finally, in the same sense, we present a zero-one mathematical linear program for solving the multi-objective problem and an algorithm that computes the true Pareto front of a feature model using SAT solvers. The computed fronts are the optimal solution for both objectives. However, we want to measure the scalability of our exact approach. This leads us to the last reserach question of this chapter (**RQ4**). Does the execution time grow with the number of features of the *FM* or with the number of products denoted by the *FM*?

## 9.2 Parallel Prioritized Pairwise Testing

In this study we present the *Parallel Prioritized product line Genetic Solver* (PPGS), a genetic algorithm for the generation of prioritized pairwise testing suites for SPLs. PPGS receives as input a *feature model* that denotes a set of valid feature combinations and computes a set of products that covers the desired pairs of feature combinations according to a priority scheme that assigns different priority weights to a set of products. This scheme has been sketched in [111] and it is currently successfully applied in an industrial setting. We performed a comprehensive evaluation of PPGS with 235 feature models with a wide range of number of features and number of products, using 3 different weight priority assignment methods and 5 product prioritization selection strategies. In addition, we compared PPGS with prioritized-ICPL [111], an alternative greedy algorithm implementation. For the statistical comparison analysis both algorithms were executed 30 times for each feature model and combinations of priority assignment and product prioritization which yielded a total of 79,800 independent runs that required about two weeks of computation on a 64-core dedicated cluster. Our study revealed that overall PPGS obtains smaller covering arrays with an acceptable performance difference with prioritized-ICPL. However, the performance difference tends to decrease as the number of products of the feature models increase. We believe these results shed light on the potential benefits that evolutionary algorithms and other search based techniques can bring for variability modeling problems such as testing.

### 9.2.1 Algorithm Description

The Parallel Prioritized product line Genetic Solver is a novel constructive genetic algorithm which follows the master-slave model to parallelize the evaluation of the individuals [7]. It computes a test suite (a list of products to test) taking into account priorities during the generation. In each iteration, PPGS adds one new product to a partial solution until all pairwise combinations are

covered. It uses as inputs a feature model *FM* and the set of prioritized products for the test suite generation. At the beginning, the test suite is initialized with an empty list and the set of remaining pairs is initialized with the weighted pairwise configurations present in at least one of the given prioritized products. PPGS represents a product by only the list of selected features, so the operators only affect the selected features. If a generated offspring individual is not a valid product (i.e., it violates any constraint derived from the feature model), it is transformed into a valid product by applying a *Fix* operation provided by the FAMA tool [203].

Since the evaluation is performed in parallel in this algorithm, the fixed individuals are stored in a structure for later evaluation of them. After we leave the inner loop, the evaluation is performed in parallel, and finally the individuals are inserted in the offspring population $Q$. The fitness value of an offspring individual is the sum of the weights of the weighted pairwise configurations that would remain to be covered after adding the offspring solution to the test suite. Thus, this fitness function must be minimized in order to first select the product that covers weighted pairwise configurations with higher weights. Notice that, as the search progresses, the cost of computing the fitness function is reduced because every time less weighted pairwise configurations remain uncovered. The internal loop is executed until a maximum of 1,000 evaluations is reached. Then, the best individual (product) found is included in the test suite and *RP* is updated by removing the weighted pairwise configurations covered by the selected best solution. Then, the external loop starts again until there is no weighted pair left in the *RP* set.

We set the configuration parameters of PPGS with values frequently observed in the literature for genetic algorithms: crossover strategy single point with a probability of 0.8, selection strategy binary tournament, population size of 10 individuals, mutation that iterates over all selected features of an individual and replaces a feature by another randomly chosen feature with a probability of 0.1, and termination condition of 1,000 fitness evaluations and full weight coverage in the external loop.

### 9.2.2 Weight Priority Assignment Methods

We considered three methods to assign weight values to prioritized products: *measured values*, *ranked based values*, and *random values*.

#### Measured values

Measured values weights were derived from non-functional properties values obtained from 16 real SPL systems, from different problem domains and implemented using different technologies, that were measured with the SPL Conqueror approach [190]. This approach aims at providing reliable estimates of measurable non-functional properties such as performance, main memory consumption, and footprint. It works by performing a set of actual property measurements on different products (usually a proper subset of all the feature combinations denoted by a feature model) with different feature interaction types. The measured values are then used to compute the estimated property values for the feature combinations that were not measured. This choice of weight priority assignment allows us to emulate more realistic scenarios whereby software testers need to schedule their testing effort giving priority, for instance, to products or feature combinations that exhibit higher footprint or performance.

For our work, we use the actual values taken on the measured products considering pairwise feature interactions. Table 9.1 summarizes the SPL systems evaluated, their measured property (`Prop`), number of features (`NF`), number of products (`NP`), number of configurations measured (`NC`), and the percentage of prioritized products (`PP`%) used in our comparison as explained shortly.

Table 9.1: Summary of the case studies measured values.

| SPL Name | Prop | NF | NP | NC | PP% |
|---|---|---|---|---|---|
| Prevayler | F | 6 | 32 | 24 | 75.0 |
| LinkedList | F | 27 | 1,344 | 204 | 14.1 |
| ZipMe | F | 8 | 64 | 64 | 100.0 |
| PKJab | F | 12 | 72 | 72 | 100.0 |
| SensorNetwork | F | 27 | 16,704 | 3,240 | 19.4 |
| BerkeleyDBF | F | 9 | 256 | 256 | 100.0 |
| Violet | F | 101 | $\approx$ 1E20 | 101 | $\approx$ 0.0 |
| Linux subset | F | 25 | $\approx$ 3E24 | 100 | $\approx$ 0.0 |
| LLVM | M | 12 | 1,024 | 53 | 5.1 |
| Curl | M | 14 | 1,024 | 68 | 6.6 |
| x264 | M | 17 | 2,048 | 77 | 3.7 |
| Wget | M | 17 | 8,192 | 94 | 1.15 |
| BerkeleyDBM | M | 19 | 3,840 | 1,280 | 33.3 |
| SQLite | M | 40 | $\approx$ 5E7 | 418 | $\approx$ 0.0 |
| BerkeleyDBP | P | 27 | 1,440 | 180 | 12.50 |
| Apache | P | 10 | 256 | 192 | 75.0 |

**F**ootprint, **M**ain memory consumption, **P**erformance, **N**umber of **F**eatures, **N**umber of **P**roducts, **N**umber of **C**onfigurations, **P**ercentage of **P**rioritized products.

For this assignment method, all the measured products were used as our prioritization products. In three cases this meant including all the products of the product line. Please refer to Table 9.1 for further details.

#### Rank based values

For this second type of weight values, we selected the products to prioritize based on how dissimilar they are when compared to all other products of an SPL, and assigned them priority weights based on their rank values. The intuition behind this assignment choice is that by giving the same weight value to two of the most SPL-wide dissimilar products, the weight values will be more likely spread among a larger number of pairwise configurations making the covering array harder to compute. In addition, this enables us to select different percentages of the number of products for prioritization. For this assignment method,the selected percentages are: 5%, 10%, 20%, 30%, and 50%.

#### Random Values

For this type of weight values, we randomly generate weights between the minimum and maximum values obtained with the ranked based values approach. For this assignment method, a percentage of the products denoted by each individual feature model was used for product prioritization. The selected percentages are: 5%, 10%, 20%, 30%, and 50%.

### 9.2.3 Experimental Setup

This section describes how our evaluation was carried out. We start with the algorithm used to compare and contrast PPGS, followed by the experimental corpus of feature models, and the software and hardware infrastructure used. Throughout this chapter, in order to check if the differences between the algorithms are statistically significant or just a matter of chance, we applied the non-parametric Wilcoxon rank-sum test [189]. The confidence level used is 95% ($p$-value under 0.05). Statistical difference is measured with the Wilcoxon test. In order to properly interpret the results of statistical tests, it is always advisable to report effect size measures. For that purpose, we have also used the non-parametric effect size measure $\hat{A}_{12}$ statistic. These tests and our statistical procedure have been described in Section 3.3.2.

#### Prioritized-ICPL (pICPL) Algorithm

In our experimental comparison we employed prioritized-ICPL, a greedy algorithm to generate $n$-wise covering arrays developed by Johansen et al. [111]. Prioritized-ICPL does not compute covering arrays with full coverage but rather covers only those $n$-wise combinations among features that are present in at least one of the prioritized products, as was described in the formalization of the problem in Section 2.4.3. We must highlight here that the pICPL algorithm uses *data parallel execution*, supporting any number of processors. Their parallelism comes from simultaneous operations across large sets of data.

We should remark that an earlier and better well-known version of a greedy algorithm for SPL pairwise testing by the same authors (Johansen et al.) is also called ICPL [110]. However that version did not consider prioritization. To avoid any confusions and as a short hand notation, henceforth we will use the term pICPL to refer to prioritized-ICPL. For further details on prioritized-ICPL please refer to [111].

#### Experimental Corpus

We created three groups based on both the number of products denoted by the feature models and how their priority was assigned as shown in Table 9.2. Group G1 was formed with 160 feature models, whose number of products ranges from 16 to 1000 products, and that were evaluated with rank based and random values. Group G2 was formed with 59 feature models, whose number of products ranged from 1000 to 80000 that were also evaluated with rank based and random values. The threshold value to divide groups G1 and G2, and the selected percentages were chosen to provide an ample variety of number of products to prioritize. Group G3 was formed with 16 feature models, with number of products ranging from 16 to ≈3E24 that were evaluated with measured values.

We obtained 16 feature models from SPL Conqueror, 5 from Johansen et al. [111], and 201 from the SPLOT website [194] (a repository for the feature model analysis research community). Thus in total we employed 222 distinct feature models. Please notice that we also incorporated 5 SPL Conqueror feature models to G1 and 8 to G2. This yields a grand total of 235 feature models to analyze[2]. For G1 and G2 the problem instances are computed considering that for each feature model two priority assignment methods are used with three different prioritization selection percentages. For example, this yields for G1 $160 \times 2 \times 3 = 960$ instances. In total 1,330 problem instances were analyzed, with two algorithms PPGS and pICPL, with 30 independent executions. This means that the data of a total of 79,800 independent runs was generated and analyzed.

[2]They are available at http://neo.lcc.uma.es/staff/javi/resources.html

Table 9.2: Evaluation case studies summary.

| | G1 | G2 | G3 | Summary |
|---|---|---|---|---|
| **NFM** | 160 | 59 | 16 | 235 |
| **NP** | 16-1K | 1K-80K | 32- ≈3E24 | 16- ≈3E24 |
| **NF** | 10-56 | 14-67 | 6-101 | 6-101 |
| **WPA** | RK,RD | RK, RD | M | |
| **PP**% | 20,30,50 | 5,10,20 | ≈0.0 – 100 | |
| **PI** | 960 | 354 | 16 | 1330 |

**NFM**: Number Feature Models, **NP**: Number Products, **NF**: Number Features, **WPA**: Weight Priority Assignment, **RK**: Rank based, **RN**: Random, **M**: Measured, **PP**%: Prioritized Products Percentage, **PI**: Problem Instances

PPGS and pICPL are non-deterministic algorithms, that is why we performed 30 independent runs for a fair comparison between them. As performance measures we analyzed both the number of products required to test the SPL and the time required to run the algorithm. In both cases, the lower the value the better the performance, since we want a small number of products to test the SPL and we want the algorithm to be as fast as possible. All the executions were run in a cluster of 16 machines with Intel Core2 Quad processors Q9400 (4 cores per processor) at 2.66 GHz and 4 GB memory running Ubuntu 12.04.1 LTS and managed by the HT Condor 7.8.4 cluster manager. Since we have four cores available per processor, we have executed only one task per single processor, so we have used four parallel threads in each independent execution of the analyzed algorithms.

### 9.2.4 Experimental Analysis

In Table 9.3 we summarize the results obtained for group G1, feature models with up to 1,000 products. Each column corresponds to one algorithm and in the rows we show the number of products required to reach 50% up to 100% of total weighted coverage. The data shown in each cell is the mean and the standard deviation of the 30 independent runs. We highlight with a light gray background those values that are better with respect to the other algorithm with a statistically significant difference. We can observe that PPGS requires a smaller number of products to test the SPL with a significant difference when we consider a coverage level of 80% up to 99%. In the rest of the cases the differences are not statistically significant, so we cannot claim that one algorithm is better than the other. Regarding the time, pICPL is around 6 times faster than PPGS with a statistically significant difference. The time is given in milliseconds in the tables.

Table 9.4 shows the results for group G2: feature models with 1,000 to 80,000 products. We use the same legend and notation as for Table 9.3. In this case the advantage of PPGS over pICPL is larger than in the previous case. First, we can observe that PPGS is better than pICPL with statistically significant difference in all the coverage percentages except 100%. Regarding the computation time, PPGS is faster than pICPL but without statistically significant difference. From these results, the trend we can observe is that as the number of products of the SPL grows PPGS is still better in quality than pICPL while it is also better in runtime. Part of our future work is to verify if this trend holds for feature models with a larger number of products.

Let us now focus on group G3, feature models with measured weight values. Table 9.5 shows

Table 9.3: Mean and standard deviation of 30 independent runs for G1 (significant differences are highlighted).

| Cov. | PPGS | pICPL | Cov. | PPGS | pICPL |
|---|---|---|---|---|---|
| 50% | $1.20_{0.40}$ | $1.20_{0.40}$ | 96% | $4.00_{1.23}$ | $4.37_{1.42}$ |
| 75% | $1.92_{0.51}$ | $1.98_{0.58}$ | 97% | $4.38_{1.32}$ | $4.71_{1.54}$ |
| 80% | $2.15_{0.59}$ | $2.25_{0.68}$ | 98% | $4.83_{1.46}$ | $5.18_{1.74}$ |
| 85% | $2.47_{0.72}$ | $2.58_{0.81}$ | 99% | $5.58_{1.71}$ | $5.87_{1.99}$ |
| 90% | $2.88_{0.86}$ | $3.13_{1.03}$ | 100% | $7.56_{2.85}$ | $7.56_{3.03}$ |
| 95% | $3.72_{1.14}$ | $4.06_{1.33}$ | TIME | $23897_{28669}$ | $10116_{18842}$ |

Table 9.4: Mean and standard deviation of 30 independent runs for G2 (significant differences are highlighted).

| Cov. | PPGS | pICPL | Cov. | PPGS | pICPL |
|---|---|---|---|---|---|
| 50% | $1.16_{0.36}$ | $1.36_{0.83}$ | 96% | $4.98_{0.97}$ | $5.83_{3.14}$ |
| 75% | $2.09_{0.42}$ | $2.47_{1.65}$ | 97% | $5.55_{1.10}$ | $6.43_{3.27}$ |
| 80% | $2.39_{0.52}$ | $2.86_{1.79}$ | 98% | $6.34_{1.34}$ | $7.23_{3.48}$ |
| 85% | $2.73_{0.59}$ | $3.27_{2.08}$ | 99% | $7.66_{1.88}$ | $8.59_{4.11}$ |
| 90% | $3.36_{0.76}$ | $3.98_{2.38}$ | 100% | $14.57_{10.65}$ | $13.79_{9.98}$ |
| 95% | $4.59_{0.90}$ | $5.42_{3.12}$ | TIME | $273728_{7.2E+5}$ | $638164_{2.1E+6}$ |

the average number of products required to cover each SPL and the time for both pICPL and PPGS over the 16 models. According to the statistically significant differences the conclusions in this group of models are similar to the conclusions in the previous ones: PPGS is better in quality (lower number of products) while pICPL is faster. In detail, regarding the quality of the solutions, PPGS is better than pICPL in 68 model-coverage combinations with statistical significant difference, while pICPL is better than PPGS only in 19 model-coverage combinations. Regarding the time, pICPL is usually faster than PPGS with a statistically significant difference, with the only exception of the `SensorNetwork` model, in which they do not have statistically significant difference.

If we take a closer look at the data in the table and taking into account the statistical significant differences, we can observe that PPGS is overall better than pICPL in 8 out of the 16 models, namely: `Apache`, `BerkeleyDBF`, `BerkeleyDBP`, `LLVM`, `PkJab`, `SensorNetwork`, `Violet` and `Wget`. On the other hand, pICPL is better only for 2 models when all coverage percentages are considered: `Prevayler` and `ZipMe`. In the remaining 6 models pICPL is better for some percentages while PPGS is better for others.

As a general conclusion of this first analysis we can say that if the number of products to test is a critical aspect for the testing engineer, PPGS should be applied to generate these products instead of pICPL. The time required by PPGS is usually no longer than a few minutes, which is a reasonable time to generate a better quality test suite. We argue this is the most common scenario in software companies with SPLs, where carrying on each product test can require hours if not days to perform, specially if they involve complex software and hardware setups [111]. On the other hand, pICPL could be employed when the number of products to test is not a critical issue and a slightly faster generation of the test suite is preferable. Nevertheless, PPGS seems to be a competitive technique for dealing with this problem (**RQ1**).

In order to properly interpret the results of statistical tests, it is always advisable to report

Table 9.5: Group G3. When considering array sizes PPGS is statistically better than pICPL in 69 cases, and pICPL is better in 18 cases.

| Model | Alg. | 50% | 75% | 80% | 85% | 90% | 95% | 96% | 97% | 98% | 99% | 100% | TIME |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Apache | PPGS | 2 | 3 | 3 | 4 | 4 | 6 | 6 | 6 | 7 | 7 | 7 | 10394 |
| | pICPL | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 7 | 7 | 8 | 8 | 7582 |
| Berk.DBF | PPGS | 2 | 4 | 4 | 5 | 5.97 | 6.97 | 6.97 | 6.97 | 7.97 | 8 | 8.17 | 11213 |
| | pICPL | 2 | 4 | 5 | 6 | 7 | 8 | 8 | 8 | 8 | 9 | 9 | 8152 |
| Berk.DBM | PPGS | 2 | 3 | 3 | 4 | 4.73 | 6.87 | 7.80 | 8.77 | 9.97 | 11.90 | 23.33 | 117607 |
| | pICPL | 2 | 3 | 3 | 4 | 6 | 7 | 8 | 8 | 10 | 11 | 21 | 94512 |
| Berk.DBP | PPGS | 1 | 2 | 2 | 3 | 3 | 4 | 4.83 | 5 | 5.93 | 7 | 10.60 | 47361 |
| | pICPL | 1 | 2 | 3 | 3 | 4 | 6 | 6 | 6 | 6 | 7 | 12 | 57291 |
| Curl | PPGS | 2 | 3 | 3 | 3.97 | 4.03 | 5.83 | 6 | 6.50 | 7.37 | 8.07 | 9.63 | 17454 |
| | pICPL | 2 | 3 | 3 | 4 | 4 | 6 | 6 | 6 | 7 | 7 | 8 | 6382 |
| LinkedList | PPGS | 1 | 2 | 2 | 2 | 3 | 4.23 | 5 | 5 | 6.13 | 7.79 | 13.37 | 60684 |
| | pICPL | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 7 | 11 | 14 | 71151 |
| Linux | PPGS | 2 | 4 | 4 | 5 | 6 | 7 | 7.67 | 8 | 8.37 | 9.40 | 11.10 | 49385 |
| | pICPL | 2 | 4 | 5 | 5 | 6 | 8 | 8 | 8 | 8 | 9 | 10 | 30522 |
| LLVM | PPGS | 2 | 3 | 3.03 | 4 | 5 | 6 | 6 | 6.07 | 7 | 8 | 8.17 | 12805 |
| | pICPL | 2 | 3 | 4 | 4 | 5 | 6 | 7 | 7 | 7 | 8 | 8 | 9032 |
| PKJab | PPGS | 1 | 2 | 2 | 3 | 3.07 | 4 | 5 | 5 | 5 | 6 | 7 | 11439 |
| | pICPL | 1 | 2 | 3 | 3 | 3 | 5 | 5 | 6 | 7 | 8 | 8 | 4661 |
| Prevayler | PPGS | 2 | 3 | 3 | 3 | 4 | 5 | 5 | 5.60 | 6 | 6 | 6 | 8091 |
| | pICPL | 2 | 3 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 6 | 6 | 2412 |
| S.Network | PPGS | 1 | 3 | 3 | 3 | 4 | 5.03 | 5.47 | 6 | 6.97 | 7.87 | 13.97 | 71971 |
| | pICPL | 1 | 3 | 4 | 5 | 6 | 8 | 9 | 9 | 10 | 11 | 17 | 74181 |
| SQL.Mem | PPGS | 1 | 2.17 | 2.90 | 3.23 | 4.07 | 6.14 | 6.97 | 7.93 | 9.23 | 11.70 | 31.53 | 903118 |
| | pICPL | 1 | 3 | 4 | 4 | 5 | 8 | 8 | 9 | 11 | 14 | 28 | 407991 |
| Violet | PPGS | 1 | 1 | 1 | 2 | 2 | 2.93 | 3 | 3.07 | 3.30 | 4.53 | 12.83 | 31376054 |
| | pICPL | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 6 | 15 | 2471691 |
| Wget | PPGS | 2 | 2.13 | 3 | 3.07 | 4 | 5.43 | 6 | 6.40 | 7 | 8.03 | 11.37 | 31525 |
| | pICPL | 2 | 3 | 3 | 4 | 4 | 6 | 6 | 7 | 7 | 9 | 11 | 19612 |
| x264 | PPGS | 1.23 | 2.23 | 3 | 3.07 | 4 | 5.30 | 6 | 6.50 | 7.23 | 8.47 | 12.10 | 37368 |
| | pICPL | 1 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 7 | 9 | 13 | 13441 |
| ZipMe | PPGS | 2 | 3 | 3 | 4 | 5 | 6 | 6 | 7 | 7 | 7 | 7.03 | 13035 |
| | pICPL | 2 | 3 | 3 | 4 | 5 | 6 | 6 | 6 | 7 | 7 | 7 | 6142 |

effect size measures. For that purpose, we have also used the non-parametric effect size measure $\hat{A}_{12}$ statistic as recommended by Arcuri and Briand [17]. Table 9.6 shows the $\hat{A}_{12}$ statistic to assess the practical significance of the results. In this table a value lower than 0.5 means that PPGS is better than pICPL, a value greater than 0.5 means pICPL is better than PPGS and 0.5 means a draw. At a first glance, we can see that most of the times (31), PPGS obtains smaller test suites for all percentages of coverage, meanwhile pICPL is only better than PPGS twice. We have highlighted with dark and light gray background the lowest and highest values of the table (0.2497 and 0.5157). The lowest value indicates that PPGS obtains a better test suite than pICPL for 98% of coverage in a model of G2 in more than 75% of the cases. The highest value indicates that pICPL obtains a better test suite for 100% coverage in a model of G1 with a probability near 0.5. In general, this statistic reconfirms that PPGS gets better test suites than pICPL in terms of the number of products when priorities are considered.

## 9.3 Seeding Strategies for Multi-Objective Pairwise Testing

There exists a wealth of literature in the context of Evolutionary Multi-Objective Optimization [49] and the application of SBSE to software testing [230]. In this study we cast CIT SPL pairwise testing as a multi-objective optimization problem and use multi-objective classical algorithms. In this chapter we study the application to SPL pairwise testing of four classical multi-objective

Table 9.6: $\hat{A}_{12}$ statistical test results for all groups. PPGS yields better test suite size values.

| Group | 50% | 75% | 80% | 85% | 90% | 95% |
|---|---|---|---|---|---|---|
| G1 | 0.4985 | 0.4729 | 0.4511 | 0.4473 | 0.3785 | 0.3501 |
| G2 | 0.4529 | 0.4193 | 0.3760 | 0.3726 | 0.3436 | 0.2887 |
| G3 | 0.5104 | 0.4562 | 0.2844 | 0.3563 | 0.3198 | 0.3239 |

| Group | 96% | 97% | 98% | 99% | 100% |
|---|---|---|---|---|---|
| G1 | 0.3410 | 0.3703 | 0.3634 | 0.4000 | 0.5157 |
| G2 | 0.2847 | 0.2647 | 0.2497 | 0.2595 | 0.4945 |
| G3 | 0.3312 | 0.3135 | 0.3927 | 0.3068 | 0.4166 |

evolutionary algorithms: NSGA-II, MOCell, SPEA2, and PAES. These four algorithms have been extensively and successfully applied to a large number of problem domains and include a diverse set of techniques and concepts of multi-objective evolutionary algorithms. The experimentation with these four MO algorithms will allow us to answer **RQ2**.

Furthermore, we analyze the impact of *seeding*, defined as any technique that exploits previous related knowledge to help the testing problem at hand [76], on the performance of these algorithms. We developed three different seeding strategies that respectively exploit the knowledge of: *i)* the size of test suites, *ii)* greedily-generated tests suites, and *iii)* the actual test suites obtained using an existing single-objective pairwise testing approach. Hence, our second research question is stated as follows: How does seeding impact the quality of the solutions obtained by the four algorithms?(**RQ3**)

For this evaluation we selected 19 feature models, from diverse problem domains, from different provenance sources, and with different structural characteristics. We compared the performance of the four algorithms on the selected models with the three seeding strategies using two well-known quality indicator measures commonly employed in the multi-objective community, Hypervolume and Generational Distance. In short, our statistical analysis reveals that the third seeding strategy, the one that exploits more domain knowledge, performs better for both of the comparison measures, and identifies the performance differences among the algorithms.

### 9.3.1 Seeding Strategies

The impact that seeding strategies have on the performance of evolutionary algorithms has been documented for instance in Paul et al. [171]. Recall that the ultimate goal of a seeding strategy is to embed domain knowledge into the individuals of the population such that this knowledge is exploited when searching for solutions. Consequently, the strategies are also domain dependent. There are seeding strategies for engineering problems [176], positioning problems [182], timetabling problems [41], to cite a few examples.

In the field of software testing, seeding has also been used. In a recent work, Fraser and Arcuri [76] compare different seeding strategies and conclude that an improvement in the performance is achieved with statistical significance when a seeding strategies is used. It should be noted though that none of these pieces of work addresses seeding in the realm of SPL testing. Our focus is on measuring the impact of seeding the initial population (the first solution in PAES) of the four algorithms. Next we explain the three seeding strategies that we study: Size-Based Random Seeding, Greedy Seeding and Single-objective Based Seeding.

### Size-Based Random Seeding

The *size-based random seeding strategy (SB)* leverages the knowledge of the size of known complete test suites. In our case, we used solutions generated by the algorithm CASA [82], at the core of the third seeding strategy which we explain shortly in Section 9.3.1. We must stress that this strategy uses from the known complete test suites *only* their *size*. We summarize this strategy with the following code snippet, where *fm* is a feature model and `n` is the size of the population to seed:

$$seed := CASA(fm)$$
$$population := sizeBasedRandom(\underline{size(seed)}, n, fm)$$

Algorithm 9 sketches how this strategy works. It receives as input the size of the test suite to generate as seed (the number of features sets of a known complete test suite), the size of the population to generate, and a feature model. The core of the algorithm is a loop (Lines 5-8) that constructs a seed test suite (of the size of a known test suite) by randomly selecting valid feature sets from a feature model (Line 6). Once the `seed` test suite is computed, it is passed to algorithm *seedPopulation* (Line 9), sketched on Algorithm 10, to generate a population of the desired size which is returned.

**Algorithm 9** Size-Based Random Seeding Strategy.

```
1: proc sizeBasedRandom
2: Input: seedSize:int, populationSize:int, fm:feature model
3: Output: population: set of test suites
4: seed← ∅
5: for i ← 1 ... seedSize do
6:    featureSet←RandomFeatureSets(fm)
7:    seed←seed ∪ featureSet
8: end for
9: population←seedPopulation(seed, populationSize)
10: return population
```

Algorithm 10 uses the `seed` test suite that it receives as input to generate a population of a given size. It creates the new test suites by randomly removing feature sets from the population `seed` (Line 6). This algorithm is also used by the other two seeding strategies.

**Algorithm 10** Seed Population.

```
1: proc seedPopulation
2: Input: seed:test suite, populationSize:int
3: Output: population:set of test suite
4: population← ∅
5: for i ← 1 ... populationSize do
6:    testSuite← RemoveFeatureSets(seed)
7:    population← population ∪ testSuite
8: end for
9: return population
```

### Greedy Seeding

The *greedy seeding strategy (GS)*, Algorithm 11, generates of a single `seed` test suite with complete coverage from which it seeds a population. Thus, this strategy leverages the knowledge of the feature sets that are part of a complete test suite. To create a seed, we use a constructive approach that selects on each iteration the best feature set (Line 9) out of 100 randomly generated ones (Lines 7-10) until complete coverage is reached (Line 6). Once the `seed` test suite is computed, it is passed to algorithm *seedPopulation* (Line 13), sketched on Algorithm 10, to generate a population of the desired size which is then returned. We summarize this strategy with the following code snippet, where *fm* is a feature model and `n` is the size of the population to seed:

$$population := greedySeeding(n, fm)$$

**Algorithm 11** Greedy Seeding Strategy.

```
1:  proc greedySeeding
2:  Input: populationSize:int, fm:feature model
3:  Output: population:set of test suite
4:  seed← ∅
5:  bestFS← ∅
6:  while not Total_Coverage(seed) do
7:    for i ← 1 100 do
8:      newFS←RandomFeatureSets(fm)
9:      bestFS← ChooseBest(bestFS, newFS)
10:   end for
11:   seed← seed ∪ bestFS
12: end while
13: population←seedPopulation(seed, populationSize)
14: return population
```

### Single-objective Based Seeding

The *single-objective based seeding strategy (SO)* consists in using a complete test suite computed by a single-objective algorithm to seed a population. Thus, this strategy leverages knowledge of test suites computed by existing single-objective SPL testing approaches. For the task of generating test suites we chose CASA, a simulated annealing algorithm that was designed to generate $n$-wise covering arrays for SPLs [82]. CASA relies on three nested search strategies. The outermost search performs one-sided narrowing, pruning the potential size of the test suite to be generated by only decreasing the upper bound. The mid-level search performs a binary search for the test suite size. The innermost search strategy is the actual simulated annealing procedure, which tries to find a pairwise test suite of size `N` for feature model `FM`. We selected CASA because in our previous work it performed better than other techniques and it is well-known in both search-based and SPL research communities. For more details on CASA and its comparison with other approaches please refer to [82].

First, we performed 30 independent runs per feature model because CASA is a non-deterministic algorithm. Then, we randomly chose one of the solutions for seeding the population. Do notice that we used the same solution of CASA for seeding all the multi-objective algorithms in order to

make a fair comparison among them. We summarize this strategy with the following code snippet, where *fm* is a feature model and $n$ is the size of the population to seed:

$$population := seedPopulation(CASA(fm), n)$$

### 9.3.2 Evaluation

In this section we give some details of the particular configuration of the algorithms, the study corpus, and how the evaluation was carried out.

#### Algorithms Details

We selected the following four algorithms because they have been extensively and successfully applied to a large number of problem domains and represent a diverse set of techniques and concepts of multi-objective evolutionary algorithms: NSGA-II, MO-Cell, SPEA2 and PAES. Since all of them were described in Section 4.2, we outline here some details of the particular configuration of the algorithms.

All algorithms use the same representation for an individual, which is a set of products, and also the same variation operators. We have used *binary tournament* as the selection scheme. This operator works by randomly choosing two individuals from the population and the one dominating the other is selected; if both solutions are non-dominated one of them is randomly selected. The crossover operator, which is executed with probability 0.8, takes two solutions, $S_1$ and $S_2$, then one cross-point is randomly selected in both solutions generating two parts per solution $S_{1a}$ - $S_{1b}$ and $S_{2a}$ - $S_{2b}$. Finally, two new individuals are created $S_{1'}(S_{1a} - S_{2b})$ and $S_{2'}(S_{2a} - S_{1b})$. The mutation operator is executed with probability 0.1. It generates ten valid products, then the product that adds more coverage to the solution is added. The product that adds more coverage to the solution is known because we apply the mutation before applying the recombination. This order of the variation operators allow us to take advantage of the information collected in the evaluation of the individual. If the resulting individual has the same coverage and more test products, at the end of the iteration, the algorithms delete it from the population because this solution is dominated.

#### Feature Models Corpus

The experimental corpus of our evaluation is formed with 19 feature models[3]. These models have two important characteristics to make our results directly applicable by software engineers. First, that the feature models are associated to actual SPLs whose code is publicly available or can be made accessible by request to the corresponding authors. This is important so that testing can be carried out on the SPL code. Second, that the feature models are explicitly and directly provided by the SPL authors, rather than, for instance, reverse-engineered from other artefacts. This is important because it guarantees that all the feature combinations are correctly captured in the feature models. We searched into three main SPL related websites: SPL Conqueror [190], FeatureHouse [71], and SPL2go [193]. In addition, we looked at recently published articles within the SPL community. For the management and analysis of feature models, we relied on three frameworks: SPLAR [153], FAMA [203], and SPLCA [109]. These tools in turn, imposed additional constraints to the selection of our corpus[4]. Table 9.7 summarizes the feature models used in our evaluation. It shows the number of features, number of products, and their application domain

[3]They are available at http://neo.lcc.uma.es/staff/javi/resources.html
[4]For example, the type of CTCs that FAMA can analyze.

Table 9.7: Feature models summary.

| Feature Model | NF | NP | PF | Domain |
|---|---|---|---|---|
| Apache | 10 | 256 | 6 | web server [190] |
| ArgoUmlSpl | 11 | 192 | 6 | UML tool [20] |
| BDB* | 117 | 32 | NA | database [71] |
| BDBFootprint | 9 | 256 | 6 | database [190] |
| BDBMemory | 19 | 3,840 | NA | database [190] |
| BDBPerformance | 27 | 1,440 | NA | database [190] |
| Curl | 14 | 1024 | NA | data trasfer [190] |
| DesktopSearcher | 22 | 462 | 8 | file search [193] |
| FameDbmsFm | 20 | 320 | 6 | database [193] |
| GPL | 18 | 73 | 12 | graph algorithms [134] |
| LinkedList | 27 | 1,344 | NA | data structures [190] |
| LLVM | 12 | 1,024 | NA | compiler library [190] |
| PKJab | 12 | 72 | 6 | messenger [190] |
| Prevayler | 6 | 32 | 6 | object persistence [190] |
| SensorNetwork | 27 | 16,704 | NA | networking [190] |
| TankWar | 37 | 1,741,824 | NA | game [71] |
| Wget | 17 | 8,192 | NA | file retrieval [190] |
| x264 | 17 | 2,048 | NA | video encoding [190] |
| ZipMe | 8 | 64 | 6 | data compression [190] |

**NF**: Number of Features, **NP**:Number of Products,
**PF**: Pareto Front size, NA if not available.
*__BDB__ prefix stands for Berkeley database.

with the source where we obtained them from. Additionally, we provide the size of the Pareto front[5], if known. Note that here we extend the previous group of feature models G3 used for the prioritization analysis.

**Experimental Setting**

Here we describe how the evaluation was carried out. The four algorithms we analyzed are non-deterministic, thus we performed 30 independent runs for a meaningful statistical analysis. In order to measure the performance of the multi-objective algorithms used here, the quality of their resulting nondominated set of solutions has to be considered. Besides the two objective functions defined in Section 2.4.4, coverage and products, we selected two well-known quality indicators that are commonly used in the multi-objective community to compare the approximated Pareto fronts of several algorithms[6], Hypervolume and Generational Distance, described in Section 3.3.1. In addition to these quality measures, we have also analyzed the time required to run the multi-objective algorithms and obtain the Pareto front, since we want the algorithms to be as fast as possible. Since we have 4 algorithms, 3 different seeding strategies, and 19 feature models the total number of independent runs is $4 \times 3 \times 19 \times 30 = 6,840$. The stopping criterion was 1,000 evaluations.

[5]Computed with algorithm presented in [135].
[6]We used a reference front whenever the true Pareto front was unknown.

### 9.3.3 Experimental Analysis

We first analyze the behavior of the multi-objective algorithms with the aim of highlighting which algorithm works better. In Table 9.8 we show the average values of the quality indicators grouped by algorithm and the execution time. Regarding the significant differences, they exist only between PAES and the rest of the algorithms for both quality indicators and performance measure. We must point out that even though PAES seems to be the worst algorithm, the average generational distance value obtained by PAES is better than the obtained by NSGA-II. This high value of generational distance requires a further analysis (addressed shortly) because NSGA-II obtains the best values for the rest of the indicators, HV and time. SPEA2 is the best in generational distance. Our evaluation indicates there is not an algorithm which is the best for all quality indicators, nonetheless NSGA-II performs best in 2 out of 3 indicators.

Table 9.8: Comparison of multi-objective algorithms using the proposed quality indicators and performance time.

| Algorithms | HV | GD | Time(ms) |
|---|---|---|---|
| NSGA-II | 0.6583 | 0.0396 | 70523 |
| MOCell | 0.6553 | 0.0293 | 74325 |
| SPEA2 | 0.6533 | 0.0289 | 71349 |
| PAES | 0.6390 | 0.0351 | 101246 |

In Table 9.9 we summarize the average results obtained grouped by seeding strategy. In this case there are clear significant differences among all the seeding strategies. We have highlighted the best value per quality indicator and performance measure, which is always obtained with the SO seeding strategy. First, we analyze the hypervolume, the higher the value, the better the quality of the obtained results. The best value is obtained with the SO seeding strategy. There are statistical significant differences between SB and GS, and also between SB and SO. Thus, it can be concluded that the quality of solutions obtained with SB are worse than those obtained with the other strategies. Second, we analyze the generational distance, the lower the value, the better the quality of the results. The best value is obtained with the SO seeding strategy again. In addition, there are significant differences with the other seeding strategies for the generational distance indicator. Finally, we analyze the time spent in the generation of the Pareto front. The results are clear, the SO strategy is the fastest. Recall that the execution of the CASA algorithm for seeding is included, which on average is 2905 milliseconds.

Table 9.9: Comparison of seeding strategies using hypervolume, generational distance, and performance time.

| Seeding Strategy | HV | GD | Time(ms) |
|---|---|---|---|
| SizeBased Random (SB) | 0.6421 | 0.0427 | 138404 |
| Greedy (GS) | 0.6556 | 0.0447 | 76783 |
| SingleObjective (SO) | 0.6568 | 0.0123 | 25800 |

Table 9.10 shows the $\hat{A}_{12}$ statistic to assess the practical significance of the results. We have highlighted the largest distance from 0.5 (equality) per quality indicator, note that 0.5 indicates no difference in the comparison. Regarding HV, there are no big differences between the algorithms. The highest difference occurs between NSGA-II and PAES. Regarding GD, the highest difference

is between MOCell and PAES. There, the generational distance is larger in 41.94% of the times. In addition, notice that NSGA-II obtains smaller values than PAES with probability 0.544 (1-0.4560).

This result indicates a tendency contrary to the deduced from the previous average value of GD of NSGA-II, in most of the comparisons it achieves a lower value than PAES. Regarding time, NSGA-II is faster with more probability than the other algorithms. In general, this statistic confirms again that NSGA-II obtains better Pareto fronts than the other algorithms, according to the selected quality indicators, and faster.

Table 9.10: $\hat{A}_{12}$ statistical test results for all algorithms. NSGA-II yields better results for HV and time measures.

| Algorithms | HV | GD | Time |
|---|---|---|---|
| NSGAII-SPEA2 | 0.5182 | 0.5172 | 0.4904 |
| NSGAII-MOCell | 0.5112 | 0.5202 | 0.4816 |
| NSGAII-PAES | 0.5626 | 0.4560 | 0.2839 |
| SPEA2-MOCell | 0.4932 | 0.5039 | 0.4910 |
| SPEA2-PAES | 0.5447 | 0.4205 | 0.3019 |
| MOCell-PAES | 0.5521 | 0.4194 | 0.3027 |

We now comment on the $\hat{A}_{12}$ statistic values shown in Table 9.11 that compares the seeding strategies. The SO strategy obtains higher values of HV than the other strategies, it obtains better values of HV than SB and GS strategies in a 54.42% (probability 1-0.4558) and a 50.23% (probability 1-0.4977), respectively. For GD, the SO strategy obtains better values (lower values) than SB and GS in a 85.62% and in a 78.39%, respectively. Therefore, SO is widely best for the generational distance indicator. Finally, SO spends less time than SB and GS in 86.19% and 82.27%, respectively. Thus, the SO strategy is the fastest without any doubt.

Table 9.11: $\hat{A}_{12}$ statistical test results for seeding strategies. SO yields better quality indicators and time values.

| Seeding Strategy | HV | GD | Time |
|---|---|---|---|
| SizedBased(SB) – Greedy(GS) | 0.4568 | 0.4795 | 0.6377 |
| SizedBased(SB) – SingleObjective(SO) | 0.4558 | 0.8562 | 0.8619 |
| Greedy(GS) – SingleObjective(SO) | 0.4977 | 0.7839 | 0.8227 |

With the results presented here we are now able to answer our research questions. Regarding **RQ2**, we didn't observe in our results a clear winner algorithm. We can claim, however, that PAES seems to provide approximated Pareto fronts of lower quality than the ones of the other algorithms. PAES is a trajectory-based algorithm (works with one solution and not a population), and probably this makes it less competitive in this problem, in which having a diverse population seems to be beneficial.

From the point of view of the testing engineer that faces this problem, our recommendation is to use any of the three best algorithms, NSGA-II, SPEA2 or MOCell, with the SO seeding strategy.The results obtained are approximated Pareto fronts like the one in Figure 9.1. Each point in that front represents a test suite for the SPL, that is, a set of products. The number of products and the coverage are given by the coordinates of the point. Thus, the multi-objective approach offers the engineer a set of test suites of different size having (almost) optimal coverage.

The engineer can, then, select the most appropriate test suite depending on the resources s/he has to test the SPL or the level of coverage s/he has to satisfy. It is important to note that this selection of the most appropriate test suite can be done after the algorithms did their job. This contrasts with the scalarization technique used in the related work (i.e. [103] and [212]) that transform the multi-objective problem in a single-objective one. In that case, only one solution is provided and the testing engineer has to provide a weight for each objective function to indicate the relative importance of the objectives. Thus, s/he is somehow selecting the solution beforehand, without the possibility of taking a look to all other, probably interesting, solutions.

Regarding **RQ3**, we can say that the seeding strategy has an impact on the performance of the search algorithm. Seeding the multi-objective algorithm with good quality solutions obtained using CASA reduces the time required in the search and finds the approximated Pareto front that is nearer to the Pareto front. These initial "seeds" include some desirable building blocks (i.e. feature sets) of the optimal solutions that help the algorithm in its search.

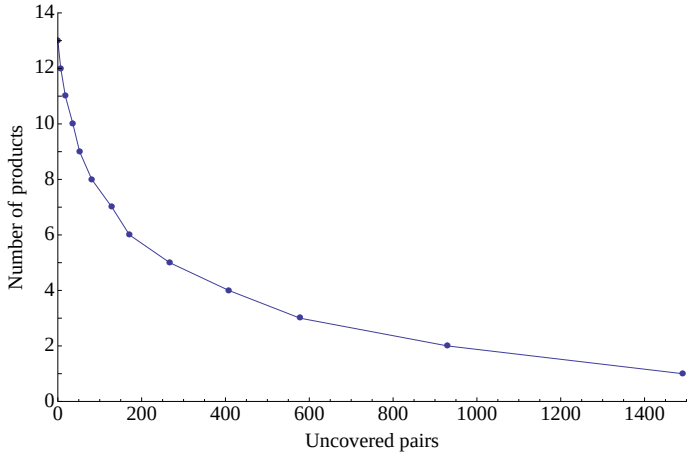



Figure 9.1: Approximated Pareto front obtained by NSGA-II in TankWar.

## 9.4 Optimal Multi-Objective Pairwise Testing

With our previous MO approach, we are able to compute near-optimal Pareto fronts in a reasonable amount of time. However, here we present an exact method, a zero-one mathematical linear program, for solving the multi-objective problem and an algorithm that computes the true Pareto front of a feature model using SAT solvers. This front is the optimal solution for both objectives. We are interested in minimizing the number of test products and maximizing the pairwise coverage. Since we want to compute the Pareto front of the multi-objective optimization problem we proceed by fixing the number of test products and defining a zero-one mathematical program that maximizes coverage. The approach presented here relates to the work by Arito et al. [21] for solving a multi-objective test suite minimization problem in regression testing.

Our evaluation found a correlation between runtime and number of products in the feature model and revealed a trade-off between reducing the number of constraints in the mathematical linear program and runtime that we plan to explore as future work.

### 9.4.1 Mathematical Linear Program

A zero-one program is an integer program in which the variables can only take values 0 or 1 [221]. The details of the algorithm applied are explained in Section 9.4.2. In this section we describe the zero-one program. Let us call $n$ to the number of test products (that is fixed) and $f$ to the number of features of the FM. We will use the set of decision variables $x_{i,j} \in \{0,1\}$ where $i \in \{1,2,\ldots,n\}$ and $j \in \{1,2,\ldots,f\}$. Variable $x_{i,j}$ is 1 if product $i$ has feature $j$ and 0 otherwise. Not all the combinations of features form valid products. Following [29], we can express the validity of any product in an FM as a boolean formula. These boolean formulas can be expressed in Conjunctive Normal Form (CNF) as a conjunction of clauses, which in turn can be expressed as constraints in a zero-one program. The way to do it is by adding one constraint for each clause in the CNF. Let us focus on one clause and let us define the Boolean vectors $v$ and $u$ as follows [196]:

$$v_j = \begin{cases} 1 & \text{if feature } j \text{ appears in the clause,} \\ 0 & \text{otherwise,} \end{cases}$$

$$u_j = \begin{cases} 1 & \text{if feature } j \text{ appears negated in the clause,} \\ 0 & \text{otherwise.} \end{cases}$$

With the help of $u$ and $v$ we can write the constraint that corresponds to one CNF clause for the $i$-th product as:

$$\sum_{j=1}^{f} v_j(u_j(1-x_{i,j}) + (1-u_j)x_{i,j}) \geq 1 \tag{9.1}$$

As an illustration, in the GPL model let us suppose that $Search$ is the 8-th feature and $Num$ is the 12-th one. The cross-tree constraint "$Num$ requires $Search$" can be written in CNF with the clause $\neg Num \vee Search$ and translated to a zero-one constraint as: $1 - x_{i,12} + x_{i,8} \geq 1$.

Our focus is pairwise coverage. This means that we want for each pair of features to cover 4 cases: both unselected, both selected, first selected and second unselected and vice versa. We introduce one variable in our program for each product, each pair of features and each of these four possibilities. The variables, called $c_{i,j,k,l}$, take value 1 if product $i$ covers the pair of features $j$ and $k$ with the combination $l$. The combination $l$ is a number between 0 and 3 representing the selection configuration of the features according to the next mapping: $l = 0$, both unselected; $l = 1$, second selected and first unselected; $l = 2$, first selected and second unselected; and $l = 3$ both selected. The values of the variables $c_{i,j,k,l}$ depend on the values of $x_{i,j}$. In order to reflect this dependence in the mathematical program we need to add the following constraints for all $i \in \{1,\ldots,n\}$ and all $1 \leq j < k \leq f$:

$$2c_{i,j,k,0} \leq (1-x_{i,j}) + (1-x_{i,k}) \leq 1 + c_{i,j,k,0} \tag{9.2}$$

$$2c_{i,j,k,1} \leq (1-x_{i,j}) + x_{i,k} \leq 1 + c_{i,j,k,1} \tag{9.3}$$

$$2c_{i,j,k,2} \leq x_{i,j} + (1-x_{i,k}) \leq 1 + c_{i,j,k,2} \tag{9.4}$$

$$2c_{i,j,k,3} \leq x_{i,j} + x_{i,k} \leq 1 + c_{i,j,k,3} \tag{9.5}$$

Variables $c_{i,j,k,l}$ inform about the coverage in one product. We need new variables to count the pairs covered when all the products are considered. These variables are called $d_{j,k,l}$, and take value 1 when the pair of features $j$ and $k$ with combination $l$ is covered by some product and 0 otherwise. This dependence between the $c_{i,j,k,l}$ variables and the $d_{j,k,l}$ variables is represented by

the following set of inequalities for all $1 \leq j < k \leq f$ and $0 \leq l \leq 3$:

$$d_{j,k,l} \leq \sum_{i=1}^{n} c_{i,j,k,l} \leq n \cdot d_{j,k,l} \tag{9.6}$$

Finally, the goal of our program is to maximize the pairwise coverage, which is given by the number of variables $d_{j,k,l}$ that are 1. We can write this as:

$$\max \sum_{j=1}^{f-1} \sum_{k=j+1}^{f} \sum_{l=0}^{3} d_{j,k,l} \tag{9.7}$$

The mathematical program is composed of the goal Equation (9.7) subject to the $4(n+1)f(f-1)$ constraints given by Equation (9.2) to Equation (9.6) plus the constraints of the FM expressed with the inequalities Equation (9.1) for each product. The number of variables of the program is $nf + 2(n+1)f(f-1)$. The solution to this zero-one linear program is a test suite with the maximum coverage that can be obtained with $n$ products.

### 9.4.2 Algorithm Details

The algorithm we use for obtaining the optimal Pareto set is given in Algorithm 12. This algorithm takes as input the FM and provides the optimal Pareto set. It starts by adding to the set two solutions that are always in the set: the empty solution (with zero coverage) and one arbitrary solution (with coverage $\binom{f}{2}$, number 2-combinations of the set of features). After that it enters a loop in which successive zero-one linear programs are generated for an increasing number of products starting at 2. Each mathematical model is solved using an extended SAT solver: `MiniSat+`[7]. This solver provides a test suite with the maximum coverage. This solution is stored in the optimal Pareto set. The algorithm stops when adding a new product to the test suite does not increase the coverage. The result is the optimal Pareto set.

**Algorithm 12** Algorithm for obtaining the optimal Pareto set.

```
optimal_set ← {∅};
cov[0] ← 0;
cov[1] ← (f 2);
sol ←arbitraryValidSolution(fm);
i ← 1;
while cov[i] ≠ cov[i − 1] do
  optimal_set ← optimal_set ∪ {sol};
  i ← i + 1;
  m ←prepareMathModel(fm,i);
  sol ←solveMathModel(m);
  cov[i] ← |sol|;
end while
```

[7]Available at URL: `http://minisat.se/MiniSat+.html`

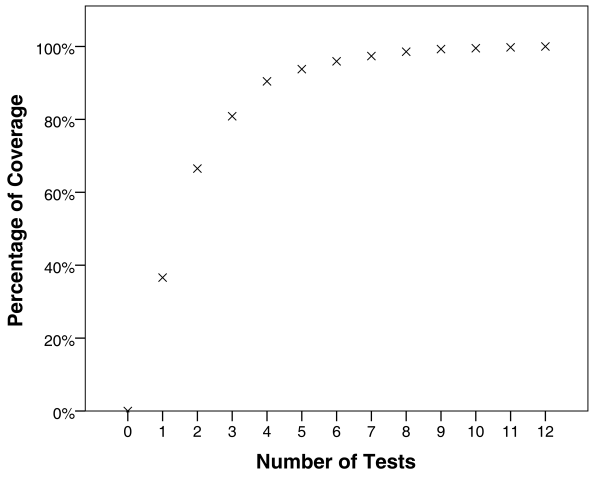



Figure 9.2: Pareto optimal front for our running example (GPL).

### 9.4.3 Experimental Setup and Analysis

The experimental corpus of our evaluation is composed by a benchmark of 118 feature models[8], whose number of products ranges from 16 to 640 products, that are publicly available from the SPL Conqueror [190] and the SPLOT [194] repositories. The objectives to optimize are the number of products required to test the SPL and the achieved coverage. Additionally, as performance measure we have also analyzed the time required to run the algorithm, since we want the algorithm to be as fast as possible.

We computed the Pareto optimal front for each model. Figure 9.2 shows this front for our running example GPL, where the total coverage is obtained with 12 products, and for every test suite size the obtained coverage is also optimal. As our approach is able to compute the Pareto optimal front for every feature model in our corpus, it makes no sense to analyze the quality of the solutions. Instead, we consider more interesting to study the scalability of our approach. For that, we analyzed the execution time of the algorithm as a function of the number of products represented by the feature model as shown in Figure 9.3. In this figure we can observe a tendency: the higher the number of products, the higher the execution time. Although it cannot be clearly appreciated in the figure, the execution time does not grow linearly with the number of products, the growth is faster than linear.

In order to check our intuition, we have performed a Spearman's rank correlation test. This test's coefficient $\rho$ takes into account the rank of the samples instead of the samples themselves. The correlation coefficient between the execution time and the number of products denoted by a feature model is 0.831. This is a very high value that confirms our expectations, the higher the number of products, the higher the execution time of the algorithm. We also computed the Spearman's rank correlation for the execution time against the number of features of the feature models which was quite lower (0.407). This is because two feature models with the same number of features could denote significantly different number of products depending on the constraints derived from the relationships between the features. In summary, the answer of the **RQ4** is that

[8]They are available at http://neo.lcc.uma.es/staff/javi/resources.html

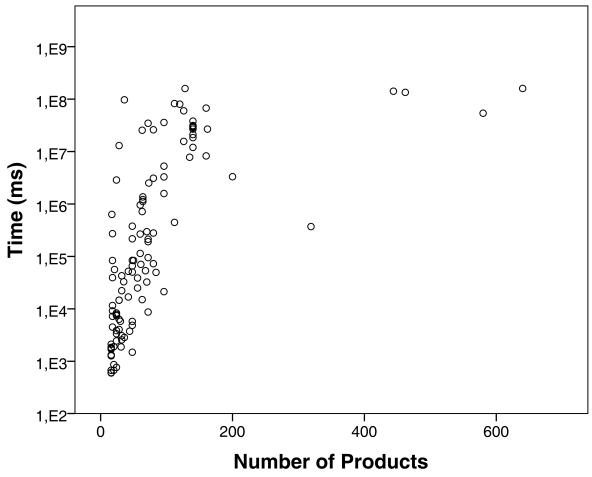



Figure 9.3: Time (log scale) required to find optimal Pareto set against the number of products of the feature models.

the best indicator of the execution time of our approach is the number of products denoted by a feature model.

## 9.5 Conclusions

Throughout this chapter we have filled several existent gaps in the SPL literature. We have tackled the pairwise test data generation problem in SPL, then we have successfully applied classical MO techniques to SPL, and finally we have presented an exact approach for computing the optimal Pareto front. Let us draw some conclusions separately for each of them.

First, we have formalized a SPL testing prioritization scheme (in Section 2.4.3) and presented its implementation with PPGS. We evaluated PPGS with 235 feature models of different characteristics using different selection criteria for product prioritization. Furthermore, we compared PPGS with greedy algorithm pICPL, a comparison that totalled 79,800 independent runs. Our analysis showed that while PPGS obtains overall shorter covering arrays it exhibits a performance difference with pICPL that tends to decrease for the feature models with larger number of products.

Second, we study the behavior of classical multi-objective evolutionary techniques applied to SPL pairwise testing. The group of algorithms were selected to cover a diverse array of techniques and concepts of multi-objective evolutionary computing. In addition, we study the impact of seeding in performance. Our evaluation unequivocally showed that seeding with knowledge from a single-objective technique produces significantly better results in less time. It also suggests that using this seeding strategy with either of NSGA-II, SPEA2 or MOCell yields results of similar quality. Our findings enable software engineers facing SPL combinatorial testing challenges to select not just one solution (as in the case of single-objective techniques) but instead to select from an array of test suite possibilities that can better match their economical or technological constraints.

Finally, we have proposed an approach to exactly obtain the optimal Pareto set of the multi-objective SPL pairwise testing problem. We defined a zero-one linear mathematical program and

an algorithm based on SAT solvers for obtaining the optimal Pareto set. By construction the solution obtained using this approach is optimal and could serve as reference for measuring the quality of the solutions proposed by approximated methods. The evaluation revealed a generally large runtime for our feature models. This fact prompted us to analyze the impact of the number of products and number of features in runtime. We found a high correlation in the first case and a low correlation in the second case. As a result of this finding our future work is twofold. First, we want to streamline the mathematical program representation in order to reduce the runtime of the algorithm. We observed that some of the constraints can be redundant. For instance, features that are selected in all the products of the product line do not need a variable since they are valid for any product. Similarly, there are pairs of feature combinations, that is $c_{i,j,k,l}$ variables, that are not valid according to the feature model and hence can be eliminated [100]. We also noticed that removing some of the redundant constraints can increase the runtime, while adding more constraints could help the SAT solver search for a solution. We plan to study the right balance of both reducing and augmenting constraints. Second, we will look at larger feature models to further study the scalability of our approach.



# Part IV

# Conclusions and Future Lines of Research

# Chapter 10

# Conclusions and Future Work

This thesis proposes a variety of contributions to the software testing field, mainly using metaheuristic techniques. We have encompassed a wide range of aspects related to testing a program: procedural and object-oriented source code, structural and functional paradigms, single-objective and multi-objective problems, isolated test cases and test sequences, and theoretical and experimental work. Regarding the analysis carried out, we have put more stress in the statistical analysis to assess the practical significance of the results. This thesis dissertation is the beginning of a research work which should be continued. For this reason, in this chapter we also describe some open issues we think are interesting to tackle in the near future.

## 10.1 Conclusions

Summarizing, we draw here the conclusions we have extracted from the main contributions of this thesis dissertation:

1. **Definition of a new distance measure for the `instanceof` operator in Object Oriented programs**. In this work we have focused on one aspect of OO Software, inheritance, to propose some approaches that can help to better guide the search of test data in the context of OO evolutionary testing. In particular, we have proposed a distance measure to compute the branch distance in the presence of the `instanceof` operator in Java programs. We have also proposed two mutation operators that change the solutions based on the distance measure defined. One of them is an adaptive mutation operator that is able to make a better search. Its main parameter $\lambda$ controls the velocity the search changes from exploration to exploitation behavior. The experimentation confirms that the search works worse with extremme values of $\lambda$. Finally, one of the main conclusions of this work is that the difficulty to test a program depends on the number of atomic conditions per logical expression and the nesting degree, since we are interested in measuring the complexity of testing a program.

2. **Definition of a new complexity measure called "Branch Coverage Expectation"**. In this work we dealt with the testing complexity from an original point of view: a program is more complex if it is more difficult to be automatically tested. Therefore, we defined the "Branch Coverage Expectation" in order to provide some knowledge about the difficulty of testing programs. The foundation of this measure is based on a Markov model of the program. The Markov model provides a theoretical background. The analysis of this measure indicates



that it is more correlated with branch coverage than the other studied static measures. This means that this is a good way of estimating the difficulty of testing a program. We think, supported by the results, that this measure is useful for predicting the behaviour of an automatic test data generator.

3. **Theoretical prediction of the number of test cases needed to cover a concrete percentage of the program computed**. Our Markov model of a program can be used to provide an estimation of the number of test cases needed to cover a concrete percentage of the program. We have compared our theoretical prediction with an average of real executions of a test data generator. The results show that our prediction is very similar to the evolution of a real execution of the test data generator. This model can help project managers to predict the evolution of the testing phase, which consequently can save time and cost of the entire project. This theoretical prediction could be also very useful to determine the coverage percentage using a particular number of test cases.

4. **Proposal of a whole test suite approach for solving multi-objective test data generation problem**. We have studied the Multi-Objective Test Data Generation Problem with the aim of analyzing the performance of a direct whole test suite multi-objective approach versus the application of mono-objective algorithms followed by a test case selection. Previous results in the literature have only focused on the coverage of a program while the oracle cost is a significant cost that has been ignored. We have evaluated four state-of-the-art multi-objective optimization algorithms: MOCell, NSGA-II, SPEA2, and PAES, two mono-objective algorithms GA, ES, and two random algorithms. In terms of convergence towards the optimal Pareto front, GA and MOCell have been the best solvers in our comparison. Although the multi-objective approach is working very well in most of the programs, we realized that dealing with only one branch at the same time (mono-objective approach) can be more effective when the program under test has high nesting degree. However, we highly recommend the direct approach if we have time restrictions.

5. **Comparison of different prioritization strategies in Software Product Lines and Classification Trees**. We have studied the Prioritized Pairwise Test Data Generation Problem with the aim of analyzing the performance of several approaches. We have compared five different approaches related to the CTM, and two related to SPL, four of them proposed by us. We have performed some experiments on a great number of different scenario/distribution combinations and for different values of weight coverage, which makes our study meaningful. The genetic algorithm outperforms the other algorithms in most scenarios and distributions, it is the best choice when one has some time restrictions or the execution of a test case is quite costly. In the SPL experimentation, our analysis also showed that while our parallel genetic approach obtains overall shorter covering arrays it exhibits a performance difference with the parallel version of ICPL that tends to decrease for the feature models with larger number of products.

6. **Definition of the Extended Classification Tree Method to generate test sequences**. We have defined an entire model (ECTM) which both industry and academia could use to completely describe all aspects needed to generate sequences of tests for testing a program. We have presented two different metaheuristic approaches to optimize the automatic generation of test sequences for the CTM. The first is a genetic algorithm with memory operator (GTSG), which is able to preserve the memory required to evaluate individuals, while also allowing the algorithm to compute a solution faster than without the operator. The second



is an ACO algorithm that is able to obtain good quality solutions using little memory. Our comparison shows that ACOts is the best algorithm in the comparison (with an state-of-the-art greedy algorithm and the GTSG). It has a good tradeoff between test suite size and coverage. Its benefits are clear, we can save costs and time executing all test steps sequentially because the previous test step puts the software in the adequate state to test the next functionality.

7. **Exploration of the effect of different seeding strategies in the computation of the Pareto fronts in SPL**. We study the behaviour of classical multi-objective evolutionary techniques applied to SPL pairwise testing. In addition, we study the impact of seeding in performance. The group of algorithms were selected to cover a diverse array of techniques and concepts of multi-objective evolutionary computing. Our evaluation unequivocally showed that seeding with knowledge from a single-objective technique produces significantly better results in less time. It also suggests that using this seeding strategy with either of NSGA-II, SPEA2 or MOCell yields results of similar quality. Our findings enable software engineers facing SPL combinatorial testing challenges to select not just one solution (as in the case of single-objective techniques) but instead to select from an array of test suite possibilities that can better match their economical or technological constraints.

8. **Proposal of an exact technique for the computation of optimal Pareto fronts in SPL**. We have proposed an approach to exactly obtain the optimal Pareto set of the multi-objective SPL pairwise testing problem. We defined a zero-one linear mathematical program and an algorithm based on SAT solvers for obtaining the optimal Pareto set. Since the solution obtained using this approach is optimal, it could serve as reference for measuring the quality of the solutions proposed by approximated methods. The evaluation revealed a generally large runtime for the feature models. This fact prompted us to analyze the impact of the number of products and number of features in runtime. We found a high correlation in the first case and a low correlation in the second one. This means that the computation time of the optimal Pareto front depends on the number of products defined by the feature model.

One of the topics that we found of particular interest in this thesis is the proposal of a new complexity measure to provide some knowledge about the difficulty of testing programs. Actually, the University of Malaga have considered to register this work as an international patent (PCT/ES2015/000100).

## 10.2 Future Work

There are different future lines possible after the work contained in the thesis. Broadly speaking, these lines are the continuation with the design of the Markov model of a program, the extension of the feature model corpus and the group of multi-objective algorithms analyzed, the proposition of trajectory search-based algorithm for the generation of test sequences, and a validation strategy based on software testing techniques for the traffic light programs. Our future proposals in them are as follows.

As to the continuation with the design of the Markov model of a program, we plan to improve the representation of a program, but without losing its simplicity. We plan to advance in the knowledge of the features of a program that occurs in a condition. The computation of the probabilities associated to a concrete decision is a great challenge to improve our measure. In

addition, we will take into account the data dependencies in the probabilities computed for the Markov model. This fact will provide more precision in the transition probabilities. Besides that, we plan to consider the amount of resources needed to execute different test cases, in particular when loops are involved in the execution of a test case. We would also like to apply our complexity measure to real-world software and compare the results with the real difficulty of testing the program by an expert.

Our work opens up several research venues, which we plan to address as part of our future work. One of them is the extension of the feature model corpus and the group of multi-objective algorithms analyzed. Besides these objectives, our goal is to expand our study beyond pairwise coverage ($t \geq 2$), to integrate domain knowledge such as control flow information as an optimization objective for test code generation, and to characterize when a particular multi-objective algorithm performs better based, for example, on structural metrics of the feature models [24].

Moreover, an interesting research topic nowadays is the generation of test sequences. We have proposed different approaches for the automatic generation of test sequences to be integrated in the CTE XL professional tool. We need to collect more real scenarios for comparison purposes, this is absolutely necessary when you are adding functionality to a professional tool. Although we have obtained great results with the ACOts algorithm, we plan to propose a trajectory search-based algorithm such as Simulated Annealing that has obtained great results in Combinatorial Testing (Covering Arrays [200]) and might suit this problem.

Regarding the evaluation of the exact approach for the computation of the true Pareto front in SPL, we found a high correlation with the number of products defined by the feature model of a program. As a result of this finding our future work, we wants to streamline the mathematical program representation in order to reduce the runtime of the algorithm. We observed that some of the constraints can be redundant. For instance, features that are selected in all the products of the product line do not need a variable since they are valid for any product. Similarly, there are pairs of feature combinations, that are not valid according to the feature model and hence can be eliminated [100]. We also noticed that removing some of the redundant constraints can increase the runtime, while adding more constraints could help the SAT solver search for a solution. We plan to study the right balance of both by reducing and augmenting constraints.

Finally, we also plan to apply the algorithms proposed in this thesis for solving real-world instance problems. In concrete, we are interested in applying them for solving problems related to the scheduling of traffic lights. We plan to propose a validation strategy for the traffic light programs to be used by the human experts of Smart Mobility. We will propose the use of a traffic feature model with priorities, which allows us not only to reduce the number of traffic scenarios to test the available cycle programs, but also to generate the most important scenarios. A first journal article following this research line is under review at the moment of writing.



# Appendices

# Appendix A

# Publications Supporting this PhD Thesis Dissertation

In this appendix, we present the set of scientific articles that have been published during the years in which this thesis has been developed. These publications speak for the interest, validity, and impact on the scientific community and literature of the work contained in this thesis, since they have appeared in impact fora, and have been subjected to peer review by expert researchers.

**International Journals indexed by ISI-JCR**

[1] Ferrer, J., Kruse P. M., Chicano F., and Alba E. (2014). *Search based algorithms for test sequence generation in functional testing.* Information and Software Technology.

[2] Ferrer, J., Chicano F., and Alba E. (2013). *Estimating software testing complexity.* Information and Software Technology. 55, 2125 − 2139.

[3] Ferrer, J., Chicano F., and Alba E. (2012). *Evolutionary algorithms for the multi-objective test data generation problem.* Software Practice and Experience 42, 1331 − 1362.

**International Journals indexed by ISI-JCR under review**

[4] Ferrer J. ,García-Nieto J., Chicano F., and Alba E. (2015). Intelligent Testing of Traffic Light Programs: Validation in Smart Mobility Scenarios. Submitted to Mathematical Problems in Engineering.

**International Journals**

[5] Ferrer, J., Chicano F., and Alba E. (2010). Correlation between static measures and code coverage in evolutionary test data generation. International Journal of Software Engineering and its Applications. 4, 57−79.

**Book Chapters and LNCS Series**

[6] R. Lopez-Herrejon, J. Ferrer, F. Chicano, A.Egyed, and E.Alba Evolutionary Computation for Software Product Line Testing: An Overview and Open Challenges (to appear)- Book chapter.

[7] Chicano, F., Ferrer J., and Alba E. (2011). Elementary Landscape Decomposition of the Test Suite Minimization Problem. Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10−12, 2011, p.48−63.

**International Conferences (Core A)**

[8] Garca-Nieto J., Ferrer, J., and Alba E. (2014). Optimising Traffic Lights with Metaheuristics: Reduction of Car Emissions and Consumption. Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN 2014), Beijing, China, July 6−11, 2014, 48−54.

[9] Lopez-Herrejon, R. Erick, Ferrer J., Chicano F., Haslinger E. Nicole, Egyed A., and Alba E. (2014). A parallel evolutionary algorithm for prioritized pairwise testing of software product lines. Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12−16, 2014. p. 1255−1262.

[10] Lopez-Herrejon, R. E., Chicano F., Ferrer J., Egyed A., and Alba E. (2013). Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing. IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22−28, 2013. p. 404−407.

[11] Ferrer, J., Kruse P. M., Chicano F., and Alba E. (2012). Evolutionary algorithm for prioritized pairwise test data generation. Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA, July 7−11, 2012. p. 1213−1220.

[12] Ferrer, J., Chicano F., and Alba E. (2009). Dealing with inheritance in OO evolutionary testing. Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Qubec, Canada, July 8−12, 2009. p.1665−1672.

**International Conferences (Core B, Core C, and non-Core)**

[13] Lopez-Herrejon, R. Erick, Ferrer J., Chicano F., Egyed A., and Alba E. (2014). Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of Software Product Lines. Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6−11, 2014. p. 387−396.

[14] Ferrer, J., Chicano F., and Alba E. (2011). Benchmark Generator for Software Testers. Artificial Intelligence Applications and Innovations - 12th INNS EANN-SIG International Conference, EANN 2011 and 7th IFIP WG 12.5 International Conference, AIAI 2011, Corfu, Greece, September 15−18, 2011, Proceedings, Part II. p. 378−388.

[15] Ferrer, J., Chicano F., and Alba E. (2010). Measuring Testing Complexity. 2nd International Symposium on Search Based Software Engineering, Benevento, Italy (Short paper online URL: http://ssbse.org/2010/fastabstracts/ssbse2010_fastabstract_02.pdf)

**National Conferences**

[16] Ferrer, J., Alba, E., and Chicano F. (2015) Sistema Inteligente para la Recogida de Residuos en las Ciudades basado en Predicciones de Llenado. Proceedings of the I Congreso Ciudades Inteligentes, Madrid, March 24−25, 2015, p. 319−324

[17] Ferrer, J., Kruse P. M., Chicano F., and Alba E. (2015) Generación de secuencias de pruebas funcionales con algoritmos bio-inspirados. X Congreso Español de Metaheursticas, Algoritmos Evolutivos y Bioinspirados (MAEB), Merida, February 4−6, 2015, p. 59−66.

[18] Ferrer, J., Garca-Nieto J., Alba E., and Chicano F. (2013). Validación Inteligente para la Sincronización de Semáforos Basada en Feature Models. IX Congreso Espaol de Metaheursticas, Algoritmos Evolutivos y Bioinspirados (MAEB), Albacete, September 17−20, p. 812−821.

[19] Ferrer, J., Chicano F., and Alba E. (2009). On the Correlation between Static Measures and Code Coverage using Evolutionary Test Case Generation. Actas de los Talleres de las JISBD, San Sebastián, September 8−9, vol 3, n 1, p.50−61.

**Articles in CoRR - not reviewed**

[20] Roberto E. Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Evelyn Nicole Haslinger, Alexander Egyed, and Enrique Alba. (2014) Towards a Benchmark and a Comparison Framework for Combinatorial Interaction Testing of Software Product Lines. CoRR abs/1401.5367

[21] Roberto E. Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Lukas Linsbauer, Alexander Egyed, and Enrique Alba. (2014) A Hitchhiker's Guide to Search-Based Software Engineering for Software Product Lines. CoRR abs/1406.2823



# Appendix B

# Resumen en Español

## B.1 Introducción

La mayoría de los países dependen de sistemas complejos basados en computadores que gobiernan las principales infraestructuras y servicios, así como la mayoría de dispositivos electrónicos usados diariamente. Consecuentemente, producir y mantener software es esencial para la sociedad actual, lo que significa un importante desafío para la comunidad científica. La ciencia de la computación es un campo de investigación en expansión que involucra el entendimiento y diseño de los computadores y su software. El principal objetivo de este campo es el estudio de los procesos algorítmicos automáticos que escalan para eliminar cuellos de botella. Uno de los más relevantes aspectos de la investigación en computadores es el diseño y desarrollo de nuevos algoritmos eficientes capaces de resolver problemas complejos en cada vez menos tiempo. Adicionalmente, la comunidad de investigadores en este campo está determinada a enfrentar estos problemas que no pudieron ser resueltos previamente.

Los problemas del mundo real son duros (en la mayoría de casos NP-duros), que significa que la complejidad del problema crece de forma exponencial con su tamaño. Cuando se usan técnicas exactas, estas aseguran encontrar la solución óptima en problemas relativamente pequeños, pero son extremadamente lentos en problemas medios y grandes. La mayoría de los problemas del mundo real tiene un espacio de búsqueda grande, a la vez que esátn sujetos a restricciones e incertidumbres, así que son difíciles de resolver por los algoritmos exactos. Con este objetivo en mente, los investigadores han aplicado técnicas emergentes como las metaheurísticas, las cuales han probado su utilidad enfrentando problemas duros. Los enfoques metaheurísticos son capaces de proporcionar soluciones cuasi-óptimas en un tiempo moderado, así que ofrecen un buen compromiso entre calidad de las soluciones y el ahorro de recursos, que es de hecho, el principal objetivo de las compañías industriales.

La Ingeniería del Software es una de las disciplinas de ingeniería que se preocupa de todos los aspectos de la producción de software, desde las etapas más tempranas de la especificación hasta el mantenimiento cuando el software ya está en uso [192]. El desarrollo de software confiable es un desafío clave para la industria del software, de hecho, se estima que alrededor de la mitad del tiempo empleado en el desarrollo del proyecto software y más de la mitad de su coste se debe a las pruebas del producto. La automatización de la generación de pruebas podría reducir el coste de todo el proyecto, esto explica porque tanto la industria del software como la academia están interesados en las herramientas de pruebas automáticas. Como la generación de pruebas adecuadas

implica un coste computacional grande, los enfoques basados en búsqueda son esenciales para lidiar con este problema.

En esta tesis aplicamos técnicas de búsqueda metaheurísticas para la optimización de problemas derivados del proceso de automatización de pruebas, particularmente el problema de automatización de la generación de datos de prueba para encontrar errores en el código fuente. Hay un rango amplio de problemas de generación de datos de prueba, en este trabajo intentamos abarcar ambos tipos de pruebas de software, de caja blanca así como de caja negra. En pruebas de caja blanca, también llamadas pruebas estructurales, los ingenieros de pruebas requieren conocimiento de cómo está implementado el software, en contraste en pruebas de caja negra o pruebas funcionales, donde los ingenieros de pruebas se concentran en qué hace el software, en vez de cómo lo hace.

Este es el contexto de este trabajo de tesis. Estamos determinados a proponer técnicas metaheurísticas para resolver las diferentes variantes de la generación de datos automatizada para software. Analizamos las diferentes técnicas con el objetivo de obtener resultados óptimos en calidad y coste, ambos aspectos muy importantes en el desarrollo software debido a la falta de recursos. Debido al carácter intrínseco hemos propuesto técnicas de optimización mono-objetivo para maximizar la calidad del conjunto de pruebas y técnicas multi-objetivo que consideran adicionalmente el coste de las pruebas. Además, pensamos que para comprender y resolver un problema necesitamos extraer todo el conocimiento posible. Consecuentemente proponemos una medida nueva de complejidad para predecir, de una mejor manera, la dificultad de probar una pieza de código con el objetivo de ayudar a los gestores de los proyectos a estimar el esfuerzo necesario para llevar a cabo esta importante tarea del desarrollo del software.

## B.2 Organización de la Tesis

Este documento de tesis se estructura en cuatro partes y varios apéndices. La primera parte se dedica a la presentación de los fundamentos de este trabajo, se describen las pruebas de software, los problemas que afrontamos, las técnicas de optimización metaheurísticas, los algoritmos concretos usados, y la metodología empleada para validar los resultados numéricos. La segunda y tercera parte se dedican a los paradigmas de pruebas de software, que son las pruebas estructurales y funcionales. En la segunda parte trabajamos el problema de generación de pruebas automáticas, proponemos una mediad de complejidad nueva para estimar la dificultad de probar una pieza de código y para finalizar tratamos el problema de generación de pruebas con dos objetivos, considerando la calidad del conjunto de pruebas y el coste de las mismas como igualmente importantes. La tercera parte explora las versiones de pruebas de caja negra del problema de la generación de pruebas, donde no tenemos información estructural del código. En esta parte lidiamos con dos representaciones diferentes de los modelos de programa, los árboles de clasificación y los modelos de características asociados a las líneas de productos software. La cuarta parte resume las principales conclusiones arrojadas a lo largo de este disertación. Al final del volumen se encuentran los apéndices.

- **Parte I. Fundamentos de las Pruebas de Software y las Técnicas Metaheurísticas**
  - El Capítulo 2 presenta los principales conceptos de las pruebas de software, enfatizando las pruebas estructurales y funcionales, los dos paradigmas principales de las pruebas de software. Después de eso, los problemas enfrentados en esta tesis son formalizados para proveer al lector los detalles exactos del problemas que se está resolviendo.

- El Capítulo 3 contiene una descripción general del campo de investigación de la optimización con metaheurísticas, incluyendo una clasificación de las principales técnicas. La última parte de esta capítulo se concentra en el procedimiento de evaluación de los resultados, incluyendo indicadores de calidad y la validación estadística estándar empleada en los experimentos.
- El Capítulo 4 presenta la versión de los algoritmos usados a lo largo de este trabajo de tesis, incluyendo técnicas mono-objetivo y multi-objetivo. Los detalles de implementación específicos (como los operadores de mutación o recombinación) de cada problema (o representación), se posponen a los capítulos correspondientes.

• **Parte II. Pruebas Estructurales**

- El Capítulo 5 se enfrenta a la herencia para la generación de datos de prueba en código orientado a objetos. Este capítulo propone una medida de distancia para el calculo de la distancia de rama en la presencia del operador `instanceof` y dos operadores de mutación basados en esta distancia. Además de estas propuestas, hemos llevado a cabo un conjunto de experimentos para contrastar nuestra hipótesis comparando contra una mutación uniforme.
- El Capítulo 6 contiene una de las investigaciones más interesantes llevadas a cabo en esta tesis, la definición de una nueva medida de complejidad llamada "Branch Coverage Expectation". Esta medida, basada en un modelo de Markov del programa, tiene como objetivo proporcionar conocimiento sobre la dificultad de probar programas. Después de esto, se evalúa con una validación teórica y experimental de la medida usando el marco de trabajo propuesto por Kitchenham et al. [120].
- El Capítulo 7 comienza describiendo dos enfoques para abordar el problema de la generación de datos de prueba multi-objetivo. Se analiza el rendimiento de un enfoque directo multi-objetivo frente a la aplicación de un algoritmo mono-objetivo seguido de una selección de pruebas. Los resultados previos de la literatura sólo se centran en la cobertura del programa mientras que el coste de oráculo ha sido ignorado en los principales estudios previos, por eso nosotros sí que lo consideramos en este trabajo.

• **Parte III. Pruebas Funcionales**

- El Capítulo 8 explora una serie de cuestiones relacionadas con el método de clasificación con árbol. Analizamos la priorización de datos de prueba para probar primero las funcionalidades más importantes. Después, definimos un modelo completo (método de clasificación con árbol extendido) que puede ser usado para la descripción de todos los aspectos necesarios para la generación de secuencia de pruebas. Nuestras propuestas han sido implementadas exitosamente en la herramienta profesional CTE XL, cuestión que da valor adicional a este trabajo.
- El Capítulo 9 aplica las técnicas metaheurísticas a las líneas de productos software. A lo largo de este capítulo completamos varios huecos existentes en la literatura de líneas de productos software: comparamos un algoritmos genético paralelo con un algoritmo del estado del arte con criterio de cobertura de pares como criterio de adecuación, exploramos el efecto de diferentes estrategias de *seeding* y proponemos algoritmos clásicos multi-objetivo y un algoritmo exacto, para computar el frente de Pareto verdadero considerando la calidad del conjunto de pruebas y el coste de oráculo.

- **Parte IV. Conclusiones**
  - El Capítulo 10 contiene una revisión global de este trabajo de tesis y revisita las principales conclusiones extraídas. Los objetivos de la tesis y las principales contribuciones son discutidas en vista de los resultados obtenidos. Finalmente, las lines de trabajo de investigación futura son discutidas someramente.
- **Apéndices**
  - Apéndice A presenta el conjunto de trabajos publicados durante los años en que esta tesis se ha llevado a cabo.
  - Apéndice B es este resumen del volumen en español.

## B.3 Fundamentos

En esta sección se explican unas breves nociones fundamentales de las pruebas de software y sobre las técnicas metaheurísticas.

### B.3.1 Pruebas de software

La generación automática de datos de prueba consiste en obtener un conjunto de pruebas adecuadas sin intervención, por tanto se descarga al ingeniero de la labor de la selección de este conjunto de pruebas. En consecuencia, esta automatización del proceso requiere la selección de esos datos de prueba adecuados por parte de un algoritmo. Esta decisión es un problema de optimización donde el algoritmo tiene que escoger la mejor solución (conjunto de pruebas) de entre un número elevado de soluciones posibles, y que puede ser formulado como un problema de búsqueda [47].

Existen dos paradigmas de pruebas complementarios que se basan en el conocimiento que el ingeniero de pruebas tiene de la estructura interna del programa: pruebas estructurales (caja blanca) y pruebas funcionales (caja negra). El paradigma estructural usa la información sobre cómo esta construido el software para la generación de las pruebas [113]. Esta información estructural generalmente viene desde el grafo de control de flujo del programa, en particular de las estructuras de control (decisiones) que dan lugar a las diferentes ramas del programa. Esta técnica se usa típicamente durante las etapas tempranas de la fase de pruebas donde el programador esta a cargo de la ejecución del conjunto de pruebas [215]. Por otro lado, las pruebas funcionales estan diseñadas sin información de la estructura del código fuente [38,39]. En este paradigma el diseño de los casos de pruebas se basa en el comportamiento externo del software. Esta técnica se usa generalmente cuando el software fue implementado de forma externa, el código fuente no está disponible, o se está probando el sistema completo.

### B.3.2 Metaheurísticas

Las metaheurísticas son estrategias de alto nivel que combinan distintos métodos para explorar el espacio de búsqueda correspondiente a un problema de optimización. Suelen definirse a modo de plantillas que se deben rellenar empleando información específica del problema sobre el cual han de aplicarse (representación de las soluciones, operadores, etc.) y son capaces de abordar problemas cuyos espacios de búsqueda son muy extensos, para los cuales, la utilización de otro tipo de técnicas como las exactas, son inabordables por el coste computacional. Las metaheurísticas pueden clasificarse dentro de dos categorías, según el número de soluciones que manejan de forma



simultánea: las basadas en trayectoria, que tienen una única solución y las basadas en población, que manejan un conjunto de soluciones, o población, de forma simultánea. Algunas metaheurísticas conocidas del primer tipo son el recocido simulado (SA), la búsqueda tabú (TS), búsqueda greedy aleatoria adaptativa (GRASP), la búsqueda de vecindario variable (VNS), o la búsqueda local iterada (ILS). Algunos ejemplos conocidos del segundo tipo son los algoritmos evolutivos (EA), los algoritmos de estimación de distribuciones (EDA), la búsqueda dispersa (SS), la optimización por colonia de hormigas (ACO) y la optimización por cúmulos de partículas (PSO). En esta tesis nos hemos centrado en el diseño y análisis de técnicas metaheurística poblacionales para optimizar problemas con uno o múltiples objetivos.

En optimización multi-objetivo se busca optimizar varios objetivos. Usualmente, dichas funciones están en conflicto entre sí, es decir, una mejora en uno de los objetivos supone un empeoramiento en alguno de los otros. Por esto, y a diferencia de la optimización mono-objetivo, el óptimo no es una única solución, sino un conjunto de soluciones conocido como el óptimo de Pareto, el cual al ser representado en el espacio de objetivos da lugar al llamado frente de Pareto. Cada solución de este conjunto es óptima en el sentido de que no es posible mejorar ninguno de sus objetivos sin empeorar alguno de los demás. A las soluciones pertenecientes al mismo se les suele conocer como soluciones no-dominadas. El objetivo de la optimización multi-objetivo es, por tanto, la obtención del conjunto de soluciones Pareto óptimas. No obstante, esto no siempre es factible; en ese caso el objetivo pasa a ser el obtener una aproximación suficientemente buena del conjunto, es decir, un conjunto de soluciones tal que se cumplen dos propiedades: cercanía al verdadero frente de Pareto, y diversidad de las soluciones a lo largo del frente. Una de las formas de medir la calidad de los frentes generados es mediante el uso de indicadores de calidad, tales como el Hypervolume [236], Spread [57], Generational Distance [207], Epsilon (Multiplicative) [237] o empirical attainment function [121].

## B.4 Problemas Abordados en esta Tesis

En las pruebas de software el ingeniero selecciona un conjunto de configuraciones iniciales para probar un programa, y a continuación se comprueba el comportamiento del software con estas. Para asegurar la corrección de un programa con esta técnica, sería necesario ejecutar el programa con todas las posibles configuraciones, pero esto es inviable en la práctica. La alternativa consiste en probar un programa con un conjunto representativo y adecuado de datos o configuraciones de prueba. El problema de la generación de datos de prueba consiste entonces en la generación de este subconjunto de pruebas que para ser adecuado debe maximizar el criterio de cobertura escogido. En este trabajo nos hemos centrado en la cobertura de ramas, que es el criterio más popular en las pruebas estructurales.

Como hemos comentado, el ingeniero debe comprobar si el comportamiento del sistema software en pruebas es correcto o no, a este esfuerzo se denomina coste de oráculo. Por tanto, otro objetivo importante es la minimización de este coste, que se puede llevar a cabo si minimizamos el tamaño del conjunto de pruebas. Consecuentemente, es obligatorio el balance entre cobertura y el coste de obtener esa cobertura. Como el coste de la fase de pruebas depende del tamaño del conjunto de pruebas, podemos definir la versión multi-objetivo del problema de generación de datos de prueba como un problema donde queremos maximizar la cobertura y minimizar el tamaño del conjunto de pruebas.

En cuanto a las pruebas funcionales, vamos a utilizar el enfoque de pruebas combinatorias de interacción [53]. Este es un enfoque efectivo para la detección de fallos causados por combinaciones concretas de componentes o entradas del sistema. Generalmente esta tarea consiste en generar,

como poco, todas las posibles combinaciones de los valores de los parámetros o características (este problema es NP-duro). El criterio de cobertura más popular utilizado en pruebas combinatorias es de pares (*pairwise*), donde todas las parejas de valores de diferentes parámetros debe estar presentes en al menos un caso de prueba.

La priorización de los casos de prueba pueden revelar fallos en etapas tempranas de la fase de pruebas, lo que es muy importante para reducir el coste de una detección tardía de un fallo. Para obtener un conjunto de pruebas ordenado por prioridades, asignamos a las diferentes características o valores de parámetros un peso según su importancia. A mayor peso, mayor importancia del elemento. Estos pesos nos van a servir para guiar la generación, para cubrir los elementos más importantes en primer lugar. Además, ya que utilizamos el criterio de cobertura de pares, deberemos asignar un peso a cada par, que es calculado como el producto de los respectivos pesos de los elementos. Por tanto, el problema de la generación de un conjunto de pruebas priorizado consiste en encontrar el conjunto de prueba que maximice la cobertura obtenida teniendo en cuenta los pesos de los pares de elementos.

Finalmente, durante esta tesis también hemos abordado el problema de la generación de secuencias de pruebas. Tradicionalmente las pruebas se ejecutan aisladamente, sin embargo, puede haber operaciones asociadas con transiciones en los sistemas software, y ejecutando esas transiciones es la única manera de comprobar el correcto funcionamiento del sistema. Por tanto, estamos interesados en comprobar los diferentes estados del sistema software pero también todas las posibles transiciones. En el problema de generación de secuencias de pruebas utilizamos dos criterios de cobertura a maximizar, la cobertura de estados y la cobertura de transiciones. En ambos casos, intentamos también que el conjunto de pruebas generadas en la secuencia tenga el tamaño mínimo posible.

La generación automática de datos de prueba es uno de los temas más estudiados en la ingeniería del software [95, 147], y los algoritmos evolutivos son las técnicas más utilizadas, por ello a esta unión se la denominan pruebas evolutivas.

## B.5 Generación de Datos de Prueba en Programas Orientados a Objetos

En este trabajo nos hemos centrado en un aspecto esencial del software orientado a objetos, la herencia. Proponemos un enfoque para guiar mejor la búsqueda de datos de prueba en el contexto de las pruebas evolutivas sobre software orientado a objetos. Particularmente, proponemos una medida de distancia para calcular la distancia de rama en presencia del operador `instanceof` del lenguaje Java. En la Figura B.1 ilustramos el cálculo de la distancia entre las clases ArrayList y TreeSet o HashSet. En este caso tenemos dos pasos de jerarquía (*hierarchical walk*) y otros dos pasos de aproximación (*approximation walk*). De esta manera podemos medir la distancia entre la clase propuesta y la clase objetivo del operador *instanceof*.

Adicionalmente hemos propuesto dos operadores de mutación basados en esta distancia definida, uno de ellos adaptativo. En los experimentos comparamos la mutación uniforme, la mutación basada en esta distancia, y un operador adaptativo cuyo comportamiento es un caso intermedio de los otros dos. Los resultados obtenidos en los experimentos nos indican que el ranking no depende del factor de adaptación de la mutación, sino del grado de anidamiento máximo que tienen los programas y el número de condiciones atómicas por decisión. Este parámetro que modifica el comportamiento del operador adaptativo si influye en los programas más complejos, como esperábamos.

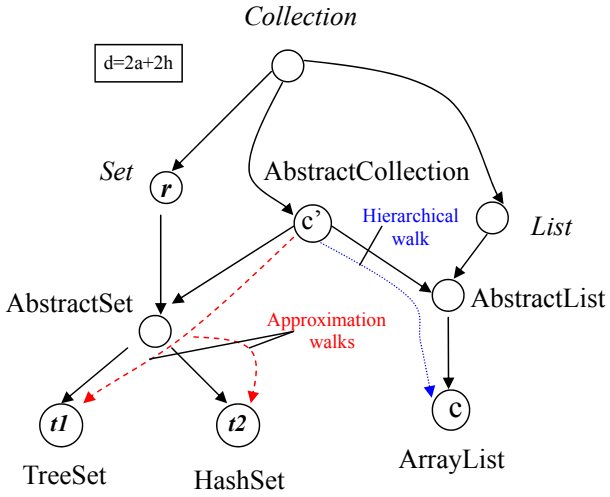



Figure B.1: Ejemplo de distancia entre una clase y un interfaz.

## B.6 Estimando la Complejidad de Probar un Programa

Estudiamos la complejidad de probar un programa desde un punto de vista original: un programa es más complejo si es más difícil probarlo automáticamente. Así que, definimos una medida para calcular la cobertura de ramas esperada, en inglés "Branch Coverage Expectation" (BCE), que proporciona conocimientos sobre la dificultad de probar programas. Los cimientos de esta medida están basados en el modelo de Markov del programa. Este modelo nos proporciona la base teórica que necesitamos. En nuestro caso construimos este modelo de Markov a partir del grafo de control de flujo del programa, donde los estados del modelo de Markov son los bloques básicos (BB) del código del programa.

Definimos BCE como la media de los valores de la esperanza matemática de atravesar de un bloque básico a otro $E[BB_i, BB_j]$ con un valor inferior a $1/2$. Si un programa tiene un valor bajo de BCE, entonces un generador de pruebas aleatorio requeriría un número grande de casos de prueba para obtener cobertura total. La medida BCE esta delimitada en el intervalo $(0, 1/2]$. Formalmente, sea $A$ un conjunto de ejes con $E[BB_i, BB_j] < 1/2$:

$$A = \left\{ (i,j) \middle| E[BB_i, BB_j] < \frac{1}{2} \right\}. \tag{B.1}$$

Entonces, BCE se define como:

$$BCE = \frac{1}{|A|} \sum_{(i,j)\in A} E[BB_i, BB_j]. \tag{B.2}$$

Basado en este modelo del programa, también podemos proporcionar una estimación del número

de casos de prueba aleatorios que deben ser generados para obtener un porcentaje de cobertura concreta. Este resultado se obtiene como la inversa del valor de BCE.

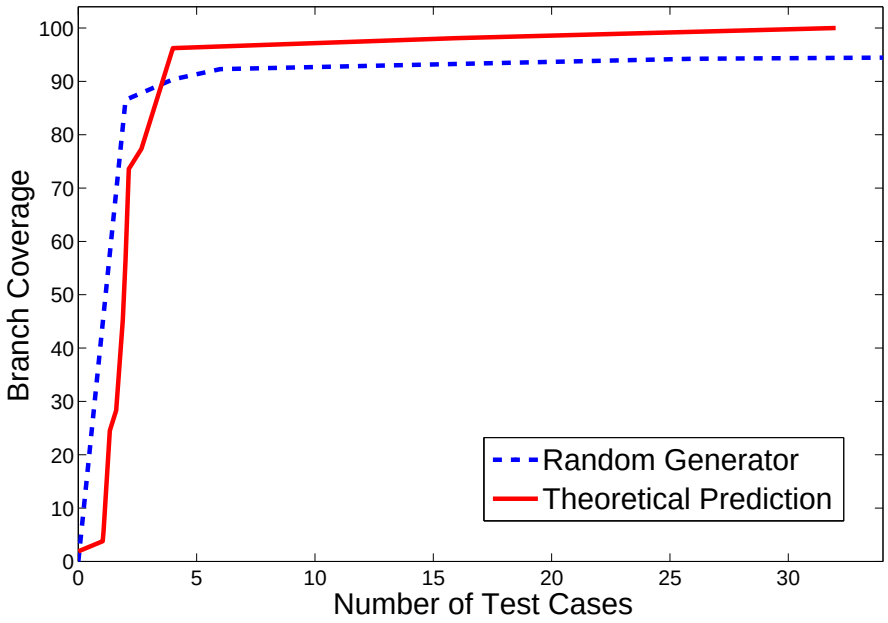



Figure B.2: Cobertura frente a número de casos de prueba de un generador aleatorio y nuestra modelo de estimación

En la Figura B.2 mostramos la cobertura teórica esperada calculada con nuestro enfoque junto con la cobertura real obtenida en 30 ejecuciones independientes de un generador de pruebas aleatorio. Podemos comprobar en la figura que nuestra estimación teórica es similar a los resultados reales.

El análisis de los experimentos indican que esta medida propuesta esta más correlacionada con la cobertura que las otras medidas de código estáticas estudiadas. Esto significa que este es un buen método para estimar la dificultad de probar un programa. Pensamos que, apoyado por los resultados, esta media es también útil para predecir el comportamiento de cualquier generador de datos de prueba automático.

## B.7 Generación de Datos de Pruebas Multi-objetivo

Los trabajos previos en la literatura se han centrado sólo en la cobertura del código, mientras que el coste de oráculo es un coste significativo que ha sido ignorado en la mayoría de los estudios. Nosotros proponemos dos enfoques diferentes para la resolución de este problema teniendo en cuenta estos dos objetivos. El primero es un enfoque multi-objetivo directo y la segunda es la aplicación de un algoritmo mono-objetivo seguido de la selección de casos de prueba.

Nuestro estudio abarca cinco algoritmos multi-objetivo (MOCell, NSGA-II, SPEA2, PAES, y un algoritmo aleatorio multi-objetivo) y tres algoritmos mono-objetivo (GA, ES , y un algoritmo

aleatorio mono-objetivo). Los experimentos se han realizado con 800 programas sintéticos generados por nuestro generador de programas que es totalmente parametrizable para obtener programas realistas, asegurando que el 100% de cobertura es posible. Adicionalmente los resultados son validados con 13 programas reales extraídos de la literatura relacionada.

Table B.1: Ranking de algoritmos acorde a la cobertura e *hypervolume* máximo agrupados por grado de anidamiento.

| Cobertura | | | | | |
|---|---|---|---|---|---|
| Rank | ND 1 | ND 2 | ND 3 | ND 4 | All |
| 1 | GA | GA | GA | GA | GA |
| 2 | ES | ES | ES | ES | ES |
| 3 | MOCell | MOCell | MOCell | NSGA-II | MOCell |
| 4 | NSGA-II | NSGA-II | NSGA-II | MOCell | NSGA-II |
| 5 | SPEA2 | SPEA2 | SPEA2 | SPEA2 | SPEA2 |
| 6 | PAES | PAES | PAES | PAES | PAES |
| 7 | RNDMono | RNDMono | RNDMono | RNDMono | RNDMono |
| 8 | RNDMulti | RNDMulti | RNDMulti | RNDMulti | RNDMulti |
| Hypervolume | | | | | |
| Rank | ND 1 | ND 2 | ND 3 | ND 4 | All |
| 1 | MOCell | MOCell | GA | GA | GA |
| 2 | NSGA-II | GA | MOCell | MOCell | MOCell |
| 3 | ES | NSGA-II | NSGA-II | ES | ES |
| 4 | GA | ES | ES | NSGA-II | NSGA-II |
| 5 | PAES | SPEA2 | SPEA2 | SPEA2 | SPEA2 |
| 6 | SPEA2 | PAES | PAES | PAES | PAES |
| 7 | RNDMono | RNDMono | RNDMono | RNDMono | RNDMono |
| 8 | RNDMulti | RNDMulti | RNDMulti | RNDMulti | RNDMulti |

En términos de convergencia hacia el frente de Pareto óptimo, el GA y MOCell han sido los mejores resolutores en la comparación. MOCell ha obtenido los mejores frentes para programas con grado de anidamiento 1 y 2, valores muy comunes en la práctica. Por otro lado, el GA es el mejor algoritmo para enfrentar programas con alto grado de anidamiento, atendiendo a los indicadores hypervolume y media de cobertura máxima. Adicionalmente, queremos destacar que ambos enfoques son buenos reduciendo el número de casos de prueba necesarios para obtener cierta cobertura. El coste de oráculo puede ser reducido significativamente ya que el enfoque mono-objetivo sólo necesita el 19.32% del límite superior de casos de prueba necesarios para obtener la cobertura máxima, mientras que el enfoque totalmente multi-objetivo es incluso mejor, sólo necesitando un 15.12% de los casos de prueba. Esta mejora justifica el uso de nuestro enfoque para lidiar con el problema de la generación de datos de prueba multi-objetivo.

## B.8 Pruebas Combinatorias usando el Método de Clasificación de Árboles

En las pruebas funcionales no existe una representación estándar del programa a probar. Una de ellas es la clasificación con árboles, cuyo método asociado se basa en el método de particionado en categorías [167]. Este enfoque para pruebas combinatorias particiona el sistema bajo pruebas en parámetros y estos a su vez se separan con valores disjuntos. En este trabajo enfrentamos el problema de priorización de la generación de datos de prueba con criterio pairwise. Comparamos cinco enfoques diferentes, tres de ellos propuestos por nosotros. Hemos realizado experimentos en 32 escenarios diferentes y para distintos pesos, lo que hace nuestro estudio significativo. Tras analizar

estos experimentos, el algoritmo genético propuesto mejora a los demás algoritmos en la mayoría de escenarios, es la mejor opción cuando tenemos restricciones de tiempo o la ejecución de los casos de prueba es costosa. Pero, si el enfoque genético no consigue unos resultados satisfactorios, podríamos usar el algoritmo ávido ya que obtiene buenos resultados en poco tiempo. Finalmente, si el conjunto de pruebas ya se ha calculado, podríamos usar el enfoque que re-ordena las pruebas para ejecutar en primer lugar los casos de prueba más importantes.

Un valor añadido de nuestro trabajo es que hemos implementado nuestras propuestas siguiendo las especificaciones de una herramienta de pruebas funcionales profesional llamada CTE XL. Sin embargo, la generación de secuencias no estaba disponible en la herramienta. Por esto, una de nuestras contribuciones es la definición de una extensión del método de clasificación por árbol para añadir transiciones al modelo. Omitimos de este resumen la definición formal del modelo completo por su extensión. Podemos asegurar que los beneficios del uso de secuencias de pruebas son claros, podemos ahorrar costes y tiempo por ejecutar todos las pruebas secuencialmente ya que la prueba anterior sitúa el sistema software en el estado adecuado para probar la siguiente funcionalidad. Presentamos dos metaheurísticas diferentes para la optimización de la generación de secuencias de pruebas. El primero es un algoritmo genético (GTSG) con operador de memoria, el cual nos permite ahorrar memoria al evaluar las soluciones candidatas, mientras que también permite calcular las soluciones más rápidamente que sin el operador. La segunda propuesta es un algoritmo de colonias de hormigas (ACOts), particularmente proponemos una variante del algoritmo ACOhg [3]. En los experimentos comparamos nuestras propuestas con un algoritmo ávido determinista del estado del arte en 12 modelos de programas diferentes extraídos de la literatura. Después de analizar las soluciones obtenidas por los tres algoritmos, podemos concluir que los enfoques metaheurísticos son significativamente mejores que el algoritmos determinista para los programas más grandes, especialmente el algoritmos ACOts.

## B.9 Líneas de Productos Software

A lo largo de esta tesis hemos completado huecos en la literatura de pruebas funcionales, particularmente podemos destacar nuestras contribuciones en la literatura de líneas de productos software. Nos hemos enfrentado al problema de generación de datos de pruebas priorizados por pares, después hemos utilizado técnicas multi-objetivo aplicadas a las pruebas de líneas de productos, y finalmente presentamos un algoritmo exacto para computar frentes óptimos de Pareto.

Las líneas de productos software se pueden representar por medio de un modelo de características (*feature model*) el conjunto total de productos viables. Por tanto, queremos probar los productos más representativos en primer lugar, siguiendo un criterio de adecuación de pares. Hemos evaluado el comportamiento de un algoritmo genético para la generación de pruebas en un total de 235 modelos de características usando diferentes tipos de priorización de productos. En nuestra comparación evaluamos nuestra propuesta contra el algoritmo pICPL, un algoritmo ávido, en un total de 79,800 ejecuciones independientes. Nuestro análisis muestra que el algoritmo genético genera conjuntos de pruebas de menor tamaño.

También estudiamos el uso de algoritmos evolutivos multi-objetivo aplicados a las pruebas pairwise de líneas de productos. Seleccionamos un conjunto amplio de algoritmos multi-objetivo (NSGA-II, SPEA2, MOCell, PAES, y un algoritmo aleatorio) para estudiar el impacto de las estrategias de *seeding*. Nuestra evaluación muestra unívocamente que el *seeding* con conocimiento del problema produce una mejora significativa en resultados y en tiempo. Conseguimos resultados similares en calidad con NSGA-II, SPEA2 y MOCell, mientras que PAES y el algoritmo aleatorio son significativamente peores.



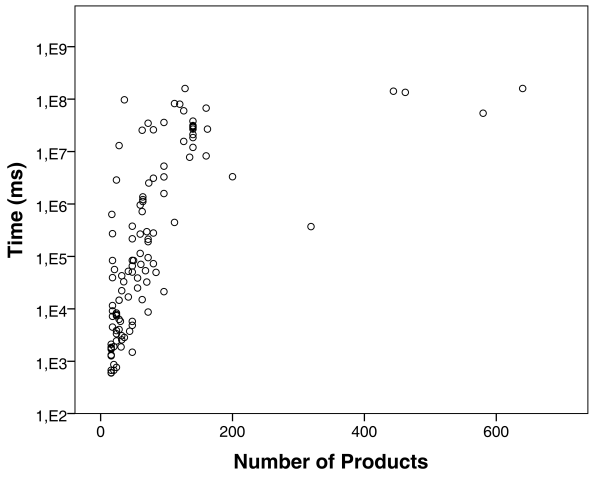



Figure B.3: Tiempo (escala logarítmica) requerido para encontrar el frente óptimo de Pareto frente al número de productos definidos por los modelos de características.

Finalmente hemos propuesto un algoritmo exacto para computar el frente óptimo de Pareto para el problema de generación de datos de pruebas por pares en líneas de productos. Definimos un programa 0-1 lineal, que junto con un algoritmo basado en resolutores SAT, es capaz de obtener frentes óptimos de Pareto. Construir una solución óptima nos permita tener una referencia para medir la calidad de las soluciones obtenidas por los métodos aproximados. La evaluación revela que nuestro enfoque exacto tiene un tiempo de ejecución bastante alto. Este hecho nos lleva a analizar el impacto del número de productos y características definidas por el modelo de características en el tiempo de ejecución del algoritmo. Como resultado, en la Figura B.3 ilustramos que existe una alta correlación del tiempo de ejecución con el número de productos.

## B.10 Conclusiones

Esta tesis propone una variedad de contribuciones al campo de pruebas evolutivas. Hemos abarcados un amplio rango de aspectos relativos a las pruebas de programas: código fuente procedimental y orientado a objetos, paradigmas estructural y funcional, problemas mono-objetivo y multi-objetivo, casos de prueba aislados y secuencias de pruebas, y trabajos teóricos y experimentales. En relación a los análisis llevados a cabo, hemos puesto énfasis en el análisis estadístico de los resultados para evaluar la significancia práctica de los resultados. En resumen, las principales contribuciones de la tesis son:

1. **Definición de una nueva medida de distancia para el operador `instanceof` en programas orientados a objetos**. En este trabajo nos hemos centrado en un aspecto relacionado con el software orientado a objetos, la herencia, para proponer algunos enfoques que pueden ayudar a guiar la búsqueda de datos de prueba en el contexto de las pruebas evolutivas. En particular, hemos propuesto una medida de distancia para computar la distancia de ramas en presencia del operador `instanceof` en programas Java. También hemos propuesto dos operadores de mutación que modifican las soluciones candidatas basadas en la medida

de distancia definida. Adicionalmente a las propuestas algorítmicas, hemos llevado a cabo un conjunto de experimentos para chequear nuestra hipótesis. Primero, hemos analizado los parámetros del algoritmo para seleccionar la mejor configuración. Después de eso, comparamos nuestros operadores con la mutación uniforme. Una de las principales conclusiones de este trabajo es que la dificultad de probar un programa depende del número de condiciones atómicas y del grado de anidamiento.

2. **Definición de una nueva medida de complejidad llamada "Branch Coverage Expectation"**. En este trabajo nos enfrentamos a la complejidad de pruebas desde un punto de vista original: un programa es más complejo si es más difícil de probar de forma automática. Consecuentemente, definimos la "Branch Coverage Expectation" para proporcionar conocimiento sobre la dificultad de probar programas. La fundación de esta medida se basa en el modelo de Markov del programa. El modelo de Markov proporciona fundamentos teóricos. El análisis de esta medida indica que esta más correlacionada con la cobertura de rama que las otras medidas de código estáticas. Esto significa que esto es un buen modo de estimar la dificultad de probar un programa.

3. **Predicción teórica del número de casos de prueba necesarios para cubrir un porcentaje concreto de un programa**. Nuestro modelo de Markov del programa puede ser usado para proporcionar una estimación del número de casos de prueba necesarios para cubrir un porcentaje concreto del programa. Hemos comparado nuestra predicción teórica con la media de las ejecuciones reales de un generador de datos de prueba. Este modelo puede ayudar a predecir la evolución de la fase de pruebas, la cual consecuentemente puede ahorrar tiempo y coste del proyecto completo. Esta predicción teórica podría ser también muy útil para determinar el porcentaje del programa cubierto dados un número de casos de prueba.

4. **Propuesta de enfoques para resolver el problema de generación de datos de prueba multi-objetivo**. En ese capítulo estudiamos el problema de la generación multi-objetivo con el fin de analizar el rendimiento de un enfoque directo multi-objetivo frente a la aplicación de un algoritmo mono-objetivo seguido de una selección de casos de prueba. La fase experimental se desarrolla utilizando 800 programas sintéticos en los cuales se puede conseguir cobertura total, junto con una validación con 13 programas reales. Hemos evaluado cuatro algoritmos multi-objetivo (MOCell, NSGA-II, SPEA2, y PAES) y dos algoritmos mono-objetivo (GA y ES), y dos algoritmos aleatorios. En términos de convergencia hacía el frente de Pareto óptimo, GA y MOCell han sido los mejores resolutores en nuestra comparación. Queremos destacar que el enfoque mono-objetivo, donde se ataca cada rama por separado, es más efectivo cuando el programa tiene un grado de anidamiento alto.

5. **Comparativa de diferentes estrategias de priorización en líneas de productos y árboles de clasificación**. En el contexto de pruebas funcionales hemos tratado el tema de la priorización de casos de prueba con dos representaciones diferentes, modelos de características que representan líneas de productos software y árboles de clasificación. Hemos comparado cinco enfoques relativos al método de clasificación con árboles y dos relativos a líneas de productos, cuatro de ellos propuestos por nosotros. Los resultados nos indican que las propuestas para ambas representaciones basadas en un algoritmo genético son mejores que el resto en la mayoría de escenarios experimentales, es la mejor opción cuando tenemos restricciones de tiempo o coste.

6. **Definición de la extensión del método de clasificación con árbol para la generación de secuencias de pruebas**. Hemos definido formalmente esta extensión para la generación de secuencias de pruebas que puede ser útil para la industria y para la comunidad investigadora. Sus beneficios son claros ya que indudablemente el coste de situar el artefacto bajo pruebas en el siguiente estado no es necesario, a la vez que reducimos significativamente el tamaño de la secuencia utilizando técnicas metaheurísticas. Particularmente nuestra propuesta basada en colonias de hormigas es el mejor algoritmo de la comparativa, siendo el único algoritmo que alcanza la cobertura máxima para todos los modelos y tipos de cobertura.

7. **Exploración del efecto de diferentes estrategias de seeding en el cálculo de frentes de Pareto óptimos en líneas de productos**. Estudiamos el comportamiento de algoritmos clásicos multi-objetivo evolutivos aplicados a las pruebas por pares de líneas de productos. El grupo de algoritmos fue seleccionado para cubrir una amplia y diversa gama de técnicas. Nuestra evaluación indica claramente que las estrategias de *seeding* ayudan al proceso de búsqueda de forma determinante. Cuanta más información se disponga para crear esta población inicial, mejores serán los resultados obtenidos. Además, gracias al uso de técnicas multi-objetivo podemos proporcionar un conjunto de pruebas adecuado mayor o menor, en resumen, que mejor se adapte a sus restricciones económicas o tecnológicas.

8. **Propuesta de técnica exacta para la computación del frente de Pareto óptimo en líneas de productos software**. Hemos propuesto un enfoque exacto para este cálculo en el caso multi-objetivo con cobertura paiwise. Definimos un programa lineal 0-1 y un algoritmo basado en resolutores SAT para obtener el frente de Pareto verdadero. La evaluación de los resultados nos indica que, a pesar de ser un fantástico método para el cálculo de soluciones óptimas, tiene el inconveniente de la escalabilidad, ya que para modelos grandes el tiempo de ejecución sube considerablemente. Tras realizar un estudio de correlaciones, confirmamos nuestras sospechas, existe una alta correlación entre el tiempo de ejecución y el número de productos denotado por el modelo de características del programa.

Uno de los temas que encontramos de mayor interés en esta tesis doctoral es la propuesta de una nueva medida de complejidad para proporcionar cierto conocimiento sobre la dificultad de probar un programa. En realidad, la Universidad de Málaga ha considerado este trabajo como susceptible de ser protegido con el registro de una patente internacional provisional (PCT/ES2015/000100). Como trabajo futuro planeamos avanzar en el diseño del modelo de Markov de un programa, pero sin perder su simplicidad. Además pensamos mejorar la estimación que realizamos cuando los bucles se ejecutan un número indeterminado de veces.

Nuestro trabajo ha abierto varias vías de investigación, las cuales planteamos abordar en un futuro próximo. Una de ellas es la generación de secuencias, donde hemos propuesto diferentes enfoques susceptibles de integrarse en una herramienta de pruebas funcionales comercial llamada CTE XL. Vamos a recabar más modelos reales para mejorar las comparaciones, ya que esta validación es esencial cuando se trabaja con un producto comercial. Aunque hemos obtenido muy buenos resultados con nuestro algoritmo de colonias de hormigas, queremos proponer un algoritmo basado en trayectoria como el SA, que ha obtenido buenos resultados en el contexto de pruebas combinatorias [200].

En relación al algoritmo exacto para el cálculo del frente de Pareto, queremos modificar el programa lineal matemático para reducir el tiempo de ejecución del algoritmo. Hemos observado que algunas restricciones pueden ser redundantes. Por ejemplo, las características esenciales (*core features*) presentes obligatoriamente en todos los productos, no necesitan variables, pues son válidas

en todos los productos. De la misma forma, hay pares de características no válidas por definición que se podrían eliminar [100]. Sin embargo, en pruebas preliminares hemos notado que eliminando restricciones el tiempo de ejecución aumenta, mientras que el aumento de las restricciones podría ayudar al resolutor SAT a ser más rápido. En nuestro trabajo futuro queremos estudiar el balance adecuado de restricciones para reducir el tiempo de ejecución.

Finalmente, planeamos aplicar las técnicas propuestas en estas tesis doctoral a la resolución de problemas asociados a la movilidad inteligente (*Smart Mobility*). Concretamente, estamos interesados en aplicarlos a la resolución del problema de configuraciones de la red semafórica. Vamos a representar el tráfico como un sistema de alta variabilidad (con modelos de características software) y proponer una estrategia de pruebas y validación para los planes de semáforos. Queremos introducir también el concepto de prioridad, para que se prueben primero las configuraciones semafóricas más utilizadas. Por ejemplo, condiciones de seco en algunas ciudades y condiciones más adversas en otras. De esta forma vamos a proporcionar una herramienta para los expertos en movilidad de la ciudad. Un primer artículo siguiendo esta línea de investigación se encuentra en revisión en el momento de esta escritura.

# List of Tables


2.1 Sample feature sets of GPL feature model. . . . . 25

5.1 Average coverage obtained changing $P_M$ and $Rf$ in the most complex programs. . 65
5.2 Average coverage obtained changing $h$ and $a$ in the most complex programs. . . . . 65

6.1 Stationary probabilities and the frequency of appearance of the basic blocks of the piece of code shown In Figure 6.2. . . . . 79
6.2 Parameters of the two EAs used in the experimental section. . . . . 83
6.3 Range of values for some static measures from the two benchmarks of programs. . 86
6.4 Characteristics of the real programs. . . . . 86
6.5 The correlation coefficients among all the measures analyzed in the benchmark 100%CP . . . . . 88
6.6 The correlation coefficients among all the measures analyzed in the benchmark ¬100%CP . . . . . 88
6.7 Correlation coefficient of the most interesting static measures in the 100%CP benchmark. We highlight the highest value per row. . . . . 89
6.8 Correlation coefficient of the most interesting static measures in the ¬100%CP benchmark. We highlight the highest value per row. . . . . 89
6.9 Relationship between the most important static measures and the average branch coverage for all the algorithms. We highlight the highest value of correlation for each algorithm and benchmark. . . . . 90
6.10 Relationship between the nesting degree and the average coverage for all the algorithms. The standard deviation is shown in subscript. We highlight the highest values of branch coverage for each algorithm and benchmark. . . . . 92
6.11 Static measures for a representative program. . . . . 95

7.1 Parameters of the multi-objective EAs used in the experimental section. . . . . 101
7.2 Parameters of the two mono-objective EAs used in the experimental section. . . . 102
7.3 Characteristics of the real programs. . . . . 103
7.4 Programs in which the median hypervolume of one algorithm is better than the others. 105
7.5 Number of programs where there exists significant difference among the HV obtained. 105
7.6 Relationship between the nesting degree and the average maximum coverage for the multi-objective algorithms. The standard deviation is shown in subscript. . . . . 107
7.7 Number of programs where there exists a significant difference among the coverage values obtained. . . . . 107
7.8 Programs in which the median hypervolume of one algorithm is better than the others. 108

7.9 Programs where a significant difference exists among the HV obtained. . . . . 108
7.10 Relationship between the nesting degree and the average maximum coverage for the mono-objective algorithms. The standard deviation is shown in subscript. . . . . 110
7.11 Number of programs where there exists a significant difference between the coverage obtained. . . . . 110
7.12 Programs in which the median hypervolume of one algorithm is better than the others. 111
7.13 Programs where a significant difference exists among the HV obtained. . . . . 111
7.14 Relationship between the nesting degree and the average coverage for all the algorithms. The standard deviation is shown in subscript. . . . . 113
7.15 Number of programs where a significant difference exists among the coverage obtained. 113
7.16 Real programs in which the median hypervolume of one algorithm is better than the others and average maximum coverage of all the real programs. . . . . 114
7.17 Real programs where a significant difference exists among the HV obtained. . . . . 115
7.18 Number of real programs where a significant difference exists among the coverage obtained. . . . . 115
7.19 Ranking of algorithms according to maximum coverage and hypervolume grouped by nesting degree. . . . . 117

8.1 Scenarios and number of factors. . . . . 125
8.2 Number of test cases needed for the GA, PPC, and PPS algorithms in eight scenarios and for four distributions. When significant differences exist between the GS and other algorithm we add an asterisk. . . . . 126
8.3 Number of times that one algorithm is better than the other two for each instance. 128
8.4 Number of observations where there exists significant difference among the GS, the PPC and PPS algorithms. . . . . 128
8.5 Number of test cases needed for the GA, DDA, and BDD algorithms in eight scenarios and for four distributions. When significant differences exist between the GS and other algorithm we add an asterisk. . . . . 129
8.6 Number of observations where there exists significant difference among the GS, the DDA, and BDD algorithms. . . . . 130
8.7 General characteristics of the benchmark of programs. . . . . 137
8.8 Parameters setting for our proposals. The parameter's values used in the experimentation are highlighted in bold. . . . . 137
8.9 Results for test sequence generation for class coverage. . . . . 138
8.10 Vargha and Delaney's statistical test results ($\hat{A}_{12}$) for class coverage. $A$ represents algorithms in rows and $B$ represents algorithms in columns. . . . . 138
8.11 Results for test sequence generation for transition coverage . . . . . 139
8.12 Vargha and Delaney's statistical test results ($\hat{A}_{12}$) for transition coverage. $A$ represents algorithms in rows and $B$ represents algorithms in columns. . . . . 139

9.1 Summary of the case studies measured values. . . . . 148
9.2 Evaluation case studies summary. . . . . 150
9.3 Mean and standard deviation of 30 independent runs for G1 (significant differences are highlighted). . . . . 151
9.4 Mean and standard deviation of 30 independent runs for G2 (significant differences are highlighted). . . . . 151



9.5 Group G3. When considering array sizes PPGS is statistically better than pICPL in 69 cases, and pICPL is better in 18 cases. . . . . 152
9.6 $\hat{A}_{12}$ statistical test results for all groups. PPGS yields better test suite size values. 153
9.7 Feature models summary. . . . . 157
9.8 Comparison of multi-objective algorithms using the proposed quality indicators and performance time. . . . . 158
9.9 Comparison of seeding strategies using hypervolume, generational distance, and performance time. . . . . 158
9.10 $\hat{A}_{12}$ statistical test results for all algorithms. NSGA-II yields better results for HV and time measures. . . . . 159
9.11 $\hat{A}_{12}$ statistical test results for seeding strategies. SO yields better quality indicators and time values. . . . . 159

B.1 Ranking de algoritmos acorde a la cobertura e *hypervolume* máximo agrupados por grado de anidamiento. . . . . 189

# List of Figures


2.1 Examples of dominated and non-dominated solutions. . . . . 15
2.2 *Video Game System* test object. . . . . 18
2.3 Video game ECTM example. . . . . 21
2.4 Graph Product Line feature model. . . . . 24

3.1 Classification of metaheuristics . . . . . 32
3.2 Examples of Pareto fronts with different behavior of convergence and diversity. . . 38
3.3 Example of hypervolume. . . . . 39
3.4 Examples of attainment surfaces. . . . . 42
3.5 Statistical validation procedure for experimental results . . . . . 42

5.1 Instanceof expression in a sentence. . . . . 58
5.2 The test data generation process. . . . . 59
5.3 Example of distance between a class and an interface. . . . . 62
5.4 Representation of one solution vector $\vec{o}$. . . . . 63
5.5 Fitness evolution with a uniformly initialized population. . . . . 67
5.6 Fitness evolution with a population near the objective solution. . . . . 67
5.7 Average number of evaluations required for 100% branch coverage in all the test programs for different values of $\lambda$. . . . . 69

6.1 The original graph of the McCabe's article. . . . . 75
6.2 A piece of code to illustrate the computation of Branch Coverage Expectation. . . 79
6.3 The CFG and the probabilities used to build a Markov Chain of the piece of code of Figure 6.2. . . . . 80
6.4 The CFG and the expectations of traversing each branch in the piece of code of Figure 6.2. . . . . 81
6.5 Illustration of the predicates transformation. . . . . 85
6.6 Boxplots showing the branch coverage against the Density of Decisions for GA in ¬100%CP. . . . . 91
6.7 Boxplots showing the branch coverage against the DLOCE for GA in ¬100%CP. . 91
6.8 Average Branch Coverage of RND against the BCE measure. . . . . 93
6.9 Coverage against the number of test cases of the random generator and the theoretical model. . . . . 94
6.10 Average Branch Coverage of GA against the Branch Coverage Expectation for the real programs. . . . . 95

7.1 The general scheme of the two proposed approaches. . . . . 100
7.2 50%-attainment surfaces: coverage against the number of test cases. . . . . 106
7.3 50%-attainment surfaces: coverage against the number of test cases. . . . . 109
7.4 50%-attainment surfaces: coverage against the number of test cases of all the algorithms. . . . . 112
7.5 50%-attainment surfaces: coverage against the number of test cases for the program *line*. . . . . 116

8.1 CTE XL professional tool. . . . . 123
8.2 Median solutions and interquartile range of ACOts, GTSG and Greedy algorithms for the Citizen example for Class Coverage. Coverage versus number of test cases in the solution. . . . . 140
8.3 Median solutions and interquartile range of ACOts, GTSG and Greedy algorithms for the Citizen example for Transition Coverage. Coverage versus number of test cases in the solution. . . . . 141

9.1 Approximated Pareto front obtained by NSGA-II in TankWar. . . . . 160
9.2 Pareto optimal front for our running example (GPL). . . . . 163
9.3 Time (log scale) required to find optimal Pareto set against the number of products of the feature models. . . . . 164

B.1 Ejemplo de distancia entre una clase y un interfaz. . . . . 187
B.2 Cobertura frente a número de casos de prueba de un generador aleatorio y nuestra modelo de estimación . . . . . 188
B.3 Tiempo (escala logarítmica) requerido para encontrar el frente óptimo de Pareto frente al número de productos definidos por los modelos de características. . . . . 191

# List of Algorithms


1 Pseudocode of Evolutionary Algorithms. . . . . 46
2 Pseudocode of ACO. . . . . 48
3 Pseudocode of NSGA-II. . . . . 50
4 Pseudocode of SPEA2. . . . . 50
5 Pseudocode of MOCell. . . . . 51
6 Pseudocode of PAES. . . . . 52
7 Pseudocode of RNDMulti. . . . . 52
8 Pseudocode of the Heuristic Rate algorithm. . . . . 135
9 Size-Based Random Seeding Strategy. . . . . 154
10 Seed Population. . . . . 154
11 Greedy Seeding Strategy. . . . . 155
12 Algorithm for obtaining the optimal Pareto set. . . . . 162




# Index


ACO Test Sequence, 134
Adequacy Criterion, 11
Ant Colony Optimization, 31

Branch Coverage, 13
Branch Coverage Expectation, 72

Class Coverage, 22
Classification Tree Method, 17
Combinatorial Interaction Testing, 17

Diversity, 30

Empirical Attainment Function, 41
Estimation of Distribution Algorithms, 35
Evolutionary Algorithms, 31, 34
Exploitation, 30
Exploration, 30
Extended Classification Tree Method, 20

Feature Model, 23
Flag Problem, 58
Function
    Fitness function, 13
    Objective function, 13
Functional Testing, 15

Generational Distance, 40
Genetic Test Sequence Generator, 132
GRASP, 33

Heuristics
    *ad hoc*, 30
    constructive, 30
Hypervolume, 39

InstanceOf Operator, 57
Intensity, 30

Iterated Local Search, 31, 34

Local optimum, 30
Local search, 30

Mathematical Linear Program, 161
Metaheuristic, 30
Metaheuristics
    formal definition, 31
    population based, 32, 34
    trajectory based, 32, 33
MM Aproach, 100
mM Aproach, 101
Mono-objective Algorithms
    Ant Colony Optimization, 48
    Evolutionary Strategies, 47
    Genetic Algorithms, 45
Multi-objective Algorithms
    Multi-Objective Cellular Genetic Algorithm, 50
    Non-dominated Sorting Genetic Algorithm-II, 49
    Pareto Archived Evolution Strategy, 51
    Random Search Multi-Objective Algorithm, 52
    Strength Pareto Evolutionary Algorithm, 49
Multi-objective Problem, 14

Neighbourhood
    in a local search method, 30

Object-Oriented, 57
Optimization problem
    binary, 13
    continuous, 13
    definition, 13

heterogeneous, 13
integer, 13
Optimization techniques
approximate, 30
exact, 29

Pairwise Coverage, 19
Pairwise Testing, 15
Parallel Prioritized product line Genetic Solver, 146
Particle Swarm Optimization, 31
Prioritized Genetic Solver, 124
Prioritized Pairwise Coverage, 20
Problems
Multi-Objective Test Data Generation Problem, 14
Multi-Objective Test Data Generation Problem in SPL, 26
Pairwise Test Data Generation Problem in SPL, 23
Prioritized Pairwise Test Data Generation Problem with Classification Tree Method, 17
Test Data Generation Problem, 11
Test Sequence Generation Problem, 20

Scatter Search, 35
Seeding Strategies, 153
Greedy Seeding, 155
Single-objective Based Seeding, 155
Size-Based Random Seeding, 154
Simulated Annealing, 31, 33
Single Point Crossover, 46
Software Product Line, 23
Spread, 39
Statement Coverage, 12
Statistical Tests, 41
$\widehat{A}_{12}$ statistic, 43
ANOVA Test, 42
Homoskedasticity, 42
Kolmogorov-Smirnov Test, 42
KruskalWallis Test, 42
Levene Test, 42
T-Test, 42
Wilkoxon Test, 42
Structural Testing, 10
Swarm Intelligence, 35

Tabu Search, 31, 33
Transition Coverage, 22

Uniform Crossover, 46
Uniform Mutation, 46

Variable Neighbourhood Search, 31, 34

# References


[1] M. A. Ahmed and I. Hermadi. GA-based Multiple Paths Test Data Generator. *Computers & Operations Research*, 35(10):3107–3124, 2008.

[2] E. Alba, C. Blum, P. Isasi, C. León, and J. A. Gómez. *Optimization techniques for solving complex problems*. Wiley, New Jersey, USA, May 2009.

[3] E. Alba and F. Chicano. Finding Safety Errors with ACO. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 1066–1073, London, UK, July 2007. ACM Press.

[4] E. Alba, F. Chicano, M. Ferreira, and J. Gomez-Pulido. Finding deadlocks in large concurrent java programs using genetic algorithms. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 1735–1742, New York, NY, USA, 2008. ACM.

[5] E. Alba and J. F. Chicano. Software Testing with Evolutionary Strategies. In *Rapid Integration of Software Engineering Techniques*, volume 3943, pages 50–65. Springer Berlin Heidelberg, 2006.

[6] E. Alba and B. Dorronsoro. *Cellular Genetic Algorithms*, volume 42 of *Operations Research/Computer Science Interfaces*. Springer-Verlag Heidelberg, 2008.

[7] E. Alba and G. Luque. *Parallel genetic algorithms*, volume 367 of *Studies in Computational Intelligence*. Springer-Verlag, 2011.

[8] N. Aleb and S. Kechid. Automatic Test Data Generation using a Genetic Algorithm. In *Proceedings of the International Conference on Computational Science and Its Applications*, volume 7972, pages 574–586, Ho Chi Minh City, Vietnam, 24-27 June 2013. Springer.

[9] S. Ali, L. Briand, A. Arcuri, and S. Walawege. An Industrial Application of Robustness Testing Using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms. In *International Conference on Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2011.

[10] S. Ali, L. Briand, H. Hemmati, and R. K. Panesar-Walawege. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, Nov. 2010.

[11] M. Alshraideh and L. Bottaci. Search-based software test data generation for string data using program-specific search operators: Research Articles. *Software Testing, Verification and Reliability*, 16(3):175–203, 2006.

[12] M. A. Alshraideh, B. A. Mahafzah, H. S. E. Salman, and I. Salah. Using Genetic Algorithm as Test Data Generator for Stored PL/SQL Program Units. *Journal of Software Engineering and Applications*, 6:65–73, 2013.

[13] S. Amland. Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software*, 53(3):287–295, 2000.

[14] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[15] A. Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11:3494–3514, 2011.

[16] A. Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, 23(2):119–147, 2011.

[17] A. Arcuri and L. Briand. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24:219–250, Nov. 2012.

[18] A. Arcuri, M. Z. Iqbal, and L. Briand. Random Testing: Theoretical Results and Practical Implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, 2012.

[19] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[20] ArgoUML-SPL Project - URL: http://argouml-spl.stage.tigris.org/, 2015.

[21] F. Arito, F. Chicano, and E. Alba. On the Application of SAT Solvers to the Test Suite Minimization Problem. In *Symposium on Search-based Software Engineering*, volume 7515 of *Lecture Notes in Computer Science*, pages 45–59. Springer, 2012.

[22] T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, Oxford, UK, 1996.

[23] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, New York NY, 1997.

[24] E. Bagheri and D. Gasevic. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19(3):579–612, 2011.

[25] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary Testing in the Presence of Loop–Assigned Flags: A Testability Transformation Approach. In *International Symposium on Software Testing and Analysis*, pages 108–118, 2004.

[26] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 1–8, 2002.

[27] V. Basili. Quantitative software complexity models: A panel summary. Tutorial on Models and methods for software Management and Engineering. *IEEE Computer society press*, pages 243—245, 1980.

[28] V. Basili and B. Perricone. Software errors and complexity: an empirical investigation. *ACM commun*, 27(1):42–52, 1984.

[29] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[30] A. Bertolino and M. Marré. How many paths are needed for branch testing? *Journal of Systems and Software*, 35(2):95–106, 1996.

[31] H.-G. Beyer, H.-G. Beyer, H.-P. Schwefel, and H.-P. Schwefel. Evolution strategies A comprehensive introduction. *Natural Computing*, 1(1):3 – 52, 2002.

[32] R. Black. *Advanced Software Testing - Vol. 1: Guide to the ISTQB Advanced Certification As an Advanced Test Analyst (Rockynook Computing)*. Rocky Nook, 2008.

[33] R. Blanco, J. Fanjul, and J. Tuya. Test case generation for transition-pair coverage using Scatter Search. *Journal of Software Engineering and Its Applications*, 4(4):37–56, 2010.

[34] R. Blanco, J. Tuya, and B. Adenso-Díaz. Automated test data generation using a Scatter Search approach. *Information and Software Technology*, 51(4):708–720, 2009.

[35] C. Blum and A. Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[36] B. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece. *Software cost estimation with COCOMO II*. Prentice-Hall, 2000.

[37] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960–970, 2006.

[38] O. Buehler and J. Wegener. Evolutionary Functional Testing of an Automated Parking System. In *Proceedings of the International Conference on Computer, Communication and Control Technologies*, pages 1–8, Orlando, Florida, 2003.

[39] O. Buehler and J. Wegener. Evolutionary Functional Testing of a Vehicle Brake Assistant System. In *Proceedings of the Metaheuristic International Conference*, pages 157–162, Vienna, Austria, 2005.

[40] C. J. Burgess and M. Lefley. Can Genetic Programming Improve Software Effort Estimation? A Comparative Evaluation. *Information and Software Technology*, 43(14):863–873, 2001.

[41] E. Burke, J. P. Newall, and R. F. Weare. Initialization strategies and diversity in evolutionary timetabling. *Evolutionary Computation*, 6(1):81–103, 1998.

[42] I. Burnstein. *Practical software testing: a process-oriented approach*, volume XLIX. Springer, 2003.

[43] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[44] K. Chang, J. Cross, W. Carlisle, and S. Liao. A Performance Evaluation of Heuristics-based Test Case Generation Methods for Software Branch Coverage. *International Journal of Software Engineering and Knowledge Engineering*, 6(4):585–608, 1996.

[45] F. Chicano. *Metaheurísticas e Ingeniería del Software*. PhD thesis, University of Malaga, 2007.

[46] H. Cichos, S. Oster, M. Lochau, and A. Schürr. Model-Based Coverage-Driven Test Suite Generation for Software Product Lines. In *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2011.

[47] J. Clarke, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating Software Engineering as a Search Problem. *IEEE Proceedings of Software Engineering*, 150(3):161–175, 2003.

[48] M. Clerc. *Particle Swarm Optimization*. Wiley, 2010.

[49] C. Coello Coello. Evolutionary Multi-Objective Optimization Website. URL: http://delta.cs.cinvestav.mx/˜ccoello/EMOO/.

[50] C. Coello Coello, G. B. Lamont, and D. Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer Series, 2007.

[51] D. Cohen and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 129–139, New York, USA, 2007. ACM.

[52] M. Cohen, M. B. Dwyer, and J. Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[53] M. Cohen, J. Snyder, and G. Rothermel. Testing across configurations: implications for combinatorial testing. *SIGSOFT Software Engineering Notes*, 31(6):1–9, 2006.

[54] M. Conrad. Systematic Testing of Embedded Automotive Software - The Classification-Tree Method for Embedded Systems. In *Perspectives of Model-Based Testing*, pages 1–12. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[55] B. Curtis, S. B. Sheppard, and P. Milliman. *Third time charm: Stronger prediction of programmer performance by software complexity metrics*. IEEE Press, Piscataway, NJ, USA, 1979.

[56] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira. A systematic mapping study of software product lines testing. *Information & Software Technology*, 53(5):407–423, 2011.

[57] K. Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, Inc., Aug. 2001.

[58] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm : NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.



[59] T. DeMarco. *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.

[60] E. Díaz, R. Blanco, and J. Tuya. Applying Tabu and Scatter Search to automated software test case generation. In *Proceedings of the Metaheuristic International Conference*, pages 290–297, Vienna, Austria, 2005.

[61] E. Díaz, R. Blanco, and J. Tuya. Tabu search for automated loop coverage in software testing. In *Proceedings of the International Conference on Knowledge Engineering and Decision Support*, pages 229–234, Porto, 2006.

[62] E. Díaz, J. Tuya, R. Blanco, and J. J. Dolado. A Tabu Search algorithm for structural software testing. *Computers & Operations Research*, 35(10):3052–3072, 2008.

[63] M. Dixon. An Objective Measure of Code Quality. *Technical Report*, pages 1–6, 2008.

[64] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[65] M. Dorigo and T. Stützle. *Ant Colony Optimization*. The MIT Press, 2004.

[66] J. J. Durillo, A. J. Nebro, F. Luna, B. Dorronsoro, and E. Alba. jMetal: A Java Framework for Developing Multi-Objective Optimization Metaheuristics. Technical Report ITI-2006-10, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, E.T.S.I. Informática, Campus de Teatinos, 2006.

[67] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb. 2002.

[68] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a Cost-Effective Test Case Prioritization Technique. *Software Quality Control*, 12(3):185–210, 2004.

[69] E. Engström and P. Runeson. Software product line testing - A systematic mapping study. *Information & Software Technology*, 53(1):2–13, 2011.

[70] F. Ensan, E. Bagheri, and D. Gasevic. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *CAiSE*, volume 7328 of *Lecture Notes in Computer Science*, pages 613–628. Springer, 2012.

[71] FeatureHouse website - URL: http://fosd.de/fh, 2015.

[72] T. A. Feo and M. G. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[73] J. Ferrer, F. Chicano, and E. Alba. Dealing with inheritance in OO evolutionary testing. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 1665–1672, New York, USA, July 2009. ACM Press.

[74] J. Ferrer, F. Chicano, and E. Alba. Evolutionary algorithms for the multi-objective test data generation problem. *Software Practice and Experience*, 42(11):1331–1362, 2012.

[75] J. Ferrer, P. M. Kruse, F. Chicano, and E. Alba. Evolutionary algorithm for prioritized pairwise test data generation. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 1213–1220, New York, USA, July 2012. ACM.



[76] G. Fraser and A. Arcuri. The Seed is Strong: Seeding Strategies in Search-Based Software Testing. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 121–130, Washington, DC, USA, 2012. IEEE Computer Society.

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

[78] G. Fraser and A. Zeller. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[79] G. Fryer. Scientific Method in Practice. *EOS Transactions*, 84:357, 2003.

[80] J. P. Galeotti, G. Fraser, and A. Arcuri. Extending a Search-Based Test Generator with Adaptive Dynamic Symbolic Execution (Tool paper). In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 421–424, New York, USA, 2014. ACM.

[81] S. García, D. Molina, M. Lozano, and F. Herrera. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the CEC'2005. *Journal of Heuristics*, 15(6):617–644, 2009.

[82] B. J. Garvin, M. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011.

[83] M. R. Girgis. Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm. *Computer*, 11(6):898–915, 2005.

[84] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, May 1986.

[85] F. Glover. A Template for Scatter Search and Path Relinking. In *Selected Papers from the European Conference on Artificial Evolution*, pages 3–54, London, UK, 1998. Springer-Verlag.

[86] F. Glover. *Handbook of Metaheuristics*. Kluwer, 2003.

[87] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[88] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co, Boston, MA, USA, 1st edition, 1989.

[89] D. Gong and Y. Zhang. Generating Test Data for Both Path Coverage and Fault Detection using Genetic Algorithms. *Frontiers of Computer Science*, 7(6):822–837, December 2013.

[90] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification*. Addison Wesley, third edition, 2005.

[91] M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[92] W. J. Gutjahr. First steps to the runtime complexity analysis of ant colony optimization. *Computers and Operations Research*, 35:2711–2727, 2008.



[93] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[94] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[95] M. Harman. The Current State and Future of Search Based Software Engineering. In *Proceedings of International Conference on Software Engineering / Future of Software Engineering*, pages 342–357, Minnesota, USA, 2007. IEEE Computer Society.

[96] M. Harman and L. Hu. Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1359–1366. Morgan Kaufmann Publishers, 2002.

[97] M. Harman and B. Jones. Search-based Software Engineering. *Information & Software Technology*, 43(14):833–839, 2001.

[98] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo. Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem. In *Proceedings of the International Workshop on Search-Based Software Testing*, pages 182–191, Paris, France, 2010. IEEE.

[99] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11, 2012.

[100] E. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. Using feature model knowledge to speed up the generation of covering arrays. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems*, page 16. ACM, 2013.

[101] S. Helke. *Verifikation von Statecharts durch struktur- und eigenschaftserhaltende Datenabstraktion*. PhD thesis, Technische Universitt Berlin, 2007.

[102] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-wise Test Suites for Large Software Product Lines. *CoRR*, abs/1211.5, 2012.

[103] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Multi-objective test generation for software product lines. In *Proceedings of the International Software Product Lines conference*, pages 62–71. ACM, 2013.

[104] A. Hervieu, B. Baudry, and A. Gotlieb. PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models. In *Proceeding of the IEEE International Symposium on Software Reliability Engineering*, pages 120–129, 2011.

[105] D. S. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Co., Boston, MA, USA, 1997.

[106] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, volume 1. MIT Press, 1975.

[107] T. Honglei, S. Wei, and Z. Yanan. The Research on Software Metrics and Software Complexity Metrics. In *International Forum on Computer Science-Technology and Applications*, volume 1, pages 131–136, 2009.



[108] IBM. Rhapsody examples - URL: https://www.ibm.com, 2013.

[109] M. F. Johansen. SPLCA Tool - URL: http://martinfjohansen.com/splcatool/, 2012.

[110] M. F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the International Software Product Lines conference*, pages 46–55, 2012.

[111] M. F. Johansen, O. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-product Lines. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2012.

[112] B. Jones and H. Sthamer. The automatic generation of software test data sets using adaptive search techniques. In *IEEE Transactions on Information and Communication Technologies*, pages 435–444, 1995.

[113] B. Jones, H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[114] C. Kaner, C. Cem, and K. All. The Impossibility of Complete Testing, 1997.

[115] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[116] J. P. Kelly. *Meta-Heuristics: Theory and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[117] J. Kennedy and R. Eberhart. *Swarm Intelligence*. Morgan Kaufmann Publishers, San Francisco, California, 2001.

[118] T. M. Khoshgoftaar and J. C. Munson. Predicting Software Development Errors Using Software Complexity Metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, 1990.

[119] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220, 4598(4598):671–680, May 1983.

[120] B. Kitchenham, S. L. Pfleeger, and Z. C. Society. Towards a Framework for Software Measurement Validation. *IEEE Transactions on Software Engineering*, 21:929–944, 1995.

[121] J. Knowles. A summary-attainment-surface plotting method for visualizing the performance of stochastic multiobjective optimizers. In *The International Conference on Intelligent Systems Design and Applications*, pages 552–557. IEEE Computer Society, 2005.

[122] J. Knowles and D. Corne. The Pareto archived evolution strategy: A new baseline algorithm for Pareto multiobjective optimisation. In *Evolutionary Computation, 1999*, volume 1, pages 98–105, Mayflower Hotel, Washington D.C., USA, 1999.

[123] J. Knowles, L. Thiele, and E. Zitzler. A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. TIK Report 214, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, 2006.



[124] H. Kobayash, B. L. Mark, and W. Turin. *Probability, Random Processes, and Statistical Analysis*. Cambridge University Press, 2011.

[125] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[126] P. M. Kruse and J. Wegener. Test Sequence Generation from Classification Trees. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 539–548. IEEE, 2012.

[127] R. Kuhn, Y. Lei, and R. Kacker. Practical Combinatorial Testing: Beyond Pairwise. *IT Professional*, 10(3):19–23, May 2008.

[128] K. Lakhotia, M. Harman, and H. Gross. AUSTIN: An open source tool for search based software testing of C programs. *Information and Software Technology*, 55(1):112–125, 2013.

[129] K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *Proceeding of the annual Conference on Genetic and Evolutionary Computation*, pages 1098–1105, New York, NY, USA, 2007. ACM.

[130] S. Lam, M. Raju, U. M, S. Ch, and P. Srivastav. Automated Generation of Independent Paths and Test Suite Optimization Using Artificial Bee Colony. *Procedia Engineering*, 30:191–200, 2012.

[131] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[132] E. Lehmann and J. Wegener. Test Case Design by Means of the CTE XL. In *EuroSTAR 2000*, pages 1–10, Kopenhagen, Denmark, 2000.

[133] X. Liu, B. Wang, and H. Liu. Evolutionary search in the context of object oriented programs. In *Proceedings of the Metaheuristics International Conference*, Vienna, 2005.

[134] R. E. Lopez-Herrejon and D. S. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *International Conference on Generative and Component-Based Software Engineering*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24. Springer, 2001.

[135] R. E. Lopez-Herrejon, F. Chicano, J. Ferrer, A. Egyed, and E. Alba. Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing. In *IEEE International Conference on Software Maintenance*, pages 404–407, Sept. 2013.

[136] J. Lozano, P. Larrañaga, P. Inza, and E. Bengoetxea. *Towards a New Evolutionary Computation. Advances in Estimation of Distribution Algorithms*. Springer Verlag, 2006.

[137] P. J. Lucas. *An Object-Oriented Language System For Implementing Concurrent, Hierarchical, Finite State Machines*. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1989.

[138] G. Luque. *Resolución de Problemas Combinatorios con Aplicación Real en Sistemas Distribuidos*. PhD thesis, University of Malaga, 2006.

[139] N. Malevris and D. F. Yates. The collateral coverage of data flow criteria when branch testing. *Information and Software Technology*, 48(8):676–686, 2006.



[140] C. Mao. Generating Test Data for Software Structural Testing Based on Particle Swarm Optimization. *Arabian Journal for Science and Engineering*, 39(6):4593–4607, June 2014.

[141] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu. Practical pairwise testing for software product lines. In T. Kishi, S. Jarzabek, and S. Gnesi, editors, *Proceedings of the International Software Product Lines conference*, pages 227–235. ACM, 2013.

[142] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.

[143] T. MathWorks. Matlab Simulink Stateflow - URL: http://es.mathworks.com/products/stateflow/.

[144] S. F. A. Mats Grindal, Jeff Offutt, M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Software Testing, Verification, and Reliability*, 15:167–199, 2005.

[145] P. S. May. *Test Data Generation: Two Evolutionary Approaches to Mutation Testing*. PhD thesis, Computing Laboratory, 2007.

[146] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

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

[148] P. McMinn. Search-Based Software Testing: Past, Present and Future. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 153–163, 2011.

[149] P. McMinn, D. Binkley, and M. Harman. Testability Transformation for Efficient Automated Test Data Search in the Presence of Nesting. In *Proceedings of the UK Software Testing Workshop*, pages 165–182, 2005.

[150] P. McMinn, D. Binkley, and M. Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering and Methodology*, 18(3):1–27, May 2009.

[151] P. McMinn and M. Holcombe. The State Problem for Evolutionary Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2724, pages 2488–2498, Chicano, Illinois, USA, 2003. Springer-Verlag.

[152] P. McMinn, M. Stevenson, and M. Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the First International Workshop on Software Test Output Validation*, pages 1–4, New York, New York, USA, July 2010. ACM Press.

[153] M. Mendoza. SPLAR library - URL: https://code.google.com/p/splar, 2013.

[154] C. C. Michael, G. McGraw, and M. A. Schatz. Generating Software Test Data by Evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[155] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer-Verlag New York, Inc., New York, NY, USA, 2004.



[156] U. Mirosamek. Two orthogonal regions (main keypad and numeric keypad) of a computer keyboard., 2009.

[157] N. Mladenovic and P. Hansen. Variable Neighborhood Search. *Computers And Operations Research*, 24(11):1097–1100, 1997.

[158] G. Myers, T. Badgett, and C. Sandler. *The Art of Software Testing*. John Wiley and Sons, New York, 2011.

[159] A. S. Namin and J. H. Andrews. The Influence of Size and Coverage on Test Suite Effectiveness. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 57–68, New York, NY, USA, 2009. ACM.

[160] U. Naumann and O. Schenk. *Combinatorial Scientific Computing*. Chapman & Hall/CRC, 1st edition, 2012.

[161] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba. Design Issues in a Multiobjective Cellular Genetic Algorithm. In *Proceedings of International Conference on Evolutionary Multi-Criterion Optimization*, volume 4403 of *Lecture Notes in Computer Science*, pages 126–140. Springer, Mar. 2007.

[162] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba. MOCell: A cellular genetic algorithm for multiobjective optimization. *International Journal of Intelligent Systems*, 24(7):726–746, 2009.

[163] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11:1–11:29, Feb. 2011.

[164] A. F. Nogueira. Predicting software complexity by means of evolutionary testing. In *Proceeding of International Conference on Automated Software Engineering*, pages 402–405, New York, NY, USA, 2012. ACM.

[165] J. Oh, M. Harman, and S. Yoo. Transition coverage testing for simulink/stateflow models using messy genetic algorithms. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 1851–1858, 2011.

[166] S. Oster, F. Markert, and P. Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In J. Bosch and J. Lee, editors, *Proceedings of the International Software Product Lines conference*, volume 6287 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2010.

[167] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[168] A. Pachauri and G. Srivastava. Automated Test Data Generation for Branch Testing using Genetic Algorithm: An Improved Approach using Branch Ordering, Memory and Elitism. *Journal of Systems and Software*, 86(5):1191–1208, May 2013.

[169] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, Apr. 2015.



[170] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. CarFast: Achieving Higher Statement Coverage Faster. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[171] P. V. Paul, A. Ramalingam, R. Baskaran, P. Dhavachelvan, K.Vivekanandan, R. Subramanian, and V. S. K. Venkatachalapathy. Performance Analyses on Population Seeding Techniques for Genetic Algorithms. *International Journal of Engineering and Technology*, 5(3):2993–3000, 2013.

[172] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[173] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. L. Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 459–468. IEEE Computer Society, 2010.

[174] P. Piwarski. A nesting level complexity measure. *Special Interest Group on Programming Languages*, 17(9):44–50, 1982.

[175] K. Pohl, G. Bockle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[176] P. Ponterosso and D. S. J. Fox. Heuristically Seeded Genetic Algorithms Applied to Truss Optimisation. *Engineering with Computers*, 15(4):345–355, 1999.

[177] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 8 edition, 2014.

[178] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Fromman-Holzboog Verlag, Stuttgart, 1973.

[179] J. C. B. Ribeiro. Search-based test case generation for object-oriented java software using strongly-typed genetic programming. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 1819–1822, 2008.

[180] J. T. Richardson, M. R. Palmer, G. E. Liepins, and M. Hilliard. Some Guidelines for Genetic Algorithms with Penalty Functions. In *Proceedings of the International Conference on Genetic Algorithms*, pages 191–197, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[181] G. Rudolph. *Evolutionary Computation 1. Basic Algorithms and Operators*, volume 1, chapter 9, Evolution Strategies, pages 81–88. IOP Publishing Lt, 2000.

[182] B. Saavedra-Moreno, S. Salcedo-Sanz, A. Paniagua-Tineo, L. Prieto, and A. Portilla-Figueras. Seeding evolutionary algorithms with heuristics for optimal wind turbines positioning in wind farms. *Renewable Energy*, 36(11):2838–2844, 2011.

[183] R. Sagarna and J. A. Lozano. On the Performance of Estimation of Distribution Algorithms Applied to Software Testing. *Applied Artificial Intelligence*, 19(5):457–489, 2005.

[184] R. Sagarna and J. A. Lozano. Scatter Search in software testing, comparison and collaboration with Estimation of Distribution Algorithms. *European Journal of Operational Research*, 169(2):392–412, 2006.

[185] A. Sakti, Y.-G. Guhneuc, and G. Pesant. Constraint-Based Fitness Function for Search-Based Software Testing. In *Proceedings of the International Conference on Integration of AI*, volume 7874, pages 378–385, Yorktown Heights, NY, USA, 18-22 May 2013. Springer.

[186] E. Salecker, R. Reicherdt, and S. Glesner. Calculating Prioritized Interaction Test Sets with Constraints Using Binary Decision Diagrams. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 278–285, Washington, DC, USA, 2011. IEEE Computer Society.

[187] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos. *Open Source Development, Communities and Quality*, volume 275 of *International Federation for Information Processing*, chapter The SQO-OS, pages 237–248. Springer, 2008.

[188] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the European software engineering conference*, pages 263–272, New York, NY, USA, 2005.

[189] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, UK, 2007.

[190] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information & Software Technology*, 55(3):491–507, 2013.

[191] C. Software. Source Monitor - URL: http://www.campwoodsw.com/sourcemonitor.html, 2012.

[192] I. Sommerville. *Software Engineering*. Pearson Addison Wesley, April 2015.

[193] SPL2go - URL: http://spl2go.cs.ovgu.de, 2013.

[194] Software Product Line Online Tools(SPLOT) - URL: http://www.splot-research.org, 2013.

[195] H. Sthamer. *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD thesis, University of Glamorgan, 1995.

[196] A. M. Sutton, D. Whitley, and A. E. Howe. A polynomial time computation of the exact correlation structure of k-satisfiability landscapes. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 365–372, 2009.

[197] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, NIST, May 2002.

[198] W. Thomson. *Mathematical and Physical Papers*. Cambridge University Press, 1882.

[199] P. Tonella. Evolutionary testing of classes. *SIGSOFT Software Engineering Notes*, 29(4):119–128, 2004.

[200] J. Torres-Jimenez and E. Rodriguez-Tello. New bounds for binary covering arrays using simulated annealing. *Information Sciences*, 185:137–152, 2012.

[201] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Software Practice and Experience*, 30(1):61–79, 2000.

[202] N. Tracey and T. N. *A search-based automated test-data generation framework for safety-critical software*. PhD thesis, University of York, 2000.

[203] P. Trinidad and D. Benavides. Fama framework. In *Proceedings of the International Software Product Lines conference*, page 359, 2008.

[204] T. Tusar and B. Filipic. Visualization of Pareto Front Approximations in Evolutionary Multiobjective Optimization: A Critical Review and the Prosection Method. *IEEE Transactions on Evolutionary Computation*, 19(2):225–245, 2015.

[205] H. Ural. Formal methods for test sequence generation. *Computation Communications*, 15(5):311–325, 1992.

[206] H. T. Uyar, A. c. Uyar, and E. Harmanci. Pairwise Sequence Comparison for Fitness Evaluation in Evolutionary Structural Software Testing. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 1959–1960, New York, USA, 2006. ACM.

[207] D. A. Van Veldhuizen. *Multiobjective evolutionary algorithms: classifications, analyses, and new innovations*. PhD thesis, Air Force Institute of Technology, USA, 1999.

[208] A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[209] S. A. Vilkomir and J. P. Bowen. Formalization of Control-flow Criteria of Software Testing. Technical Report SBU-CISM-01-01, South Bank University, SCISM, London, UK, 2001.

[210] T. Vos, F. Lindlar, B. Wilmes, A. Windisch, A. Baars, P. M. Kruse, H. Gross, and J. Wegener. Evolutionary functional black-box testing in an industrial setting. *Software Quality Changes*, 2012.

[211] G. H. Walton, J. H. Poore, and C. J. Trammell. Statistical testing of software based on a usage model. *Software Practice and Experience*, 25(1):97–108, 1995.

[212] S. Wang, S. Ali, and A. Gotlieb. Minimizing test suites in software product lines using weight-based genetic algorithms. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 1493–1500, 2013.

[213] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In *IEEE Congress on Evolutionary Computation*, pages 851–858, 2006.

[214] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 1925–1932, 2006.

[215] J. Watkins and S. Mills. *Testing IT: An Off-the-shelf Software Testing Process*, volume 36. ACM, New York, NY, USA, May 2011.



[216] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[217] J. Wegener, H. Sthamer, B. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6:127–135, 1997.

[218] E. J. Weyuker. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

[219] A. W. Williams and R. L. Probert. A measure for component interaction test coverage. In *IEEE International. Conference on Computer Systems and Applications*, pages 304–311, 2001.

[220] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[221] L. A. Wolsey. *Integer Programming*. Wiley, 1998.

[222] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of Genetic Algorithms to Software Testing. In *Proceedings of the International Conference on Software Engineering and its Applications*, pages 625–636, Toulouse, France, 1992.

[223] M. Xiao, M. El-Attar, M. Reformat, and J. Miller. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering*, 12(2):183–239, 2007.

[224] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *Proceedings of the Annual IEEE International Conference on Dependable Systems and Networks*, pages 359–368, Lisbon, Portugal, June-July 2009. IEEE.

[225] Z. Xu, M. Cohen, W. Motycka, and G. Rothermel. Continuous test suite augmentation in software product lines. In *Proceedings of the International Software Product Lines conference*, pages 52–61. ACM, 2013.

[226] S. Yoo and M. Harman. Pareto Efficient Multi-Objective Test Case Selection. *Proceedings of the International Symposium on Software Testing and Analysis*, pages 140–150, 2007.

[227] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[228] L. Yu and A. Mishra. Experience in predicting fault-prone software modules using complexity metrics. *Quality Technology and Quantitative Management*, 9(4):421–433, 2012.

[229] P. Zave. FAQ sheet on feature interaction - URL: http://www.research.att.com/pamela/faq.html, 1999.

[230] Y. Zhang. Search Based Software Engineering Repository. http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/.

[231] Y. Zhang and J. Clark. The state problem for test generation in Simulink. In *Proceedings of the annual Conference on Genetic and Evolutionary Computation*, pages 1941–1948. ACM Press, 2006.



[232] Y. Zhou, B. Xu, and H. Leung. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software*, 83(4):660–674, 2010.

[233] E. Zitzler. Evolutionary Multiobjective Optimization. In G. Rozenberg, T. Bäck, and J. N. Kok, editors, *Handbook of Natural Computing*, pages 871–904. Springer, 2012.

[234] E. Zitzler, K. Deb, and L. Thiele. Comparison of multiobjective evolutionary algorithms: empirical results. *Evolutionary computation*, 8(2):173–95, Jan. 2000.

[235] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. In *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems*, number 103 in 1, pages 95–100, 2001.

[236] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

[237] E. Zitzler, L. Thiele, M. Laumanns, F. C., and d. F. V. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE, Transactions on Evolutionary Computation*, 7:117–132, 2003.