

**PLETS - A PRODUCT LINE OF
MODEL-BASED TESTING
TOOLS**

ELDER DE MACEDO RODRIGUES

Thesis presented as partial requirement for
obtaining the degree of Ph. D. in Computer
Science at Pontifical Catholic University of
Rio Grande do Sul.

Advisor: Prof. Avelino Francisco Zorzo

Dados Internacionais de Catalogação na Publicação (CIP)

R696P Rodrigues, Elder de Macedo
PLETS – A product line of model-based testing tools /
Elder de Macedo Rodrigues. - Porto Alegre, 2013.
130 p.

Tese (Doutorado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Avelino Francisco Zorzo.

1. Informática. 2. Engenharia de Software.
3. Software – Análise de Desempenho. 4. Dados de Teste
(Informática). I. Zorzo, Avelino Francisco. II. Título.

CDD 005.1

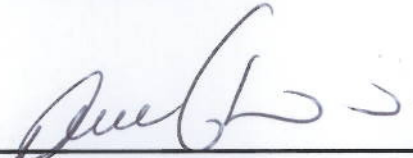
**Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS**




Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE TESE DE DOUTORADO

Tese intitulada "PLeTs - A Product Line of Model-based Testing Tools", apresentada por Elder de Macedo Rodrigues, como parte dos requisitos para obtenção do grau de Doutor em Ciência da Computação, aprovada em 21/08/2013 pela Comissão Examinadora:


Prof. Dr. Avelino Francisco Zorzo
Orientador

PPGCC/PUCRS


Prof. Dr. Rafael Prikladnicki

PPGCC/PUCRS

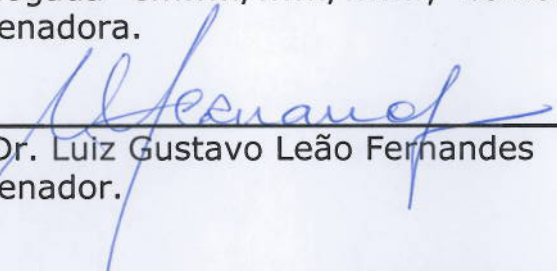

Profa. Dra. Itana Maria de Souza Gimenes

UEM

Prof. Dr. Adenilso da Silva Simão

USP

Homologada em 05/11/2013, conforme Ata No. 020 pela Comissão Coordenadora.


Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

PUCRS

Campus Central

Av. Ipiranga, 6681 - P. 32 - sala 507 - CEP: 90619-900

Fone: (51) 3320-3611 - Fax (51) 3320-3621

E-mail: ppgcc@pucrs.br

www.pucrs.br/facin/pos

To my family, colleges and friends, who offered me unconditional support during the course of this thesis.

“Passam às mãos da minha geração
Heranças feitas de fortunas rotas
Campos desertos que não geram pão
Onde a ganância anda de rédeas soltas”
(Engenheiros do Hawaii)

ACKNOWLEDGMENTS

I would like to thank my parents for their endless love and support. I would like a special thank to my wife Ildevana and to my daughter Ana Carolina for their love, kindness and support during the past four years - actually in the last 10 and 8 years. I love you girls!!

A special thank to Prof. Avelino Zorzo for believe in my work and for being my supervisor and mentor in the last 6 years. Thank you AZ!

A thank to Prof. Itana Gimenez, Prof. Adenilso da Silva Simão and Prof. Rafael Prikladnicki for being part of my committee and for being available to discuss and contribute with this thesis.

I would like to thank all my colleagues and friends at Cepas: Prof. Flávio Oliveira, Leandro Costa, Maicon Bernardino, Artur Freitas, Priscila Guarnieri, Anderson Domingues, Tiago Chagas, Ana Luiza Cunha e Murilo Arantes. Thank you guys for the collaboration, discussion, support and barbecues.

I would like to thank Prof. Krzysztof Czarnecki for give me the opportunity to conducted my research at the Generative Software Development lab at University of Waterloo. Moreover, a special thank to all my "canadian" friends: Stephen Liu, Leonardo Passos, Aline Ferreira, Leopoldo Teixeira, Zubair Aktar, Arrah Leonardo, Pedro and Aline Drimel.

I also acknowledge, again, Prof. Itana Gimenez for her valuable feedback and for give me the opportunity to spend a whole month in her research lab at UEM. Furthermore, I would like to thank Prof. Edson Junior for his support during my stay in Maringá and his valuable contributions to this work.

Last but not least, I would like to thank Capes, Dell Inc., INCT-Sec and Procad.

PLETS - A PRODUCT LINE OF MODEL-BASED TESTING TOOLS

RESUMO

O teste de software é uma atividade fundamental para garantir a qualidade de software. Além disso, teste de software é uma das atividades mais caras e demoradas no processo de desenvolvimento de software. Por esta razão, diversas ferramentas de teste foram desenvolvidas para apoiar esta atividade, incluindo ferramentas para Teste Baseado em Modelos (TBM). TBM é uma técnica de teste para automatizar a geração de artefatos de teste a partir de modelos do sistema. Esta técnica apresenta diversas vantagens, tais como, menor custo e esforço para gerar casos de teste. Por este motivo, nos últimos anos, diversas ferramentas para TBM foram desenvolvidas para melhor explorar essas vantagens. Embora existam diversas ferramentas TBM, a maioria delas tem o seu desenvolvimento baseado em um esforço individual, sem a adoção de técnicas de reuso sistemático e com base em uma única arquitetura, dificultando a integração, evolução, manutenção e reutilização dessas ferramentas. Uma alternativa para mitigar estes problemas é adotar os conceitos de Linhas de Produto de Software (LPS) para desenvolver ferramentas de TBM. LPS possibilitam gerar sistematicamente produtos a custos mais baixos, em menor tempo e com maior qualidade. A principal contribuição desta tese de doutorado é apresentar uma LPS de ferramentas de teste que suportam TBM (PLeTs) e um ambiente automatizado para apoiar a geração dessas ferramentas (PlugSPL). Além disso, esta tese apresenta uma abordagem para gerar ferramentas para TBM, que foram aplicadas em dois exemplos de uso. Com base nos resultados obtidos nos exemplos de uso, podemos inferir que LPS pode ser considerada uma abordagem relevante para melhorar a produtividade e o reuso durante a geração de ferramentas de TBM. Além disso, também foi realizado um estudo experimental com o objetivo de avaliar o esforço para se utilizar uma ferramenta derivada da PLeTs para geração de *scripts* de teste. Os resultados apontaram que o esforço para gerar *scripts* de teste foi reduzido consideravelmente, quando comparado com a uma ferramenta de *Capture and Replay*.

Palavras Chave: Linha de Produto de *Software*, Teste de *Software*, Teste Baseado em Modelos.

PLETS - A PRODUCT LINE OF MODEL-BASED TESTING TOOLS

ABSTRACT

Software testing is recognized as a fundamental activity for assuring software quality. Furthermore, testing is also recognized as one of the most time consuming and expensive activities of software development process. A diversity of testing tools has been developed to support this activity, including tools for Model-based Testing (MBT). MBT is a testing technique to automate the generation of testing artifacts from the system model. This technique presents several advantages, such as, lower cost and less effort to generate test cases. Therefore, in the last years a diversity of commercial, academic, and open source tools to support MBT has been developed to better explore these advantages. In spite of the diversity of tools to support MBT, most of them have been individually and independently developed from scratch based on a single architecture. Thus, they face difficulties of integration, evolution, maintenance, and reuse. In another perspective, Software Product Lines (SPL) offers possibility of systematically generating software products at lower costs, in shorter time, and with higher quality. The main contribution of this Ph.D thesis is to present a SPL for testing tools that support MBT (PLeTs) and an automated environment to support the generation of these tools (PlugSPL). Furthermore, our strategy was initially applied to generate some MBT testing tools which were applied in two examples of use performed in collaboration of an IT company. Based on the feedback from the examples of use we can infer that SPL can be considered a relevant approach to improve productivity and reuse during generation of MBT testing tools. Moreover, we also performed an experimental study carried out to evaluate the effort to use an MBT tool derived from our SPL to generate test scripts and scenarios. Thus, the results point out that the effort to generate test scripts, when compared with a Capture and Replay based tool, was reduced considerably.

Keywords: Software Product line, Software Testing, Model-based Testing.

LIST OF FIGURES

| | | |
|------|--|----|
| 2.1 | Error Propagation (adapted from [ALRL04]) | 22 |
| 2.2 | Levels of Testing [UL06] | 26 |
| 2.3 | MBT Main Activities [EFW01] | 29 |
| 2.4 | Software Product Line Engineering Life-cycles [LSR07] | 33 |
| 2.5 | Example of a Car Feature Model - Slightly Adapted from [KCH ⁺ 90] | 37 |
| 2.6 | Realising Variability at Architecture [LSR07] | 38 |
| 3.1 | Systematic Mapping Studies Process (adapted from [PFMM08]) | 42 |
| 3.2 | Bubble Plot of the Domain Studies Distribution by Publication Year and Testing Level | 51 |
| 3.3 | Venn Diagram of the Test Models Categorization | 52 |
| 4.1 | MBT Tools Process | 61 |
| 4.2 | PLeTs Feature Model | 62 |
| 4.3 | PLeTs UML Component Diagram with SMarty | 65 |
| 4.4 | PlugSPL Modules | 70 |
| 4.5 | PlugSPL FM Editor | 71 |
| 4.6 | Component Management Module | 72 |
| 4.7 | Code Before the Replacement | 73 |
| 4.8 | Code After the Replacement | 73 |
| 5.1 | An Approach for Generating Performance Scripts and Scenarios | 76 |
| 5.2 | PLeTsPerfLR Tool Architecture | 79 |
| 5.3 | PLeTsPerfVS Tool Architecture | 80 |
| 5.4 | TPC-W Use Case Diagram | 80 |
| 5.5 | TPC-W Activity Diagram - Shop Use Case | 81 |
| 5.6 | Abstract Test Scenario of the Actor <i>Customer</i> | 82 |
| 5.7 | Example of Abstract Test Case Generated from the <i>Shop</i> Use Case | 83 |
| 5.8 | Example of Data File Containing Some Parameters | 83 |
| 5.9 | XML of Test Scenario Generated for the Visual Studio (*.LoadTest) | 84 |
| 5.10 | Test Script Generated for the Visual Studio | 84 |
| 5.11 | Test Script Generated for the LoadRunner | 84 |
| 5.12 | An Approach for Generating Structural Test Cases | 86 |
| 5.13 | PLeTsStructJabuti Tool Architecture | 89 |
| 5.14 | PLeTsStructEmma Tool Architecture | 89 |
| 5.15 | Sequence Diagram | 92 |

| | | |
|------|--|-----|
| 5.16 | Code Snippet of the Abstract Structure | 93 |
| 5.17 | Code Snippet of the Data File for JaBUTi | 93 |
| 5.18 | Code Snippet of TestDriver.java class | 93 |
| 5.19 | Coverage Information for JaBUTi | 95 |
| 6.1 | Experiment Run Timeline | 109 |
| 6.2 | Experiment Time - Session 1 | 112 |
| 6.3 | Experiment Time - Session 2 | 112 |
| 6.4 | Experiment Time - Session 3 | 112 |

LIST OF TABLES

| | | |
|-----|--|-----|
| 3.1 | Definition of Search Strings | 43 |
| 3.2 | Search Engines, Retrieved and Selected Primary Studies | 48 |
| 3.3 | Quality Studies Scores | 48 |
| 3.4 | Research Type Facets | 50 |
| 3.5 | List of Model-based Testing Tools | 56 |
| 5.1 | Classes and Methods to be Tested | 91 |
| 6.1 | Scales of Experiment Variables | 102 |
| 6.2 | Assigning Subjects to the Treatments for a Randomized Design | 110 |
| 6.3 | Summarized Data of the Effort | 110 |
| 6.4 | Design Mistake/error Data of the Performance Testing Scripts and Scenarios | 111 |
| 6.5 | Distribution Using Shapiro-Wilk Normality Test | 113 |
| 6.6 | Wilcoxon Matched Pairs Signed-ranks Test Results - Effort | 113 |

LIST OF ACRONYMS

CR – Capture and Replay
COTS – Commercial-off-the-shelf
XML – eXtensible Markup Language
FM – Feature Model
FODA – Feature Oriented Domain Analysis
FROM – Feature-oriented Reuse Method
IT – Information Technology
IEEE – Institute of Electrical and Electronics Engineers
IDE – Integrated Development Environment
LSTS – Labelled State Transition System
MBT – Model-based Testing
ROI – Return of Investment
SPT – Schedulability Performance and Time
SPL – Software Product Line
SPLA – Software Product Line Architecture
SPLE – Software Product Line Engineering
SMARTY – Stereotype-based Management of Variability
SUT – System Under Test
SLR – Systematic Literature Review
SMS – Systematic Mapping Study
TDL – Technology Development Lab
UML – Unified Modelling Language
V&V – Verification and Validation

CONTENTS

| | | |
|----------|--|-----------|
| 1 | INTRODUCTION | 16 |
| 1.1 | PROBLEM STATEMENT | 17 |
| 1.2 | THESIS CONTRIBUTIONS | 17 |
| 1.3 | ORGANIZATION | 18 |
| 2 | BACKGROUND | 20 |
| 2.1 | SOFTWARE TESTING | 20 |
| 2.1.1 | TEST CONCEPTS AND TERMINOLOGY | 21 |
| 2.1.2 | TESTING TECHNIQUES | 22 |
| 2.1.3 | LEVELS OF TESTING | 25 |
| 2.1.4 | MODEL-BASED TESTING | 28 |
| 2.2 | SOFTWARE PRODUCT LINES | 30 |
| 2.2.1 | SOFTWARE PRODUCT LINE ENGINEERING | 32 |
| 2.2.2 | SPL VARIABILITY MANAGEMENT | 35 |
| 2.3 | RELATED WORK | 38 |
| 2.4 | CHAPTER SUMMARY | 39 |
| 3 | A SYSTEMATIC MAPPING STUDY ON MBT | 40 |
| 3.1 | RELATED RESEARCH | 40 |
| 3.2 | PLANNING REVIEW | 41 |
| 3.2.1 | SCOPE AND OBJECTIVE | 41 |
| 3.2.2 | QUESTIONS STRUCTURE | 42 |
| 3.2.3 | RESEARCH QUESTIONS | 42 |
| 3.2.4 | SEARCH PROCESS | 43 |
| 3.2.5 | INCLUSION AN EXCLUSION CRITERIA | 44 |
| 3.2.6 | QUALITY ASSESSMENT CRITERIA | 44 |
| 3.2.7 | SELECTION PROCESS | 45 |
| 3.2.8 | DATA ANALYSIS | 45 |
| 3.3 | CONDUCTION | 46 |
| 3.3.1 | SEARCH | 46 |
| 3.3.2 | STUDY QUALITY ASSESSMENT | 48 |
| 3.4 | RESULT ANALYSIS | 49 |
| 3.4.1 | CLASSIFICATION SCHEMES | 49 |

| | | |
|----------|---|-----------|
| 3.4.2 | MAPPING | 49 |
| 3.5 | THREATS TO VALIDITY | 51 |
| 3.6 | DISCUSSION | 52 |
| 3.7 | CHAPTER SUMMARY | 55 |
| 4 | A SOFTWARE PRODUCT LINE OF MODEL-BASED TESTING TOOLS - PLETS | 57 |
| 4.1 | PLETS | 57 |
| 4.1.1 | REQUIREMENTS | 58 |
| 4.1.2 | DESIGN DECISIONS, PROCESS AND VARIABILITY CONTROL | 60 |
| 4.1.3 | ARCHITECTURE AND IMPLEMENTATION | 64 |
| 4.2 | PLUGSPL - AN ENVIRONMENT TO SUPPORT A COMPONENT BASED SPL ... | 66 |
| 4.2.1 | REQUIREMENTS | 66 |
| 4.2.2 | DESIGN DECISIONS | 69 |
| 4.2.3 | PLUGSPL ENVIRONMENT | 69 |
| 4.3 | FINAL CONSIDERATIONS | 73 |
| 5 | EXAMPLES OF USE | 75 |
| 5.1 | GENERATING PERFORMANCE MBT TOOLS | 75 |
| 5.1.1 | AN APPROACH FOR PERFORMANCE TEST CASE AND SCRIPTS GENERATION | 76 |
| 5.1.2 | EXAMPLE OF USE | 79 |
| 5.1.3 | PERFORMANCE MBT TOOLS: CONSIDERATIONS AND LESSONS LEARNED ... | 85 |
| 5.2 | GENERATING STRUCTURAL MBT TOOLS | 85 |
| 5.2.1 | AN APPROACH FOR STRUCTURAL TEST CASE GENERATION | 86 |
| 5.2.2 | EXAMPLE OF USE | 88 |
| 5.2.3 | STRUCTURAL MBT TOOLS: CONSIDERATIONS AND LESSONS LEARNED | 94 |
| 5.3 | FINAL CONSIDERATIONS | 95 |
| 6 | EMPIRICAL EXPERIMENT | 97 |
| 6.1 | DEFINITION | 97 |
| 6.1.1 | RESEARCH QUESTIONS | 97 |
| 6.1.2 | OBJECTIVE DEFINITION | 98 |
| 6.2 | PLANNING | 99 |
| 6.2.1 | CONTEXT SELECTION | 99 |
| 6.2.2 | HYPOTHESIS FORMULATION | 100 |
| 6.2.3 | VARIABLES SELECTION | 101 |

| | | |
|----------|--|------------|
| 6.2.4 | SELECTION OF SUBJECTS | 103 |
| 6.2.5 | EXPERIMENT DESIGN | 103 |
| 6.2.6 | INSTRUMENTATION | 104 |
| 6.2.7 | THREATS TO VALIDITY | 105 |
| 6.3 | OPERATION OF EXPERIMENTAL STUDY | 107 |
| 6.3.1 | PREPARATION | 107 |
| 6.3.2 | EXECUTION | 108 |
| 6.3.3 | RESULTS | 109 |
| 6.4 | ANALYSIS AND INTERPRETATION | 111 |
| 6.5 | CONCLUSIONS | 113 |
| 7 | THESIS SUMMARY AND FUTURE WORK..... | 115 |
| 7.1 | THESIS CONTRIBUTIONS | 115 |
| 7.2 | LIMITATIONS AND FUTURE WORKS | 116 |
| 7.3 | PUBLICATIONS | 117 |
| | REFERENCES | 119 |

1. INTRODUCTION

The demand for software systems from different areas of knowledge has increased in the past years. This fact has led companies to produce software on an almost daily basis, putting a strong pressure on software engineers that have to ensure that the software is working properly. This situation has increased due to the competition pressure as several companies are paying attention to the mistakes made by their rivals. High quality software is, therefore, a very important asset for any software company nowadays. Hence, software engineers are looking for new strategies to produce and verify software in a very fast and reliable manner.

One of the most important activities for assuring quality and reliability of software products is by means of fault¹ removal, *e.g.*, **Software Testing** [MS04] [Har00]. Software testing is applied to minimize the number and severity of faults in a software. Complementary approaches, such as fault tolerance, fault forecasting, fault prevention, or even fault removal in the sense of software verification, could also be applied to increase software quality [ALRL04].

Software testing can contribute to increase software quality; however it is important to systematically undertake the testing activity by using well-defined testing techniques and criteria [DLS78] [MS04] [RW85]. Despite the variety of testing techniques, criteria and tools that are available, most of them are applied in industry in an *ad hoc* manner, because software engineers still have to develop test scripts, provide test cases, and understand and write the tools configuration files. These activities can be error prone and faults can be injected in one of these tasks of software testing.

In this context, **Model-based Testing (MBT)** is one technique that has gained considerable attention to support the testing activity by taking into account information represented in system models [UL06]. MBT has several advantages when compared to other testing techniques. For instance, the use of MBT can reduce the probability of misinterpretation of system requirements by a test engineer and also can reduce the overall testing workload, since its adoption can support the automation of the test case generation and execution, and the reuse of the testing artifacts (*e.g.*, system models)

Therefore, in the last years a diversity of commercial, academic, and open source MBT tools has been developed to better explore these advantages [Hui07] [BGN⁺04] [HN04] [PMBF05] [BBM02] [Hui07] [ABT10] [Ben08] [Bob08] [VCG⁺08] [KMPK06]. For instance, Conformiq Qtronic tool [Hui07] derives tests automatically from behavioural system models, *e.g.*, UML statecharts [OMGb] and Qtronic Modeling Language. In turn, the AGEDIS suite [HN04] generates tests from AGEDIS Modeling Language (AML) and UML models to test component-based distributed systems. Likewise, the COW suite [BBM02] generates test cases from UML diagrams. Despite the fact that most of these tools have been individually developed by different companies or research groups, most of them use a similar process and in some cases the same system models, even when generating tests for different test levels.

¹We use in this thesis the definition of fault, error and failure in according to [ALRL04].

1.1 Problem Statement

Although there are a diversity of commercial, academic, and open source testing tools that automate software testing tasks, most of these tools have been individually and independently implemented from scratch based on a single architecture. Thus, they face difficulties of integration, evolution, maintenance, and reuse.

In order to reduce the difficulties that MBT faces, it would be relevant to have a strategy to automatically generate specific products, *i.e.*, testing tools, for executing all MBT phases based on the reuse of assets and on a core architecture. This is one of the main ideas behind **Software Product Lines (SPL)**. Basically, a SPL is composed of a set of common features and a set of variable parts that represent later design decisions [CN01]. The SPL adoption has increased in the past years and several successful cases have been reported in the literature [SEI] [BCK97] [Nor02].

Despite the increasing number of success cases of adoption of SPL and the great amount of works related to testing software product lines [OG05] [ER11] [LKL12], there is no relevant investigation on how the SPL concepts can be applied to support the development of testing tools. Moreover, to the best of our knowledge, there is a lack of investigation on the use of SPL concepts to build testing tools that use MBT.

1.2 Thesis Contributions

The main focus of this thesis is to apply **SPL** concepts to systematize and support the generation of Model-based Testing Tools - **MBT**. The main contributions of our work are:

- To propose, design and develop a **Product Line of Model-based Testing tools (PLeTs)**. To support the elicitation of the domain requirements, design decisions and domain realization, we empirically investigated several state-of-the-art papers and academic, open-source and commercial MBT tools. We also based our domain design decisions on the investigation of several works focusing on SPL foundations, adoptions concerns, variability mechanism and tools support to the SPL adoption.
- Development of an environment to support the SPL life-cycle. SPL adoption has many challenges in industrial settings. One particular challenge regards the use of *off-the-shelf* tools to support the adoption of SPLs. Frequently, these tools do not address the company's specific needs. To elicit concrete requirements we availed from our experience when using SPL to derive testing tools in a research collaboration, between a Technology Development Lab (TDL) of an IT company and our university. We present the requirements and our design decisions in the development of an in-house tool addressing those specific requirements. These decisions, in turn, can be re-applied in different settings sharing similar requirements.

- An approach to easily integrate the tools derived from PLeTs with commercial, open-source or academic testing tools already used by a testing team. The reuse of these tools presents several benefits, such as, reduce the effort and the cost during the development of SPL core assets and less investment in training while the tool is used by testing teams.
- Present two examples of use, where the testing tools generated from PLeTs are used to generate test scripts to test two web applications. Furthermore, we also perform an experiment, which was focused to understand the effort to use a generated tool from PLeTs and compare it with the effort when using an industrial standard testing tool. The results show that in some scenarios the use of our MBT tool requires less effort than the standard tool used by several software development companies.

Furthermore, we also claim that our research presents some additional outcomes, such as:

- A novel approach to generate performance scripts and scenarios: as the requirements to generate the tools are based on an industrial setting, and there is little investigation on how MBT can be applied in performance testing, we propose a novel approach that uses UML models, a modelling notation widely used in the industry. This approach has been used to develop some performance testing tools, that have been successfully used in several academic works and industrial use cases to automatically generate performance scripts and scenarios.
- A novel approach to generate structural test cases: there are different tools that can be used to support the automation of structural testing, *e.g.*, JaBUTi [VMWD05], EMMA [Rou], and Poke-Tool [Cha91]. However, despite the benefits of these tools, some tasks still have to be performed manually, *e.g.*, the description of test cases. This makes the test process time consuming and prone to injection of faults. To overcome this limitation we proposed an approach that uses UML models to derive structural test cases for different structural testing tools.

Ultimately, we expect that the discussions, approaches, results, and tools presented in this thesis will be a real contribution to the software testing and software product line research fields. Moreover, we believe that the use of PLeTs, PlugSPL environment and the generated MBT testing tools in an industrial setting represent a valuable feedback on the use of these approaches in a real scenario.

1.3 Organization

This thesis is organized as follows: **Chapter 2** presents the background information about software testing, Model-based Testing and Software Product Lines. **Chapter 3** presents a systematic mapping study on Model-based Testing, which provide a broad overview about the relevant contributions on MBT field, in especial on MBT tool support. **Chapter 4** presents our **Product Line**

of Model-based Testing tools (PLeTs), as well as, the elicitation of the domain requirements, design decisions and domain realization. In this chapter we also introduce the requirements and our design decisions to develop an environment to support the generation of testing tools from PLeTs. **Chapter 5** presents two examples of use where testing tools derived from PLeTs are used to generate and execute performance and structural testing cases and scripts. **Chapter 6** presents our experimental study that aims to evaluate the effort to create performance test cases and scripts when using an MBT tool generated from PLeTs (PLeTsPerfLR). In order to evaluate the results we compare the PLeTsPerfLR MBT tool with a tool used in industry that uses Capture-Replay testing technique, *i.e.*, HP LoadRunner. Finally, conclusions and future works directions are presented in **Chapter 7**.

2. BACKGROUND

In this chapter, we provide some background information on Software Testing and Software Product Lines (SPL). The background on Software Testing includes definitions on test concepts and terminology, testing techniques, testing levels and Model-based Testing. We also provide in this chapter some essential background on SPL, product line engineering and SPL variability management.

2.1 Software Testing

Nowadays, almost all human activities take advantage of the use of software systems, from a trivial system to manage a local business to a complex and critical system, such as those used to control a nuclear power plant, financial operations or a modern airplane. Although, in all of these systems, trivial or complex, some level of dependability and security is required. For instance, in the last few years it has not been acceptable for a critical system, or even in some cases an ordinary software system, to have a low degree of availability, dependability and safety, because a service failure may result in a significant financial loss, or even worse, human lives loss. For this reason, software development companies have been trying to reach an increasing level of dependability in their software. Thus, they are making a substantial effort and financial investment to ensure that the software is working properly. Consequently, high quality software is a very important asset for any software development organization.

However, there are several attributes that these organizations must consider to ensure that software is of high quality. In the Avizienis's *et al.* [ALRL04] work they claim that a high quality software (dependable) must encompass six attributes: availability, reliability, confidentiality, safety, integrity and maintainability. Furthermore, Avizienis's *et al.* also categorized the four main techniques that might be used to reach dependability in a software system:

- **Fault prevention:** this category focused on preventing the occurrence or introduction of faults;
- **Fault tolerance:** this focuses on avoiding service failures if a fault is present;
- **Fault forecasting:** its main focus is to predict likely faults, so they can be removed or their effects on the system can be minimized or circumvented;
- **Fault removal:** this category focuses on reduces the number of faults, by detecting the failures and then removing the faults¹, during the software development and use.

Although all these techniques are used to achieve software dependability, one the most used techniques in the software development process in industry is fault removal. In accordance

¹The definitions of fault, error and failure are presented in Section 2.1.1

with [ALRL04], this category is subdivided into verification and validation (V&V). Validation is the process of evaluating the software to ensure compliance with intended usage (requirements) and verification is the process to evaluate if the software has met the specified requirements for all software development phases. Static analysis, theorem proving, model checking, symbolic execution and software testing are examples of verification and validation techniques [ALRL04].

In addition to the other verification and validation techniques, software testing is one of the most used techniques, constituting one key element to reach reliability in a software system [ERM07]. Software testing can be defined as the process of systematically evaluate a system by its controlled execution and observation [AO08]. Software testing and its objective can be defined as:

Testing is performed to evaluate and improve product quality by identifying defects and problems. Software testing consists of the dynamic verification of a program's behavior on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior [Ber].

2.1.1 Test Concepts and Terminology

In this section, we present some relevant test concepts and terminology that will be used throughout this thesis. Most of this information has been taken from the IEEE Standard Glossary of Software Engineering Terminology [Com] and IEEE Standard for Software and System Test Documentation [IEE]. However, to maintain consistency with the taxonomy and concepts presented above, we have chosen to use the definitions of Fault, Error and Failure as presented by Avizienis *et al.* [ALRL04]. Furthermore, we believe that the concepts and taxonomy presented in that work are comprehensive and are structured in a way that makes them easier to read and understand. The main concepts and terminology used in this thesis are:

Fault: this is a static defect introduced into a software system, usually during its coding phase. It is often caused by human errors, such as, mistakes with typing, lack of knowledge, and misinterpretation of requirements. In case a fault is present in a system, but it is never executed, the fault is *dormant*. Otherwise, the fault is *active* and can cause an error (Figure 2.1 a)).

Error: this can be defined as an incorrect internal state of a system. In case this incorrect internal state leads to a failure, the error was *propagated* outside and can result in a service failure (Figure 2.1 b)). If an error is present, but is not active and detected, it is classified as a *latent* error.

Failure: failure is defined as an observable event that occurs when the system's actual behavior deviates from the system's expected behavior (Figure 2.1 c)). Since software testing consists of the dynamic verification of a program's behavior, it is focused on detecting failures to remove faults.

Test specification: it is a document that is used by the test analyst to describe the test environment, test inputs, execution conditions and expected behavior of the system under test (SUT). This document is often based on an agreement between the customer and the development/testing team.

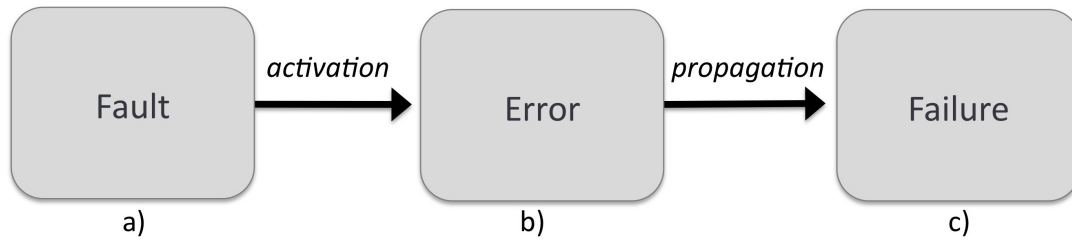


Figure 2.1: Error Propagation (adapted from [ALRL04])

Test case: is a set of test inputs and execution conditions with the objective of exercising a particular system (or part of a system) and then monitoring the system behavior to verify if it is in accordance with its requirements.

Test suite: it is a set of test cases (normally concrete test cases).

Abstract test case: it describes the testing execution and conditions without concern for the programming language or the tool that will be used to execute the test. Normally, it is an initial step to create a concrete test case. For example, an abstract test case for performance testing might be a textual document (*e.g.*, Word or XML) that describes the activities that can be performed by users when they are interacting with the SUT.

Concrete test case: it is an abstract test case that has all the concrete information necessary to execute the test. Typically, a concrete test case addresses a program language or a script format. For instance, a XML file that describes an abstract test case for performance testing (activities and its parameters) can be used to create a set of performance scripts and scenarios to a specific performance tool (*i.e.*, LoadRunner [Hew]), which in turn can be executed by a performance testing tool (*e.g.*, Costa *et al.* [CCO⁺12] propose an approach to generate performance scripts and scenarios from abstract test cases).

Test oracle: it is a testing artifact that contains all information about the SUT's expected behavior. Thus, during or after test execution, this information is compared with the observed system behavior to try to detect failures.

2.1.2 Testing Techniques

Based on the test concepts and terminology presented 2.1.1, in this section, we present and discuss the techniques that can be used to try to find and remove faults that might be present in the software. Software testing is focused on revealing as many failures as possible. Thus, to do that, most of the testing literature claims the use of two complementary testing techniques: White-box and Black-box [MS04] [ERM07].

White-box Testing

The White-box testing technique, also called Structural testing or Logic-driven testing, is focused on ensuring that the internal components of a software system are working properly by the analysis of its inner structure. Thus, the testers can take advantage of the access to the source code and then use their knowledge about the software's internal structure to generate test cases. In an ideal scenario, the generated test cases would cover all software's code paths, conditions, loops, and data flow. However, it is impractical in an industrial setting, since the time that would be spent to execute the test is in most cases bigger than the estimated software lifetime. Thus, to overcome this limitation, the tester must use some coverage criteria [ERM07] to generate relevant test cases. For instance, a coverage criterion can be used to define what the percentage of the system source code was exercised during the test. This means that the tester can define, for instance, that at least 60% of the code must be exercised. The following coverage criteria can be used to improve the test case selection when applying White-box testing [MS04] [ERM07]:

- *Statement coverage*: it is a simple coverage criterion that is focused on determining the percentage of the system source code that was executed during the test. The assumption is that the greater the statement coverage there is, the smaller possibility of faults that remain present in the software.
- *Branch coverage*: its goal is to determine a percentage of the system source code branches points (decisions) that were executed during the test, *i.e.*, an *IF* decision has two branches (True and False). This means that in case a tester wants 60% coverage of a system source code that has 2,000 branches, the test must exercise 1,200 branch points. The assumption here is the same as the statement coverage: a greater branch coverage means that there is less possibility of faults remaining present in the software.
- *Path coverage*: this criterion aims to determine the coverage percentage of the source code paths of a system are executed during the test. Everett and Raymond [ERM07] state that: "*a source code path is the sequence of program statements from the first executable statement through a series of arithmetic, replacement, input/output, branching, and looping statements to a return/stop/end/exit statement.*". The coverage percentage is measured from the estimated total of the SUT's code paths. Thus, if the SUT has 1,500 code paths and the tester traversed 300 code paths, the path coverage is 20%. The assumption is that the higher the path coverage percentage is, the smaller the possibility of faults still present in the software.
- *Loop Coverage*: this criterion aims to determine the coverage percentage of loops, *e.g.*, *DO*, *FOR*, *WHILE*, present in the SUT after the test execution. Thus, during the test the system must execute a loop zero (the tester must try to avoid the execution of a loop), one, $n/2$, n and $n + 1$ times (n represents the limit value to stop the loop condition). The objective of the Loop coverage criterion is to find unexpected and inappropriate looping conditions, adopting a similar approach than the Boundary-value analysis for Black-box technique (see

next subsection). The percentage of achieved coverage is calculated over the SUT estimated loops and the assumption is that a greater loop coverage means less faults might be present in the code.

Black-box Testing

Black-box testing, also called Functional testing, Data-driven or Input/output Driven test, is a testing technique focused on deriving test cases from the external specification of the SUT, such as specifications, requirements and design documents. Its goal is to induce the system to deviate from its expected behavior, by submitting external inputs (simulating daily business activities) and unlikely Structural testing, Functional testing is not concerned with the SUT inner structure. Despite the fact that the Black-box testing technique can be applied to find all faults introduced into the software, it is impractical because the tester would have to submit every possible input accepted by the software. Furthermore, in general it is humanly impossible to submit all inputs because it can be very large or even infinite [DMJ07]. Following, we briefly introduce some Black-box coverage criteria that can be used by the tester to reduce the input set size and to improve the possibility of test cases revealing the existence of a fault [MS04] [ERM07]:

- *Equivalence partitioning*: this criterion is based on dividing the input domain of the SUT, identified from the specification, into valid and invalid equivalence classes (subsets of the input domain) from which test cases can be generated [Ber]. In the case that an input value from an equivalence class reveals a failure, it is expected that all other inputs from the equivalence class will find the same failure. Based on this, the tester can assume that an input value of each class is equivalent to test any other input value present in the class. Thus, this assumption will reduce the size of the test inputs and hence the effort and time spent to verify the SUT.
- *Boundary-value analysis*: this criterion is considered as a complement to the Equivalence partitioning criterion because unlike the former that is based on choosing a random input value from the equivalence class, the Boundary-value analysis is focused on defining test inputs that lie along data boundaries. Some authors, such as, Everett and Raymond [ERM07] and Myers [MS04], claim that the Boundary-value analysis is important to testers because a large amount of functional failures occur in these boundaries or close to it. Based on this statement, testers can assume that test cases that take advantage of Boundary-value analysis have a higher possibility to find failures than those test cases that do not use it.
- *Cause-effect graphing*: Equivalence partitioning and Boundary-value analysis are well-known criteria and have been used for many years to reduce the size of the test inputs and the time and effort to verify a system. However, these criteria fail on to not explore combinations of input conditions. To overcome this limitation, the tester can apply the Cause-effect graphing criterion that establishes the testing requirements based on the possible combinations of input conditions. In accordance to Myers [MS04], to apply Cause-effect graphing criterion, the tester

must investigate the possible input conditions (causes) and possible actions (effects) of the SUT. After that, a graph is constructed linking the identified causes and effects and then it is converted into a decision table from which the test cases are derived. One of the main issues related to Cause-effect graphing (actually it might be an issue for all functional criteria) is that often the program specifications are made in an informal way. Thus, the testing requirements derived from such specifications are also somewhat inaccurate.

- *Error guessing*: it is an *ad hoc* criterion focused on generating the test case inputs based on the tester's/developer's previous experience with similar code, programming language and application domain [Bur03]. Thus, case the testing data of a similar code or even a past release was kept and there is some document identifying and mapping the failures to faults, the tester is able to, in an *ad hoc* way, "guess" which faults may be present in the code. Although this criterion does not provide the same advantages as those presented previous, it is well known that many testers have been using Error guessing criterion in their everyday testing activities, such as discussed in [ERM07] [MS04] [Bur03].

2.1.3 Levels of Testing

In this section, we present the different testing levels and how testing is usually performed during the software development process, from the requirements and specifications to the complete system and its acceptance by the customer. Although there are many software development models present in the literature and also works that present the testing levels and discuss their relation to each activity of the software development process [Som11] [AO08] [UL06] [NT11], most of these works define 4 levels or major phases: Unit testing, Integration testing, System testing, Acceptance testing (see Figure 2.2).

Unit testing

Unit testing is focused on testing a small testable piece of a system, such as, methods, classes and functions, and is usually performed by the developer (top of the Figure 2.2). In this testing level the main objective of the tester is to try to detect functional and structural failures on the code unit. Since the code unit under test is normally small and the tester/developer has a high knowledge about the code, it is quite simple to design and to write relevant tests and also to analyze the results. Furthermore, an efficient unit testing improves the possibility that the majority of and the most relevant faults will be removed in the early stages of the software development. This means less cost and effort because it is cheaper and easier to remove faults during the Unit testing than at any other testing level [UL06].

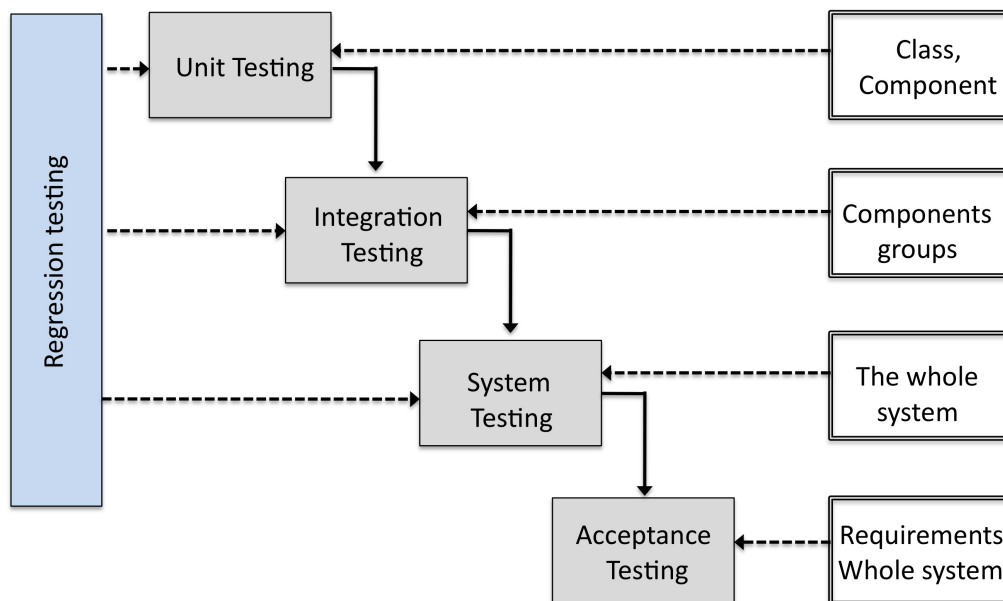


Figure 2.2: Levels of Testing [UL06]

Integration testing

The objective of the Integration testing is to verify the incompatibility and interfaces between individual components that were already tested and approved during the Unit testing. Despite the fact that components were already individually tested during the Unit testing, it cannot guarantee that the interfaces were correctly implemented and also that the components will work properly when combined with other components - in some cases they are designed and coded by a different development team. Thus, the main focus of the Integration testing is to integrate a set of individual components into a huge working component or module. There are some approaches to integrate and test the components, such as those presented in [NT11]:

- *Incremental*: In this approach, the integration of the components is incremental per cycle. This means that some components are completed, integrated and tested. Then, a new set of components are added and tested. Thus, this cycle continues until the whole system is completed integrated and tested.
- *Top down*: This integration approach focus on systems where the components must have a hierarchical relationship. For instance, there is a high-level component that can be decomposed into a set of components, which in turn, can be decomposed into other components, and so on. Each component that is not decomposed is a terminal component. On the other hand, components that perform some operations and invoke their decomposed components are non-terminal components. In a high level abstraction, the main idea behind this approach is to add one component each time to the component in the highest level, test the integration and then add another non-terminal component. The cycle must be repeated until the added component is a terminal component.

- *Bottom up*: unlike the Top down approach, the Bottom up approach focuses on starting the integration by the components that are terminal components. To perform the integration, it is necessary to develop a test drive to invoke the components that will be integrated (the terminal components). When the tester decides that the components under test are ready to be integrated, the test drive is removed, and another test drive is developed to test the components that will be integrated with the components already tested. Thus, the process continues until all modules have been tested and integrated.
- *Big bang*: in this approach, all the modules are individually tested and are then integrated to construct the whole system. After that, the system is tested as a complete system.

System Test

After the system modules are integrated and tested, it is necessary to establish if the system's implementation meets the customer requirements. Therefore, the testing team tests the whole system based on its requirements. Motivated by the fact that the System test requires a high amount of resources and is usually a time consuming activity, it is normally performed by a dedicated testing team, at least in medium and large companies. As there are several types of System tests, testing teams can have some highly skilled professionals focused in some types of System Test. Some of these types are:

- *Functional testing*: its goal is to ensure that the behavior of the developed system meets the requirements. To perform Functional testing, the tester must submit proper and illegal inputs to the system and then analyse its output (Black-box testing - see Subsection 2.1.2). Furthermore, the tester should test all the system's functions/methods and all the system's states and transitions must be exercised.
- *Performance testing*: the main focus of Performance testing is to verify if the system meets the performance requirements. For instance, a performance requirement might name that the system's response time must be less than two seconds or that the system must handle up to two hundred simultaneous users. Thus, the performance team can simulate the users load and then use the test results to optimize the system performance. An example of the system optimization is the adjustment of the system resources, such as, the amount of available system memory.
- *Stress testing*: its main goal is to determine the maximum amount of load that the system can handle before it breaks, e.g., an abnormal load of users. Thus, the testing team uses this information to ensure that the system behavior meets the requirements even under the worst supported load peak [NT11].
- *Security testing*: it is applied to verify if the system meets the security requirements. A system meets the security requirements when the system and its data are protected from unauthorized

access (confidentiality), its data is protected from unauthorized modification (integrity), and when the system and its data are available to authorized users (availability).

- *Recovery testing*: it is performed in order to verify if the system recovered properly from a crash, such as hardware failures. Normally, the tester simulates some hardware failure (removing a processor), connective loss (removing the network cable) or abruptly restarting the computer and then analyses how the system recovers and how much time it took to recover.

Regression Testing

Regression testing cannot be considered as a testing level since it is performed throughout the software testing process, like a sub-phase of Unit, Integration and System levels (see Figure 2.2). It is executed after the development/testing teams have made any functional improvements or repairs to the system. The Regression testing purpose is to ensure that the modified version of the system maintains the same functionalities and to ensure that any fault was introduced during the system modification [Bur03]. Another regression testing particularity is that it is not necessary to design and write new tests. Instead, the tests are selected from the set of tests that were already designed, developed and executed. Furthermore, as the regression test represents a substantial effort during the testing process, usually just a few tests are selected and executed. However, the tester must precisely define how many tests must be chosen to improve the possibility of failure detection [JG07] [LHH07].

Acceptance testing

After the System test, the system is ready to be delivered to the customer, but before it must be accepted by the customer. The Acceptance testing is performed by the customer, and its main goal is to check the system behavior against the customer's requirements [Ber]. Despite the fact that this test is run by the customers, usually the development and testing teams are involved in the test preparation, helping them in the testing execution activities and results evaluation. If the customer identifies that some requirements were not satisfied, the system must be corrected, or the requirements can be changed. Otherwise, the customer can accept the software and then it is ready to be delivered.

2.1.4 Model-based Testing

Model-based Testing (MBT) is a technique used to support the (semi)automatic generation of test case/scripts from the SUT's models [EFW01]. Normally, the SUT can be modelled from three different perspectives: *Data model* - the model represents the data input to the SUT; *Tester model* - the model represents the interaction between an user and the SUT, and; *Design Model* - which represents the dynamic behavior of the SUT. MBT can accept as an input a wide amount of system

models, such as, Unified Modelling Language (UML) [OMGb], State Diagrams and Specification and Description Language (DSL).

The MBT adoption requires more activities than the traditional activities of software testing [UL06]. For instance, the MBT adoption requires that test engineers adjust their testing process and activities, and invest in the training of the testing team. El-Far and Whittaker [EFW01] defined the main activities of the MBT process as Build Model, Generate Expected Inputs, Generate Expected Outputs, Run Tests, Compare Results, Decide Further Actions and Stop Testing. Figure 2.3 presents a brief overview of these activities:

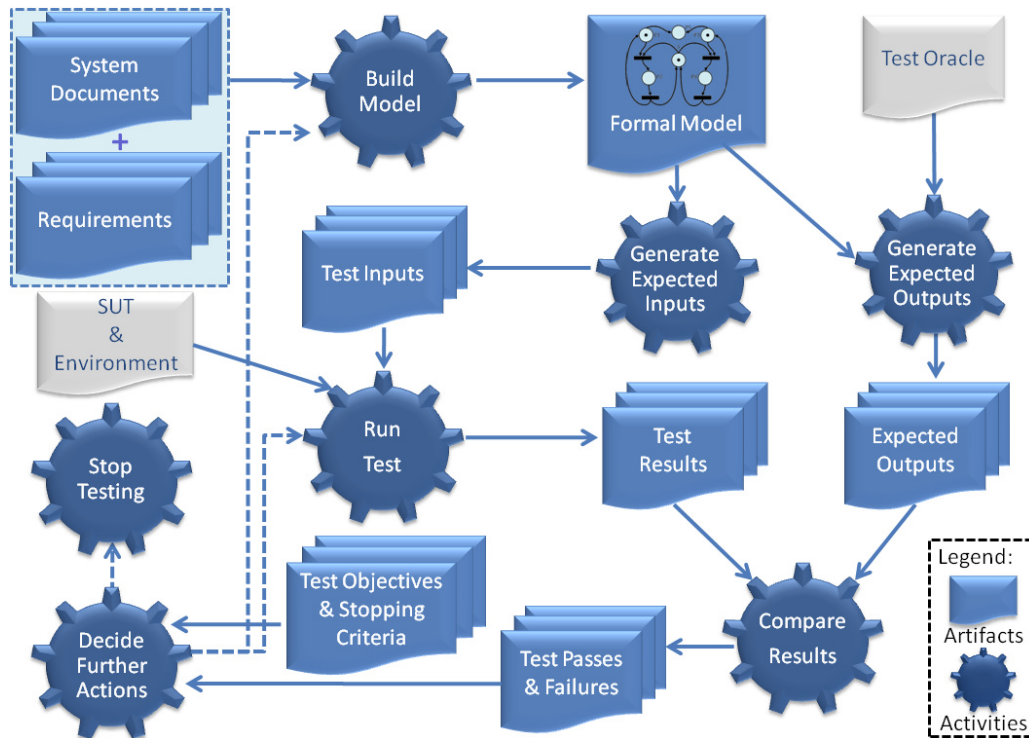


Figure 2.3: MBT Main Activities [EFW01]

- *Build Model*: consists of constructing a model based on the specification of a system. This step defines the choice of model, according to the application being developed;
- *Generate Expected Inputs*: uses the model to generate test inputs (test cases, test scripts, application input data);
- *Generate Expected Outputs*: generates some mechanism that determines whether the results of a test execution are correct. This mechanism is a test oracle, and it is used to determine the correctness of the output;
- *Run Tests*: executes test scripts and stores the results of each test case. This execution can be performed on the system under test (SUT) and/or system's environment;
- *Compare Results*: compares the test results with expected outputs (test oracle), generating reports to alert the test team about failures;

- *Decides Further Actions*: based on the results, it is possible to estimate the software quality. Depending on the achieved quality, it is possible to stop testing (quality achieved), to modify the model to include further information to generate new inputs/outputs, to modify the system under test (to remove remaining faults), or to run more tests;
- *Stop Testing*: concludes the testing and releases the software system.

As the MBT process supports the (semi) automation of the testing activities, it helps to reduce the cost of software testing since this cost is related to the number of interactions and test cases that are executed during the testing phase. As the testing phase costs between 30% and 60% of software development effort [MS04], MBT is a valuable approach to mitigate these problems [VCG⁺08]. Furthermore, the MBT adoption can bring several other advantages to the test team, such as [EFW01]:

- Shorter schedules, lower cost, and better quality;
- Early identification of ambiguities in the model specification;
- Enhanced communication among developers and testers;
- Automatic test script generation from test cases;
- Test mechanisms to automatically run generated scripts;
- Easiness to update the test cases when the requirements are changed;
- Less effort to perform regression tests;
- Capability to assess software quality and reliability (due to the possibility of comparing quality and reliability using the same formal model, e.g., Markov Chains).

Although a testing team takes all advantages proportioned by the MBT adoption, a tool support is mandatory. Nowadays, a number of commercial, academic and open-source MBT tools are available [UL06]. However, it is not an easy task for a testing engineer to define what tool will be adopted since several tools can be based on a variety of models, coverage criteria, approaches and notations. For this reason, some works have been published in recent years comparing MBT tools and approaches [UL06] [DNSVT07] [SL10] [SMJU10].

2.2 Software Product Lines

Throughout a few decades, more and more software development companies have been using some software engineering strategies, such as reuse-based software engineering, to develop software with less cost, faster delivery and increased quality. Reuse-based software engineering is a strategy in which the development process is focused on the reuse of software components, reducing

the effort and improving the software quality. In recent years, many techniques have been proposed to support the software reuse, such as, Component-based development, COTS product reuse and Software Product Lines (SPL) [Som11].

The Component-based development is centred on developing a software system by integrating components, where each component can be defined as an independent software unit that can be used with other components to create a system module or even a whole software system. In a similar manner, COTS product reuse is focused on developing a software system by configuring and integrating existing application systems, but without changing its source code. On the other hand, Software Product Line is focused on developing a family of applications based on a common architecture and a shared set of components, where each application is generated from these components and architecture in accordance with the requirements imposed by different customers [Som11]. Another SPL definition is presented by Clements and Northrop [CN01]:

A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

In past years, SPL has emerged as a promising technique to achieve systematic reuse and at the same time to decrease development costs and time-to-market [PBL05]. During this time many successful cases studies have been reported [SEI] [LSR07]. For instance, Nokia reported that the SPL adoption resulted in a significant increase in production of new models of mobile phones and Hewlett Packard reported a productivity improvement and a reduction of the time-to-market of their printers. However, in addition to the benefits reported by these companies, SPL adoption by an organization can present several others benefits, such as [LSR07] [PBL05]:

- *Reduction of the development costs:* as the development of each application is made based on a common set of core assets, it implies the reduction of the overall development cost. However, there is an initial additional effort and cost to develop a common set of core assets and to structure the organization development process. Thus, the development of the firsts applications implies a higher cost than developing them as single applications. Several authors advocate that the equalization point, where the costs are the same for developing each system separately as for developing them as a product line, is between 3 and 4 systems [PBL05] [LSR07] [WL99]. After that, the system developed by the organization will require less investment and effort.
- *Reduction of the development time:* in the same way that the development cost is higher, the initial development time is higher because it is first necessary to develop the common core assets of the family of systems. However, after the development of the core assets, there are significant reductions in the effort to develop product variants. This reduction means that the product's variants will have a reduced time-to-market and a faster return of investment (ROI).
- *Improve the reliability of products:* the system reliability it is not a target benefit of the SPL adoption, but as the reused components are normally tested individually or when integrated

with other components several times during the development of products variants, there is a high possibility that a large number of failures can be detected and the faults removed.

- *Reduction of the maintenance effort and cost:* as in the SPL the systems share a common platform, so it is easier for a development company to maintain/change the platform than maintain a set of individual systems. Furthermore, as the software maintenance is normally performed over a large period of time and could represent a significant effort to the development teams, the reduction of the maintenance cost is relevant to reduce the effort and the cost during the software life-cycle.
- *Facilitate the evolution:* similar to the maintenance, in the SPL the evolution of the software products is easier than for a single product. For instance, in case that the developers want to introduce a new feature into a set of software systems, it is only necessary to add the new artifacts that address this feature in the SPL platform. Thus, this simplifies the evolution of all the systems that share these artifacts.
- *Improve the cost estimation:* since the SPL is a family of applications based on a common architecture and a shared set of components, it is simple for a development company to estimate the risk and consequently the cost of the development of a new software product from the platform.

However, if an organization decides to take advantage of the benefits presented above, it is necessary to define the best way to introduce SPL concepts in their development process. In accordance with Krueger [Kru02], there are three adoption approaches for an organization moving from a traditional development process to a SPL process: **proactive**, **reactive**, and **extractive**. With the **proactive** approach, an organization defines the requirements and designs and develops from scratch all the assets of a software product line. Thus, it is an expensive and a high risk approach because the requirements may have been misinterpreted or they can even change during or after the SPL assets development. On the other hand, a **reactive** approach incrementally increases the SPL when a new software product is demanded by a client. Unlike the **proactive** approach, a **reactive** approach allows an organization with a low cost and less effort to develop a new product, since some assets are already developed. In turn, an **extractive** approach is an ideal option when the organization already has a related set of software because it is easy to identify and document the commonalities and the differences among them. Furthermore, during the Product Line Engineering (SPLE), the common artifacts can be extracted from the systems and can be used as part of the SPL core assets.

2.2.1 Software Product Line Engineering

Traditional software engineering is focused on designing and developing only an individual system at a time and with a few opportunistic reuse of the code assets. On the other hand, Software

Product Line Engineering (SPL) is a distinct approach, which is focused on the reuse of all assets developed during the software development life-cycle, such as, the requirements, architecture, source code and testing artifacts, to generate a set of related systems. The collection of all these reusable assets composes the product line infrastructure. Another key difference is that SPL is based on two separate life-cycles: Domain Engineering and Product Engineering (see Figure 2.4) [PBL05].

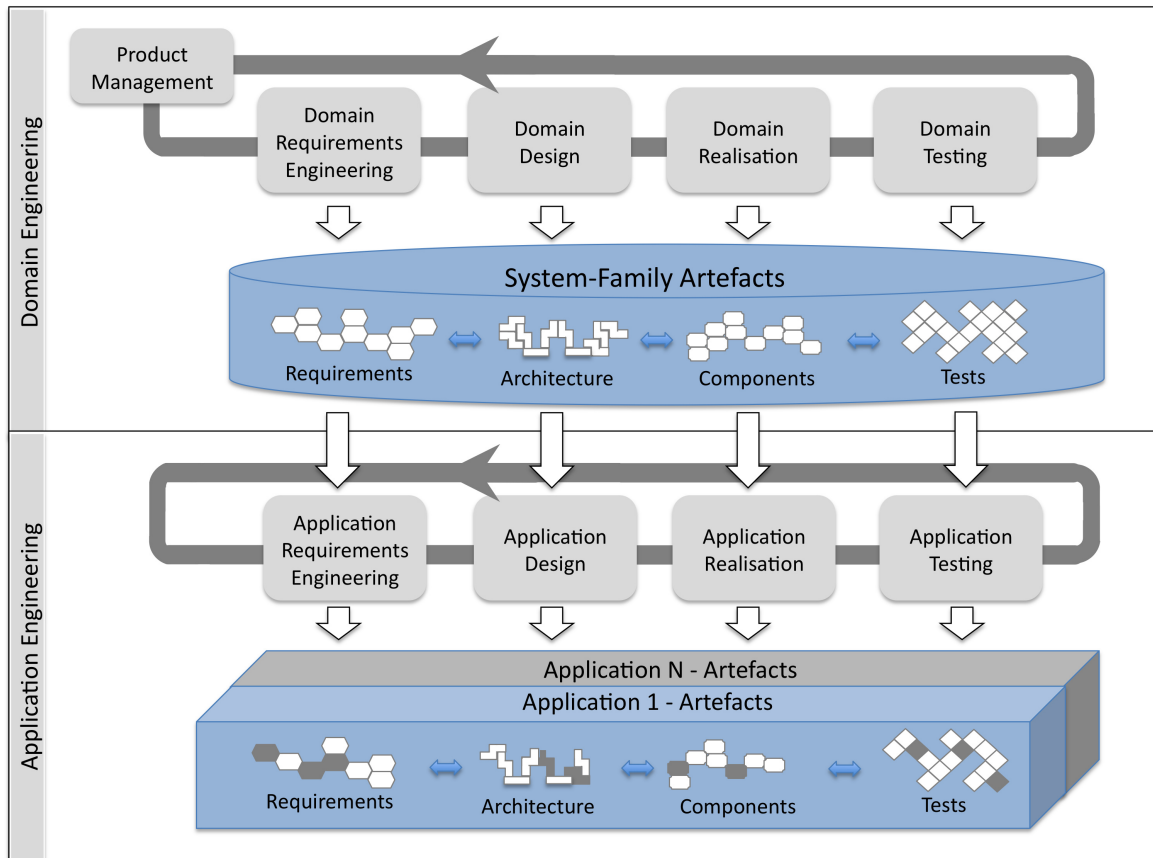


Figure 2.4: Software Product Line Engineering Life-cycles [LSR07]

Domain Engineering

The Domain Engineering life-cycle is focused on defining the SPL scope and generating all common assets, from requirements to testing, that compose the SPL platform. Thus, based on these common assets it defines the platform variability to support a generation of the different systems. Next we briefly introduce the five sub-processes that compose the Domain Engineering life-cycle [PBL05] (represented by the top five gray boxes in Figure 2.4):

- *Product Management*: the main goals of the product management sub-process are the definition of the scope of the SPL and the management of the organization's product portfolio. The results of this process are the product roadmap, which defines the common and variable features of products, the schedule to release each product, the definition of all generated products and the list of all reusable artifacts [KCH⁺90].

- *Domain Requirements*: requirement engineering uses the product roadmap and the requirements of each desired system to define and document the common and variable requirements of the SPL. To achieve these goals, five phases are performed: *Elicitation*, which is the analysis of the users requirements; *Documentation*, where the users requirements are formalized in textual documents or models; *Negotiation*, where the requirements are discussed and negotiated with the users/stakeholders to reach a consensus; *Validation and Verification*, in this phase the objective is to verify if the requirements are correct and understandable; *Management*, its goal is to maintain and review the changes in the requirements during the product line life-cycle. As result of these phases, during the domain requirements, we can name the set of common variables and well-defined requirements and the SPL variability model.
- *Domain Design*: the main focus of this sub-process is to use the domain requirements and the product line variability model to define the Software Product Line Architecture (SPLA). The SPLA is a common and high level structure that will be used for all the products of the SPL.
- *Domain Realization*: in the domain realization, the SPLA and the previously developed domain assets are used to design and implement the domain software components. At this point, the components are developed to be easily configurable and loosely coupled, but they are not yet an executable application.
- *Domain Testing*: during the domain testing the developed components at the domain realization are tested against their specification (e.g., requirements and SPLA). To reduce the time and effort during the domain testing, the developed test artifacts (e.g., test plan and test cases) to test the components are also reusable and can be used to test a new or a modified component.

It is important to highlight that the Domain Engineering sub-process are presented and separated in this thesis, they are directly connected to the respective application engineering sub-process (represented by an arrow - Figure 2.4). As show in Figure 2.4, Domain Engineering generates and manages a set of common assets that in turn are used in the application engineering to generate the SPL products. On the other hand, the application engineering sub-process provides feedback that can be used to guide changes and/or the development of new assets in the Domain Engineering [PBL05].

Product Engineering

In the application engineering life-cycle, the common and variable assets developed in the Domain Engineering are combined with specific assets to create a SPL product. The application engineering is composed by four sub-process, which receive as an input its contra-part in Domain Engineering to generate application artifacts [PBL05]. The application engineering sub-process, which is represented by the four bottom gray boxes in Figure 2.4, is briefly introduced bellow:

- *Application Requirements*: the application requirements sub-process uses the domain requirements and the product line roadmap in addition to the target application's specific requirements to generate the requirements of a specific product.
- *Application Design*: during the application design, the SPLA is used to instantiate the architecture of a specific application. Therefore, the desired parts of the SPLA are selected (its variations points are resolved) and some adaptations related to the specific product are added.
- *Application Realisation*: the main focus of the application realisation is to use the application architecture, the domain realisation components and the realisation of specific application components to generate an executable product. For instance, in a simple component-based SPL some domain realization components provide interfaces (variation point) that can be required by several domain variable components and application specific components (variants). Then, a product can be generated by the simple selection of one variable component to realize each provided interface.
- *Application Testing*: as in the traditional software development process, the applications generated from a product line must also be tested. Although the individual components were usually tested in the domain testing, the whole application must be tested during the application testing because it is almost impossible to test all the component combinations and some specific components might have been added. In the last years, several works that present approaches, techniques and tools related to SPL testing can be found in the literature [OG05] [ER11] [LKL12].

2.2.2 SPL Variability Management

As previously mentioned, during the Domain Requirement sub-process (see Figure 2.4), the requirements of each desired system are elicited to define the common and variable requirements of the SPL. After that, during the other Domain Engineering sub-processes the variability is spread through all domain artifacts, such as, architecture and components.

Since SPL engineers, users, developers and testers have a different view of the same SPL and its products, there are different approaches to document variability in requirements (SPL engineers and users), design (SPL engineers/architectures), realisation (developers) and testing artifacts (testers) [PBL05]. For instance, to document variability in domain requirements textual documents or models can be used. On the other hand, in domain design, UML component, class or package diagrams can be used [OGM10]. To avoid misinterpretation among many variability documentation, it is important to have a way to communicate the SPL variability among different stakeholders. Feature models (FM) is an example of an intuitive notation, for both customers and developers, that can be used to express commonalities and variabilities of a software product line [KCH⁺90] [SHTB07].

Feature Model

In the seminal work of Kang *et al.* [KCH⁺90] introduces the Feature Oriented Domain Analysis (FODA) and presents the FM definition as follows:

A feature model represents the standard features of a family of systems in the domain and relationships between them...The feature model serves as a communication medium between users and developers. To the users, the feature model shows what the standard features are, what other features they can choose, and when they can choose them. To the developers, the feature model indicates what needs to be parameterized in the other models and the software architecture, and how the parameterization should be done.

Furthermore, Kang *et al.* [KCH⁺90] presented the basic feature diagram modelling elements as features and the relationship between the child features and its parent feature. In addition, the relationship between a feature and the FM cross-tree constrains are also classified. Three kinds of relationship among features and two kinds of FM constraints are described next:

- *Mandatory*: a mandatory relationship means that whenever the parent feature is selected, the child feature must be present in the product. For instance, in Figure 2.5 the *Body* feature will be always present in all products derived from that SPL.
- *Optional*: the optional relationship between a child feature and its parent feature means that the child feature can be present or not when its parents feature is selected. For instance, in the FM diagram presented in Figure 2.5 the *Cruise Control* feature is optional, so it can be present or not in a car.
- *Alternative*: an alternative relationship means that in the case that a parent feature was selected, one and just one child feature must be selected from the set of child features. In Figure 2.5 the *Transmission* and its child feature *Automatic* and *Manual* are an example of an alternative relationship. Thus, one and just one transmission can be present in a car: *Automatic* or *Manual*.
- *Excludes*: excludes is a cross-tree constraint that can be used to exclude a feature in the presence of another feature. The use of exclude constraint is exemplified in Figure 2.5, where feature *Manual* excludes *Electric* feature. Thus, in the case that a product from this product line has a manual transmission, the electric engine will not be available.
- *Requires*: a requires constraint means that in the case of the presence of a specific feature, it implies in selecting another feature. For instance, if a car from the product line has an automatic transmission (*Automatic* feature), the cruise control (*Cruise Control* feature) must be present in the car.

Since Kang's work, various extensions of the FODA feature model were introduced to reduce ambiguity and to support different methods, such as, the cardinality-based feature models [CHE05], the FeatureRSEB [GFA98] and Feature-oriented Reuse Method (FORM) [KKL⁺98]. For instance,

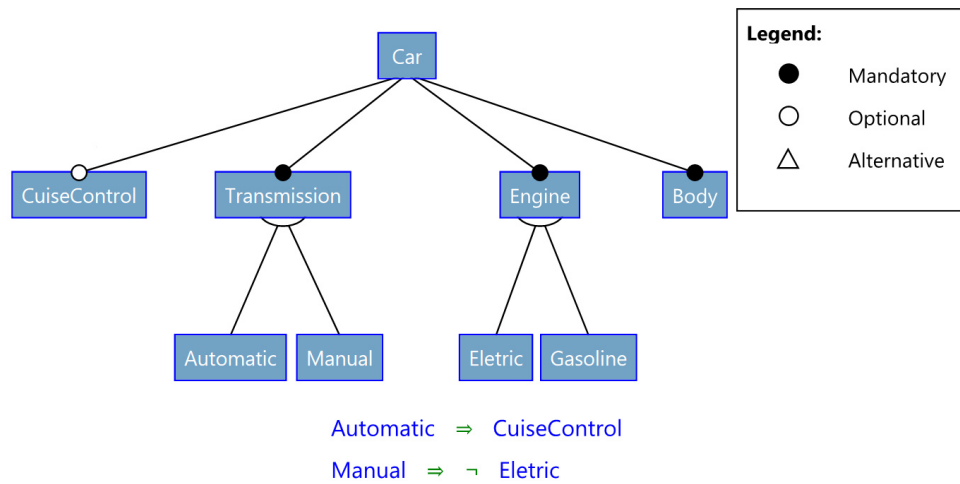


Figure 2.5: Example of a Car Feature Model - Slightly Adapted from [KCH⁺90]

in Griss *et al.* [GFA98], an extension to the FODA feature model to support an *Or* relationship between features was presented. An *Or* relationship means that one or more child features can be included in a product when its parent feature is included. On the other hand, Czarnecki *et al.* [CHE05], proposed an extension to the FODA notation to support feature cardinality. Thus, based on this extension an optional feature can be modelled as a feature cardinality relationship with cardinality [0..1] and a mandatory feature can be modelled as a feature cardinality relationship with cardinalities [1..1]. Thum's *et al.* [TKES11] discussed the explicit definition of abstract features in the FM. They also claim that kind of features are just used to structure a feature model but do not have any impact on domain/product realisation. This approach simplifies the mapping between features and components, improves the identification of the valid combinations of features and also the combinations of realisation components.

Variability mechanisms

As previously presented in Section 2.2.1, during the Domain Requirement the common and variable requirements of the SPL are elicited and then the variability is spread throughout all domain artifacts, such as, architecture and components (see Figure 2.4). Linden *et al.* [LSR07] argue that in an abstract level, there are three techniques to realise variation of architecture and several kinds of concrete variation mechanisms (see Figure 2.6):

- *Adaptation*: it is a technique in which a component has only a single implementation, but it changes its behavior by the use of configuration files. An example of the concrete use of adaptation to realise the variation points in software assets is by the use of a compile-time configuration mechanism. In compile-time configuration, the variability is implemented in the software artifacts through the use of selective compilers mechanism, such as *ifdefs* statements.
- *Replacement*: the replacement technique is based on the conception that there are several implementations of a component, where each one of the implementations follows a different

component specification. An example of a concrete use a replacement in software assets is the use of component replacement as a variability mechanism. The component replacement mechanism allows the substitution, at compile time, of the standard component by each one of the different implementation of the components (since it implements the same interface).

- *Extension*: this technique requires that generic interfaces are provided to support the addition of several components. Plug-ins [Som11] is a concrete example of an extension technique in software artifacts since it can be used to extend a system functionality by its addition through a generic system interface.

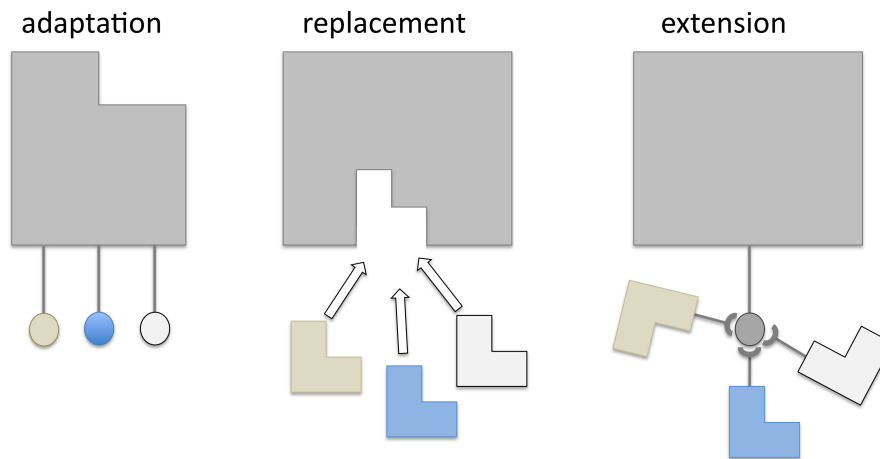


Figure 2.6: Realising Variability at Architecture [LSR07]

2.3 Related Work

Several Model-based Testing (MBT) supporting tools are discussed in the literature [UL06] [DNSVT07] [SL10] [SMJU10]. Utting and Legard [UL06] presented a work where eleven MBT tools were discussed and categorized by modelling perspectives and modelling notations. In the Dias Neto's *et al.* work a systematic review on MBT approaches is presented, where 78 MBT works were evaluated and classified according to their testing level, software domain, behavior model, test coverage criteria, test case generation criteria and automation level. Shafique and Labiche [SL10] performed a systematic review of MBT tools, where twelve tools were selected and evaluated in accordance with their test criteria, automation coverage, model type, category and platform. In contrast, Sarma *et al.* [SMJU10] presented an industrial case study where the commercial MBT tools Conformiq's Qtronic [Hui07] and Microsoft's SpecExplorer [VCG⁺08] were evaluated according to the following criteria: model representation (type of model), model validation (detect or not requirement's defects), test generation strategy (coverage criteria), test data generation strategy (fully automated or not), concurrency support (support physically distributed process), cost and complexity of applying the MBT tool, testing level and usability.

Despite the crescent amount of commercial, academic and open-source MBT tools available and the fact that some of these tools share some common features, the use of SPL concepts to generate MBT tools has not received much attention by field practitioners. In addition to our work, we only identified one initial work investigating the development of a product line of testing tools [TAR]. However, this work is in an initial stage and focus only on the generation of testing tools to support the generation of textual test cases from textual specification. Furthermore, the generated tools do not address the generation of concrete test cases for different testing techniques or testing levels. On the other hand, there are an increasing amount of works related to testing a software product line [ER11], [OG05]. For instance, the Engestrom's work [ER11] is focused on identifying the main challenges in SPL testing and also what topics have been investigated in the last years.

2.4 Chapter Summary

As we discussed in this chapter, software testing techniques, in addition to other verification and validation techniques, is one of the most used techniques in the software development industry, constituting one key element to provide evidence of reliability of a software system. However, the software testing process has a high cost and effort when compared to the other phases of the software development process. Thus, automation of software testing through reuse of software artifacts is a good alternative for mitigating the effort and cost, but its full automation still a *dream* [Ber07]. Model-based Testing (MBT) is a technique to support the (semi)automatic derivation of testing cases and scripts from the software models. MBT can help to reduce the cost of software testing since it automates the testing phase [MS04]. Although, to a testing team to take advantage of MBT, tool support is mandatory. Nowadays, a number of commercial, academic and open-source MBT tools are available, and some of these tools are based on the same system models, coverage criteria, methods and notations. However, despite the fact that some of these tools use a similar process and in some cases the same system models, even when generating tests for different testing levels, they have been individually developed for different companies or research groups. In this context, it would be relevant to apply SPL concepts to reuse assets to support the generation of testing tools that use MBT. However, to adopt a SPL approach to support the generation of tools that use MBT, it is necessary to understand the domain (and available tools that use MBT) to define the domain requirements and also to support the identification of its common and variable features, e.g., models and testing levels [PBL05].

Chapter 3 will present a Systematic Mapping Study (SMS) [PFMM08], to map out MBT, in order to find evidence that could be used to improve the MBT adoption. Furthermore, we will focus on map academic and industry tools that use MBT, test coverage criteria and the automation of various testing activities, such as, model creation, model verification, test case generation and test case execution.

3. A SYSTEMATIC MAPPING STUDY ON MBT

A Systematic Mapping Study (SMS) is a secondary study method used to provide evidence about a specific topic of interest [PFMM08]. Therefore, it provides an overview of a research area, identifying the quantity and type of available research and mapping its results. It is important to mention that apart from others secondary studies, SMS was recently discovery as a method to aggregate and categorize primary studies in software engineering. On the other hand, Systematic Literature Review (SLR) is a method to examine in-depth and describe the methodology and the results of the primary studies, that has received a lot of attention in the last decade by software engineering practitioners. Although appearing similar, there are significant differences between systematic literature reviews and systematic mapping studies, as those discussed by the Kitchenham *et al.* [KBB]. For instance, while an SLR has specific research questions related to the studies' outcomes and the scope is focused on empirical papers related to the research questions, an SMS has generic research questions and the scope is broad and consequently focuses on a large number of papers related to a research topic or field.

In this chapter a Systematic Mapping Study of primary studies about Model-based Testing is presented. This SMS was performed in order to provide an overview about the MBT field. Furthermore, the SMS was also performed in order to map out the main tools used to support MBT, its requirements and common and variables features. Variability identification is a challenging activity during the design and development of a SPL, as well as, the definition of the features granularity [CGR⁺12]. Therefore, it is relevant that the SPL specialist understands the domain and the legacy applications (in the context of the thesis, the MBT tools) to extract its features. Thus, we will use the results of this investigation on MBT and MBT tools to support our decisions on the requirements, design and development of an SPL of MBT tools.

3.1 Related research

In the past, a few papers presented an SLR on MBT [DNSVT07] [SL10]. Dias Netto *et al.* [DNSVT07] performed a systematic review on MBT approaches in academic initiatives, which includes representation models, support tools, test coverage criteria, level of automation, intermediate models, and complexity of models. Shafique and Labiche [SL10] presented a SLR focused on MBT tools support in academic and industry, specifically on tools that rely on state-based models. The study compares test coverage criteria (for instance, adequacy criteria supported by the selected tools, script flow, data coverage and requirements coverage), automation coverage comparison for various testing activities (model creation, model verification, test case debugging, sub-modeling, test case generation, test case execution, requirements traceability), and support of test scaffolding (like adapter creation, oracle automation, stub creation, on-line testing, off-line testing).

Surveys of some MBT approaches and tools are discussed in some works [Bob08] [SD10] [MOSHL09]. Boerg [Bob08], for example, explores MBT on system level; the author applies MBT to a subsystem of an e-mail gateway system and presents results starting from early phases of software development that show that MBT significantly increased the number of faults identified during system testing.

Saifan and Dingel [SD10] performed another survey on MBT, which was focused on distributed systems. The authors highlight how to apply testing in distributed systems using MBT, thereunto, different quality attributes of distributed systems have been tested, such as security, performance, reliability, and correctness. Their proposal is to add three new attributes (the purpose of testing, the test case paradigm, and the type of conformance checking) in order to provide criteria to the classification. Finally, based on this classification, they perform a simple comparison between different MBT tools.

The last study provides a survey on model-driven testing techniques, specifically MBT approach [MOSHL09], and aims to compare the techniques presented in more than fifteen (15) MBT approaches. The authors perform a comparison among these different techniques, using the following criteria: modelling notation, automatic test generation, testing target, and tool support. Their main idea was to propose a reference model for someone that is willing to build an MBT tool.

Although the works discussed in this section cover several aspects regarding MBT approaches, methods, techniques, models, specifications and tools that support MBT, most of them do not systematically investigate the available tools and its common and variable features. Thus, it is relevant to map what are the actual MBT tools and also what are its main features, such as, modelling notation and test case generation methods.

3.2 Planning Review

In order to achieve the expected outcomes from our SMS, it is fundamental to follow a well-defined process that includes searching, screening, assessing and analysing the primary studies. In our SMS we follow the process proposed by Petersen *et al.* [PFMM08], which describes the process of SMS in Software Engineering. Therefore, our process is divided in three phases (see Figure 3.1): Planning, Conduction and Reporting. Each phase has two activities and each activity in turn results in an artifact. As shown in Figure 3.1 the final artifact of the process is the systematic map.

3.2.1 Scope and Objective

In our SMS we look for the main contributions to provide an overview of the MBT field, its process, models and tools from several works that were produced between 2006 and 2010 (inclusive) in academic and industrial domains. The main objective of this SMS is to provide a broad overview on MBT works, the main tools used to support MBT and also to map the main features that are

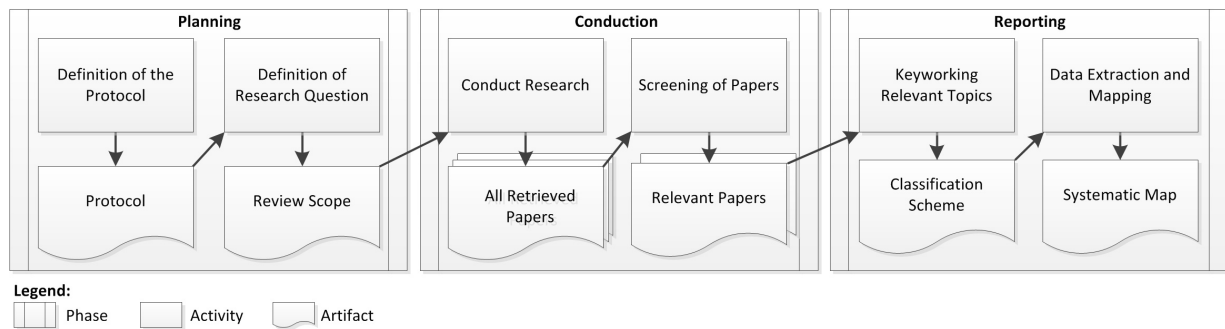


Figure 3.1: Systematic Mapping Studies Process (adapted from [PFMM08])

provided by these MBT tools. These results can provide insight to researchers or test engineers to decide to adopt an existing tool or even to define the requirements to develop a new one.

3.2.2 Questions Structure

The research questions were structured in four criteria:

- *Population*: Research on Software Testing.
- *Intervention*: Model-based Testing approaches.
- *Outcome*: The expected results are approaches, methods, methodologies, techniques, models, specifications and tools that are currently used in the MBT process.
- *Context*: Identify which MBT approaches, methods, methodologies and techniques and tools have been proposed between 2006 and 2010.

3.2.3 Research Questions

As mentioned before, the objective of this SMS is to map out the MBT field and also the main tools that have been used to support MBT, its requirements and common and variables features. In addition, we also focus on identifying in which domains MBT is applied and what the most used models or specifications are. To support the definition of the SMS research scope, we used the topics already addressed by the related research as a basis (see Section 3.1) and also the contributions from MBT researchers and industrial practitioners. Based on that, we defined the following research questions (RQ):

RQ1. *How many MBT works were published between 2006 and 2010?*

RQ2. *What are the industrial and academic MBT supporting tools?*

RQ3. *In which application domains is MBT applied to?*

RQ4. *What the models or specifications are used in MBT?*

3.2.4 Search Process

Databases

In order to perform our search, we have used databases that (1) have a web-based search engine; (2) have a search mechanism able to use keywords; and, (3) contain computer science papers. Our selection includes the ACM Digital Library, IEEE Xplore, Springer Link, SCOPUS and Compendex ¹.

Terms and Synonymous

We used structured questions to construct search strings [KC07].

Table 3.1: Definition of Search Strings

| Structure | Terms | Synonyms |
|---------------------|------------------------|---|
| Population | Software | |
| Intervention | Model-based Testing | MBT Model-based Test Model-based Software Testing |
| Outcome | Approach | Method Methodology Technique |

String

We used the boolean operation "OR" to select alternate words and synonyms, and boolean operation "AND" to select terms for population, intervention and outcome.

```
((MBT OR "model-based testing" OR "model based testing" OR "model-based test" OR "model based test" OR "model-based software testing" OR "model based software testing") AND (approach OR method OR methodology OR technique) AND (software))
```

However, the constructed string could not be used as planned, since some of the databases have some specific ways to deal with search strings. In such cases, particular strategies were adopted to construct one or more strings for those search engines (see Section 3.3.1).

¹dl.acm.org; ieeeexplore.ieee.org; link.springer.com; scopus.com; engineeringvillage.com

3.2.5 Inclusion and Exclusion Criteria

An important activity during the SMS planning is the definition of the Inclusion (IC) and Exclusion Criteria (EC). These criteria support the selection of the appropriate papers (IC) and to reduce the number of papers that are returned by the search engines (EC). The IC and EC of our SMS are:

- **IC1:** The paper must have been published between 2006 and 2010;
- **IC2:** The study must be available on the Web;
- **IC3:** The paper must be written in English;
- **IC4:** Research papers and technical reports that present some MBT contribution;
- **EC1:** Studies that describe how to apply some MBT tool to perform test activities, but the work does not present a relevant contribution to the MBT process;
- **EC2:** Studies that do not contain some type of evaluation: examples, case study, experiment, or proof of correctness. This evaluation must contain some kind of analysis, showing the achieved results.

3.2.6 Quality Assessment Criteria

The quality/assessment criteria (QA) are:

QA1. *Does the study describe a new contribution to MBT process?*

QA2. *Is there some type of evaluation?*

QA3. *Do the authors present some kind of analysis, showing the achieved results?*

QA4. *Does the study describe the used models?*

QA5. *Does the study use an MBT tool?*

For each of the above questions, the following scoring was used: Y (yes) = 1; P (partly) = 0.5, N (no) = 0. Hence, the total score, sum of five questions, could result in: 0 to 1.0 (very poor); 1.5 or 2.0 (fair); 2.5 or 3.0 (good); 3.5 or 4.0 (very good) and 4.5 or 5.0 (excellent).

In order to grade each paper, the reader has to respect the following criteria:

QA1. Y: relevant contributions are explicitly defined in the study; P: relevant contributions are implicit; N: relevant contributions cannot be identified and are not clearly established;

- QA2.** Y: the study has explicitly applied evaluation (for example, a case study, an experiment, or proof of correctness); P: the evaluation is a simple “toy” example; N: no evaluation has been presented;
- QA3.** Y: the authors present some kind of analysis or show the achieved results; P: only a summary of the achieved results is presented; N: neither analysis nor results are specified;
- QA4.** Y: the models or modelling languages are clearly specified; P: the models or modelling languages are slightly described; N: the models or modelling languages cannot be identified;
- QA5.** Y: the study presents a proposal of an MBT tool or demonstrates its use; P: the study either describes or demonstrates only a proposal of the MBT tool, never both; N: a proposal of an MBT tool is not shown in study.

3.2.7 Selection Process

Our selection process is divided in six steps:

1. **Search databases:** Initially, strings were generated by means of the selected keywords and synonyms. Initial selection: an initial selection was carried out based on the criteria mentioned in Section 3.2.5;
2. **Eliminate redundancies:** There were some redundancies since some studies were returned by different search engines. In this step, we eliminated and recorded those redundancies;
3. **Intermediate selection:** The title and the abstract (reading the introduction and conclusion when necessary) were read for each study returned by the search engines;
4. **Final selection:** In this step, all studies were completely read, and we applied the same criteria as the intermediate step;
5. **Eliminate divergences:** If there were any divergences or doubts about the studies, a group of specialists would read the studies and discuss whether the study should or should not be included in the final selection;
6. **Quality analysis:** Based on the quality criteria (see Section 3.2.6), we evaluate the quality of studies that were read in the Final Selection step. The quality criteria were evaluated independently by two researchers; therefore, reducing the likelihood of erroneous results and/or bias.

3.2.8 Data Analysis

The data was tabulated to show:

- The number of selected studies per year (RQ1);
- The percentage of selected studies per source type (RQ1);
- Identify the MBT tools, classified as industrial or academic, on testing level, on testing techniques, and their characteristics (e.g., modelling notation and test case generation) according to the MBT process (RQ2 and RQ3);
- The number of selected studies per models or specifications, according to the MBT taxonomy presented in [UPL12] (RQ4).

3.3 Conduction

The SMS was conducted for a period of three months (February/2011 to April/2011), according to the plan presented in the previous sections. In all, 803 papers were retrieved. In this section, we present details of steps "Search databases" and "Quality analysis" presented in Section 3.2.7.

3.3.1 Search

The constructed string (see Section 3.2.4) could not be used as planned, since some of the databases have some specific ways to deal with search strings. Hence, in this section we present the strings that were used in each of the web search engines. Actually, there is a mapping between the constructed string and a string for each database.

We limited our search to the "Abstract", "Title" and "Keywords" (when available) fields in all databases, excluding, for example, the article "Body".

The ACM Digital Library, IEEE Xplore and SpringerLink search engines allow refining a search, *i.e.*, to determine the range of publication year. The range was defined as from 2006 and to 2010, inclusive. In the other search engines, this filter was embedded as part of the search string. We observed, also, that using "model based" or "model-based" as part of the search string would not change the output in the IEEE Xplore, Scopus and Compendex search engines.

IEEE Xplore

```
((MBT OR "model based testing" OR "model based test" OR
"model based software testing") AND (approach OR method
OR methodology OR technique) AND (software))
```

ACM Digital Library

```
(Abstract:(MBT OR "model-based testing" OR "model based testing" OR "model-based test" OR "model based test" OR "model-based software testing" OR "model based software testing") AND Abstract:(approach OR method OR methodology OR technique) AND Abstract:(software)) OR (Title:(MBT OR "model-based testing" OR "model based testing" OR "model-based test" OR "model based test" OR "model-based software testing" OR "model based software testing") AND Title:(approach OR method OR methodology OR technique) AND Title:(software))
```

SpringerLink

```
ab:((MBT or "model based testing" or "model based test" or "model based software testing") and (approach or method or methodology or technique) and (software))
```

Scopus

```
(TITLE-ABS-KEY(mbt OR "model based testing" OR "model based test" OR "model based software testing") AND TITLE-ABS-KEY(approach OR method OR methodology OR technique) AND TITLE-ABS-KEY(software)) AND (LIMIT-TO(PUBYEAR, 2010) OR LIMIT-TO(PUBYEAR, 2009) OR LIMIT-TO(PUBYEAR, 2008) OR LIMIT-TO(PUBYEAR, 2007) OR LIMIT-TO(PUBYEAR, 2006)) AND (LIMIT-TO(LANGUAGE, "English"))
```

Compendex

```
(TITLE-ABS-KEY(mbt OR "model based testing" OR "model based test" OR "model based software testing") AND TITLE-ABS-KEY(approach OR method OR methodology OR technique) AND TITLE-ABS-KEY(software)) AND ( LIMIT-TO(PUBYEAR,2010) OR LIMIT-TO(PUBYEAR,2009) OR LIMIT-TO(PUBYEAR,2008) OR LIMIT-TO(PUBYEAR,2007) OR LIMIT-TO(PUBYEAR,2006) ) AND ( LIMIT-TO(LANGUAGE,"English"))
```

After submitting the search strings to each search engine, a total of 803 studies were returned. After the elimination of redundancies, 448 studies were excluded since they were duplicated. In the next step we excluded the papers based on the title and the abstract, 355 studies were analysed to identify studies that present contributions to the MBT field, resulting in 50 studies. After full paper readings we had 46 papers that were analysed to verify if they would meet the quality criteria (see Table 3.2).

Table 3.2: Search Engines, Retrieved and Selected Primary Studies

| Databases | Retrieved | Selected | % |
|--------------|-----------|----------|--------|
| ACM | 104 | 7 | 21,21% |
| Compendex | 219 | 11 | 33,33% |
| IEEE | 160 | 13 | 39,39% |
| Scopus | 289 | 0 | - |
| SpringerLink | 31 | 2 | 6,07% |

3.3.2 Study Quality Assessment

Table 3.3 provides information of the quality studies scores included in the SMS. Each study can be identified through the column **ID** and its references presented in the column **Reference**, as well as publication year in the column **Year**. Columns **1**, **2**, **3**, **4** and **5** show scores based on the Quality Assessment (QA). Column **Sc** shows the final score for each study and column **Des** presents some subjective information to help in the understanding of the score assigned to each primary study.

Each of the 46 studies was assessed independently by two researchers according to the five QA shown in Section 3.2.6. Taken together, these criteria provided a measure of the extent to which we could be confident that a particular study could give a valuable contribution to the mapping study. We used the studies' quality assessment as a threshold for the inclusion/exclusion decision, to identify the primary studies that would form a valid foundation for our study. Finally, papers that scored at least 2.5 points were selected, *e.g.*, 33 studies were selected (see Tables 3.2 and 3.3).

Table 3.3: Quality Studies Scores

| Studies | | | QA | | | | | Quality | | Studies | | | QA | | | | | Quality | |
|---------|-----------------------|------|----|---|---|---|---|---------|-----|---------|----------------------|------|----|---|---|---|---|---------|-----|
| ID | Reference | Year | 1 | 2 | 3 | 4 | 5 | Sc | Des | ID | Reference | Year | 1 | 2 | 3 | 4 | 5 | Sc | Des |
| 01 | [ABT10] Abbors | 2010 | Y | P | P | Y | Y | 4.0 | V | 24 | [HBAA10] Hemmati | 2010 | N | Y | Y | N | N | 2.0 | F |
| 02 | [ADM06] Allen | 2006 | P | N | P | P | P | 2.0 | F | 25 | [IAB10] Iqbal | 2010 | Y | Y | Y | Y | Y | 5.0 | E |
| 03 | [AB06] Andaloussi | 2006 | Y | Y | P | Y | Y | 4.5 | E | 26 | [JTG+10] Jiang | 2010 | P | Y | Y | P | Y | 4.0 | V |
| 04 | [AS10] Athira | 2010 | P | P | P | P | N | 2.0 | F | 27 | [KKP06] Kandl | 2006 | P | Y | N | Y | Y | 3.5 | V |
| 05 | [APUW09] Aydal | 2009 | P | Y | Y | Y | P | 4.0 | V | 28 | [KMPK06] Kervinen | 2006 | P | Y | P | Y | Y | 4.0 | V |
| 06 | [Ben08] Benz | 2008 | Y | P | Y | Y | Y | 4.5 | E | 29 | [LG10] Louchau | 2010 | Y | Y | N | Y | Y | 4.0 | V |
| 07 | [Bob08] Boberg | 2008 | P | Y | Y | N | P | 3.0 | G | 30 | [LMG10] Loffler | 2010 | Y | Y | P | Y | Y | 4.5 | E |
| 08 | [BK08] Bringmann | 2008 | P | Y | P | P | P | 3.0 | G | 31 | [MSK+07] Mathaikutty | 2007 | P | P | P | P | N | 2.0 | F |
| 09 | [BDLRM09] Brito | 2009 | N | Y | P | P | N | 2.0 | F | 32 | [Mem07] Memon | 2007 | P | P | Y | P | P | 3.0 | G |
| 10 | [CNM07] Cartaxo | 2007 | P | P | P | Y | N | 2.5 | G | 33 | [NZR10] Naslavsky | 2010 | P | P | N | P | P | 2.0 | F |
| 11 | [CLS+09] Chinnapongse | 2009 | P | Y | P | P | P | 3.0 | G | 34 | [NSS10] Nguyen | 2010 | P | Y | P | Y | Y | 4.0 | V |
| 12 | [CAM10] Cristia | 2010 | N | P | P | P | P | 2.0 | F | 35 | [Oli08] Olimpiew | 2008 | Y | Y | Y | Y | Y | 5.0 | E |
| 13 | [DNT09] Dias Neto | 2009 | P | Y | Y | N | N | 2.5 | G | 36 | [Par06] Paradkar | 2006 | N | Y | P | P | P | 2.5 | G |
| 14 | [DEFM+10] Dorofeeva | 2010 | N | Y | P | P | N | 2.0 | F | 37 | [Puo08] Poulitaival | 2008 | Y | P | P | P | P | 3.0 | G |
| 15 | [EAXD+10] Elariss | 2010 | P | Y | Y | N | P | 3.0 | G | 38 | [SMJU10] Sarma | 2010 | Y | Y | Y | Y | Y | 5.0 | E |
| 16 | [FIMR10] Farooq | 2010 | Y | Y | Y | Y | Y | 5.0 | E | 39 | [SHH07] Schulz | 2007 | P | P | Y | Y | Y | 3.5 | V |
| 17 | [FL09] Farooq | 2009 | P | Y | Y | P | N | 3.0 | G | 40 | [SWK09] Stefanescu | 2009 | Y | P | P | Y | Y | 4.0 | V |
| 18 | [FLG10] Feliachi | 2010 | N | Y | N | P | P | 2.0 | F | 41 | [SWW10] Stefanescu | 2010 | P | P | P | Y | Y | 3.5 | V |
| 19 | [GHV07] Gonczy | 2007 | Y | P | P | Y | Y | 4.0 | V | 42 | [VCG+08] Veanes | 2008 | Y | P | N | Y | Y | 3.5 | V |
| 20 | [GFF+10] Grasso | 2010 | P | Y | P | N | N | 2.0 | F | 43 | [VSM+08] Vieira | 2008 | P | Y | P | N | N | 2.0 | F |
| 21 | [Gro10] Groenda | 2010 | P | P | P | P | P | 2.5 | G | 44 | [YX10] Yu | 2010 | P | P | N | P | N | 1.5 | F |
| 22 | [HGB08] Hasling | 2008 | Y | Y | P | Y | Y | 4.5 | E | 45 | [ZLLW09] Zeng | 2009 | P | P | P | P | N | 2.0 | F |
| 23 | [HJK10] Heiskanen | 2010 | Y | Y | N | P | N | 2.5 | G | 46 | [ZSH09] Zhao | 2009 | Y | P | P | P | N | 2.5 | G |

Legend - **Y**: Yes, **N**: No, **P**: Partly **Sc**: Score, **Des**: Description, **F**: Fair, **G**: Good, **V**: Very Good, **E**: Excellent.

3.4 Result Analysis

3.4.1 Classification Schemes

As shown in Figure 3.1, which presents the systematic process we followed, our Classification Schemes are generated by the activity "Keywording Relevant Topics". Keywording is performed in two steps. The former one is the one in which we read abstracts (introduction and conclusion when necessary) and identified keywords, concepts and context of the research that corroborate the contribution of the papers. In the latter step, keywords are merged and combined to develop an abstract level understanding from different selected papers. This step helps to define the categories that represent the population of the selected studies. It is also responsible for clustering these categories in the mapping.

During the "Keywording" activity, six main categories were created: *i*) the testing level, which for example could be unit, integration, system, acceptance and regression testing; *ii*) software domain, *e.g.*, education, automotive, health care, service etc.; *iii*) research type, *i.e.*, industrial experience, theoretical, proof of concept, experimental study and empirical study; *iv*) contribution type, *i.e.*, model, language, tool, method, technique, approach, framework and strategy; and, *v*) MBT process, *i.e.*, test modeling, model transformation, test case generation, test case instantiation, test case selection, test oracle and validation; *vi*) type of model, *i.e.*, UML, Markov Chain, Petri Nets, Labeled Transition System (LTS), etc.

Some categories were derived from the keywords, *e.g.* testing level, software domain and contribution type. However, the research type facet, which reflects the research approach used in the papers, is general and independent from a specific focus area.

One example of a detailed description of categories is shown in Table 3.4 [PFMM08] [DNTSV07]. We believe the other facets are self-explanatory.

3.4.2 Mapping

The qualitative evaluation of the literature is described in this section. This evaluation is related to the research questions presented in Section 3.2.2.

Figure 3.2 shows the bubble graph with the domain distribution (central Y axis) of primary studies in relation to the publication year (left side of X axis) and testing level (right side of X axis). The intersection bubble between axes contains the reference of the primary studies, and the size of the bubble depicts the number of studies.

From the 33 resulting studies, the number of primary studies describing each testing level was: 1 (Acceptance), 6 (Integration), 20 (System), 5 (Regression) and 1 (Not Applicable). As can be seen in the figure, there was no work on unit testing. On the other hand, most of the works

Table 3.4: Research Type Facets

| Category | Description |
|------------------------------|---|
| Experimental Study | “Techniques investigated are novel and have not yet been implemented in practice. Techniques used are for example experiments, <i>i.e.</i> , work done in the lab.” |
| Empirical Study | “Techniques are implemented in practice and an evaluation of the technique is conducted. That means, it is shown how the technique is implemented in practice (solution implementation) and what are the consequences of the implementation in terms of benefits and drawbacks (implementation evaluation). This also includes to identify problems in industry.” |
| Industrial Experience | “Experience papers explain on what and how something has been done in practice. It has to be the personal experience of the author”. |
| Proof of Concept | “A solution for a problem is proposed, the solution can be either novel or a significant extension of an existing technique. The potential benefits and the applicability of the solution is shown by a small example or a good line of argumentation”. |
| Theoretical | “These papers sketch a new way of looking at existing things by structuring the field in form of a taxonomy or conceptual framework”. |

(60%) are based on system testing. For the publication year of the selected primary studies, the distribution per year was: 4 (2006), 4 (2007), 7 (2008), 6 (2009) and 12 (2010).

The analyses of the graph shown in Figure 3.2 aims to answer the research questions RQ1, *i.e.*, 33² works were published between 2006 and 2010, and RQ3, *i.e.*, studies that use MBT in the following application domains: ATM, automotive, CRM, education, e-mail, ERP, office, game, project, protocol, health care, service and telecommunications. Apparently, MBT can be applied in several application domains, but some of them stand out compared to others, such as: automotive, health care and telecommunications.

Another facet from the “Keywording” activity (see Section 3.4.1) is related to the MBT process, which resulted in the following number of studies: 5 (test modeling), 2 (model transformation), 18 (test case generation), 1 (test case instantiation), 1 (test case selection), 3 (validation) and 3 (others). Most of the studies describe techniques and methods for generating test cases and only a few describe details on how to create test models.

From the type of model, the primary studies were organized using the following categories:

- **UML:** papers describing MBT that use UML diagrams (including profiles and similars) as the mechanism to describe test models;
- **Formal Models (FM):** papers describing MBT that use Formal Models (*e.g.*, Finite State Machines (FSM) or LTS) as the mechanism to describe test models;
- **UML-FM:** papers describing MBT that combines UML and Formal Models as the mechanism to describe test models;

²We considered only the final selected papers, but actually 355 were returned at the beginning of the search process.

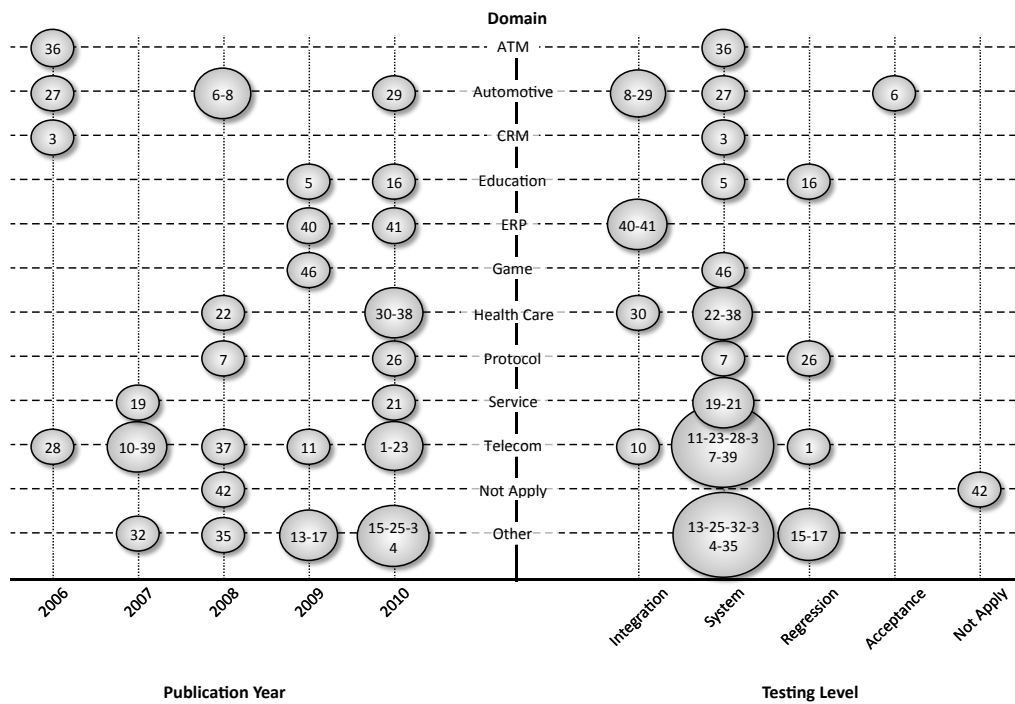


Figure 3.2: Bubble Plot of the Domain Studies Distribution by Publication Year and Testing Level

- **Others (OT):** papers describing MBT that use another proposed model (*ad hoc* model) instead of using UML or Formal Models as the mechanism to describe test models;
- **UML-OT:** papers describing MBT that combines UML and another proposed model as the mechanism to describe test models;
- **FM-OT:** papers describing MBT that combines Formal Models and another proposed model as the mechanism to describe test models.

Figure 3.3 presents a Venn diagram that represents this test model categorization: 24.2% (8) UML; 6.1% (2) UML-FM; 9.1% (3) UML-OT; 39.4% (12) OT. In turn, the FM aggregates 30.3% for formal models, 18.2% (6) only FM and 6.1% (2) for each combining UML and another (OT) proposed model. Nevertheless, 51.5% (17) of papers used another (OT) proposed model, distributed in: 36.4% (12) only OT, 9.1% (3) combining UML+OT and 6.1% (2) combining FM+OT.

3.5 Threats to validity

The main threats that we have identified that can compromise the validity of our SMS on MBT are:

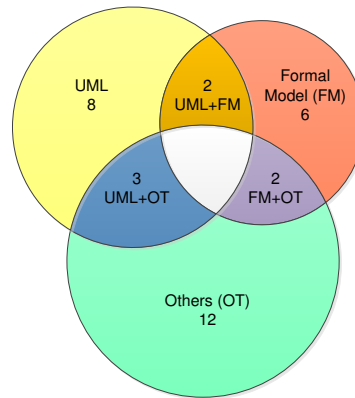


Figure 3.3: Venn Diagram of the Test Models Categorization

Publication bias: refers to the possibility of some papers presenting, for instance an MBT tool or model, which are not published because the research results did not yield the desired outcome, company-confidential results, or because the research was conducted on topics that do not fit into the common computing conferences and journals. As we analysed 355 papers on MBT, our SMS was not restricted to a small sample of the available papers, thus it minimizes the risk that some unpublished or unreturned papers during the searching process impact the SMS results.

Primary studies selection bias: usually, the SMS authors cannot guarantee that all relevant primary studies were returned during the searching process, as well as, during the paper evaluation by the authors. In order to mitigate this threat, the quality assessment criteria, as well as, the weight used to quantify each of them were evaluated independently by two researchers (step 6 of selection process); hence, this will hopefully help to reduce the likelihood of erroneous results and/or bias.

Vested interests of the authors: We are not aware of biases we may have had when analysing and categorizing the papers, but the reader should be aware of the possible impact of our own interests on the analyses. In particular, it is possible that the recommendations we make are affected by our interests and opinions.

Unfamiliarity with other fields: we defined the search strings based on our experience and the investigation of some systematic mapping studies on MBT, but we cannot completely avoid the possibility that some terms defined in the search strings have synonyms that we have not identified.

3.6 Discussion

In this section, we present and discuss the answers of our research questions.

1. *How many MBT works were published between 2006 and 2010?* After submitting the search strings to each search engine, 803 studies were recovered, which were reduced to 355 studies after the exclusion of duplicated papers. Thus, we consider that 355 papers were published

in the interval defined in our SMS. However, after we have applied the selection criteria and quality analysis, 33 papers remained. For purposes of comparison, Dias Neto et. al. [DNSVT07] presented an SMS in 2006, which focused on MBT works published until 2006. That SMS recovered 202 studies (already excluded studies whose scope was not related to MBT, or were repeated, or were unavailable) where 72 papers were selected to be analysed quantitatively and qualitatively. However, we cannot directly compare the results of that work with our SMS because the works used different search strings and were focused on different intervals of years. In our SMS we defined an interval of 5 years and Dias Neto et. al. [DNSVT07] did not define an interval. However, the SMS results show that MBT is still an active research field.

2. What are the industrial and academic MBT tools?

After the analysis of the selected papers, we identified 26 MBT tools, where seven (7) are commercial tools and nineteen (19) are academic tools, as follows:

- **Industrial:** Conformiq Qtronic [ABT10] [Puo08] [SMJU10] [SHH07]; AspectT (BMW) [Ben08], QuickCheck (Quviq) [Bob08], Spec Explorer [VCG⁺08] [JTG⁺10] [NSS10] [SMJU10], QTP (*Quick Test Pro*) [KMPK06] [NSS10], TPT (*Time Partition Testing*) [BK08], TDE/UML [HGB08].
- **Academic:** MATERA [ABT10], MagicDraw [ABT10], Alloy Analyzer [APUW09], NModel [CLS⁺09]³, UMLAUT (*Unified Modeling Language All pUrposes Transformer*) [CNM07], TGV (*Test Generation with Verification technology*) [CNM07], START [FIMR10], Groove [GHV07], LTSA (*Labelled Transition System Analyzer*) [GHV07], TEMA [HJK10], TargetLink [KKP06], TVT (*Tampere Verification Tool*) [KMPK06], GUITAR (*GUI Testing FrAmewoRk*) [Mem07], GTG (*GUI Test Generator*) [NSS10], CADeT (*Customizable Activity diagrams, Decision tables and Test specifications*) [Oli08], GOTCHA [Par06], MBT4Chor [SWK09], FOKUS!MBT [SWW10], MbtTigris [SWW10].

After the extraction and categorization of the commercial and academic tools from the papers, we focused our attention on identifying what the main tools used to support MBT are, as well as, their main features. However, detailed information about the tools implementation and their features is not present in most papers. Thus, we had to search the tools on the internet to try to find the tool's website or repository to download the tool's binaries or source and, when available, the tool's documentation. Table 3.5 presents the returned list of MBT tools and their main features. Basically, most of this information was extracted from the tools website or documentation, but in some cases it was necessary to install and execute the tools to try to identify some features. Based on this information, it was possible to identify, for instance, that most of these MBT tools use UML and FSM as modelling notations and that all tools are focused on functional testing.

³NModel is an Open Source Software to Model-based Testing and analysis framework for model programs written in C# <http://www.codeplex.com/NModel>

It is important to highlight that the tools information was extracted only from the selected works (33 papers). Thus, as the papers must present a relevant contribution and show some kind of evaluation and analyses (e.g., an experiment or a case study), it means that the tools are at some point consolidated and are not only a toy example. Furthermore, as the tools details presented in the Table 3.5 were based on the tool usage and/or their documentation, it also helps to support that these tools are at some point consolidated and useful. The downside of this approach is that there is a possibility that some papers that present an MBT tool were not identified, since the works were in an initial stage.

3. *In which application domains is MBT applied to?*

The SMS results show that the MBT works addressing several domains, such as ATM [Par06], automotive [Ben08] [BK08] [KKP06] [LG10], CRM [AB06], education [APUW09] [FIMR10], ERP [SWK09] [SWW10], games [ZSH09], network protocol [Bob08] [JTG⁺10], health care [HGB08] [LMG10] [SMJU10], services [GHV07] [Gro10] and telecommunications [ABT10] [CNM07] [CLS⁺09] [HJK10] [KMPK06] [Puo08] [SHH07]. Figure 3.2 shows that despite the fact that MBT is applied in several application domains, in some of these domains the use of MBT stands out, such as in the automotive (4 papers) and telecommunications (7 papers) domains.

4. *What are the modelling notations or specifications used to model the SUT?* The SMS results pointed out that several modelling notations or specifications have been used in the past years to model the SUT. We clustered these notations and specifications into three groups: UML, FM (Formal Model) and OT (Others) - see Figure 3.3.

- **UML:** UML [AB06] [CNM07] [FIMR10] [FL09] [GHV07] [HGB08] [LMG10] [Oli08] [SHH07] [SWK09], SysML [ABT10], UML Marte [IAB10], U2TP [AB06] [SWW10];
- **FM:** AsmL (*Abstract State Machine Language*) [VCG⁺08], TTCN-3 [AB06] [ZSH09], ASM (*Abstract State Machine*) [Bob08], Simulink/Stateflow [BK08] [KKP06] [LG10], LTS (*Labeled Transition Systems*) [CNM07] [KMPK06], FSM (*Finite State Machine*) [KKP06] [SWW10], EFSM (*Extended FSM*) [CLS⁺09] [JTG⁺10];
- **OT:** QML (*Qtronic Modeling Language*) [ABT10] [Puo08] [SMJU10], Z Model [APUW09], Alloy Modelling Language [APUW09], Ecore Model [Ben08], Erlang [Bob08], GUI Model [EAXD⁺10], AIAM (*Accuracy Information Annotation Model*) [Gro10], PCM (*Palladio Component Model*) [Gro10], Event Flow [Mem07], Mapping Model [NSS10], SALT (*Specification and Abstraction Language for Testing*) [Par06], Lyra modeling [SHH07], MCM (*Message Choreography Models*) [SWK09] [SWW10], TestingMM (*Testing Meta-Model*) [SWW10], Spec# [VCG⁺08], MSC (*Message Sequence Chart*) [ZSH09].

These results show that several UML profiles were adopted, such as: MARTE [IAB10], SysML [ABT10], U2TP [AB06] [SWW10]. The results also pointed out that UML is the most used modelling notation (representing 24.2% of the papers). In additional, some works [AB06]

[CNM07] proposed the mixed use of UML and formal models (representing 6.1%). On the other hand, some authors [SHH07] [SWK09] [SWW10] proposed the use of UML in combination with another modelling approach (representing 9.1% of the all papers). In turn, the use of FM as the system model is presented by [Bob08] [BK08] [CLS⁺09] [JTG⁺10] [KKP06] [KMPK06] (representing 18.2%). Few authors [VCG⁺08] [ZSH09] also presented the mixed use of FM and another model (representing 6.1%). Furthermore, several papers proposed the use of several different models (OT), representing 36.4% of the selected papers [APUW09] [Ben08] [DNT09] [EAXD⁺10] [FIMR10] [Gro10] [HJK10] [Mem07] [NSS10] [Par06] [Puo08] [SMJU10].

3.7 Chapter Summary

Model-based Testing is a huge and "alive" research field and every year a significant number of papers presenting different kinds of contributions are published (*e.g.*, approaches, process, modelling notations and tools support). Throughout the last decade some studies were focused on characterizing and analysing these contributions [DNSVT07] [Bob08] [Esl08] [SD10] [MOSHL09]. In this SMS we were interested in mapping out the MBT field, mapping the industrial and academic MBT supporting tools, the application domains in which MBT is applied and the most used modelling notations. We also analysed the MBT supporting tools to identify their main features, such as, input models, model redundancy, test generation criteria and the testing level. Thus, a tester or a testing analyst could use the SMS results to support the selection of a tool or even to define the requirements and the design of an MBT supporting tool. In the context of this thesis the SMS results will be used to provide, among others findings, domain expertise and also to support the identification of the features that must be present in an MBT tool. For instance, based on the list of MBT tools presented in Table 3.5, we can infer that all MBT tools must accept as an input some kind of SUT model (*e.g.* UML and FSM) and apply some test case generation technique (*e.g.* random generation) to generate test cases for some testing level (*e.g.* system and integration). Thus, we can identify, at a high level, which are the tools basic features and also their cross-tree constraints. Furthermore, we can also use the SMS results to define the tools requirements and to support some decisions during the SPL design, such as, the chosen of an implementation variability mechanism. Therefore, the SMS results is the starting point to define the requirements and to support our decisions on the design and development of our SPL of Model-based Testing tools.

Table 3.5: List of Model-based Testing Tools

| Tool Name | Type | Owner | Model | | | | Test Generation | | | Testing | | | |
|---------------------------|------|----------------------------------|---|---------|-------------------------|--------------------------------------|------------------|--|--|---------------------------|-----------|------------|---------------------|
| | | | Modelling Notation | Subject | Redundancy | Characteristics | Paradigm | Test Selection Criteria | Test Generation | Test Case Paradigm | Execution | Technique | Testing Level |
| Qtronic | C | Conformiq | Qtronic Modeling Language (QML), UML state machines with blocks of Java or C# | SUT | separated test model | non-deterministic, timed | transition-based | structural model coverage, requirements coverage | symbolic execution | symbolic execution, TTCN3 | both | functional | system |
| Spec Explorer | A | Microsoft Research | AsmL (Abstract State Machine Language), Spec#, FSM | SUT | separated test model | non-deterministic, untimed, discrete | state-based | random, shortest path, transition coverage, traversal algorithms | traversal algorithm | FSM | both | functional | system |
| QuickCheck | C | Quviq | Abstract State Machine (ASM), Erlang | SUT | Separated model | non-deterministic, untimed, discrete | transition-based | randomly, fault-based | symbolic execution, random generation | symbolic execution | offline | functional | system |
| TDE/UML | A | - | UML | SUT | shared test & Dev model | - | transition-based | data coverage | random generation | - | online | functional | system |
| TEMA | A | Tampere University of Technology | labelled state transition system (LSTS) | SUT | separated test model | deterministic, untimed, discrete | state-based | requirements coverage | random generation | action words | online | functional | system, acceptance |
| Tampere Verification Tool | A | Tampere University of Technology | labelled state transition system (LSTS) | SUT | separated test model | non-deterministic, untimed, discrete | state-based | requirements coverage | random generation | action words | online | functional | system |
| FOKUS!MBT | A | Fraunhofer Institut | UML model with U2TP, testing meta model (TestingMM), FSM | SUT | separated test model | non-deterministic, untimed, discrete | state-based | structural coverage criteria, state and transition coverage, shortest path, random | random generation, graph search algorithms | FSM with TestingMM | offline | functional | integration |
| MbtTigris GraphWalker | A | GraphWalker | FSM, extended FSM | SUT | separated test model | non-deterministic, untimed, discrete | state-based | structural coverage criteria | graph search | GraphML | both | functional | system |
| Time Partition testing | C | PikeTec | MATLAB/Simulink, Stateflow, TargetLink models | SUT | separated test model | deterministic, timed, continuous | data-flow | Ad-hoc test case specification | symbolic execution | - | offline | functional | integration, system |

Legend: A: Academic, C: Commercial, -: not available

4. A SOFTWARE PRODUCT LINE OF MODEL-BASED TESTING TOOLS - PLETS

In the previous chapter, we focused our research effort on identifying the main MBT supporting tools and their main features. In this chapter we introduce the motivation and discuss the requirements, which were extracted from our SMS and from our collaboration with a Technology Development Lab (TDL) of Dell Computer Brazil, the design decisions and the architecture of our Software Product Line of Model-based Testing Tools: PLeTs. The requirements and design decisions on the development of the PlugSPL environment, which is used to support the automatic generation of products from PLeTs are also presented in this chapter.

4.1 PLeTs

As already discussed in Chapter 2, the cost and effort of the software testing activities are related to many factors, such as, the definition of the coverage criteria used to generate the test cases that will be executed during the testing process and the tool support. As it is one of the most time consuming and expensive phases of software development, it is relevant to automate the testing activities. MBT technique and their supporting tools can be used to mitigate this problem by automating the process of generating testing data, *e.g.*, test cases and scripts. However, although we identified several works of MBT tools (see Chapter 3), and most of these tools have similar features (*e.g.*, coverage criteria and modelling notations - see Section 3.6), the effort to design and develop a new MBT tool is high since, usually, it has to be constructed from scratch. In addition, most of the available MBT tools are limited to generate testing data in a proprietary format and to a specific technology or domain [HJK10] [HGB08].

Another issue that a testing team has to face is that in some situations it could be necessary to adopt complementary testing technology and consequently use a different testing tool to test an application. Moreover, the necessity to adopt a different testing tool could also be motivated by non-technical factors, such as, the tool's cost, the tool's available features, and market decisions. Furthermore, in these situations, it is usual that a large investment is made in a software license and to train the testing team. Besides that, a lot of pressure is put on the testing team to quickly learn the new technology and then explore all the tool's features. Furthermore, sometimes the testing team is already motivated to move from some testing approach to an MBT approach and its supporting tools, but when they realize that the already purchased tools and the related team knowledge are almost useless to apply MBT, they give up or postpone MBT adoption.

To overcome these issues, during the design and development of an MBT tool, the *Build Model*, *Generate Expected Inputs*, *Generate Expected Outputs* steps should be designed and developed bearing in mind that an external tool (*e.g.*, LoadRunner, Jabuti or even an in-house tool) could be used to execute the *Run Test* step. Thus, the testing team could reuse these testing tools to apply

MBT, reducing time and cost of the development of an MBT tool. Although the possibility of using an external tool to execute MBT is useful to the testing community, this approach does not systematize the reuse of software artifacts already developed to support each MBT step. Another point is that it is necessary to control and plan the variability between the artifacts.

Based on this, an approach to mitigate these limitations and systematize the artifacts reuse could be to design and develop an SPL that could be used to generate MBT tools to cover all steps of the MBT process [EFW01], *i.e.*, a tool in which it would be possible to describe the system model, that would generate test cases/scripts, that would execute test scripts and also compare the results. An even better situation would be if the test team could generate a testing tool for each different application domain, or different testing level and then execute it over the same application. Furthermore, it is desirable that the testing team reuse previously implemented artifacts (*e.g.*: models, software components, scripts).

Therefore, based on the context and motivation presented above, which in turn are based on information about MBT tools extracted from the mapping study depicted in the Chapter 3 and our expertise on the design and development of an MBT tool in collaboration with a TDL, we defined the basic requirements of our SPL of MBT tools. In the next section we introduce the MBT tools requirements, our approach to develop PLeTs and their variability identification.

4.1.1 Requirements

During the domain engineering, an SPL engineer uses the product roadmap and the requirements of desired systems to define and document the common and variable requirements of the SPL [LSR07]. Thus, our starting point in the development of our SPL was to define the PLeTs requirements and identify whether they are common or variable.

However, identifying the MBT tools' requirements is a challenging task, since to the best of our knowledge there are few updated contributions in the MBT literature about tools characteristics [DNSVT07] [SL10] and there are a significant number of MBT tools available. Because of this, we performed an SMS (see in Chapter 3) to map out the available MBT tools and their main features. Furthermore, we resorted to a TDL lab of an industrial partner to discuss the requirements and validate the identified tools' features.

Based on this, we defined the following basic requirements to design our SPL of MBT tools:

RQ1) *The MBT tools must support automatic generation of testing data*

Reducing effort and investment and at the same time improving software quality are always the goals of a testing team. Thus, the MBT tools generated from PLeTs must support automatic generation of test cases and/or test scripts based on the system model (see Chapter 2). These tools must accept a system model as an input, generate test cases/scripts (*Generate Expected Inputs*), execute test scripts and then compare results. After that, the PLeTs' product loads the generated

scripts and starts the test (*Run Tests*). Certainly, in some situations it could be desirable that a tool supports only some of these activities. For instance, in some situations a functional MBT tool could only generate a set of abstract test cases. It is important to note that the full automation of the testing process is still a dream [Ber07]. Thus, the use of the MBT tools generated from PLeTs require that some activities must be performed manually, such as creation of system models and results analyses.

RQ2) *Tools generated from PLeTs must support integration with other testing tools*

The generation of MBT tools must take advantage of the integration with other testing tools to run the testing scripts that were generated using MBT. Therefore, the last activities of the MBT process (*Generate Expected Outputs* and *Run Test*) should be designed to support an external tool, such as LoadRunner [Hew], Jabuti [VMWD05] or even an in-house tool. This requirement addresses the reduction of development time and cost in the development of MBT tools since the SPL engineer only has to design and develop components related to the testing script generation. Another motivation to include this requirement is that the testing team's knowledge about an already used testing tool is still a valuable skill since in some situations the use of an MBT tool would not be desirable. For instance, in some scenarios it is required that a specific feature must be implemented in an MBT tool to test an application. However, this kind of application or technology is not common in the company's portfolio. Thus, a test engineer could decide to save effort and investment necessary to implement this new feature and just use the legacy testing tool to test the application.

RQ3) *PLeTs artifacts must be designed and developed to support automatic generation of MBT tools*

Based on the fact that MBT tools generated from PLeTs could be adopted by a company as an alternative to reducing their testing effort, and that nowadays it is common for a company to have geographically distributed testing teams, changes to SPL artifacts must be easily documented and self-contained to facilitate the development and maintenance. Furthermore, usually testers evolve a tool (e.g., by adding an extra tool capability) it must not require any knowledge about other tool's artifacts inner structure nor require any change in the source code of the previous developed artifacts. This means that the feature implementation must extend a tool in just one place [KAK08]. Moreover, the SPL must be designed to require a minimal manual support to generate the MBT tools. Furthermore, distributed teams can access the same PLeTs repository and generate their own tools using these components developed by different teams. To alleviate the effort required by a testing team that wants to reuse components developed by others teams, such as, understand the components inner structure, we defined that the use of these components on the generation of a MBT tool from PLeTs must require minimal manual intervention.

RQ4) *PLeTs must not be bound to any proprietary SPL supporting environment or tool*

One particular concern regards the use of *off-the-shelf* tools, such as, FeatureIDE [TKB⁺12], to support a company development life-cycle, since sometimes these tools do not meet specific requirements of the companies aiming to adopt them. Furthermore, the adoption of an *off-the-shelf* tool would require a specific platform expertise that could limit the company potential, and also

lead to high training costs. Moreover, most of the available tools require a reasonable financial investment in a software licence and in some cases an expensive investment in SPL consulting. Thus, the decision to adopt a tool to support the automatic generation of a product from PLeTs must not cause a meaningful extra investment and should not require complex expertise.

It is important to note that the requirements *RQ3* and *RQ4* do not directly address the definition of features to MBT tools, but they address how we design our SPL and our choice of an SPL variability mechanism. In the following section we present and discuss our design decisions, which are based on the requirements presented in this section. Furthermore, requirements *RQ1* and *RQ2* are high level requirements, which could be decomposed into more specific requirements. To facilitate the understanding and readability of this thesis, we introduce and discuss the specific requirements along the text without mentioning it as a specific *RQ*.

4.1.2 Design Decisions, Process and Variability control

The basic requirement *RQ1* defines that MBT tools generated from PLeTs must support automatic generation of cases and scripts from system models. Although El-far [EFW01] has initially presented and discussed MBT steps, we used the SMS results in order to identify the tool's common and variable characteristics. Based on that, we noticed that MBT tools share the following set of basic characteristics: extract test information from system models, generate the test cases/scripts and run the test. Since the requirement *RQ2* defines that MBT tools must take advantage of the integration with others testing tools, it leads us to split the generation of test cases and scripts into two features. The first generates abstract test cases, which are not related to a technology, and then the latter instantiates them to executable test scripts of a testing tool. Thus, an MBT tool generated from PLeTs could have the following basic features: extract test information from system models, generate abstract test cases, generate scripts and run the test.

Figure 4.1 shows the adopted MBT process, which is based on the MBT process proposed by El-far [EFW01]. Thus, an MBT tool generated from PLeTs must parse testing information from a file¹ (Figure 4.1 (a)) and based on this information generate the abstract test cases (Figure 4.1 (b)). After that, the MBT tool can instantiate the abstract test to a tool format (Figure 4.1 (c)) and then run the test (Figure 4.1 (d)).

After the identification of the basic features and the definition of the MBT tools' process, we focus on the analysis of the MBT tools, recovered from the SMS, to identify which set of features are related to each basic feature. For instance, we identify that some MBT tools use common test case generation techniques to generate the test cases, e.g., random generation [Kor90] and graph search algorithms [YLS12], or could deal with common modelling notations, e.g., UML and Labelled State Transition System (LSTS). It is important to highlight that we did not focus on identifying fine-grained features, such as, common classes or methods necessary to parse a file, but rather

¹Most software modelling tools export/store models information using a structure file format, e.g., XML.

identifying coarse-grained variability, such as a parser to extract the test information from some common modelling notations or a common test case generator implementation. This decision partially met requirement *RQ3*, which defines that each feature implementation must be self-contained and extend a tool in just one place.

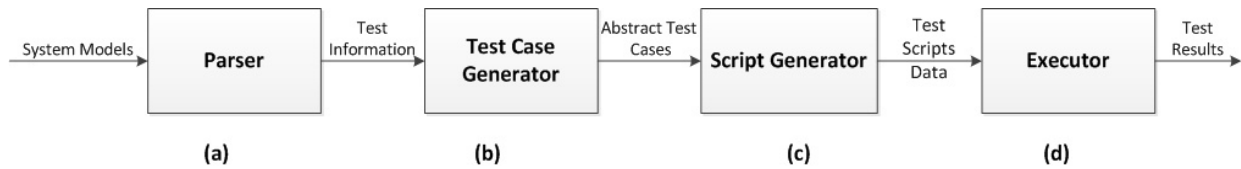
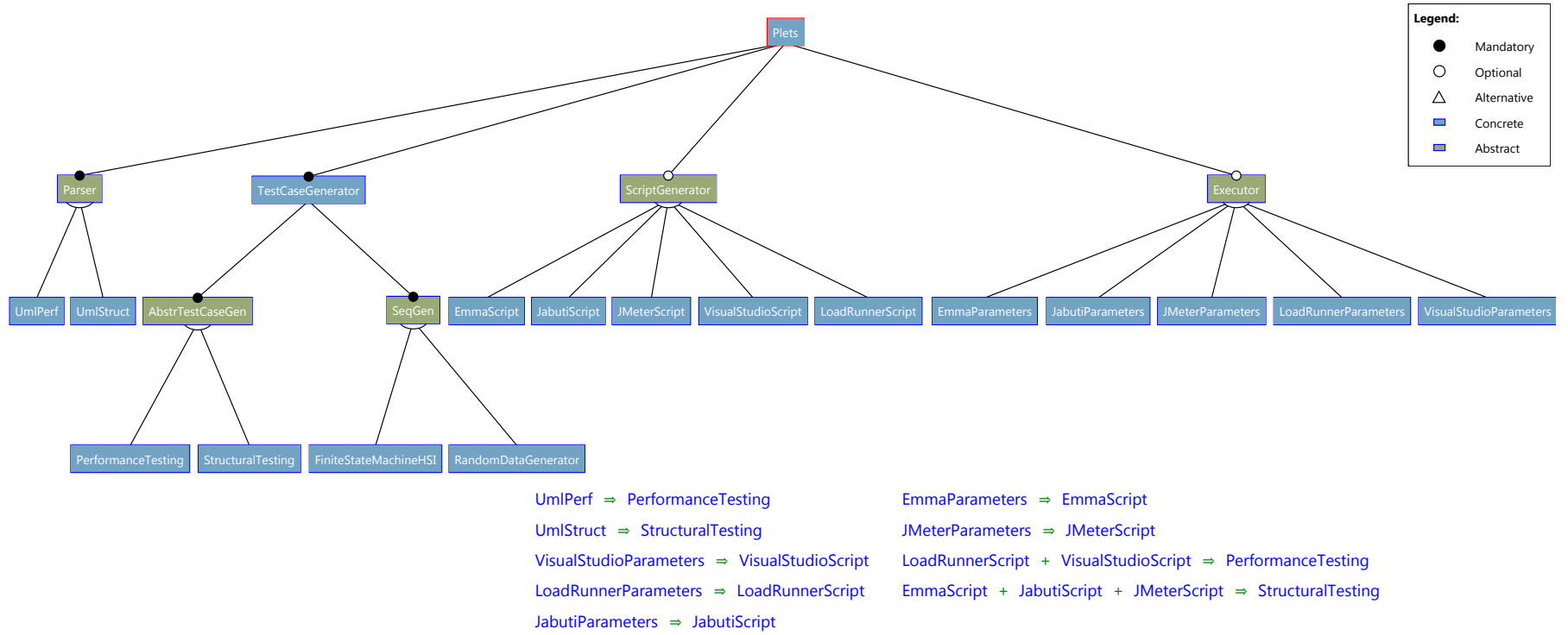


Figure 4.1: MBT Tools Process

Since we have identified the tools' basic and specific features, we use an extend FODA feature model to represent the relationship between the child features and their parent feature and the dependency between features (cross-tree constrains). Figure 4.2 presents the PLeTs feature model, which is composed of four basic features: *Parser*, *TestCaseGenerator*, *ScriptGenerator*, and *Executor* (notice that each basic feature represents a step in the MBT tool process - see Figure 4.1). It is important to mention that, even though our current feature model has a well-defined number of features, this model can, and will be expanded to include new features as new features are identified and their inclusion is demanded. A description of the PLeTs features is as follows:

- *Parser* is analogous to the *Build Model* step in the MBT main activities (see Figure 2.3). It is a mandatory feature with two child features, *UmlPerf* and *UmlStruct*. The former is related to extract performance testing information from UML diagrams and the latter is related to extract structural testing information from UML diagrams. UML models were chosen as a feature since it is a well-established notations and has been used by several researchers [AB06] [CNM07] [FIMR10] [FL09] [GHV07] [HGB08] [LMG10] [Oli08] [SHH07] [SWK09]. Furthermore, in several companies, UML notation has been used by their development and testing teams.
- *TestCaseGenerator* represents the *Generate Expected Inputs* step in the MBT process. It is a mandatory feature, since every MBT tool must support the generation of test cases and has two child features: *AbstrTestCaseGen* and *SeqGen*. The former has two features: *PerformanceTesting* and *StructuralTesting*. The latter has two child features: *FiniteStateMachineHSI* and *RandomTestData*. The *AbstrTestCaseGen* feature encompasses the generation of the abstract test cases: for performance testing if the SPL engineer/testing engineer selected the *PerformanceTesting* feature, or structural test if the *StructuralTesting* feature is selected. The *SeqGen* feature encompasses the generation of the test sequences. The test sequences generation can be based on the HSI sequence generator method when the *FiniteStateMachineHSI* feature is selected or based on random test data generation when the *RandomTestData* feature is selected. It is important to note that the definition of a test sequence method and the generation of abstract test case was based on our investigation of MBT tools. Furthermore, the decision to include the feature *FiniteStateMachineHSI*, among others sequence generator

Figure 4.2: PLEts Feature Model



methods, was also motivated by a collaboration project with a research group from ICMC-USP² (Instituto de Ciências e Matemáticas e de Computação da Universidade de São Paulo), which has expertise in the development of test sequences methods. Thus, the *FiniteStateMachineHSI* feature was developed by a master student in the context of this collaboration project.

- *ScriptGenerator* is an optional feature, since some MBT tools can just generate abstract test cases, that encompass the instantiation of the abstract test cases into executable scripts for a testing tool that will be used to run the test over the System Under Test (SUT). The *ScriptGenerator* feature has five child features: *VisualStudioScript*, *LoadRunnerScript*, *JMeterScript*, *JabutiScript* and *EmmaScript*. These features encompass the instantiation of abstract test cases to concrete test scripts to the following testing tools: Visual Studio [PG06], LoadRunner [Hew] and Jmeter [Apa] for performance testing and Emma [Rou] and Jabuti [VMWD05] for structural testing. Based on the assumption that a testing team should use at least one testing tool for each testing level, we expected that new scripts generator features will be continuously added in a reactive way.
- *Executor* is an optional feature that embraces the automatic execution of the external testing tool and also the automatic execution of the tests. This feature also has five features: *VisualStudioSParameters*, *LoadRunerParameters*, *JMeterParameters*, *JabutiParamters* and *EmmaParameters*. These features encompass the automatic execution of the testing tool and running the test scripts to the respective external testing tools: Visual Studio, LoadRunner and Jmeter for performance testing and Emma and Jabuti for structural testing.

With regard to Figure 4.2, it is important to notice that we identified several dependencies among features, which are denoted using Propositional Logic. For instance, if the feature *LoadRunnerParamters* is selected to compose an MBT tool, feature *LoadRunnerScript* must be selected, since the generated tool is not able to run tests without test scripts. In turn, the selection of *LoadRunnerScript* implies the selection of feature *PerformanceTesting*, which encompasses the generation of abstract test cases for performance testing, which in turn requires the *SeqGen* feature. Furthermore, since our SPL adoption approach is proactive and reactive, the PLeTs feature model will evolve and new features will be included. For instance, if the SPL owner/manager wants to support the generation of scripts to another external testing tool and also to support the automatic execution of the testing, it will have to include two new features, one to the parent *ScriptGenerator* and another to *Executor*. Moreover, it might be necessary to define their dependencies to other features.

Requirement *RQ3* defines that PLeTs artifacts must be designed and developed to support automatic generation of MBT tools. Thus, it led us to define the following design decisions:

- The PLeTs FM must support the definition of abstract and concrete features (green features on Figure 4.2 are abstract and blue are concrete) [TKES11]. Abstract features on PLeTs are used

²This collaboration project is funded by PROCAD/CAPES - Programa Nacional de Cooperação Acadêmica

only for readability purposes and concrete feature must be mapped to exactly one component, as it do not require maintaining explicit traceability links, and reduces mistakes, development effort, and costs. In case of feature interaction, the PLeTs engineer can rely on *#ifdef* based notations. Binding is done by matching feature and component names, in a 1:1 mapping

- To use component replacement as a variability mechanism. To choose this variability mechanism will require that the PLeTs engineer implements several versions of a component, where each version follows a different component specification, *e.g.* several parser implementations, where each implementation parses a different set of information. As discussed in Chapter 2, the component replacement mechanism allows the substitution, at compiling time, of a standard component by each implementation of the components (since it implements the same interface). Thus, a test engineer can derive an MBT tool just by selecting the desired tools' features, which are directly mapped to components.

Requirement *RQ4* defines that PLeTs artifacts must not be bound to any proprietary SPL supporting environment or tool. This requirement led us to design and develop an environment to support the design, development and product generation of an SPL using a component replacement mechanism. Furthermore, the adoption of our environment does not require complex tool expertise or extra costs. Section 4.2 presents more details about the environment specific requirements, design decisions and their implementation. Furthermore, it presents how the environment can be applied to support the automatic generation of MBT tools from PLeTs.

4.1.3 Architecture and Implementation

As presented in Section 4.1.2, we defined the use of a replacement mechanism to develop each concrete feature of the PLeTs feature model. Thus, an MBT tool derived from PLeTs is assembled by selecting a set of components and a common software base. We chose this approach to generate PLeTs products because it presents some advantages, such as, high-level of modularity and a simple 1:1 feature to code mapping.

To manage the dependencies among components and to represent the variability in the PLeTs architecture, we chose to apply the **S**tereotype-based **M**anagement of **V**ariability (SMarty) approach [OGM10]. SMarty is composed of an UML profile and a process for managing variabilities in an SPL. The SMarty profile contains a set of stereotypes and tagged values to denote the SPL variability at the architecture level. The SMarty process consists of a set of activities that guide the user to trace, identify, and control variabilities in an SPL. Our main motivations to choose the SMarty approach, among other approaches to manage variability using UML models, are the fact that it can be easily extended and it has a low learning curve [OGM10]. Figure 4.3 shows the PLeTs component model in accordance to SMarty, which in turn reflects the features depicted in the PLeTs feature model presented in Figure 4.2. Since we are using a component replacement mechanism, each provided interface represents a variation point and each variable component implementation

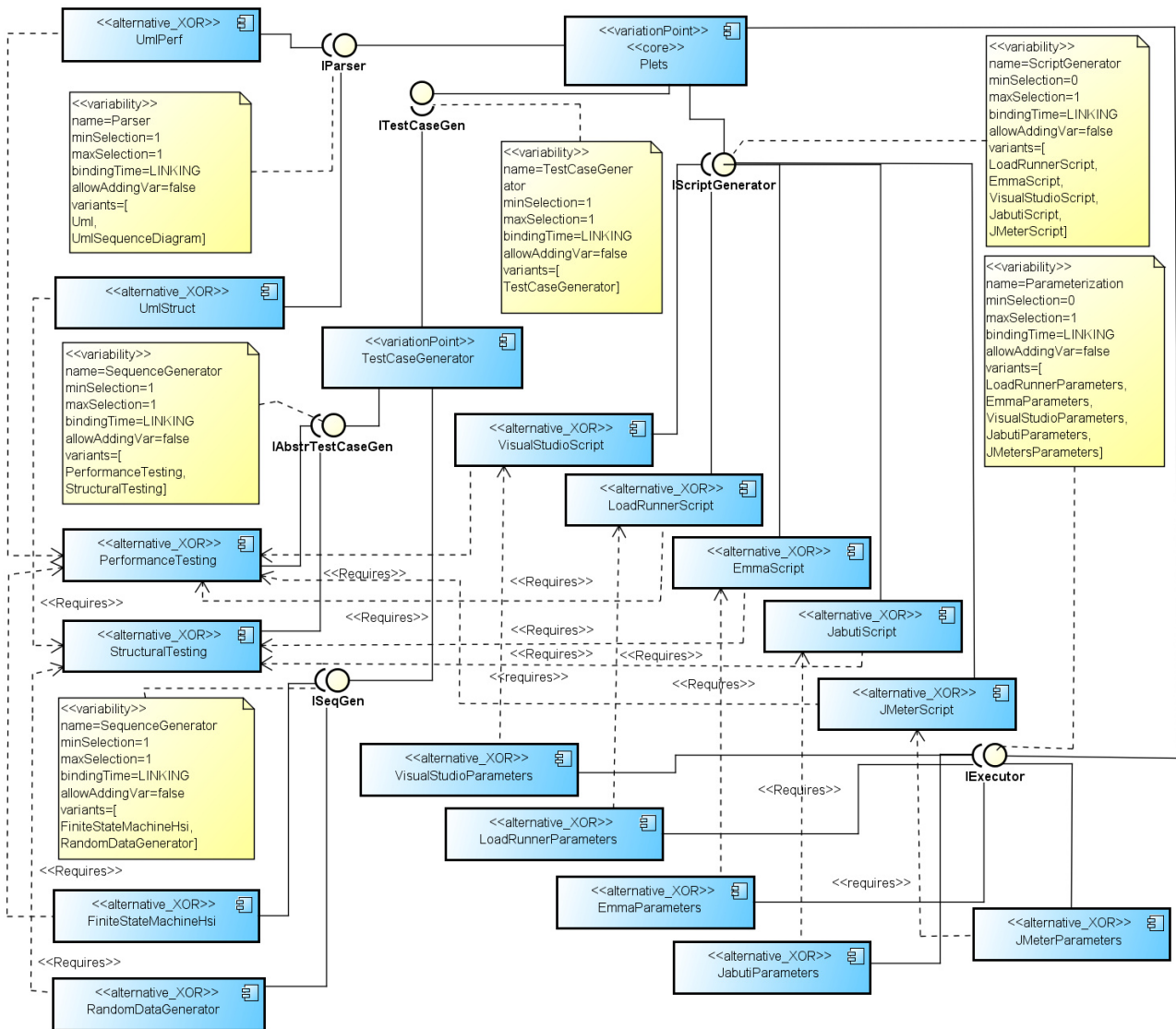


Figure 4.3: PLeTs UML Component Diagram with SMarty

represents a variant. The interfaces provided by the *PLeTs* components are as follow (see Figure 4.3):

- *IParser* is both a mandatory variation point that has two exclusive variant components, *UmlPerf* and *UmlStruct*. It is important to note that the associated variability indicates that the minimum number of variants is one ($minSelection = 1$) and the maximum is one ($maxSelection = 1$).
- *ITestCaseGen* is a mandatory variation point that has one mandatory component: *TestCaseGenerator*. This component provides two interfaces: *IAbstractTestGen* and *IseqGen*. The former interface can be realized by one of the following components: *PerformanceTesting* or *StructuralTesting*. The latter interface can be realized by one of the following components: *FiniteStateMachineHsi* or *RandomDataGenerator*. The minimum number of variants that can realize the interface is one ($minSelection = 1$) and the maximum is one ($maxSelection = 1$).

- *IScriptGenerator* is an optional variation point that can be realized by one of the following components: *VisualStudioScript*, *LoadRunnerScript*, *EmmaScript*, *JabutiScript* and *JmeterScript*. Thus, the minimum number of variants that can be realized by the interface is zero ($minSelection = 0$) and the maximum is 1 ($maxSelection = 1$).
- *IExecutor* is an optional variation point that can be realized by one of the following components: *VisualStudioScript*, *LoadRunnerScript*, *EmmaScript*, *JabutiScript* and *JmeterScript*. The minimum number of variants that can be realized by the interface is zero ($minSelection = 0$) and the maximum is 1 ($maxSelection = 1$).

In all components, apart from *PLeTs*, each associated variability indicates that one variant can be exclusively selected to resolve the variability. The SMarty approach allows to represent scenarios where the selection of a variant forces the selection of another variant, as a constraint among the variants. For instance, if one selects component *LoadRunnerParameters* to compose a *PLeTs* product, it requires the selection of component *LoadRunnerScript*. In turn, the selection of the latter component requires that the *PerformanceTesting* component must be selected. It is important to notice that the constraints presented in the variability component model will be used as input by our environment to resolve the dependencies among features to generate a product. For example, a valid configuration of a product derived from *PLeTs* could have the following components: *PLeTs*, *UmlParser*, *TestCaseGenerator*, *PerformanceTesting*, *FiniteStateMachineHsi*, *LoadRunnerScript*, *LoadRunnerParameters*.

4.2 PlugSPL - An environment to Support a Component based SPL

This section presents the requirements, design decisions and development of an environment to support an SPL using a component replacement approach. Some requirements described in this section are based on our experience deriving an SPL of testing tools in a research collaboration between a Technology Development Lab (TDL) of Dell Computers and PUCRS.

Basically the design and development of our environment provides two contributions: the identification of a set of requirements, which are specific to our research context, that highlight industrial needs that, in our understanding, are not fully addressed by any existing tool, either commercial or open-source. Thus, they elicit practical scenarios that tool vendors and/or implementers may consider supporting; and, we report our design decisions when implementing these requirements in an in-house solution.

4.2.1 Requirements

This section presents the requirements we identified with regards to enabling the configuration (component selection) and derivation of products (gluing selected components).

RQT1) *The adopted tool must not be bound to any specific IDE (already defined as a PLeTs requirement).*

Based on the fact that in most companies the development and testing teams, usually, take advantage of a significant number of different solutions, e.g., IDEs, programming language, version control system, they use solutions from different distributors/vendors (e.g., Eclipse for Java and Visual Studio for C#). The adopted tool must not impose any IDE, as that would require a specific platform expertise that could limit the company's potential, and also lead to training costs unrelated to the adoption of an SPL-based solution alone.

RQT2) *The adopted tool must support a graphical-based notation for designing FMs.*

Following from the fact that features are an effective communication medium across different stakeholders and feature-model-based notations are a widespread mechanism to capture variability, to support for variability modelling we use feature models (FMs). For example, in a company the FMs can be used as the main communication mechanism with stakeholders outside testing teams, and possibly non-technical staff (e.g., project managers). In addition, we consider that FMs provide testers with a quick visualization of the existing features in the current snapshot of the testing infrastructure and how each feature relates to one another. In some cases, stakeholders could state a preference towards graphical notations, as they consider textual ones to be linear, i.e., one element is only known when another one ends, and as such, hinders an immediate grasp of the underlying structure.

RQT3) *Structural architectural models must be kept in synchronization with the FM and code base (and vice-versa).*

Since testing teams could be geographically distributed, changes to the underlying test infrastructure must be documented at all times to facilitate communication and future maintenance. Furthermore, both models should be kept in sync with each other, to facilitate the communication between the teams.

RQT4) *The adopted tool must support a graphical-based notation for designing structural architectural models.*

In addition to keeping FMs, changes should also be documented in terms of structural architectural models that closely resemble the coding artifacts in the testing infrastructure (e.g., UML component diagrams). These models capture what feature models alone would otherwise miss (e.g., a class method). In addition, both models should be kept in sync with each other and the code base (full round-tripping). As before, models should be presented/edited graphically.

RQT5) *The tool must be extensible to support different structural architectural modelling and FM notations.*

Currently, structural architectural modelling is mostly based on UML diagrams. However, a company that is adopting our SPL would benefit if the SPL tool was flexible enough to support other notations in the future (e.g., custom DSLs). Likewise, the tool should also allow different notations for FMs to be supported, as extensions will be added as needed.

RQT6) *FMs should be derivable from the structural architectural models.*

Since SPL is something new for most companies, testers are familiar with standard UML structural models, while less so with feature models. To prevent initial mistakes and to minimize the effort in extending the testing infrastructure, the tool must be able to recover an FM from the defined structural architecture, which in turn can be tuned accordingly.

RQT7) *Structural architectural models should be derivable from FMs.*

As time progresses and FMs become more common among stakeholders, testers can start extending the testing infrastructure by first changing the FM, and then deriving the corresponding structural architectural model, which can be tuned accordingly (round-trip is already requested by RQT4).

RQT8) *For each product of the testing infrastructure, it should be possible to derive its corresponding structural architectural model.*

Products result from the selection and gluing of components. For each product it should be possible to generate its structural architectural model, which results from selecting specific elements from the architectural model of the whole product line.

RQT9) *Traceability links among models and implementation assets should require minimal human intervention/effort.*

Traceability is normally an important concern to any company that adopts an SPL, as FMs and structural architectural models need to be mapped to implementation assets, and vice-versa. To prevent a high burden on manually keeping such links, traceability links among models and implementation must require minimal human intervention.

RQT10) *When extending the existing infrastructure with new features, the initial code skeleton should be automatically generated.*

Currently, when testers evolve the testing infrastructure (e.g., by adding an extra capability) they often duplicate code (e.g., when implementing the interface of a core capability of the testing infrastructure), by either copying and adapting an existing implementation or writing a new one from scratch. The automatic generation of the initial code skeleton reduces the development effort and the probability of mistakes, e.g., to implement a wrong interface.

RQT11) *The adopted tool must allow creation of new glue code generators, that should be pluggable into the system without intrusive changes.*

The adopted tool must allow to hook code generators to produce glue code for specific programming languages, since the support of only one programming language could limit the company potential and led to extra costs.

As no tool currently meets all presented requirements, the option was to create a component-based environment for our MBT product line.

4.2.2 Design Decisions

In this section, we report our design decisions based on the set of requirements discussed in the previous section. As mentioned before, to the best of our knowledge, such requirements are not fully addressed by any existing tool, either commercial or open-source, hence the option for developing an in-house solution. For each design decision, we highlight the associated requirements.

DDT1) We define an extensible environment for using different feature models. The tool includes a feature model editor that currently supports a modified FODA notation together with abstract features [TKES11]. (RQT 1, 2, 5)

DDT2) We define an extensible environment for using different structural models. Currently, we use UML component models as structural models, as they are one of the most used modelling notation [DGMR06]. Therefore, this adoption mitigate a need for extensive training, as it is the standard notation taught in universities in our country. The editor supports SMarty notation [OGM10] to capture underlying variability in the model. (RQT 3, 5)

DDT3) Every concrete feature is mapped to exactly one component, as it dispenses maintaining explicit traceability links, and thus reduces mistakes, development effort, and costs. In case of feature interaction, we rely on *#ifdef* based notations. Binding is done by matching feature and component names, in an 1:1 mapping. (RQT 4, 9)

DDT4) We synchronize the feature and structural models to maintain consistency after performing changes in each of them. Consistency in this case, is eased, by relying in the 1:1 mapping, which makes transformation among models straightforward. (RQT 4, 6, 7, 9)

DDT5) Every component corresponds to a single compilation unit. Again, the binding is done by matching names. Based on the product configuration and relying on the 1:1 mapping, we create a program that instantiates the components of the given configuration. (RQT 8)

DDT6) Each component corresponds to a project, (e.g., Visual Studio). The environment supports the generation of an initial code skeleton (classes realizing the designed components), to which developers must complete, .e., to provide implementation to current empty methods. (RQT 10)

DDT7) We define an extensible environment for using different target languages. Currently, we use C#. (RQT 11)

4.2.3 PlugSPL Environment

In this section we describe how we developed the PlugSPL environment, based on the requirements and design decisions presented in Section and . PlugSPL is a modular environment written in C#, in which a test manager can design an SPL, configure and develop its components, define a valid product configuration and generate a product. Figure 4.4 shows the environment

modules and their main activities, *i.e.*, SPL Design, Component Management, Product Configuration and Product Generation.

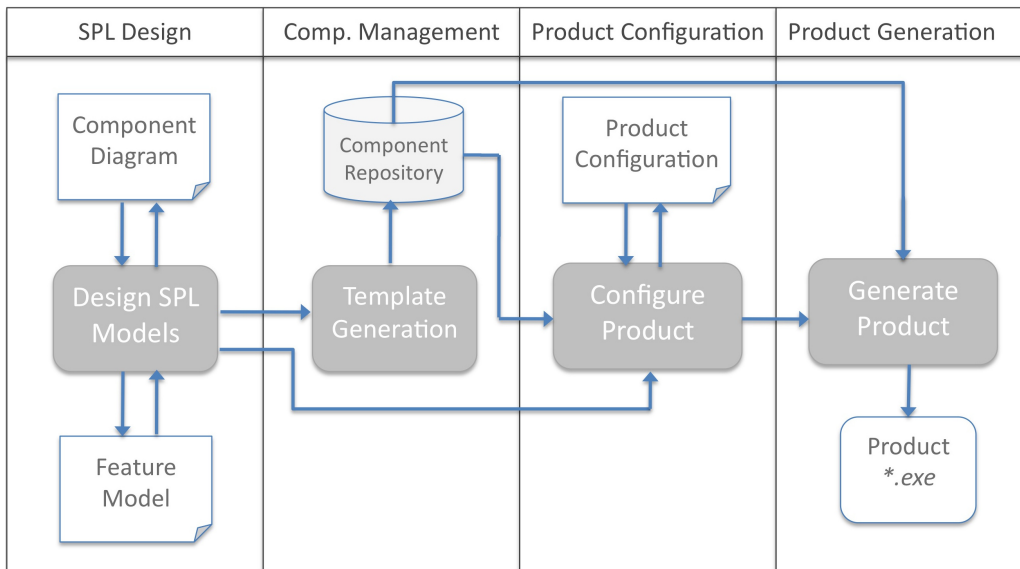


Figure 4.4: PlugSPL Modules

Product Line Design Module

The SPL Design module is the starting point when using PlugSPL and aims to support the SPL design using a graphical FM notation or an UML Component Diagram (CD) annotated following the SMarty variability profile [OGM10]. This flexibility allows a test engineer to start modelling the SPL using a graphical FM notation or a component diagram with annotations using the integrated UML component modelling environment.

The Design module also supports the automatic mapping among features in the feature model and components in the component diagram. Thus, the domain engineer might design an FM and then automatically generate the component diagram or model a component diagram and then generate the FM (full round-trip). As discussed in Section 4.2.3, the mapping from FM to component diagrams is 1:1, based on the features and components names. Furthermore, both the FM and component diagram may contain cross-tree constraints. Therefore, the environment provides a constraint editor to create or edit constraints for FM and component diagrams. The created or edited constraints are also automatically kept in sync with each other.

To support potential changes in the requirements, this module supports extending the system without intrusive changes. The design module can be extended using components to support different formats, such as SPLOT [MBC09], any other FM modeling notation, such as cardinality-based FM [CHE05], or UML diagrams, *e.g.*, class diagram.

The PlugSPL standard FM editor also provides support to model abstract features [TKES11], which are features used only for readability purposes and are not mapped to implementation artifacts, *e.g.*, software components (see Section 4.5). Support to define abstract and concrete features was

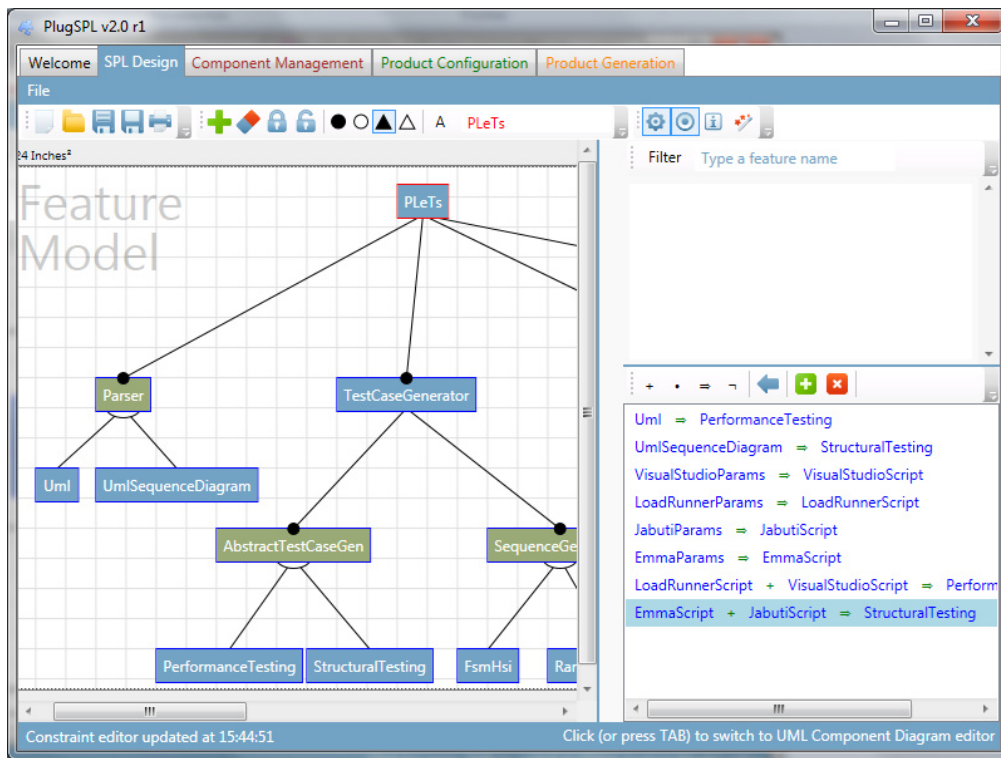


Figure 4.5: PlugSPL FM Editor

implemented to facilitate the mapping from features to UML diagram components and then to implementation artifacts. Concrete features are automatically mapped to a software component and their information is used by other PlugSPL modules to support the component management and product configuration.

Component Management Module

This module aims to manage the SPL software components, generate code templates, and validate their integration. Thus, based on the features or components specified during SPL analyses, the Component Management module generates a set of C# templates (e.g., classes and interfaces) for each concrete feature or component element (see Section 4.6). Based on the generated templates, the test engineer can define different testers/developers to code each component. Then, the tester can import each component template into an IDE and code the component. After that, the component source code (a Visual Studio project) must be added into the Component Management Module repository. Furthermore, when a new component is inserted in the PlugSPL repository, its source code is automatically checked to verify whether there are integration problems or not.

The Component Management module was developed to support C# template generation and validation. However, this module can also be extended to support template generation and integration in different languages, e.g., to support Java or C++. Thus, to use PlugSPL for supporting the life-cycle of a Java-based SPL, it is necessary to develop a Java template generator component and then replace the standard C# generator component.

Product Configuration Module

In this module, the test engineer must select what features must be present in the target product. This module uses, as input, the feature or component models created in the SPL Design module and the components (C# projects) stored in the components repository. Based on these inputs, the Product Configuration module automatically maps the features or components to code artifacts and then generates, in execution time, a tree view in which the test engineer can select the desired features. Furthermore, during feature selection, the tool automatically manages the dependencies among features according to constraints defined in the SPL design.

Once a product is configured, the configuration is saved in the project workspace, and test engineers proceed to generate target products.

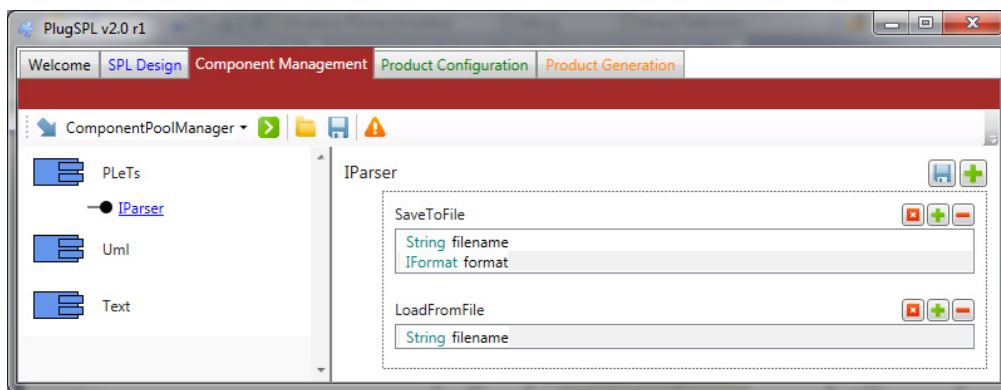


Figure 4.6: Component Management Module

Product Generation Module

The product generation module takes an existing product configuration and selects the associated components in the project workspace. Based on the product configuration, the module automatically copies the source code of the selected components from the component workplace to a temporary directory. Then, the temporary tags in the source code are replaced by the selected components data. We use temporary tags in our approach motivated by the fact that when the developers/testers are developing the SPL components, they do not know which component will be chosen by the SPL/test engineer to resolve the variability of an interface. Thus, we use temporary tags that must be defined in each interface (only in those that are a variation point) and that will be replaced during the product generation by the component name chosen to resolve the variation point variability.

Figure 4.7 presents a snapshot of a class developed following our approach. Line 5 shows how the *DummyIParser* temporary tag is applied to represent some type of parser that must be instantiated. In contrast, Figure 4.8 presents a snapshot of the same class during the product generation, in which the temporary tag is replaced by the chosen component name, e.g. the *UML* parser. We have observed that the use of temporary tags to support the automatic product generation

avoids human errors during component development and also reduces time and effort to generate SPL products.

The Component Management and Product Generation module was developed to support only C# templates. However, in the event of changes to the requirements, the module also supports extensions in the tag replacement and product compilation. Therefore, if some testing team want to use PlugSPL to support an SPL with components developed in Java, for instance, it is also necessary to develop a Java tag replacement and compilation component.

```

1 public class PLeTs{
2     string fileName, newFileName;
3     //omitted code
4     Console.WriteLine("Initializing <Parser> component...");
5     IParser parser = new DummyIParser();
6     parser.LoadDocument(fileName);
7     parser.ConvertStructures();
8     parser.SaveDocument(newFileName);
9     //omitted code
10 }

```

Figure 4.7: Code Before the Replacement

```

1 public class PLeTs{
2     string fileName, newFileName;
3     //omitted code
4     Console.WriteLine("Initializing <Parser> component...");
5     IParser parser = new Uml();
6     parser.LoadDocument(fileName);
7     parser.ConvertStructures();
8     parser.SaveDocument(newFileName);
9     //omitted code
10 }

```

Figure 4.8: Code After the Replacement

4.3 Final Considerations

In this chapter we presented the requirements, design decisions and the development of a Software Product Line of Model-based Testing tools: PLeTs. The main benefits and issues identified while developing our SPL are:

- The MBT tools derived from PLeTs can manage the whole MBT process. The derived tools should be able to accept some kind of SUT models as input, and based on that generate an output. The output could be a test case suite or a script to a specific testing tool. However, we designed PLeTs bearing in mind that in most cases a company that will adopt our MBT tool/PLeTs to support their testing process could already have a defined testing process and therefore some kind of testing tool. Furthermore, to design and develop an MBT tool from scratch is a time-consuming and costly activity. Because of this, we have designed/developed PLeTs to support the automatic creation and execution of the scripts to academic or commercial testing tools. The only functionality required is that these testing tools import scripts or use some kind of template files. It is important to highlight that up to this moment we have already developed components to generate scripts and execute them on several testing tools. Furthermore, our collaboration with the TDL gives us valuable feedback about the use of our SPL in an industrial setting. Thus, this feedback is used to update the MBT tools' requirements and then adjust the tools' features. Although we have had success in our collaboration, we are aware that other companies may have different tool requirements. However, we really believe that our SPL is flexible enough to support changes to meet these different requirements.
- A feature can be easily added to the SPL to support a new MBT tool functionality. In order to allow the implementation of new features, as well as making their integration flexible, we

have used a component based variability mechanism. Based on this, a new component can be developed from scratch, as a standard C# project, and easily added to the PLeTs repository. Additionally, we can select a pre-existent component and if necessary modify it to support a different functionality, and then add it to the SPL as a new feature. However, when these features are added to the SPL, it is necessary to manage their variability. Figure 4.3 (Section 4.1.3) represents this situation, where some components have a dependency relationship with another component, denoted by `«requires»`. An undesired consequence of this is that the complexity of managing the dependency grows along with PLeTs. Furthermore, a manual selection of the features, which are mapped to components during the product generation, and resolving their constraints is an error prone activity and requires highly skilled SPL engineers. Thus, to mitigate this issue we defined a simple mapping to map features to components (1:1) and implemented features in a self-contained and code replacement way, to support automatic generation of the products. The down side of the use of this coarse-grained variability is that during the development of some components we cannot take full advantage of their common class and methods. For instance, components *UmlPerf* and *UmlStruct* extract testing information from an UML diagram, then they can share some common methods, such as, a method used to open an UML file. Therefore, these two components could be developed as a single component, reusing common methods. However, as currently we identify and manage variability at component level, we lose fine-grained reuse. A solution would be to use another variability mechanism, in combination with the actual mechanism, to support fine-grained variability, such as, preprocessor directives. Actually, `#ifdef` based notations already can be used as a variability mechanism inside the PLeTs components, but the PlugSPL environment must be changed to support the SPL design and the automatic product generation.

- Furthermore, we also discussed a set of requirements and our design decisions addressing the development of PlugSPL to support the SPL development life-cycle. We also presented how the PlugSPL environment can be used to support the design, product configuration and generation of component-based SPLs. It is important to highlight that this environment has been successfully applied to generate several Model-based Testing tools for a TDL of a global IT company. Thus, the tools have also been used in some case studies and student projects to support the generation of testing tools to support different testing techniques and approaches [Cos12] [Sil12]. Furthermore, the user's feedback indicates that the tool usage requires low investment in user's training and presents a low learning curve. However, we have to better investigate and understand the effort an learning curve when using our environment, *e.g.*, an empirical experiment to compare the effort an learning curve when using our environment and an *off-the-shelf* environment.

5. EXAMPLES OF USE

This chapter presents two examples of use of MBT tools generated from PLeTs. These tools were used to generate test data and scripts to test two different applications in the context of a collaboration project between a Technology Development Lab (TDL) of an IT company and our university¹. In the first example of use, we introduce an approach and two MBT tools, PLeTsPerfLR and PLeTsPerfVS, which are used to generate performance scripts from UML Use Case and Activity diagrams to test an application [SRZ⁺11] [CCO⁺12] [Cos12] [Sil12]. Later, we also present an approach and two MBT tools, PLeTsStructJabuti and PLeTsStructEmma, which are used to generate structural test cases from UML Sequence diagrams to test a web application [CORS12] [Cos12]. Finally, we present our final considerations and the lessons learned during the generation of these tools.

5.1 Generating Performance MBT Tools

In this section, we present two performance MBT tools generated from PLeTs using the PlugSPL environment. Each MBT tool generates performance scripts and scenarios from UML Use Case and Activity diagrams. These tools have some common features and use the same approach: receive annotated UML Use Case and Activity diagrams as input and automatically derive abstract test cases. After that, the abstract test cases can be instantiated to a script format (concrete test cases and scripts) and then the scripts can be executed using an existing load generator tool (*e.g.* LoadRunner) to test a web application. It is important to note that the main use of MBT has been directed to the test of functional aspects of software and only lately researchers have applied MBT to investigate techniques towards non-functional testing. Therefore, lately, an increasing number of works [BLG11] [RVZ10] [DTA⁺08] [SRZ⁺11] discuss how to apply MBT and UML models to automatically generate performance test cases and scripts. Most of them apply MBT in conjunction with an UML profile, such as, Modelling and Analysis of Real Time and Embedded Systems (MARTE) [BM] or Schedulability Performance and Time (SPT) [OMGa], which are extensions of UML for modelling performance requirements. The use of these profiles provide modelling patterns with clear semantics allowing automated generation of test cases and scripts.

Based on UML models, one can extract information for tests and then analyse the performance counters (*e.g.* throughput, transactions per second and response time). Such information is distributed throughout several UML diagrams, *e.g.*, Use Cases (UC), Activity (AD) and Sequence diagrams (SD). An approach that depicts how these diagrams are used to represent the information of performance testing is presented by [DTA⁺08], in which a model designed in SPT and composed by UML diagrams was tagged with specific properties such as probabilities on the incidence of use cases, response times for each activity and resource availability.

¹Part of these work was developed in the context of two master dissertation [Cos12] [Sil12]

5.1.1 An Approach for Performance Test Case and Scripts Generation

Usually, the first step adopted by industry when testing applications for non-functional properties is to choose a given workload generator among the various available tools. The decision on which tool to better test an application involves several factors such as familiarity (has been used before), price or target software platform. Furthermore, every workload generator has its own peculiarities, configurations and tweaks as well as common operations to be performed when mimicking user behavior within the SUT. Thus, test scenarios and scripts generated for some tools cannot be reused by another tool.

Figure 5.1 shows our approach, in which the main idea is to receive as an input the UML diagrams (Figure 5.1(a)) and generate abstract test cases and scenarios (Figure 5.1 (b)) suitable for the derivation of test scenarios and scripts to a workload generator (Figure 5.1 (c)) and then execute the test (Figure 5.1 (d)). The main difference between our approach and MBT abstract test suites is the fact that we focus on creating intermediate models to use for performance testing rather than functional testing. Another contribution is that we apply MBT to construct a performance abstract test scenario. Based on that, test scenarios and scripts can be generated for a wide range of workload generators. Thus, to generate an MBT tool, from PLeTs, that generates scripts to a workload generator, the SPL engineer can reuse the previously developed components (e.g., Parser and Test Case generator) and only design and develop the script generator and the executor components.

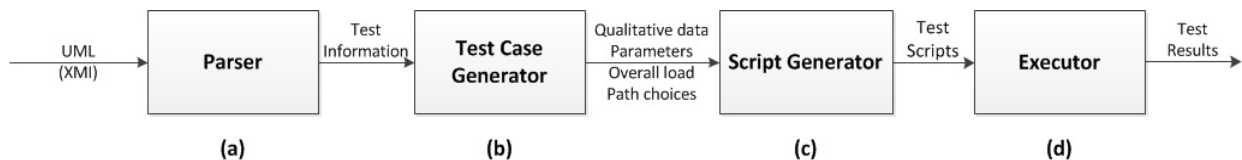


Figure 5.1: An Approach for Generating Performance Scripts and Scenarios

Abstract Intermediate Models

A performance test study usually entails a set of operations that must be properly and orderly executed to enhance the quality of the SUT execution (e.g., improve response time). For instance, once the SUT is stable in terms of implementation (in code-freeze mode to support better test performance measures), a performance analyst defines a performance test plan based on performance requirements, which is an important document that have been already discussed in earlier project phases. Then, the analyst must define which workload generator will be used for executing the test over the SUT. As stated earlier, the choice must take into account several aspects, such as proprietary tools versus open-sources ones, and tool availability in the performance testing infrastructure. Our initial focus is to generate a technology-independent testing description from our MBT approach. Thus, we reused the application models to generate an abstract intermediate model that is a description of the test scenarios. It contains information needed to instantiate the performance scripts, such as loads, ramp-up time, total execution time, which can be derived from

the model, while abstracting script details such as machine locations and API calls. An important issue when using MBT to generate performance test scenarios and scripts to define which UML diagram and which UML profile will be used to annotate the needed information.

Transformation Methodology

Before describing the overall methodology, we discuss the UML annotation provided by other parties. Our approach relies on the assumption that UML models are carefully annotated with high-quality information. We are trusting that all stakeholders are interested in the process of discovering the application's problems and bottlenecks and also that all involved personnel are committed to creating comprehensive test scenarios allowing further reliable analysis.

The process of generating test scenarios from annotated UML models must encompass a set of operations that must be performed prior to the test execution. Figure 5.1 shows our methodology, which involves basically four steps, where each step is implemented by a PLeTs component:

- UML Annotations Verification (Figure 5.1 (a))

We focus our attention on two broadly used diagrams present in UML: Use Cases (UC) and Activity Diagrams (AD). This step entails evaluating if the models were annotated with information that can be used in the generation of the abstract model in the next step. Thus, to annotate this information on the UML models, we defined some UML tags based on the UML profile to Schedulability Performance and Time (SPT) [OMG_a]: a) **«TDtime»**: limits the performance test duration(s) for the scenario(s); b) **«TDpopulation»**: maps the number of virtual users that populates the system; c) **«TDhost»**: describes the name of the host to connect to; d) **«TDaction»**: specifies an action that must be taken, e.g., the execution of a script; e) **«TDparameters»**: represents two information: name and value, that must be filled out to proceed, e.g., a login screen, or a search form; f) **«TDprob»**: indicates the probability to decide the next activity to be executed. It is used in the decision element model within the ADs or annotated in the association element model between actor and use case within the use cases diagram; g) **«TDthinkTime»**: defines the amount of time units each virtual user waits before taking another action. This tag is used by a large set of workload generators to mimic users behavior.

After this step, the UML models should satisfy the following conditions: a) every UC has at least one or several ADs; b) the AD is well-formed, *i.e.*, contains an initial and an end state; c) test plan with information regarding the test itself, e.g., the type of test, the number of virtual users; d) every action in the AD is annotated with context information, *i.e.*, user form and query string parameters and user think time information.

- Abstract Test Case and Scenarios Generation (Figure 5.1 (b))

Our abstract intermediate model is designed to be extensible and modular, using hierarchical textual structures to represent activities readily available in UML diagrams or other similar

representations of business processes, *e.g.*, Business Process Model Notation (BPMN) [DDO08]. In fact, it is straightforward to take any high-level business process representation and directly turn it into a test scenario; however, the test must be instantiated having a specific workload generator in mind.

In this step are used two abstract models: abstract test scenarios and abstract test cases. Each abstract model is a hierarchical structure that is used to map the AD to a textual abstract representation retaining the order of events and the parallel activities that must take place. This structure uses a sequential numbering convention with appended activities, *e.g.*, 1, 1.1, 2.1 and so forth. Each activity is mapped to a number, here defined as an *activity number*. The numbered mapping is a visual aid to inspect sequential and parallel relations within the abstract intermediate model.

Note that for the abstract model to function according to our specification, it should contain only the fundamental information to instantiate different test scenarios. The abstract format suggested here is extensible and could be enhanced to contain more information regarding performance tests for a more complex test suite.

It is important to highlight that the *FiniteStateMachineHsi* component used to generate the performance abstract test cases and scenarios was developed in the context of a master dissertation [Sil12]. Furthermore, we are aware that there are several methods in the literature that can be applied to generate test sequences. However, it is not the scope of this thesis to discuss the methods to generate test sequences to apply performance testing. Thus, our main focus is to present how to develop MBT tools from a set of common and specific artifacts. For instance, if a testers want to use a different method to generate test sequences, they have to develop a new component, based on the guidelines defined during the PLeTs development², and include the component into the PLeTs' repository.

- Script and Scenario Generator (Figure 5.1(c))

This step is tool-oriented because it generates specific test scenarios that must be created from the abstract intermediate model. We explain how this is performed in the following sections. Our approach shifts the concern on the performance test execution itself to the description of an abstract model that captures the needed test information looking up only to high-level UML models.

- Executor (Figure 5.1(d))

The last step of our methodology consists of loading the test scripts and scenarios into a specific workload generator and running the SUT.

²More information about the PLeTs' models, source code and implementation guidelines can be found in [CePa]

5.1.2 Example of Use

This section describes an application scenario in which our approach is applied to generate abstract performance test cases and scenarios. For this purpose, we used the TPC-W benchmark [Men02] as an application example. The TPC-W is a transactional web service benchmark that implements a benchmark and an e-commerce application that is used by the benchmark (or by any other workload generator). Thus, to generate scripts and scenarios to test the performance of the TPC-W application, we derived two performance MBT tools from PLeTs: PLeTsPerfLR and PLeTsPerfVS. Figure 5.2 and 5.3 present the tools architecture, which depicted the common components shared by these tools, (*i.e.*, UmlPerf, PerformanceTesting, TestCaseGenerator and FiniteStateMachineHsi). Since the tools share a set of common components and each step of our methodology is implemented in a component, both tools accepted UML models as input (UMLPerf component in the Figure 5.2 and 5.3) and automatically generates the abstract scenarios (components FiniteStateMachine, Performancetesting and TestCaseGenerator - see Figure 5.2 and 5.3). The tools' specific components, which are load generator-oriented because they generate specific test scripts, are related to the script generation and test execution. Therefore, in order to generate the PLeTsPerfLR, we developed the LoadRunnerScript and the LoadRunnerParameters components, which are specific to the MBT tool that generates scripts and scenarios to the LoadRunner load generator (Figure 5.2). In order to derive the MBT tool that generates scripts to the Visual Studio, the PLeTsPerfVS, we had to develop two specific components: VSScripts and VSParameters (Figure 5.3). It is important to highlight that the common components were coded with 1,646 lines of code, the PLeTsPerfLR specific components have 258 lines of code and the PLeTsPerfVS specific components have 344 lines of code. These numbers illustrate that the use of our SPL to generate MBT tools can reduce the development effort of a MBT tools, *e.g.* the effort to develop the PLeTsPerfLR was decreased almost 80%. The following section presents how we applied PLeTsPerfLR and PLeTsPerfLR MBT tools to generate performance scenarios and scripts to test the TPC-W web application.

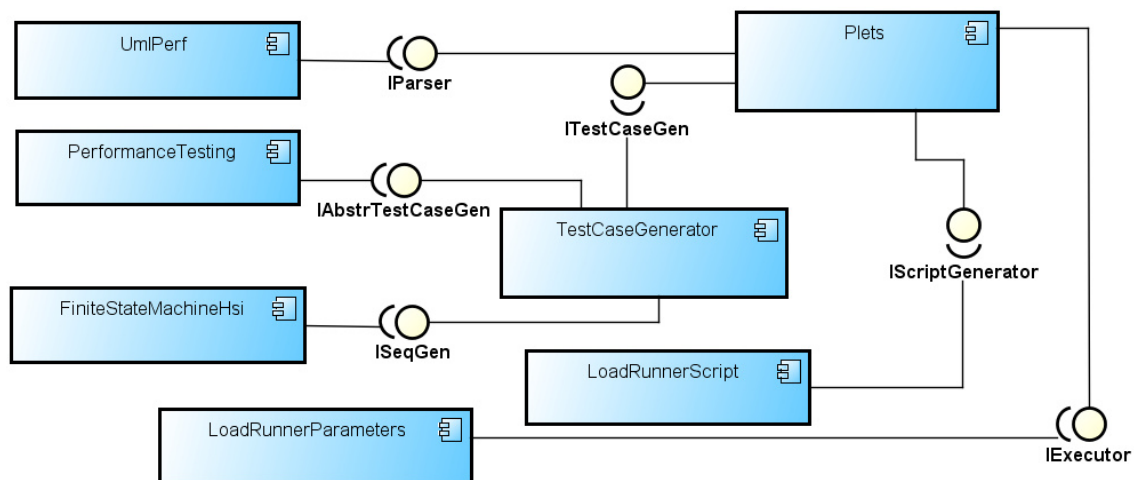


Figure 5.2: PLeTsPerfLR Tool Architecture

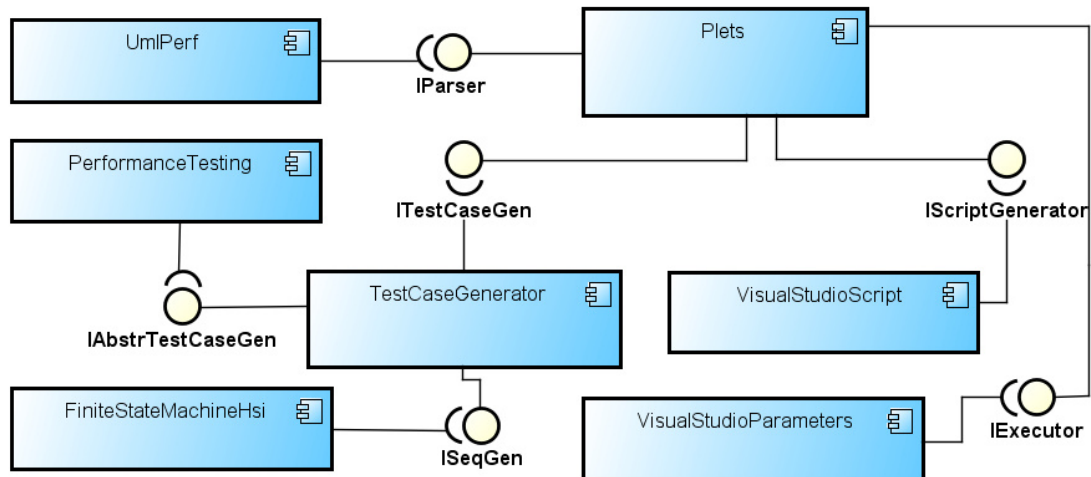


Figure 5.3: PLeTsPerfVS Tool Architecture

TPC-W UML Diagrams

In order to test the TPC-W application using our approach, first we have to create and annotate the application models. For this task, we have created several UML based tags to represent the performance information needed to generate the abstract intermediate models. As a starting point we have annotated three different use cases (shown in Figure 5.4): a) *Browser*: the users perform browsing interactions; b) *Registration*: the user fill out a registration form; and c) *Shop*: the users perform purchase interactions.

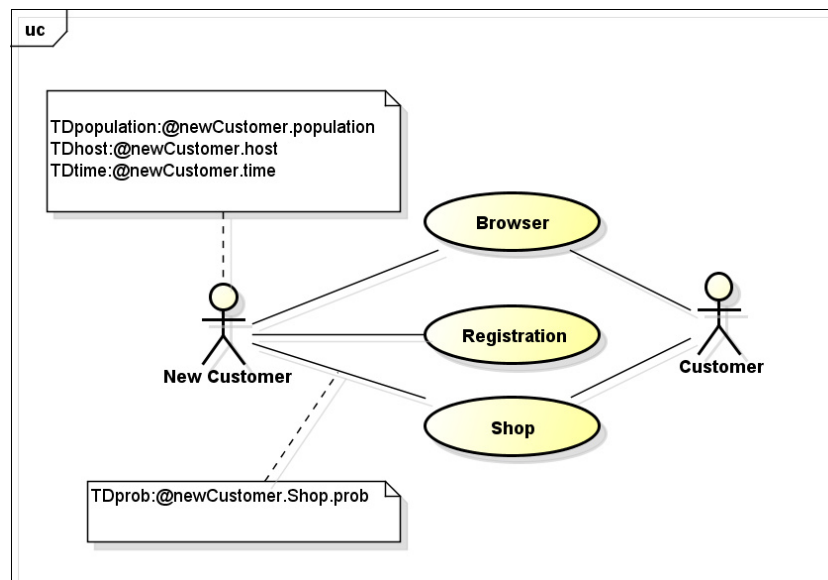


Figure 5.4: TPC-W Use Case Diagram

Each actor modelled in the UC diagram contains information about a specific test scenario. Thus, we defined two different test scenario interactions for the actors: *Customer* and *New Customer*. The test scenario for *New Customer* is a set of interactions that will be performed over the set of use case elements (*Browser*, *Registration* and *Shop*). It is important to notice that each UC is related

to a specific AD, e.g., the AD depicted in Figure 5.5 is related to the *Shop* use case depicted in Figure 5.4. Basically, each AD represents the sequence of activities performed by an actor over the application. As depicted in Figure 5.5, the first activity is *Home Page*, which is the TPC-W home page. After that, the user could perform the following actions: select a specific book category (*New Products*) or perform a search for a particular book (*Search Request* and *Search Results*). The activity *Search Request* shows a list of books to the user as a result. Hence, when selecting a book from this list (*Search Results*), several pieces of information about the selected book are displayed to the user (*Product Detail*). After that, the user must perform one of the following activities: finish his access to the application or continue on the website and make a purchase (*Shopping Cart*). If the user decided on the latter option, the next step is related to the registration of the customer (*Customer Registration*) in the application. Then, the user fills out some purchase information, such as, credit card information and delivery date (*Buy Request*). The last step checks whether all information is correct, then the purchase is confirmed (*Buy Confirm*) and the user finishes his access to the application. This diagram also has two decision elements in its flow that represent the probability of executing different paths in the system, e.g., the tag *@homePage.searchRequest.prob* between activities *Home Page* and *Search Request* in the Figure 5.5.

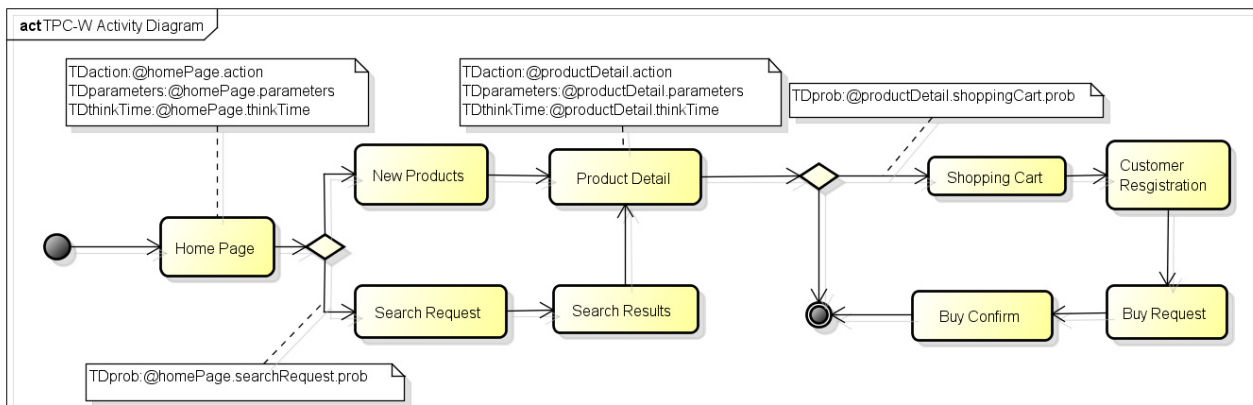


Figure 5.5: TPC-W Activity Diagram - Shop Use Case

As described in Section 5.1.1, UML diagrams can be initially annotated with seven tags in our approach. The UC shown in Figure 5.4 has four tags: *TDpopulation*, *TDhost*, *TDtime* and *TDprob*. Each one has its respective parameter, *@customer.population*, *@customer.host*, *@customer.time* and *@newCustomer.BrowsingMix.prob*. In relation to our AD we also have included four different tags and their respective parameters (see Figure 5.5).

Abstract Test Scenarios Generation

Once all UML diagrams (see Figures 5.4 and 5.5) were annotated with performance information, we applied our approach to generate abstract test scenarios for the TPC-W application. The creation of abstract test scenarios allows a later definition of a workload generator and its test script templates. In the context of this thesis, the main motivation to generate abstract test scenarios is to make easier the generation of new variants of MBT tools from PLeTs.

Basically, an abstract test scenario defines the amount of users that will interact with the SUT and also specifies the users' behavior. The abstract test case contains information regarding the necessary tasks to be performed by the user.

Figure 5.6 shows an example of an abstract test scenario generated from test information annotated in the actor *Customer*. The amount of abstract test scenarios generated based on an UC diagram is directly related to the amount of actors modelled in the UC model, e.g., for the TPC-W example there are two abstract test scenarios: *Browser* and *Shop*.

```

Abstract Test Scenario: Customer
## Test Setting
Virtual Users : <<TDpopulation: @customer.population>>
Host of SUT : <<TDhost: @customer.host>>
Test Duration : <<TDtime: @customer.time>>
## Test Cases Distribution:
<<TDprob: @customer.Shop.prob>>
1. Shop
1.1. Shop 1
1.2. Shop 2
1.3. Shop 3
1.4. Shop 4
<<TDprob: @customer.Browser.prob>>
2. Browser 1
2.1 Browser 2
2.2 Browser 3
2.3 Browser 4

```

Figure 5.6: Abstract Test Scenario of the Actor *Customer*

As mentioned in Section 5.1.1, the abstract test scenario has information related to the test context and definitions of the abstract test cases that must be instantiated, including the number of virtual users distribution for each abstract test case. Thus, our annotation approach is divided in two groups: 1) *Test Setting* – describes general characteristics applied to the test context as a whole (extracted from the UC); 2) *Test Cases Distribution* – represents specific information about the abstract test cases generated from each AD. In order to accomplish that, every test case represents an user path in the SUT. It is important to notice that the header of each abstract test case contains probability information (see Figure 5.6).

As show in Figure 5.7, the abstract test cases are built in a hierarchical approach, in which activities are listed and organized according to the dependency between the AD activities (represented by *activity number*). Figure 5.7 shows the abstract test case based on a test sequence derived from the AD (Figure 5.5). Furthermore, in the description of abstract test cases there is a variation of parameters added to each activity, which are composed by the name and the value of each tag, showing the flexibility of the configuration models. A parameter is the concatenation of two pieces of information: activity name and tag name, preceded by the delimiter @. Although, the tag *TDprob* has three pieces of information, the first information is the tag name preceded by the name of two UML elements. An example of this is presented in Figure 5.5 where the tag *TDprob @homePage.searchRequest.prob* is tagged in the UML association element between the UML decision

node and the target activity (*Search Request*). The same notation is applied in the UC diagrams, where the tag is applied between the UC *Shop* and the actor *New Customer*.

```
#Abstract Test Case: Shop 3
1. Home Page
  <<TDmethod : @HomePage.method>>
  <<TDaction : @HomePage.action>>
  <<TDparameters : @HomePage.parameters>>
  <<TDthinkTime : @HomePage.thinkTime>>
2. New Products...
3. Product Detail...
4. Shopping Cart...
5. Customer Registration...
6. Buy Request...
7. Buy Confirm
  <<TDmethod : @BuyConfirm.method>>
  <<TDaction : @BuyConfirm.action>>
  <<TDparameters : @BuyConfirm.parameters>>
  <<TDthinkTime : @BuyConfirm.thinkTime>>
```

Figure 5.7: Example of Abstract Test Case Generated from the *Shop* Use Case

It is important to notice that each tag parameter refers to a data file (Figure 5.8) that is automatically generated. Thus, when abstract test scenarios and scripts are instantiated to a concrete test scenario and scripts for a specific workload generator, the tag parameter is replaced by a value extracted from that data file.

```
@BuyConfirm.method:"POST"
@BuyConfirm.action:"http://localhost/tpcw/buy_confirm"
@BuyConfirm.parameters:[$ADDRESS.CITY, $ADDRESS.STATE]
@BuyConfirm.thinkTime:5
```

Figure 5.8: Example of Data File Containing Some Parameters

Test Scenarios and Scripts Generation

Based on the abstract test scenarios and test cases presented previously, the next step is to generate concrete instances (scripts and scenarios). This is a technology dependent step, since the concrete scenarios and test cases are strongly related to a specific workload generator that will directly execute test cases.

This is an important step to the automation of the performance testing because it allows the flexibility of choice to a workload generator or technology only when generating a new variant of an MBT tool. However, it is necessary for an advanced tool knowledge to create scripts and scenarios. Therefore, to demonstrate how our approach could be valuable to a performance testing team, we present how to generate test scenarios and scripts for two load generators that are used by the PLeTsPerfVS and PLeTsPerfLR MBT tools. Basically, the Visual Studio (VS) and the LoadRunner (LR) set up their test scenarios and scripts in two files. One of them is a scenario file that is used to

store the test configuration, workload profile distribution among test scripts and the performance counters that will be monitored by the tool. The other file is a script configuration file that is used to store the information about users' interaction with the application, including HTTP requests, as well as its parameters and transactions defined between requests. Figure 5.9 shows the VS scenarios file that was generated by the PLeTsPerfVS to test TPC-W. In this case, the *MaxUser* property corresponds to the parameter *@customer.population*. Another tag that has changed is the *RunConfiguration* with attribute *RunDuration* that is related to the tag *@customer.time*. The process to instrument a test scenario is based on a template.

```
<LoadTest ...>
  <Scenarios>
    <Scenario Name="Customer" ...>
      <ThinkProfile Value="0" Pattern="On" />
      <LoadProfile Pattern="Step" InitialUsers="0" MaxUsers="50" StepUsers="10"
        StepDuration="0" StepRampTime="60" />
      <BrowserMix>...</BrowserMix><TestMix>...</TestMix>
      <NetworkMix>...</NetworkMix>
    </Scenario>
  </Scenarios>
  <CounterSets>...</CounterSets>
  <RunConfigurations>
    <RunConfiguration RunDuration="7200" WarmupTime="300" TestIterations="100" ...>...</
      RunConfiguration>
  </RunConfigurations>
</LoadTest>
```

Figure 5.9: XML of Test Scenario Generated for the Visual Studio (*.LoadTest)

```
<WebTest Name="Shop 3" ...>
  <Items>
    <TransactionTimer>...</TransactionTimer>
    <TransactionTimer Name="Buy Confirm">
      <Items>
        <Request Url="http://localhost:8080/tpcw/
          TPCW_buy_confirm_servlet" ThinkTime="5"
          ...>
          <QueryStringParameters>
            <QueryStringParameter Value="{{ $ADDRESS.
              CITY }}" Name="CITY" ... />
            <QueryStringParameter Value="{{ $ADDRESS.
              STATE }}" Name="STATE" ... />...
          </QueryStringParameters>
        </Request>
      </Items>
    </TransactionTimer>
  </Items>
  <ValidationRules>...</ValidationRules>
</WebTest>
```

Figure 5.10: Test Script Generated for the Visual Studio

```
Action ()
{
  ...
  lr_think_time(5);
  web_submit_data("buy_confirm.jsp",
    "Action=http://localhost:8080/tpcw/
      TPCW_buy_confirm_servlet",
    "Method=POST",
    "RecContentType=text/html",
    "Referer=",
    "Mode=HTML",
    ITEMDATA,
    "Name=CITY", "Value={{ $ADDRESS.CITY }}",
    ENDITEM,
    "Name=STATE", "Value={{ $ADDRESS.STATE }}",
    ENDITEM,
    LAST);
  ...
}
```

Figure 5.11: Test Script Generated for the Load-Runner

Figure 5.10 presents a snippet of VS test scripts generated to test TPC-W. In turn, Figure 5.11 shows a snippet for LR. These test scripts were instantiated based on abstract test cases presented in Figure 5.7. Basically, the test scripts are a set of several HTTP requests. Among the features

correlated in the set of test artifacts, we highlight the following example: 1) *Tag Request* - in the VS the attribute *Url* and the attribute *web_submit_data* in the LoadRunner are related to the parameter *@BuyConfirm.action*; the VS attribute *ThinkTime* and the LoadRunner parameter *lr_think_time* are correlated to the parameter *@BuyConfirm.thinkTime*; 2) *Tag QueryStringParameter* - the VS and LoadRunner attributes *Name* and *Value* are related to the parameter *@BuyConfirm.parameters*.

5.1.3 Performance MBT tools: Considerations and Lessons Learned

Throughout Section 5.1 we presented and discussed an approach to generate performance test scenarios from UML models. We also presented how we applied our approach to support the generation of two performance MBT tools from PLeTs: PLeTsPerfLR and PLeTsPerfVS. Furthermore, we showed how we applied these tools to generate performance test scripts and scenarios to test the TPC-W application .

Furthermore, our example of use provides an indication that generating abstract models is a powerful means to derive effective technology-independent test scenarios. It is important to highlight that the creation of an abstract model for later definition of a test script and scenario using a chosen workload generator needs only to annotate a few selected data in the UML models. Translating UML models to this representation is also more comprehensible for end-users when they are tracking bugs or trying to understand the flow of operations for a functionality.

Based on the lessons learned during the execution of those examples, we envision several future works. One could, for example, seamlessly translate a different UML model (e.g. Sequence Diagram) using our abstract model to generate scripts to some tool that is based on a different testing technique, e.g, structural or functional testing.

5.2 Generating Structural MBT Tools

This section presents two Model-based Testing tools³, generated by PLeTs, which generate structural test cases from UML Sequence diagrams. Since these tools are derived from PLeTs, they have common features and use the same approach: accept a system model as an input and automatically derive test cases using a random technique. After, the test cases can be run to test the corresponding code, *i.e.*, measure structural coverage using existing testing tools, *e.g.* JaBUTi [VMWD05] or EMMA [Rou]. Our structural MBT tools are composed of the following features (see Figure 5.12): (a) Parser: extracts test information about the classes and methods to be tested from UML sequence diagrams; (b) Test Case Generator: applies a random test data generation technique to generate an abstract structure, which has a technology-independent format (abstract test cases) and describes the test case information; (c) Script Generator: generates script/test driver for a specific testing tool from information contained in the abstract structure; (d) Executor:

³Most part of the structural MBT tools were developed in the context of a master dissertation [Cos12]

represents the test execution for a specific testing tool using the test driver generated in the previous step [CORS12] [Cos12].

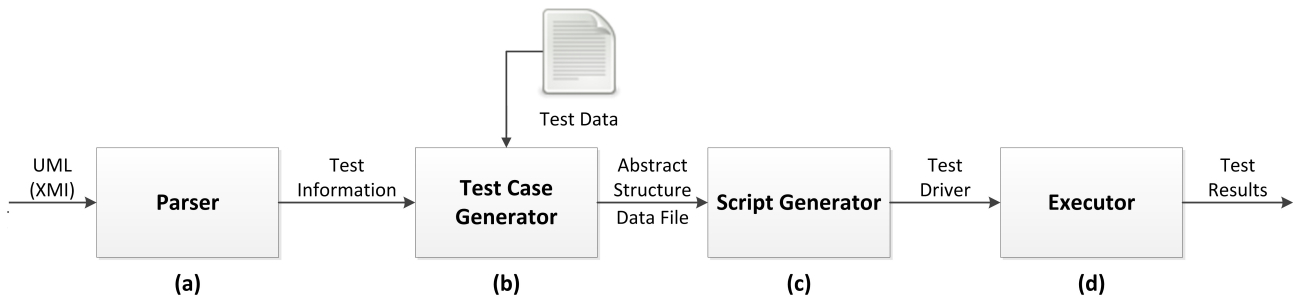


Figure 5.12: An Approach for Generating Structural Test Cases

It is important to notice that during the domain implementation we first developed the common structural components and then we focused on the development of the specific components (application implementation) required to generate the PletsStructJabuti tool, which is an MBT tool that generates script/test driver and executes it using the JaBUTi tool. After that we developed the specific structural components required to generate the PletsStructEmma MBT tool, which generates script/test driver and executes it using the EMMA tool [Rou].

In summary, both generated tools extract test information from annotated UML Sequence diagrams (test models), generate an abstract structure, instantiate the information present in this structure to generate and execute concrete test cases/test drivers, respectively, for the target tools: JaBUTi or EMMA. Furthermore, once the common components are developed and the first tool was generated, it was easy to generate the second one, because both tools share several features, which in turn are mapped to components, *e.g.*, parser and test data generation. Therefore, we were able to develop the second tool with less effort due to reuse of components already developed for the first tool.

5.2.1 An Approach for Structural Test Case Generation

As discussed in Chapter 3, MBT tools can take advantage of a variety of system models and modelling notations to support the automation of test case generation. For instance, UML models, which are one of the most used modelling notations during the software development life-cycle in the industry, can be used to automate the test case generation through annotation of test information on stereotypes and tags. Stereotypes and tags can be included in different elements of an UML model to represent test case information [OMG12].

In order to generate and execute scripts/test drivers from models, our approach must retrieve information about classes and methods, annotated in UML Sequence diagrams. It is important to highlight that the diagrams must be well-defined, *i.e.*, they have to contain information about classes and methods, parameters (name, type), as well as, each method return type. Besides, it is also necessary to annotate the diagrams with additional information, *e.g.*, a variable that will be

used to specify the path of the classes that will be tested. This information will be used to generate the abstract structure (practical details about how the diagram is annotated will be presented in Section 5.2.2). The UML sequence diagram is annotated with the following tags:

- **«TDexternalLibray»**: specifies the path of libraries of the SUT;
- **«TDclassPath»**: specifies the path of the classes that will be tested;
- **«TDtechnologyPath»**: specifies information about the chosen testing technology (such as JaBUTi, EMMA, etc.), e.g., the installation directory and the path of its launcher;
- **«TDimportList»**: specifies a list of imported classes (*import*, *package*).

Each tag can define a fixed value or a variable that can be replaced when generating the actual test case or driver for a specific structural testing tool. For example, the previously mentioned four tags must be annotated in the sequence diagram with the following parameters: *@externalLibrary*, *@classPath*, *@technologyPath* and *@importList*. However, these parameters are just a reference and have no actual information about technology, class path, external library or import list. After this annotation process, it is necessary to export all information described in the UML Sequence diagram to a XML file, which is the input of the first step in our approach.

The first step (*Parser*) of our approach consists of parsing a XML file to a parser in order to extract the information necessary for generating a data structure in memory, which we called Test Information (see Figure 5.12 - a). The Test Information describes the methods and classes that will be tested. The second step (*Test Case Generator*) receives as input the Test Information and a XML file called *Test Data* (see Figure 5.12 - b).

The *Test Data* has the actual values about libraries used to the application execution, the path of classes to be tested and the package list to be imported. However, the *Test Data* file has no technology information because the first two steps of our approach are technology-independent. Moreover, the *Test Data* also describes a set of different parameter values for all classes and methods of the application to be tested.

Based on that, the *Test Case Generator* applies a random test data generation technique [Kor90] under the parameter values contained on *Test Data* and only for the classes and methods described on Test Information. The reason for choosing this technique consists of selecting a set of specific parameters for each one of those classes and methods. This technique was used because it is less expensive and simpler to automate. However, other techniques for test data generation are presented in the literature, e.g., data generation based on symbolic execution [LCYW11] and data generation based on dynamic execution [DLL⁺09].

After applying the random test data generation technique, the *Test Case Generator* also produces the Abstract Structure and the Data File, which are inputs to the third step in our approach. The Abstract Structure is a text file that describes, in a sequential and technology-independent format, the entire data flow of the classes and methods to be tested (see Figure 5.16 for an example of file that contains an abstract structure). The Abstract Structure is divided in three groups:

- Technology Configuration: defines the *@technologyPath* parameter, which specifies the information about the technology that will be used for the test;
- Test Configuration: defines the *@classPath*, *@externalLibrary* and *@importList* parameters, which define the information used for a specific test case;
- Sequential Flow Configuration: defines the sequential flow of the methods that will be tested.

Each one of these parameters is a reference to the actual data that is stored in the Data File, which is a text file that contains the information (values) used to instantiate test cases for a given technology (see Figure 5.17 for an example of a file that contains actual values for a testing tool). The information of specific technology is provided in the step *Script Generator* (see Figure 5.12 - c), when the user must provide all information necessary to generate scripts using a specific testing tool, e.g., the path of its launcher. Therefore, when the Abstract Structure and Data File are instantiated to generate test scripts/test driver for a specific testing tool, a new class file⁵ is generated. This file contains a class that makes calls to the methods that will be tested and also includes a set of information to be used as input of those methods. In our approach, the input information is generated, automatically, using the random test data generation technique previously mentioned.

One of the advantages of using a file to store the actual values, which are used in the instantiation of the class file, is that it is not necessary to include in the UML Sequence diagram the parameter values of the methods that will be tested. Thus, to generate new test cases with different input values, it is only necessary to generate new test data using any kind of data generation technique.

Moreover, the advantage of using an abstract structure is related to the ability to reuse information for different structural testing tools, e.g., JaBUTi [VMWD05], Semantic Designs Test Coverage [Sem], Poke-Tool [Cha91], IBM Rational PurifyPlus [IBM] or EMMA [Rou]. In this sense, if a company decides, due to management strategy, to migrate to another structural testing tool it will be able to use the test cases previously generated. Besides that, the abstract structure presents the test information in a clear format, making it simple and easy to understand. Therefore, it is easier to automate the generation of test scripts for several tools.

The last step of our approach (*Executor* - see Figure 5.12 - d) consists of performing the test with a specific structural testing tool. Therefore, all the class files generated in step three are used for the test execution. The generation of the class files will be further described in Section 5.2.2.

5.2.2 Example of Use

This section describes how we have applied our approach to test an application to manage professional profiles of employees from an IT company *i.e.*, Skills. The main goal is to assess the

⁵In the testing tools in which we have applied our approach, this file is called *TestDriver.java*.

efficacy and the functionality of our approach and present how we derived the PletsStructJabuti and PletsStructEmma structural MBT tools from PLeTs. Figure 5.13 and Figure 5.14 present the tools' architecture and show the tools' common (UmlStruct, TestCaseGenerator, RandomDataGenerator and StructuralTesting) and specific components (JabutiScript, EmmaScripts, JabutiParameters and EmmaParameters) used to implement the steps of our approach. As mentioned in Section 5.2.1, the first two steps of our approach are related to the extraction of information from the UML Sequence diagram and to generate an abstract test structure. The abstract test structure contains test case information and is used to generate scripts to different structural testing tools, in this case the PletsStructJabuti and PletsStructEmma. The last two steps are technology-dependent, one to extract information from the abstract structure and to generate scripts and scenarios, and the other to execute scripts and collect results from the execution of a structural testing tool. Thus, to generate the MBT tools from PLeTs we reused the common components and developed two components, which support the two last steps of our approach.

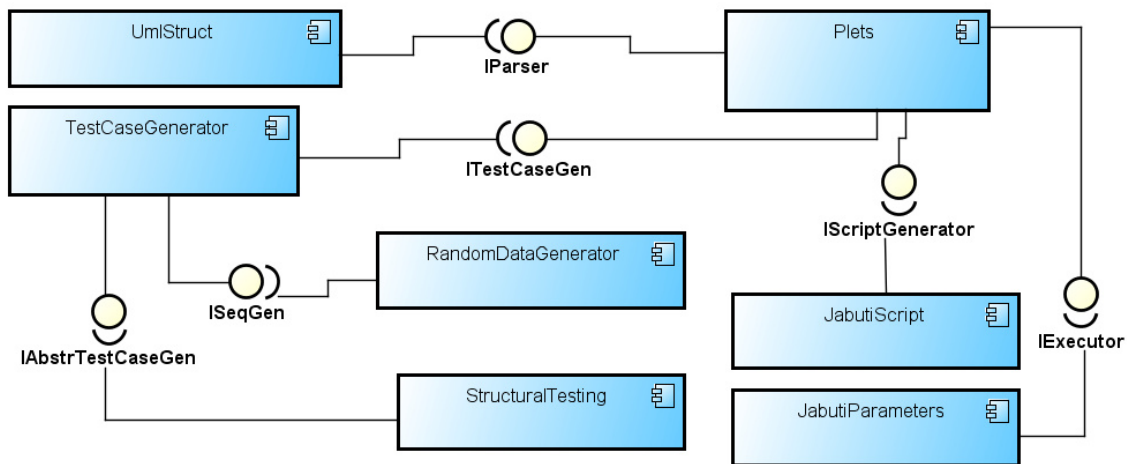


Figure 5.13: PLeTsStructJabuti Tool Architecture

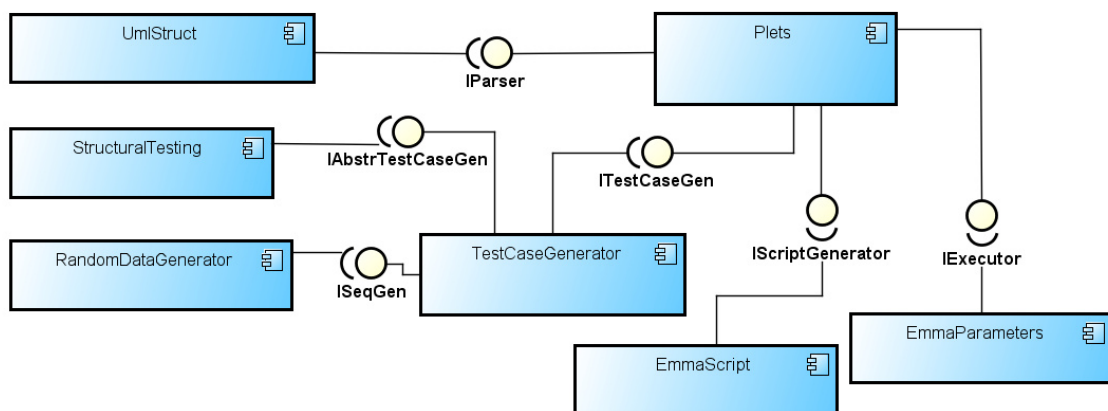


Figure 5.14: PLeTsStructEmma Tool Architecture

Test Cases for JaBUTi

JaBUTi [VMWD05] (Java Bytecode Understanding and Testing) is a tool used to perform structural testing for Java applications. Different from other structural testing tools, JaBUTi can run tests without requiring an analysis of source code. The execution of tests is performed based on the analysis of Java Bytecode. JaBUTi performs coverage analysis of classes and components of programs to be tested based on eight structural testing criteria, four criteria for data flow analysis and four for control flow analysis [VMWD05].

In order to automate test execution using JaBUTi, it is necessary to create a project file, whose extension is *jbt*. All information related to the Bytecode of the classes that will be tested and the path to these classes are stored in this file. This file also describes the paths to all libraries belonging to the application to be tested, as well as the Bytecode of the *TestDriver* class, which contains information that will be used to test the classes and methods from the SUT. Based on information of the classes' Bytecode, JaBUTi builds the Definition-Use Graph (DUG) for each class.

After creating the project file, JaBUTi performs the instrumentation of classes that will be tested. It runs the test cases described in the file *TestDriver.java* through a call for the *probe.DefaultProber.probe* method, which stores the program information that will be tested. Then, another method is invoked (*probe.DefaultProber.dump*) and all data collected in the previous method call are stored in a trace file (*.trc*). The data written to the trace file correspond to the paths taken by the program during the test run. JaBUTi extracts information from this trace file, updates the test data and recalculates the coverage information. Based on the coverage information the tester can decide conclude the test or re-execute the test.

Test Cases for EMMA

EMMA [Rou] is a testing tool that performs Java code analysis based on structural criteria and, like JaBUTi, it executes tests through analysis of the Java Bytecode. However, differently from JaBUTi, which uses a graphical interface for running test cases, EMMA performs structural analysis through a command line interface. In order to generate test cases, EMMA needs to call a Java process and pass some parameters, such as the path to the classes that will be tested, the application internal library, the *TestDriver* class and the *emmarun* command. This command is responsible for calling three other subcommands: (a) *instr*: used for the instrumentation of classes that will be tested; (b) *run*: used for executing the test cases described in the *TestDriver* class and for the trace file generation (*.em*), which has information about the paths covered during the program execution; (c) *report*: extracts information from the trace file, updates the testing data and recalculates the test coverage information. Finally, a report is generated (HTML or text format) with the test result information.

Skills tool - Workforce Planning

The application used as use case is called Skills (*Workforce Planning: Skill Management Tool*) [SRZ⁺11]. The main objective of this application is to manage and to register skills, certifications and experiences of employees for a given company.

Table 5.1: Classes and Methods to be Tested

| Classes | Methods |
|----------------------|---|
| ServletCertification | boolean searchCertification(String certification, String provider) String checkName(String certification, String provider) int getProvider(String certification, String provider) |
| ServletSkill | boolean searchSkill(String name) String checkName(String nome) |
| ServletExperience | ArrayList<String> getUserExperiences(int userID) |
| ServletProfile | ArrayList<String> getUsers(String name) String printResult(ArrayList<String> array) |
| ServletPassword | boolean checkPassword(String userName, String currentPasswd) boolean changePassword(String userName, String newPasswd) |

In order to verify the functional aspects of our approach, we have tested a set of classes and methods of Skills. These classes and methods are represented by a UML Sequence diagram (see Figure 5.15) that describes a process, in which an user performs the following operations: (a) search for particular certification information; (b) search for particular skill information; (c) display a list of registered experiences; (d) display information about the user profile; and (e) change the login password. As can be seen in Table 5.1, these operations are performed through calls of 10 methods of 5 classes. Note that our approach consists in automating the generation and execution of test cases, in which only the systems' internal methods are analyzed. Therefore, no method called from the user interaction will be analyzed, since our approach does not implement this feature. In this context, only the information about the methods described in Table 5.1 will be used to automatically generate and execute scripts/test driver.

In order to generate and execute test scripts, initially, we had to annotate the sequence diagram with the tags *TDexternalLibray*, *TDclassPath*, *TDtechnologyPath*, *TDimportList* and their respectively parameter values: *@externalLibrary*, *@classPath*, *@technologyPath* and *@importList*. These tags and values were annotated in the classifier elements, which represent the five classes used for this example of use. After annotating the UML Sequence diagram with test information, we export the model to a XMI file. This XMI file is an input for the PletsStructJabuti and PletsStructEmma tools. During their execution, the tools parse the XMI file, extracting information from the methods and classes that will be tested in order to generate a data structure in memory (Test Information). Based on the Test Information and the Test Data (a XML file with different parameter values for all classes and methods of the SUT), the tools apply a random test data generation in order to generate the Abstract Structure (Figure 5.16) and Data File (Figure 5.17).

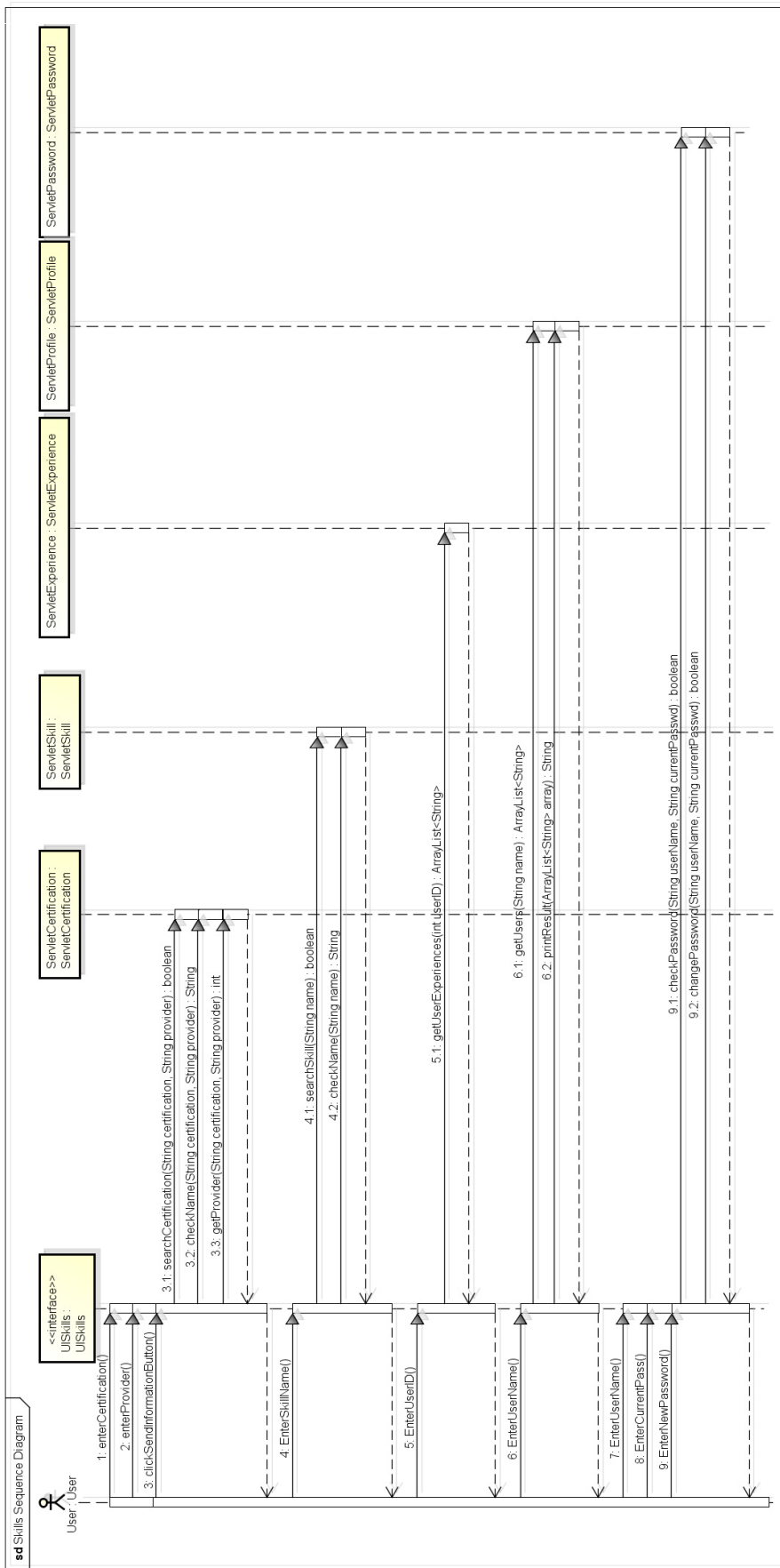


Figure 5.15: Sequence Diagram

```

Abstract Structure: Search for Certification
## Technology Configuration ##
Technology Information : <<TDtechnologyPath: @technologyPath>>

## Test Configuration ##
External Libraries : <<TDexternalLibray: @externalLibrary>>
Path Classes : <<TDclassPath: @classPath>>
Imported Classes : <<TDimportList: @importList>>

## Sequential Flow Configuration ##
1. ServletCertification
1.1. searchCertification(String certification, String provider):
boolean
1.2. checkName(String certification, String provider): String
1.3. getProvider(String certification, String provider): int
...

```

Figure 5.16: Code Snippet of the Abstract Structure

```

@technologyPath = C:\Jabuti\bin; C:\Jabuti\lib\bcel-5.2.jar;
C:\Jabuti\lib\capi.jar; ...
@externalLibrary = C:\Tomcat 6.0\lib\jsp-api.jar; ...
@classPath = C:\CmTool_SkillsTest\web\WEB-INF\classes; ...
@importList = servlets.*; java.io.*; java.util.StringTokenizer
1. ServletCertification
1.1. searchCertification("ActiveX", "BrainBench")
1.2. checkName("ActiveX", "BrainBench")
1.3. getProvider("ActiveX", "BrainBench")
...

```

Figure 5.17: Code Snippet of the Data File for JaBUTi

Figure 5.16 shows a code snippet of the Abstract Structure, which is divided into three information groups: *Technology Configuration*, *Test Configuration* and *Sequential Flow Configuration*. As mentioned in Section 5.2.1, all the parameters present in each information group are a reference to the actual data that is stored in the Data File, which contains all values that will be used to instantiate test cases for a given technology (JaBUTi or EMMA). Figure 5.17 presents a code snippet with information regarding the parameter values of this file. In the example depicted in Figure 5.17, information about JaBUTi technology path (*@technologyPath*) is defined.

```

import servlets.*;
import java.io.*;
import java.util.StringTokenizer;

public class TestDriver {
    static public void main(String args[]) throws Exception {
        ServletCertification servletcertification = new ServletCertification();
        servletcertification.searchCertification("ActiveX", "BrainBench");
        servletcertification.getProvider("ActiveX", "BrainBench");
        servletcertification.checkName("ActiveX", "BrainBench");
    }
}

```

Figure 5.18: Code Snippet of TestDriver.java class

Based on the information described in the Abstract Structure and Data File, the *TestDriver.java* class is generated. This class is the same for both JaBUTi and EMMA. Since JaBUTi and EMMA perform structural analysis on the Bytecode, PletsStructJabuti and PletsStructEmma create a Java process to compile the driver class. Figure 5.18 shows a code snippet of the *TestDriver.java*

file. In order to perform test cases with EMMA, automating the generation of the *TestDriver.java* class is enough. However, in order to perform test cases with JaBUTi it is necessary to generate a project file. PletsStructJabuti generates this project file by creating a Java process. This process runs a JaBUTi's internal class called *br.jabuti.cmdtool.CreateProject*, in which some information such as paths of the JaBUTi's internal libraries are used as an input parameter.

Once these two files are generated, the test execution consists in the internal call of the *probe.DefaultProber.probe* and *probe.DefaultProber.dump* methods for JaBUTi. At the end, the PletsStructJabuti creates a Java process to run JaBUTi, which is responsible to calculate and to show the updated coverage information for the defined test case. After viewing the coverage information, the user/tester has the possibility of terminating the PletsStructJabuti execution and then, finalizing the test or continuing to run the tool, in order to generate and execute more test scripts. Figure 5.19 shows the coverage results after four test runs. All classes and methods were analysed based on All-nodes criterion. As can be seen in the figure, the blue bar represents the coverage percentage after one run; the red bar represents the coverage percentage after two runs; the green bar represents the coverage percentage after three runs and the gray bar represents the coverage percentage after four runs. Note that the *searchCertifications*, *checkName* (*ServletCertification* class), *checkName* (*ServletSkill* class), *getUserExperiences*, *printResult*, and *checkPassword* methods were fully covered after one run; *searchSkill* after two runs; and *changePassword* after three runs. However, the *getProvider* and *getUsers* methods could not obtain a full coverage percentage even after three runs. Four runs were required for these two methods to achieve full coverage.

In order to generate and execute test scripts using PletsStructEmma, we have used the same UML Sequence diagram annotated with the same test information. Furthermore, all abstract test cases generated for PletsStructJabuti were also used for our second tool. In the same way as PletsStructJabuti, the user/tester has the possibility of continuing to run the tool in order to generate and execute more test scripts. The results for EMMA are similar to the ones for JaBUTi presented in Figure 5.19.

5.2.3 Structural MBT tools: Considerations and Lessons Learned

The examples presented in the Section 5.2.2 show that our approach allowed the use of the same diagrams and test cases to be used in two tools producing similar results. Furthermore, the generation of the PletsStructJabuti and PletsStructEmma were generated with less effort since they are derived from PLeTs, which allowed the reuse of some common components (*UmlStruct*, *StructuralTesting*, *RandomDataGenerator* and *TestCaseGenerator*). Although we have to develop specific components for both our tools (*EmmaScript*, *EmmaParameters* for PLeTsStructEmma and *JabutiScript*, *JabutiParameters* for PLeTsStructJabuti), this task required less effort compared to the development of an MBT tool without the adoption of a reuse-based strategy. For instance, the structural common components have 731 lines of code, the specific components of the PLeTsStructEmma tools have only 355 lines of code and the components which are specific to the PLeTsStructJabuti have

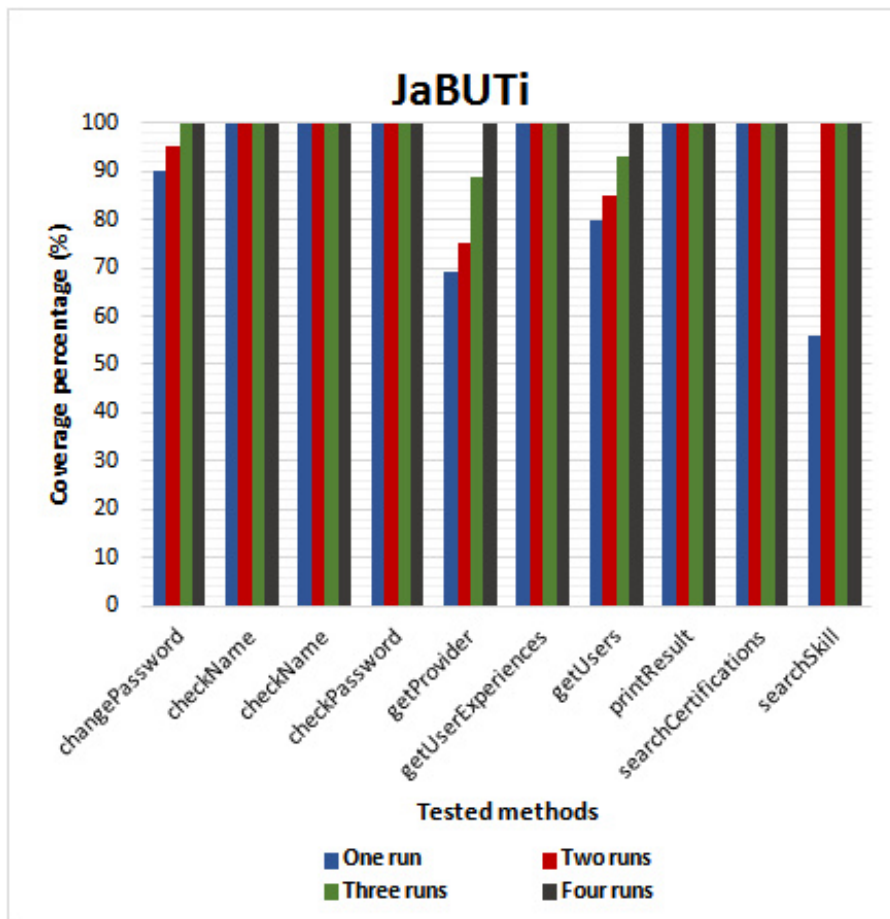


Figure 5.19: Coverage Information for JaBUTi

354 lines of code. Furthermore, once familiar with the functional features of the PletsStructJabuti tool, it was possible to perform tests with little learning effort using PletsStructEmma, since both tools share several features, e.g., parser, test data generation technique and the Abstract Structure format.

5.3 Final Considerations

This chapter presented two approaches to automate the generation of testing data from UML diagrams. Based on these approaches we developed some components to our SPL. We also showed that our SPL is able to generate MBT tools that can be integrated with academic or commercial tools to execute performance or structural tests. Thus, we generated two tools, PLeTsPerfLR and PLeTsPerfVS, to apply performance testing for a web application (TPC-W) and two tools, PLeTsStructEmma and PLeTsStructJabuti, to apply structural testing for an application from an IT company.

The main advantages of our approach is related to the possibility of reusing test information described in UML diagrams and in the abstract structure. Furthermore, the definition of an abstract structure is the central point of our approach to improve the reuse. Because of this, it is possible

to easily develop specific components which address the generation of scripts to a different test technology and consequently improve the reuse during the product generation and also the reuse of the test cases previously defined by another tool. For instance, the technologies used to exemplify our approach were two structural testing tools: JaBUTi and EMMA. However, commercial tools such as Semantic Designs Test Coverage, Rational PurifyPlus or other academic tools such as Poke-Tool could be used for this purpose. Furthermore, based on the number of lines of code that were written to develop the SPL components, we can assume that deriving tools from PLeTs requires less effort when compared to the development of an MBT tool without the adoption of a reuse-based strategy. In addition, once familiar with the functional features of one of the generated tools, it was possible to perform tests with little learning effort using other tools generated from PLeTs, since both tools share the same processes and consequentially the same main features, *e.g.*, parser, test data generation technique and the abstract structure format.

It is important to note that although our approach promotes the reuse of SPL components and the generated MBT tools promotes the reuse of the testing models, in the example UML diagrams annotated with testing information, the examples of use described in this section do not provide enough evidence about the required effort to use a MBT tools generated from PLeTs. Thus, it is relevant to compare the effort required to generate testing data using a tool generated from PLeTs with a standard testing tool used in the industry. Therefore, in the next chapter we perform an empirical experiment to understand the effort required to generate performance scenarios and scripts to test the performance of an application.

6. EMPIRICAL EXPERIMENT

This chapter presents an experimental study carried out to evaluate the effort to use MBT tools derived from PLeTs [RVZ10] [SRZ⁺11] [CCO⁺12] for the generation of test scripts and scenarios in the context of a performance testing activity. The MBT tool used in our experiment is the PLeTsPerfLR (see Section 5.1). We compared the effort of modelling the UML models and the generation of its corresponding script/scenario using PLeTsPerfLR MBT tool against the effort of creating the script/scenario with LoadRunner, a tool widely used in the industry that uses the Capture and Replay (CR) technique. Capture and Replay based tools require that the test engineer executes the tests manually once on the application to be tested, using the load generator in “record” mode, then defines scenario parameters (number of virtual users, test duration etc.) and runs the generator. Nevertheless, the planning and design of test scripts and scenarios for CR based tools is a highly specialized task - it requires that a test engineer understands the performance automation tools that are employed during the test phase, as well as knowledge about the application to be tested, its usage profile, and about the infrastructure(s) where it will operate. Thus, we see in current practice a bottleneck in productivity of performance engineering teams due to the workload and complexity involved.

The Capture and Replay based tools limitations and the MBT adoption benefits were some of our main motivation to propose PLeTs, since we can take the advantages of the MBT to automatically generate scripts and test data and at the same time reuse the testing tools and all the related testing team knowledge to generate testing data.

This empirical experiment is organized as follows. In Section 6.1 the experiment definition and its research questions and objectives are presented. Section 6.2 describes the experiment planning, its hypotheses, research questions and variables. We also describe the selection of the subjects, the experiment design and the threats to the experiment validity. In Section 6.3, the Preparation and Execution steps performed during the experiment operation are discussed; and, Section 6.4 describes the analysis of our findings and presents a general analysis and the relation of the results against our hypotheses. Finally, Section 6.5 summarizes the experiment results and presents our conclusions on the experiment.

6.1 Definition

6.1.1 Research Questions

The motivation of this experiment is to evaluate the effort when using an MBT tool, in our case PLeTsPerfLR, to test performance of web applications. It is well known that the software testing process is time consuming and one of the most expensive phases in the software development process. The main advantage when using MBT is that it provides support for automatic generation

of test scripts and scenarios, thus reducing effort during the software testing activity. However, test engineers usually face some key issues, such as, to understand the learning curve and effort to modelling the system, when they have to decide whether to adopt or not a MBT tool. Thus, we performed this experiment to support this decision and to provide empirical evidence to answer the following questions: is the effort to learn modelling less costly than developing test cases and scripts directly with a CR-based tool? That is, what is the evidence that our MBT tool improves overall testing productivity?

Our strategy to address these questions was to define a set of testing tasks, execute those tasks using PLeTsPerfLR MBT tool and a CR-based tool, and measure the efforts to use each one of them. For the execution with CR, we chose LoadRunner [Hew], which is widely used by many companies in the field. Each testing task is composed of a sequence of steps such that, when the subjects execute the task correctly using PLeTsPerfLR, the tool generates the same script and scenario produced when performing the same task correctly using LoadRunner. An important point arises here: since the correct execution was necessary to produce the same results, in order to make a meaningful comparison of efforts, we observed the rate of mistakes made by subjects during the execution of tasks and defined this rate as a control variable (see Subsection 6.2.3).

Our evaluation was conducted to answer the following research questions:

RQ1. *What is the effort of applying performance testing using the PLeTsPerfLR tool or the LoadRunner tool for generating simple test scripts and scenarios?*

RQ2. *What is the effort of applying performance testing when using the PLeTsPerf MBT for re-generating test scripts and scenarios due to changes in the application to be tested, when compared to LoadRunner?*

RQ3. *What is the effort to generate a complex set of test scripts and scenarios using PLeTsPerfLR or LoadRunner*

6.1.2 Objective Definition

Object of Study

The object of study is the effort of using the PLeTsPerfLR for testing performance. Basically, the user interaction with the MBT tool is limited to constructing/editing an UML model following some design guidelines and loading this model using the tool graphical user interface. Once constructed, UML models can be used by different PLeTs performance testing tools to generate scripts and scenarios. Thus, the effort presented in this study can be generalized to any MBT performance tool derived from PLeTs.

Purpose

The purpose of the experiment is to evaluate the effort of generating test scripts and scenarios when using PLeTsPerfLR.

Summary of Definition

Analyze the use of *PLeTsPerfLR* together with *LoadRunner* tools for the purpose of *evaluation* with respect to their *effort of generating scripts and scenarios* from the perspective of the *performance tester and the performance test engineers* in the context of *undergraduates, M.Sc. and Ph.D. students, performance testers and performance test engineers* for *generating performance test scripts and scenarios*.

6.2 Planning

In this section we present the plan of our experiment, its hypotheses, research questions and variables. We also describe the selection of the subjects, the experiment design and the threats to the experiment validity.

6.2.1 Context Selection

The context of the experiment is characterized according to four dimensions:

- **Process:** we used the *in-vitro* approach, since it refers to an experiment in a laboratory under controlled conditions. This experiment is not an industrial software testing, *i.e.*, it is off-line;
- **Participants:** undergraduate, master and doctoral students, testers and test engineers;
- **Reality:** the experiment addresses a real problem, *i.e.*, the differences in individual effort to create performance scripts and scenarios;
- **Generality:** it is a specific context since the MBT tool used in this experiment is an MBT performance testing tool derived from PLeTs SPL, but the ideas can be expanded to different MBT tools generated from PLeTs.

6.2.2 Hypothesis Formulation

In this section we stated our central hypotheses and also defined what measures will be used to evaluate the hypotheses. For each hypothesis, we use the following notation:

Φ_{pp} : represents the effort when using PLeTsPerfLR to generate performance scripts and scenarios;

Φ_{lr} : represents the effort when using LoadRunner to generate performance scripts and scenarios.

Recalling our research questions from Section 6.1:

RQ1. *What is the effort of applying performance testing using the PLeTsPerfLR or LoadRunner for generating simple test scripts and scenarios?*

Null hypothesis, H_0 : effort is the same when using PLeTsPerfLR or LoadRunner to generate simple performance testing scripts and scenarios.

$$H_0: \Phi_{pp} = \Phi_{lr}$$

1. **Alternative hypothesis, H_1 :** the effort is higher when using PLeTsPerfLR to generate simple performance testing scripts and scenarios than when using LoadRunner.

$$H_1: \Phi_{pp} > \Phi_{lr}$$

2. **Alternative hypothesis, H_2 :** the effort is lower when using PLeTsPerfLR to generate simple performance scripts and scenarios than when using LoadRunner.

$$H_2: \Phi_{pp} < \Phi_{lr}$$

Measures: effort (time spent).

RQ2. *What is the effort of applying performance testing when using the PLeTsPerf MBT for re-generating test scripts and scenarios due to changes in the application to be tested, when compared to LoadRunner?*

Null hypothesis, H_0 : effort is the same to re-generate performance test scripts and scenarios using PLeTsPerfLR or LoadRunner.

$$H_0: \Phi_{pp} = \Phi_{lr}$$

1. **Alternative hypothesis, H_1 :** the effort is higher to re-generate performance testing scripts and scenarios when using PLeTsPerfLR than when using LoadRunner.

$$H_1: \Phi_{pp} > \Phi_{lr}$$

2. **Alternative hypothesis, H_2 :** the effort is lower to re-generate performance testing scripts and scenarios when using PLeTsPerfLR than when using LoadRunner.

$$H_2: \Phi_{pp} < \Phi_{lr}$$

Measures: effort (time spent).

RQ3. *What is the effort to generate a complex set of test scripts and scenarios using PLeTsPerfLR or LoadRunner?*

Null hypothesis, H_0 : effort is the same using PLeTsPerfLR or LoadRunner to generate a complex set of test scripts and scenarios.

$$H_0: \Phi_{pp} = \Phi_{lr}$$

1. **Alternative hypothesis, H_1 :** the effort is higher when using PLeTsPerfLR to generate a complex set of test scripts and scenarios than when using LoadRunner.

$$H_1: \Phi_{pp} > \Phi_{lr}$$

2. **Alternative hypothesis, H_2 :** the effort is lower when using PLeTsPerfLR to generate a complex set of test scripts and scenarios than when using LoadRunner.

$$H_2: \Phi_{pp} < \Phi_{lr}$$

Measures: effort (time spent).

6.2.3 Variables Selection

In this section, we present the dependent and independent variables that are used to represent the experiment treatments and their measured values (see Table 6.1). The independent variables describe the treatments and are, thus, the variables for which the effects should be evaluated. The measured values of the dependent variable represent mainly the performance of the different software testing tools, and hence describe the effectiveness of the different tools.

Independent Variable

The main independent variable of interest in this study is the choice of a performance testing approach. It has a nominal scale and can take one of two values: PLeTsPerfLR or LoadRunner.

Dependent Variables

The main dependent variable defined in this experiment is effort, measured as the amount of time that the subjects spend on tasks for generating the test scripts and scenarios.

Control Variables

The control variables used in the experiment are the degree of formal education, the experience of the subjects in performance testing and the number of mistakes/errors (error rate)

made during task execution. We call it error rate for short. The degree of formal education and the experience of subjects are blocking variables and are included to reduce sources of variability and thus lead to improved precision. The error rate is used to ensure that the level of difficulty of the tasks was independent of the tool employed, thus ensuring that effort comparisons are meaningful.

Table 6.1: Scales of Experiment Variables

| Experiment variables | | |
|----------------------|----------------------------|------------|
| Variable Type | Variable Name | Scale Type |
| Independent | Software testing tool | Nominal |
| Control | Experience of subjects | Ordinal |
| Control | Degree of formal education | Ordinal |
| Control | Error rate | Ratio |
| Dependent | Effort | Ratio |

Since we are interested in errors that would lead the tools to yield different scripts as results, the error rate measure must be explained a little further. In order to cope with the characteristics of each technique, we defined a set of specific structural criteria for classifying errors made by subjects in each tool, PLeTsPerfLR and LoadRunner. For the PLeTsPerfLR MBT tool, the errors were classified as follows:

- * *Non-Critical errors (Typing errors)*: typing errors when inserting information in applications (SUT) forms are considered non-critical errors;
- * *Critical errors (Modelling errors)*: these are errors related to performance modelling and can be clustered in:
 - *Tag errors*: the tester forgot some tag or wrote an incorrect tag name. For instance, errors in the name of some *TDaction* tag do not generate performance scripts correctly;
 - *Stereotype errors*: the tester defined a wrong number of stereotypes and then created an incorrect stereotype or missed a stereotype;
 - *Class Base errors*: these are the modelling errors related to define a stereotype in a wrong UML element.

For the model-based approach, the following formulas were defined:

* *Non critical errors (Typing Errors)*: $\gamma = 1 - (\text{tagged values with errors} / \text{total tagged values})$;

* *Critical errors (Modeling Errors)*: $\Phi = 1 - (\Psi + \Omega + \Gamma) / 3$, where:

Ψ Tag errors = TD names with errors / total TD names;

Ω Stereotypes errors = stereotypes names with errors / total stereotypes names;

Γ Class Base errors = class base with errors (missing or exchanged) / total class base;

- * *Error rate*: the error rate of the generated scripts and scenarios when using PLeTsPerfLR is μ_{pp}
 $= 1 - [(\text{Tagged values with errors} + \text{TD names with errors} + \text{Stereotypes names with errors} + \text{Class Base with errors}) / (\text{Total Tagged values} + \text{Total TD names} + \text{Total Stereotypes names} + \text{Total Class Base})]$

On the other hand, to measure the error rate for when using LoadRunner we defined the following error classification:

- * *Non Critical errors (Typing errors)*: $\gamma = \text{form fields with errors} / \text{total form fields}$;
- * *Critical errors*: $\Phi = 1 - (\Theta + \Xi)$, where:
 - Θ Scripting errors = missing record operations on TPC-W / total record operations on TPC-W;
 - Ξ Scenario errors = fields configured with wrong values / total fields (e.g., VUsers, ramp-up, etc.).
- * *Error rate*: the error rate of the generated scripts and scenarios when using LoadRunner is μ_{lr} (error rate) = $1 - [(\text{missing record operations on TPC-W} + \text{fields configured with wrong values} + \text{form fields with errors}) / (\text{total record operations on TPC-W} + \text{total fields} + \text{total form fields})]$.

6.2.4 Selection of Subjects

The subject selection was defined by the availability of academic and professional performance testers/engineers. We invited doctoral, master and undergraduate students to participate in our experiment as subjects. The undergraduate students were second year, or later, students from a Computer Science or Information Systems courses. They come from one university (PUCRS) and from one college (Senac). Each subject had different experience knowledge, such as: experience in the industry as software analyst or as developer, or just experience developing software in an IT undergraduate course. The professional subjects were from only one IT company.

6.2.5 Experiment Design

The experiment design addressed the following general principles:

Randomization: The subjects were randomly allocated to each performance testing tool - PLeTsPerfLR or LoadRunner. Moreover, as all subjects execute all treatments (paired comparison design), we randomly defined their execution sequence.

Blocking: as mentioned before, the selected subjects for this experiment had different background in performance testing. Thus, to minimize the effect of those differences, the subjects were classified in two groups according to their skills in software testing (beginner and advanced groups - see Figure 6.1).

Balancing: the subjects are randomly grouped into each group (randomized block design), so that each tool is performed by the same number of subjects (PLeTsPerfLR or Loadrunner).

Standard design types: The design type presented aims to evaluate whether the values of ϕ_{pp} and ϕ_{lr} are different for similar values of μ_{pp} and μ_{lr} . Thus, it is necessary to compare the two treatments against each other. As defined in [WRH⁺00], the One Factor with Two Treatments design type must be applied. The factor is the software testing tool that will be used and the treatments are the PLeTsPerfLR and LoadRunner tools. The response variable is measured on a ratio scale, *i.e.*, to allow us to rank the items that are measured and to quantify and compare the sizes of differences between them.

6.2.6 Instrumentation

The background and experience of the individuals were gathered through a survey applied before the experiment started. The collected information provided the profile of the subjects.

Objects: The main objects are the test scripts and test scenarios generated for testing the TPC-W application [WRH⁺00]. Other documents were provided for the execution of the experiment, such as: non-functional requirements, test specification and test plan. We have provided a supporting tool, called Argo UML [Zha06], for modelling the Use Cases and Activity diagrams.

Guidelines: The tools were presented to the subjects through a printed manual, with an overview of the tools and detailed instructions on how to use them to create performance scripts and scenarios to test the application. We performed a training phase in a laboratory room for all the experiment's subjects. During the training, the subjects could ask open questions about the tools, the modelling process and the creation of scripts and scenarios described in the manual. It is important to note that in the training phase, a different application is used to create scripts and scenarios. Questions and answers were shared among all the subjects in the training room. During the experiment execution, a printed guide is used by the subjects. It includes information about the process for generating test scripts and scenarios using both testing tools, *i.e.*, PLeTsPerfLR (a guideline to modeling performance using Argo UML tool was also included) and LoadRunner.

Measurements: We collected effort metrics for each subject. All subjects performed the tasks using the same computational resources.

6.2.7 Threats to validity

The experimental process must clearly identify the concerns about the different types of threats to the experiment validity. Defining a clear experimental process and describing each threat and how we work to mitigate it allows further analysis of the experiment by researchers, and also helps to ease the study replication. There are different classification schemes for different types of threats to validate the experiment. We adopted the classification scheme published by [CC79], which is divided in four types of threats:

Conclusion validity: this type of threat affects the ability to draw conclusions about relations between the treatment and the outcome of an experiment.

- *Measures reliability:* this perspective suggests that objective measures are more reliable than subjective measures, *i.e.*, they do not depend on human judgement. In our study, the measurements of effort and error rate do not involve human judgement;
- *Treatment implementation reliability:* even though we use the same application to apply both treatments, there is the risk that the treatment implementation is not similar between different subjects. This risk cannot be completely avoided in our study, since we cannot interfere with the subjects when they are generating test scripts and test scenarios. Certainly, different subjects could define distinct tests scripts and scenarios;
- *Random irrelevancies in experimental setting:* the experiment was executed in an isolated laboratory, to avoid external interaction, such as the use of mobile phones, interruptions, etc;
- *Random heterogeneity of subjects:* the variation due to the choice of heterogeneous participants with different experiences and academic degrees may be a threat to the validation of experiment results. To mitigate this threat we defined academic degree and experience in performance testing as blocking variables.

Internal validity: focus on the threats to the internal validity of the experiment.

- *History:* the date to start the experiment execution was defined to avoid periods in which subjects may be exposed to external influences, *e.g.*, avoid running the experiment during the exam period (students subjects) and close to the start/end of an important project (industry subjects);
- *Maturation:* each experiment session was applied in the morning because the subjects are more motivated and less tired by the day's workload;
- *Selection:* a questionnaire was applied to assess the knowledge and the experience of subjects and then used to select and group (block) the subjects.

External validity:

- *Subjects*: a threat to the experiment's external validity was to select a group of subjects that may not be representative to the performance testing community. Thus, to mitigate this threat, we selected students with some skills, basic programming and UML modelling knowledge, to work with performance testing and professionals with different levels of expertise in performance testing. We also categorized the subjects into four groups, two groups for academic subjects and two for industry subjects. Eventually, we obtained a balanced group of students and professionals, but the low number of graduated students and the beginner professionals may have influenced the results. Because of this, in our experiment design, each subject executes two treatments, thus we were able to compare the results from the two tools. To avoid the effect of a learning curve influencing the results, the subjects of each block were divided into two groups, where each group started with a different treatment;
- *Tasks*: another threat to the experiment is that the tasks defined to generate the performance scripts during the experiment execution may not reflect the activities performed by a performance tester when testing an application. To mitigate this threat we interviewed some senior performance testers and performance engineers, from different companies, to define how the scripts must be generated and what is a reasonable task size (defined by the number of requests). The performance consultants did not participate and did not have any contact with the experiment subjects;
- *Experiment effects*: to mitigate the fact that the author of PLeTsPerfLR could know some of the subjects, and this fact could influence the results, the information about the author of the tool was not provided to the subjects until the end of the experiment. Also, the author was not directly involved in the experiment execution, preserving his identity;

Construct validity: a possible threat to the validity of our experiment is the fact that some subjects may know that one of the tools used in the experiment was developed for someone related to the experiment team. Another threat that may be present in every experiment is the fact that some subjects can erroneously conclude that their personal performance is measured to, for instance, rank them. To mitigate these threats, before each session we explained that we were evaluating the tools, not the subjects.

Another possible threat is that the complexity and the size of the scripts generated during the experiment may not be representative enough to generalize the results. To minimize this threat, we interviewed senior performance testers and performance engineers to define the complexity and the size of the scripts.

6.3 Operation of Experimental Study

This section discusses the preparation and execution steps performed during the experiment operation.

6.3.1 Preparation

Even when an experiment is perfectly designed and the data are properly analysed, the results will be invalid if the subjects do not compose a balanced sample or are not committed to the experiment [WRH⁺00]. In this section, we present how the experiment was conducted, how the documentation was prepared and how the experiment environment was configured. We also describe how the experiment subjects were involved and motivated.

Our first personal contact with the experiment subjects was made through a presentation session, where the experiment was described. As mentioned previously, when defining the schedule of sessions, we took special care to avoid running the experiment during the exam period (an issue for student subjects) and close to the start/end of a project (an issue for industry subjects). The presentation session was divided into two parts: an initial explanation about the general idea of the experiment, and the second part was reserved for questions and suggestions. Other points addressed during the presentation session were related to certify that all the subjects understood the research objective and how and which results would be published, making clear that any personal data would be kept confidential. At the end of the session a subject profile form was provided to all the subjects. The information extracted from the forms was used to classify and distribute the subjects through the blocks before running the experiment.

Another issue we discussed during the experiment preparation was related to how the experiment data should be collected. To avoid human mistakes we defined that the data collection related to effort (time spent to perform an activity) should be automated. Thus, we used simple software that measures how much time each subject spent to generate each script using PLeTsPerfLR and LoadRunner. To collect error rate data we used scripts generated by the subjects. Thus, after every experiment session, the generated scripts were moved to a server and the computer used in the experiment was restored to its previous state. After running the experiment, the generated scripts were independently verified by two senior performance test engineers. The verification was focused on the validation of the generated scripts against the testing documentation trying to find any misinterpretation or inconsistency.

6.3.2 Execution

The experiment execution took place in November and December of 2012 and was composed of two phases: training and experiment execution. The training phase was divided into two sessions: one was used to train subjects in modelling performance testing using UML and to use PLeTsPerfLR; another was used to train on the use of LoadRunner to generate scripts to test a web-based application. Furthermore, each session was divided in two parts: in the first part, we presented and demonstrated how to use the tool; in the second part, each subject used the tool to generate a performance script in accordance with the training guidelines. For the training phase, the SUT was Skills (see Section 5.2.2)

The experiment execution phase was split in three sessions, in which each session is composed of two tasks. These tasks were performed using the PLeTsPerfLR and LoadRunner tools, one for each task, and both tasks should generate an equivalent performance script as an output. For the experiment execution phase, the SUT was TPC-W [Men02], a standard benchmark that simulates an e-commerce website. Each session of the execution phase is described next:

- **Session 1:** to generate a simple performance script and scenario in accordance to the experiment guidelines:
 - *Task 1:* edit a simple UML model, annotating it with performance stereotypes and tags, and generate a standard performance test script using PLeTsPerfLR;
 - *Task 2:* to generate a simple performance testing script using LoadRunner.
- **Session 2:** to edit a script/model in accordance with the experiment guidelines:
 - *Task 1:* to edit an UML model and to generate scripts using PLeTsPerfLR;
 - *Task 2:* to edit the performance script and to re-generate the scripts using LoadRunner.
- **Session 3:** to generate a set of test scripts and scenarios in accordance with the experiment guidelines:
 - *Task 1:* to edit an UML model and to generate a more complex set of test scripts and scenarios using PLeTsPerfLR MBT tool;
 - *Task 2:* to edit the performance scenario and to generate a more complex set of test scripts and scenarios using LoadRunner.

Figure 6.1 represents the experiment timeline and the order that each group of subjects (block) performed the tasks of the sessions. Thus, Group 1 started the sessions using PLeTsPerfLR (model-based approach - M) and then used LoadRunner (Capture and Replay - C). Alternately, Group 2 started each session using LoadRunner and after PLeTsPerfLR. The experiment sessions

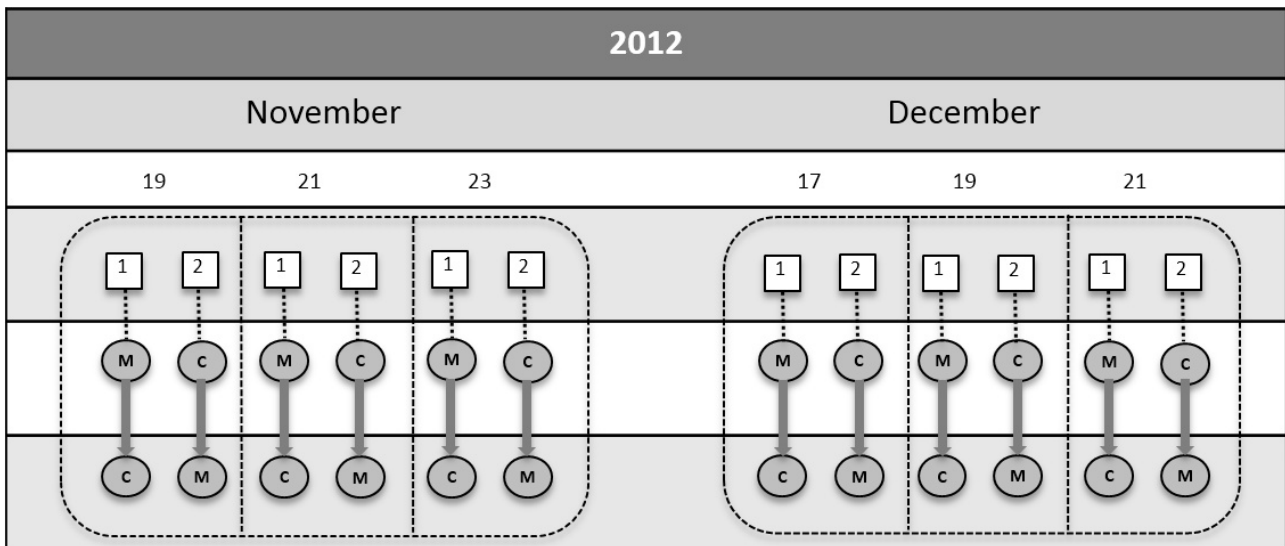


Figure 6.1: Experiment Run Timeline

were conducted over a two-month period: in November sessions were executed by the students (we split the groups to make it easier to define a time frame to execute the sessions) and the subjects from the industry executed the sessions in December. Moreover, to mitigate the fact that the time between each session and also the time between the end of the training phase and the experiment may influence the results, we defined a break of one day between each subjects' session and a two-day break between phases.

Table 6.2 represents how the experiment subjects were distributed, the number of subjects in each block and their academic or professional background. It is important to highlight that the subjects were randomly selected to start using one of two treatments (paired comparison design). In spite of the fact that all the subjects executed both treatments, we were concerned about the subjects' background distribution. The number of subjects from the industry with an advanced background was almost three times higher than subjects from academy. In spite of the subjects' different background, the general results achieved by the advanced industry subjects were similar to those achieved by beginner students. For instance, the advanced industry subjects generated a complex set of scripts using PLeTsPerfLR in less time than using LoadRunner; the same result was achieved by the beginner students.

6.3.3 Results

In this section we present the effort and error rate data collected. Table 6.3 presents the effort data collected while the subjects performed the tasks of each session presented in Section 6.3.2. Columns **1**, **2** and **3** present the average session time per block and the **Total** column presents the overall time of the sessions per subject block. To better summarize the results, we also present the average time (**Avg.**) spent per block to complete all the sessions. Furthermore, columns **Avg.1**, **Avg.2** and **Avg.3** show the average time spent by the subjects to apply each treatment (PP and LR

Table 6.2: Assigning Subjects to the Treatments for a Randomized Design

| Subjects Assignment | | | |
|---------------------|----------|----------|------------------|
| | Blocks | | Num. of subjects |
| PLeTsPerfLR | Academic | Beginner | 10 |
| | | Advanced | 5 |
| | Industry | Beginner | 4 |
| | | Advanced | 11 |
| LoadRunner | Academic | Beginner | 10 |
| | | Advanced | 5 |
| | Industry | Beginner | 4 |
| | | Advanced | 11 |

tools) to execute the session activities. The table shows that the average effort using PLeTsPerfLR was higher than with LR in Session 1 (38,97 min vs 14,49 min), and lower in Sessions 2 and 3 (14,05 vs 19,29 and 20,86 vs 39,20 respectively).

Table 6.3: Summarized Data of the Effort

| | | Effort | | | | | | | | |
|-----------|----------|----------|-------|-------|-------|-------|-------|-------|-------|-------|
| Treatment | Blocks | 1 | 2 | 3 | Total | Avg. | Avg.1 | Avg.2 | Avg.3 | |
| PP | Academic | Beginner | 47.90 | 13.70 | 21.70 | 83.30 | 27.77 | 38.97 | 14.05 | 20.86 |
| | | Advanced | 43.20 | 14.40 | 22.60 | 80.20 | 26.73 | | | |
| | Industry | Beginner | 36.25 | 16.25 | 20.25 | 72.75 | 24.25 | | | |
| | | Advanced | 28.55 | 12.82 | 18.91 | 60.28 | 20.09 | | | |
| LR | Academic | Beginner | 15.60 | 20.10 | 35.00 | 70.70 | 23.57 | 14.49 | 19.29 | 39.20 |
| | | Advanced | 15.60 | 19.40 | 34.80 | 69.80 | 23.27 | | | |
| | Industry | Beginner | 14.50 | 19.50 | 46.75 | 80.75 | 26.92 | | | |
| | | Advanced | 12.27 | 18.18 | 40.27 | 70.72 | 23.57 | | | |

Legend - **PP**: PLeTsPerfLR; **LR**: LoadRunner; **1, 2, and 3**: experiment sessions; **Avg.:** all sessions/block average time; **Avg.(1, 2, 3)**: treatment/session average time

Table 6.4 presents the summarized data of the error rate of the performance testing scripts and scenarios. We collected the error rate data from the performance scripts and scenarios generated by each subject while performing the tasks described in Section 6.3. Some considerations about the retrieved error rate data:

- A wrong configuration of any information related to the test scenario counts as an error for both approaches;
- A model inconsistency is characterized as an error because it will generate incorrect scripts and scenarios (e.g., accidentally deleting a transition).

Table 6.4: Design Mistake/error Data of the Performance Testing Scripts and Scenarios

| | | Error rate (%) | | | | | | | | | |
|----|----------|---------------------|-------|-------|-----------------|------|------|-------|-------|-------|-------|
| Tr | Blocks | Non critical errors | | | Critical errors | | | Total | | | |
| | | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | |
| PP | Academic | Beginner | 15.42 | 5.45 | 5.00 | 3.17 | 0 | 0 | 11.05 | 2.50 | 2.60 |
| | | Advanced | 8.14 | 7.27 | 10.77 | 1.11 | 1.54 | 0 | 5.68 | 4.17 | 5.60 |
| | Industry | Beginner | 21.61 | 4.55 | 4.81 | 1.22 | 1.92 | 0 | 13.95 | 3.12 | 2.50 |
| | | Advanced | 11.40 | 5.79 | 8.39 | 2.28 | 1.40 | 0 | 8.23 | 3.41 | 4.36 |
| LR | Academic | Beginner | 1.48 | 2.05 | 0.67 | 5.21 | 5.60 | 6.53 | 2.93 | 2.14 | 1.54 |
| | | Advanced | 2.96 | 1.93 | 5.14 | 1.25 | 6.60 | 2.21 | 2.44 | 2.50 | 4.85 |
| | Industry | Beginner | 18.52 | 19.28 | 20,77 | 6.25 | 2.00 | 0.92 | 14.63 | 15.18 | 16.60 |
| | | Advanced | 2.36 | 4.49 | 6.53 | 1.52 | 0.91 | 1.42 | 2.00 | 3.73 | 5.51 |

Legend - **Tr**: *treatments*; **PP**: *PLeTsPerfLR*; **LR**: *LoadRunner*; **1**, **2**, and **3**: experiment sessions.

6.4 Analysis and Interpretation

In this section we summarize our general findings on the data described in Section 6.3.3, mainly the measured effort for task execution. Figure 6.2 presents the box-plot graph of the data set relative to Session 1. In this data set, the median of execution time with the model-based approach was 35 minutes, while the median with LoadRunner was 13 minutes, expressively lower than the result with PLeTsPerf.

An issue that needs further attention is the high variability found in the data describing effort with PLeTsPerfLR: standard deviation is about 3 times higher than the other one (13,31 vs 4,43). This difference did not occur in the other two tasks. A possible explanation could be that, apparently, for some subjects (undergraduate students) the real training in modelling was the first task, which established an uniformity in their modelling skill. Nevertheless, this should be further investigated. There is a change in the box-plot graph of Session 2 (Figure 6.3). In this data set, the median of execution time with PLeTsPerfLR is 13.5 minutes, while the median with LoadRunner is 19 minutes.

The relatively small gain for PLeTsPerfLR observed in data set for Session 2 is more visible in the data set for Session 3 (Figure 6.4). Here, the median for this approach is 20 minutes, against 36 minutes for LoadRunner. There is an outlier in the data for PLeTsPerfLR (represented by the * symbol in the Figure 6.4), which we chose to keep because it does not affect the value of the median.

We performed hypothesis testing for all data sets using the PortalAction statistical package [Por], which is based on the R platform [R P]. First, we evaluated the hypothesis of distributions being normal with the Shapiro-Wilk normality test (Table 6.5). We are considering a significance level $\alpha = 0,05$ [Por].

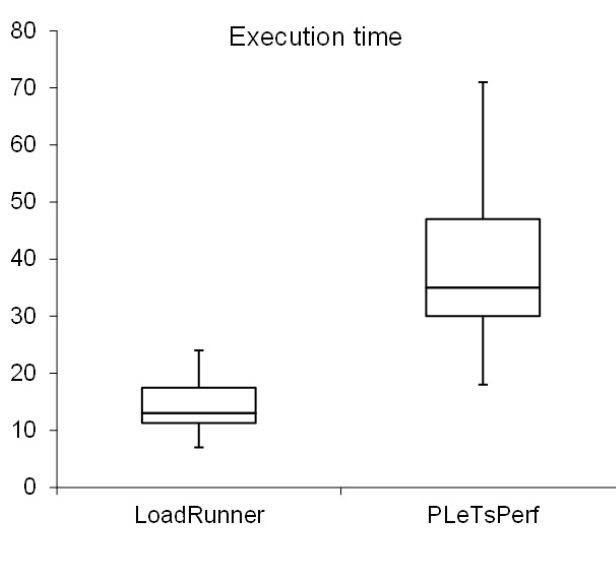


Figure 6.2: Experiment Time - Session 1

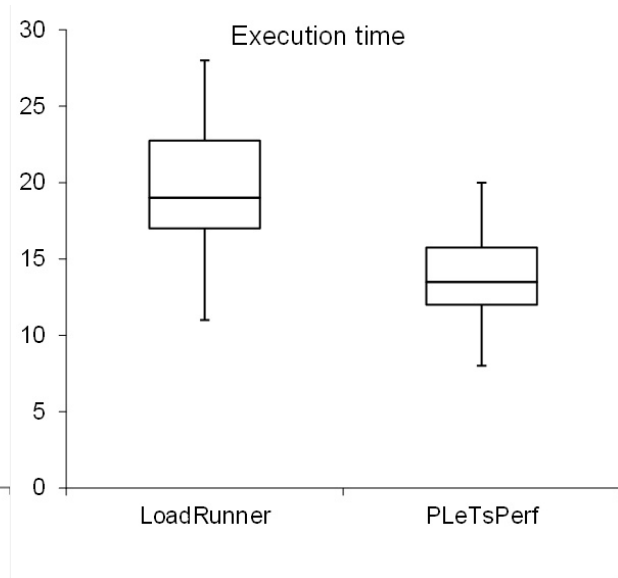


Figure 6.3: Experiment Time - Session 2



Figure 6.4: Experiment Time - Session 3

The results for the sample collected in the first session with LoadRunner show p-value less than α , indicating that the distribution cannot be assumed to be normal. Also, the sample for PLeTsPerfLR tool in the third data sets (related to Session 3) has a p-value of 0.045, thus rejecting the normality hypothesis. The other distributions all have p-values higher than 0.05, allowing the assumption that they have normal distributions. Since only two samples rejected the normality hypothesis, and with values below but very close to α , we also applied the Kolmogorov-Smirnov test, which resulted in accepting the normality hypothesis for all samples. Even so, for the hypothesis testing, we decided that it was safer not to assume normal distributions.

Since some of the distributions in the datasets do not satisfy the normality assumption when using the Shapiro-Wilk test [WRH⁺00], in order to test the null hypothesis, we applied the Wilcoxon matched pairs signed-ranks test, which does not assume normal distribution of data. For each data set (Sessions 1, 2 and 3) we applied the test to the paired samples (LoadRunner, PLeTsPerfLR), in

Table 6.5: Distribution Using Shapiro-Wilk Normality Test

| Treatment | Session 1 | Session 2 | Session 3 |
|------------------|------------------|------------------|------------------|
| LoadRunner | 0.021 | 0.383 | 0.267 |
| PLeTsPerfLR | 0.062 | 0.403 | 0.045 |

this way comparing the effort of PLeTsPerfLR with the effort using LoadRunner. The results are shown in Table 6.6.

For all pairs of samples involving PLeTsPerfLR and LoadRunner, the results of the Wilcoxon test was much smaller than 0.05, by 3 to 5 orders of magnitude. Therefore, we reject the null hypothesis and conclude that there was a significant difference in effort between the PLeTsPerfLR and LoadRunner tools. In Session 1 data set, the effort was higher using PLeTsPerfLR; in the data sets of Sessions 2 and 3, the effort with the PLeTsPerfLR was lower. Thus, in Session 1, we confirm the alternative hypothesis **H1**, and confirm the alternative hypothesis **H2** in Sessions 2 and 3.

Table 6.6: Wilcoxon Matched Pairs Signed-ranks Test Results - Effort

| Treatment | Session 1 | Session 2 | Session 3 |
|------------------|------------------|------------------|------------------|
| PLeTsPerfLR/LR | 0.00000086 | 0.00000558 | 0.00000188 |

We compared the total error rates in the three sessions, as well as the error rates by degree of formal education (academy vs industry). We again applied the Wilcoxon matched-pairs signed ranks test. The only significant difference was in Session 1, where the error rate using LoadRunner was slightly better than with PLeTsPerfLR. In all the other cases, the test confirmed the null hypothesis, meaning that the difference was not significant - in other words, the difficulty level of the tasks was the same with both tools; therefore, the effort comparison is meaningful. In Session 1, the difference in error rate was consistent with the difference in effort - subjects completed the task using LoadRunner with less effort and with fewer errors than using PLeTsPerfLR. In the comparison between academic and industry, in all sessions the Wilcoxon test confirmed the null hypothesis, indicating that the error rates of both classes of subjects was the same for both tools.

6.5 Conclusions

Automation of software testing through reuse of software artifacts is a good alternative for mitigating the testing costs and increasing the efficiency and effectiveness of testing. Model-Based Testing is a technique to support the generation of testing artifacts based on software models. In the case of performance testing, which is intensively based on automation, the design of scripts and test scenarios is a highly specialized and costly task. Thus, it is likely that performance testing could benefit from model-based techniques. Nevertheless, there is lack of empirical evidence of such claims in the literature. In particular, in the software industry, engineers are sceptical of the benefits of

MBT, mainly due to the needed learning and modelling time and effort. The main contribution of our experiment is to present results of a controlled experiment comparing the effort of creating a model of a system to be tested, and then using PLeTsPerfLR to automatically generate the performance test script from the models, against the effort of creating a script for the same system using a Capture and Replay based tool: LoadRunner. The results pointed out that, for a simple testing task, the effort of using the LoadRunner tool was lower than using PLeTsPerfLR, but as the complexity of the testing task increased, the advantage of using our MBT tool increased significantly.

Since we focused on understanding the effort of using PLeTsPerfLR in the context of a cooperation project with industry, we chose to focus on the comparison of the effort in completing testing tasks using our Model-based Testing tool and a Capture and Replay based tool. Fortunately, it was possible to ensure that the results of tasks - the scripts - were the same for both tools. That was because, in the case of PLeTsPerfLR, the scripts were derived automatically, and in the case of LoadRunner they were created with a Capture and Replay wizard. Therefore, when creating the tasks guidelines, we defined the modelling steps and the capture steps in such a way that, if executed correctly, they would produce the same results. But what if they were not executed correctly? What if subjects completed tasks faster but with many errors with one tool and not with the other? The effort comparison would be meaningless. In order to control this, we defined a (somewhat *ad hoc*) measure of correct execution. Fortunately, the results allowed us to draw conclusions from the effort data.

Another important point was the heterogeneity of our population of subjects, which could have threaten the validity of the experiment. Unfortunately, we did not have enough subjects available to make a perfect balance of beginner and advanced experience levels. The sample sizes of these two subsets were different and small (14 and 16 subjects respectively), not allowing us to draw strong conclusions. Despite that, we applied the Wilcoxon test to the beginners and advanced data sets separately. The results were the same as the results for the entire data sets, for both tools, in all sessions. We did not present this as a formal result due to sample sizes; nevertheless, it seems that any possible disturbances due to heterogeneity of subjects were not enough to invalidate the found results.

7. THESIS SUMMARY AND FUTURE WORK

In this thesis, we have presented, defined the requirements and developed a Software Product Line of Model-based Testing tools. Thus, along this thesis we showed that the adoption of SPL has increased in the past years since several companies report successful cases and that SPL has become a well-known and wide-applied approach to promote reuse, minimize time-to-market and cost of software systems. We also discussed the advantages and the relevance of Model-based testing as an alternative to automate software testing activities. Furthermore, based on a Systematic Mapping Study results we have identified that there are several tools to support MBT process and in most cases these tools share a common set of features. However, to the best of our knowledge, there is no academic or commercial work proposing a Software Product Line to derive Model-based Testing tools. Based on this, we identified SPL requirements, defined the design decision and developed the artifacts of a SPL of MBT tools: the PLeTs SPL. Furthermore, we also discussed in this thesis the requirements, design decisions and the development of an environment to support the automatic generation of products for PLeTs. Moreover, we described two examples of use, where we generated two MBT tools to apply performance testing and two MBT tools to apply structural testing. Finally, we presented an experimental study carried out to evaluate the effort to use MBT tools derived from PLeTs to generate test scripts and scenarios in the context of performance testing activities. The examples of use results pointed out that the number of lines of code required to develop a MBT tool is reduced when the tool is generated from PLeTs, which means that less effort is required to develop a tool. Furthermore, the experiment results show that in some situations the use of an MBT tool generated from PLeTs requires less effort to test an application than when using a Capture and Replay based tool.

7.1 Thesis Contributions

As already stated, the contributions of this thesis can be summarized as:

- Propose, design and develop a **Product Line of Model-based Testing tools: (PLeTs)**. The lesson learned from two examples of use and from an empirical experiment pointed out that deriving a set of related MBT tools from a SPL requires less effort than developing these tools from scratch. Furthermore, our experimental study indicates that the use of a performance MBT tool derived from PLeTs requires less effort than a Capture and Replay based tool: LoadRunner. Moreover, in the context of a collaboration project, some generated MBT tools are currently under the analysis of the TDL and if they are approved, these tools will be adopted by the Dell company as one of their standard testing tool.
- Development of an environment to support the SPL life-cycle. SPL adoption has many challenges in industrial settings. One particular challenge regards the use of *off-the-shelf*

tools to support the adoption of SPLs because in some scenarios these tools do not address some company's specific needs. Along the thesis we presented this specific requirements and our design decisions in the development of our environment. These decisions, in turn, can be re-applied in different settings sharing similar requirements. Furthermore, we have been working to make the environment more flexible and capable to support different programming language and modelling notations. An initial version of the environment source code and its documentation are available in the PlugSPL website [CePb].

- An approach to support the integration of the tools derived from PLeTs with commercial, open-source or academic testing tools. The reuse of these tools presents several benefits, such as, reduce the effort and the cost during the development of SPL core assets and less investment in training while the tool is used by testing teams. Currently our SPL supports the generation of scripts for five testing tools, three for performance testing and two for structural testing. It is important to note that we are already working, in collaboration with the TDL, to support the generation of others MBT testing tools, such as, the generation of MBT tools to apply functional testing.
- We presented the use of the MBT tools generated from PLeTs, in an industrial setting, to perform two examples of use. The main objective of these examples of use was to apply the testing tools generated from PLeTs to generate test scripts to test applications. The results showed that the generation of these tools from a common set of artifacts requires less effort than developing these tools in an isolated manner. Furthermore, we also performed an experiment, selecting students and performance tester from industry as subjects. This experiment focused on the understanding of the needed effort to generate performance scripts using a tool generated from PLeTs and to compare it with the effort when using an industrial testing tool: LoadRunner. The results showed that in complex scenarios the use of our tool requires less effort than using LoadRunner.
- Two approaches to generate scripts from UML diagrams. As the requirements to generate the tools are based on an industrial setting, and there is little investigation on how MBT can be applied in performance testing, we proposed an approach that uses UML diagrams as an input to generate the performance test scripts. Furthermore, we also proposed an approach to generate structural test cases and than instantiated in some structural testing tools, e.g., JaBUTi [VMWD05], EMMA [Rou].

7.2 Limitations and Future Works

Although this thesis presented real contributions to the software testing and software product line research fields, we identified the following limitations and opportunities of future works:

- Component-based variability was useful and facilitated the PLeTs development and adoption, but at the same time it was a limitation in situations when we want to take advantage of fine-grained reuse. Because of this, we cannot reuse common classes and methods, which could impact the effort to implement a new feature. However, to implement fine-grained features requires more effort on mapping features to source code and in some cases it can obfuscate de code readability. To mitigate this limitation, we already defined a requirement to the next version of PLeTs: PLeTs/PlugSPL must support a fine-grained mechanism, *e.g.*, preprocessor directives, in conjunction with the actual component replacement mechanism.
- Although we performed two examples of use and an empirical experiment to validate our results, we are aware that we have to better investigate the benefits presented by our SPL and by the MBT tools derived from PLeTs. For instance the heterogeneity of our experiment subjects and the size of the sample could threat the validity of the experiment. Furthermore, the examples of use are performed in accordance with the requirements of a specific IT company, which could not represent another company settings. Because of that, we are already planning the replication of the experiment with another Capture and Replay tool, and with a large sample of subjects from an IT company. It is important to note that this IT company demonstrated high interest in our initial results, then they encouraged their employees to participate in the experiment.
- The fact that we do not use a software product line testing approach to test the components and tools generated from PLeTs can imposed some quality issues to the generated tools and can lead to increase the testing cost and effort. Thus, the absence of use an approach to test our SPL lead us to spend more time on testing each PLeTs component and also testing each derived tool. Therefore, a Ph.D student from our research group is currently investigating an approach to generate test cases to test products generated from a software product line.

7.3 Publications

During the development of this thesis we presented and discussed our research results in the following papers:

- Rodrigues, E. M.; Viccari, L. D.; Zorzo, A. F. “PLeTs-Test Automation using Software Product Lines and Model Based Testing”. In: Proceedings of the 22th International Conference on Software Engineering and Knowledge Engineering, 2010, San Francisco, EUA.
- Silveira, M. B.; Rodrigues, E. M.; Zorzo, A. F.; Vieira, H.; Oliveira, F. “Model-Based Automatic Generation of Performance Test Scripts”. In: Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering, 2011 Miami, EUA.
- Costa, L. T.; Czekster, R. M.; de Oliveira, F. M.; de M. Rodrigues, E.; da Silveira, M. B.; Zorzo, A. F. “Generating performance test scripts and scenarios based on abstract intermediate

models". In: Proceedings of the 24rd International Conference on Software Engineering and Knowledge Engineering, 2012, San Francisco, EUA.

- Costa, L. T.; Oliveira, F. M.; Rodrigues, E. M. ; Silveira, M.; Zorzo, A. F. Uma Abordagem para Geração de Casos de Teste Estrutural Baseada em Modelos. In: Workshop de Teste e Tolerância a Falhas, 2012, Ouro Preto - MG. WTF 2012, 2012.
- Rodrigues, E. M.; Zorzo, A. F.; Oliveira, E. A.; Gimenes, I. M.; Maldonado, J. C.; Domingues, A. R. P. PlugSPL: An Automated Environment for Supporting Plugin-based Software Product Lines. In: 24th International Conference on Software Engineering and Knowledge Engineering, 2012, San Francisco, EUA.

Under Review

- Rodrigues, E. M.; Zorzo, A. F.; Nakagawa, E. Y.; Gimenes, I. M.; Maldonado, J. C.; "A Software Product Line for Model-Based Testing Tools", Journal of Universal Computer Science, **Under review**.
- Rodrigues, E. M.; Oliveira, F. M.; Costa, L. T.; Silveira, M. B.; Souza, S. R. S.; Saad, R.; Zorzo, A. F.; Model-based Testing applied to Performance Testing: An Empirical Study, Journal of Empirical Software Engineer, **Under review**.

To be submitted

- Costa, L. T.; Oliveira, F. M.; Rodrigues, E. M. ; Silveira, M.; Zorzo, A. F. Structural Test Case Generation Based on System Models, **To be submitted to ACM Symposium on Applied Computing**.
- Rodrigues, E. M.; Domingues, A. R. P.; Zorzo, A. F.; Oliveira, F. M.; Czarnecki, K.; Passos, L. PlugSPL: A Tool Supporting Component-Based Software Product Lines, **To be submitted to SPL 2014**.
- Rodrigues, E. M.; Zorzo, A. F.; Oliveira, F. M.; Czarnecki, K.; Passos, L. On the Requirements and Design Decisions of an In-House Component-Based SPL Automated Environment, **To be submitted to SPL 2014**.

BIBLIOGRAPHY

- [AB06] Andaloussi, B. S.; Braun, A. "A Test Specification Method for Software Interoperability Tests in Offshore Scenarios: A Case Study". In: Proceedings of the IEEE International Conference on Global Software Engineering, 2006, pp. 169–178.
- [ABT10] Abbors, F.; Backlund, A.; Truscan, D. "MATERA - An Integrated Framework for Model-based Testing". In: Proceedings of the 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2010, pp. 321–328.
- [ADM06] Allen, W.; Dou, C.; Marin, G. "A Model-based Approach to the Security Testing of Network Protocol Implementations". In: Proceedings of the 31st IEEE Conference on Local Computer Networks, 2006, pp. 1008–1015.
- [ALRL04] Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C. "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE Transaction on Dependable Secure Computing*, vol. 1, Jan-Mar 2004, pp. 11–33.
- [AO08] Ammann, P.; Offutt, J. "Introduction to Software Testing". Cambridge University Press, 2008, 344p.
- [Apa] Apache. "JMeter Performance Test". Available in: <http://activemq.apache.org/jmeter-performance-tests.html>, July 2013.
- [APUW09] Aydal, E.; Paige, R.; Utting, M.; Woodcock, J. "Putting Formal Specifications under the Magnifying Glass: Model-based Testing for Validation". In: Proceedings of the 2nd of the International Conference on Software Testing Verification and Validation, 2009, pp. 131–140.
- [AS10] Athira, B.; Samuel, P. "Web Services Regression Test Case Prioritization". In: International Conference on Computer Information Systems and Industrial Management Applications, 2010, pp. 438–443.
- [BBM02] Basanieri, F.; Bertolino, A.; Marchetti, E. "The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects". In: *UML 2002 — The Unified Modeling Language*, Jezequel, J.-M.; Hussmann, H.; Cook, S. (Editors), Springer Berlin Heidelberg, 2002, pp. 383–397.
- [BCK97] Bass, L.; Clements, P.; Kazman, R. "Software Architecture in Practice". Addison-Wesley Longman Publishing Co., Inc., 1997, 452p.
- [BDLRM09] Brito, P. H. S.; De Lemos, R.; Rubira, C. M. F.; Martins, E. "Architecting Fault Tolerance with Exception Handling: Verification and Validation", *Journal of Computer Science and Technology*, vol. 24, Mar 2009, pp. 212–237.

- [Ben08] Benz, S. "AspectT: Aspect-oriented Test Case Instantiation". In: Proceedings of the 7th international conference on Aspect-oriented software development, 2008, pp. 1–12.
- [Ber] Bertolino, A. "Knowledge Area Description of Software Testing SWEBOK". Available in: <http://www.computer.org/portal/web/swebok>, April 2013.
- [Ber07] Bertolino, A. "Software Testing Research: Achievements, Challenges, Dreams". In: 2007 Future of Software Engineering, 2007, pp. 85–103.
- [BGN⁺04] Barnett, M.; Grieskamp, W.; Nachmanson, L.; Schulte, W.; Tillmann, N.; Veanes, M. "Towards a Tool Environment for Model-Based Testing with AsmL". In: *Formal Approaches to Software Testing*, Petrenko, A.; Ulrich, A. (Editors), Springer Berlin Heidelberg, 2004, pp. 252–266.
- [BK08] Bringmann, E.; Krämer, A. "Model-Based Testing of Automotive Systems". In: Proceedings of the 1st 2008 International Conference on Software Testing, Verification, and Validation, 2008, pp. 485–493.
- [BLG11] Barna, C.; Litoiu, M.; Ghanbari, H. "Model-based Performance Testing". In: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 872–875.
- [BM] Bernardi, S.; Merseguer, J. "A UML Profile for Dependability Analysis of Real-time Embedded Systems, year = 2007". In: Proceedings of the 6th international workshop on Software and performance, pp. 115–124.
- [Bob08] Boberg, J. "Early Fault Detection with Model-based Testing". In: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, 2008, pp. 9–20.
- [Bur03] Burnstein, I. "Practical Software Testing: A Process-Oriented Approach". Springer, 2003, 709p.
- [CAM10] Cristiá, M. and, M.; Albertengo, P.; Monetti, P. "Pruning Testing Trees in the Test Template Framework by Detecting Mathematical Contradictions". In: Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods, 2010, pp. 268–277.
- [CC79] Cook, T. D.; Campbell, D. T. "Quasi-experimentation: Design and Analysis Issues for Field Settings". Houghton Mifflin, 1979, 405p.
- [CCO⁺12] Costa, L. T.; Czekster, R. M.; Oliveira, F. M.; Rodrigues, E. M.; da Silveira, M. B.; Zorzo, A. F. "Generating Performance Test Scripts and Scenarios Based on Abstract Intermediate Models". In: Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering, 2012, pp. 112–117.
- [CePa] CePES/PUCRS. "PLeTs SPL". Available in: <http://www.cepes.pucrs.br/plets/>, May 2013.

- [CePb] CePES/PUCRS. “PlugSPL SPL”. Available in: <http://www.cepes.pucrs.br/plugspl/>, May 2013.
- [CGR⁺12] Czarnecki, K.; Grünbacher, P.; Rabiser, R.; Schmid, K.; Wąsowski, A. “Cool Features and Tough Decisions: a Comparison of Variability Modeling Approaches”. In: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, 2012, pp. 173–182.
- [Cha91] Chaim, M. L. “POKE-TOOL: A Tool to Support Data Flow Based Structural Test of Programs”, Master’s Thesis, DCA/FEEC/UNICAMP, São Paulo, Brazil, 1991, 176p.
- [CHE05] Czarnecki, K.; Helsen, S.; Eisenecker, U. “Formalizing Cardinality-based Feature Models and their Specialization”, *Software Process: Improvement and Practice*, vol. 10, Nov 2005, pp. 7–29.
- [CLS⁺09] Chinnapongse, V.; Lee, I.; Sokolsky, O.; Wang, S.; Jones, P. L. “Model-Based Testing of GUI-Driven Applications”. In: Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems, 2009, pp. 203–214.
- [CN01] Clements, P.; Northrop, L. “Software Product Lines: Practices and Patterns”. Addison-Wesley Longman Publishing Co., Inc., 2001, 608p.
- [CNM07] Cartaxo, E.; Neto, F.; Machado, P. “Test Case Generation by Means of UML Sequence Diagrams and Labeled Transition Systems”. In: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, 2007, pp. 1292–1297.
- [Com] Committee, I. C. S. S. E. T. “IEEE Standard Glossary of Software Engineering Terminology, year = 1983”. Institute of Electrical and Electronics Engineers, 1-84p.
- [CORS12] Costa, L. T.; Oliveira, F. M.; Rodrigues, E. M.; Silveira, Maicon Bernardino; Zorzo, A. F. “Uma Abordagem para Geração de Casos de Teste Estrutural Baseada em Modelos”. In: Proceedings of Workshop de Teste e Tolerância a Falhas, 2012, pp. 87–100.
- [Cos12] Costa, L. T. “Conjunto de Características para Teste de Desempenho: Uma Visão a Partir de Ferramentas”, Master’s Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil, 2012, 113p.
- [DDO08] Dijkman, R. M.; Dumas, M.; Ouyang, C. “Semantics and Analysis of Business Process Models in BPMN”, *Information and Software Technology*, vol. 50, Nov 2008, pp. 1281–1294.
- [DEFM⁺10] Dorofeeva, R.; El-Fakih, K.; Maag, S.; Cavalli, A. R.; Yevtushenko, N. “FSM-based Conformance Testing Methods: A survey Annotated with Experimental Evaluation”, *Information and Software Technology*, vol. 52, Dec 2010, pp. 1286–1297.

- [DGRM06] Debnath, N.; Garis, A.; Riesco, D.; Montejano, G. "Defining Patterns Using UML Profiles". In: IEEE International Conference on Computer Systems and Applications, 2006, pp. 1147–1150.
- [DLL⁺09] Dara, R.; Li, S.; Liu, W.; Smith-Ghorbani, A.; Tahvildari, L. "Using Dynamic Execution Data to Generate Test Cases". In: Proceedings of the IEEE International Conference on Software Maintenance, 2009, pp. 433–436.
- [DLS78] DeMillo, R.; Lipton, R.; Sayward, F. "Hints on Test Data Selection: Help for the Practicing Programmer", *Computer*, vol. 1, Apr 1978, pp. 34–41.
- [DMJ07] Delamaro, M. E.; Maldonado, J. C.; Jino, M. "Introdução ao Teste de Software". Campus, 2007, 408p.
- [DNSVT07] Dias Neto, A. C.; Subramanyan, R.; Vieira, M.; Travassos, G. H. "A Survey on Model-based Testing Approaches: A Systematic Review". In: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies, 2007, pp. 31–36.
- [DNT09] Dias Neto, A. C.; Travassos, G. H. "Model-based Testing Approaches Selection for Software Projects", *Information and Software Technology*, vol. 51, Nov 2009, pp. 1487–1504.
- [DNTSV07] Dias Neto, A. C.; Travassos, G.; Subramanyan, R.; Vieira, M. "Characterization of Model-based Software Testing Approaches", Technical Report, Technical Report ES-713/07, PESC-COPPE/UFRJ, 2007, 114p.
- [DTA⁺08] Demathieu, S.; Thomas, F.; André, C.; Gérard, S.; Terrier, F. "First Experiments Using the UML Profile for MARTE". In: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, 2008, pp. 50–57.
- [EAXD⁺10] El Ariss, O.; Xu, D.; Dandey, S.; Vender, B.; McClean, P.; Slator, B. "A Systematic Capture and Replay Strategy for Testing Complex GUI Based Java Applications". In: Proceedings of the 7th International Conference on Information Technology: New Generations, 2010, pp. 1038–1043.
- [EFW01] El-Far, I. K.; Whittaker, J. A. "Model-based Software Testing". In: *Encyclopedia of Software Engineering*, Marciniak, J. (Editor), Wiley, 2001, pp. 825–837.
- [ER11] Engström, E.; Runeson, P. "Software Product Line Testing - A Systematic Mapping Study", *Information and Software Technology*, vol. 53, Jan 2011, pp. 2–13.
- [ERM07] Everett, G.; Raymond McLeod, J. "Software Testing: Testing Across the Entire Software Development Life Cycle". John Wiley & Sons, 2007, 280p.

- [Esl08] Eslamimehr, M. M. "The Survey of Model-based Testing and Industrial Tools", Master's Thesis, Linköping University, Department of Computer and Information Science, Linköping, Sweden, 2008, 65p.
- [FIMR10] Farooq, Q.; Iqbal, M.; Malik, Z.; Riebisch, M. "A Model-Based Regression Testing Approach for Evolving Software Systems with Flexible Tool Support". In: Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems, 2010, pp. 41–49.
- [FL09] Farooq, Q.; Lam, C. P. "Evolving the Quality of a Model-based Test Suite". In: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops, 2009, pp. 141–149.
- [FLG10] Feliachi, A.; Le Guen, H. "Generating Transition Probabilities for Automatic Model-Based Test Generation". In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation, 2010, pp. 99–102.
- [GFA98] Griss, M. L.; Favaro, J.; Alessandro, M. d. "Integrating Feature Modeling with the RSEB". In: Proceedings of the 5th International Conference on Software Reuse, 1998, pp. 76–86.
- [GFF+10] Grasso, D.; Fantechi, A.; Ferrari, A.; Becheri, C.; Bacherini, S. "Model-based Testing and Abstract Interpretation in the Railway Signaling Context". In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation, 2010, pp. 103–106.
- [GHV07] Gonczy, L.; Heckel, R.; Varro, D. "Model-based Testing of Service Infrastructure Components". In: Proceedings of the 19th IFIP TC6/WG6.1 International Conference, 7th International Workshop Testing of Software and Communicating Systems, 2007, pp. 155–170.
- [Gro10] Groenda, H. "Usage Profile and Platform Independent Automated Validation of Service Behavior Specifications". In: Proceedings of the 2nd International Workshop on the Quality of Service-Oriented Software Systems, 2010, pp. 6:1–6:6.
- [Har00] Harrold, M. J. "Testing: A Roadmap". In: Proceedings of the Conference on The Future of Software Engineering, 2000, pp. 381.
- [HBAA10] Hemmati, H.; Briand, L.; Arcuri, A.; Ali, S. "An Enhanced Test Case Selection Approach for Model-based Testing: An Industrial Case Study". In: Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering, 2010, pp. 267–276.
- [Hew] Hewlett Packard - HP. "Software HP LoadRunner". Available in: <https://h10078.www1.hp.com/cda/hpms/>, April 2013.

- [HGB08] Hasling, B.; Goetz, H.; Beetz, K. "Model-based Testing of System Requirements using UML Use Case Models". In: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, 2008, pp. 367–376.
- [HJK10] Heiskanen, H.; Jääskeläinen, A.; Katara, M. "Debug Support for Model-based GUI Testing". In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, 2010, pp. 25–34.
- [HN04] Hartman, A.; Nagin, K. "The AGEDIS Tools for Model Based Testing", *SIGSOFT Software Engineering Notes*, vol. 29, Jul 2004, pp. 129–132.
- [Hui07] Huima, A. "Implementing Conformiq Qtronic". In: *Testing of Software and Communicating Systems*, Petrenko, A.; Veanes, M.; Tretmans, J.; Grieskamp, W. (Editors), Springer Berlin Heidelberg, 2007, pp. 1–12.
- [IAB10] Iqbal, M.; Arcuri, A.; Briand, L. "Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies". In: *Model Driven Engineering Languages and Systems*, Petriu, D.; Rouquette, N.; Haugen, S. (Editors), Springer Berlin / Heidelberg, 2010, pp. 286–300.
- [IBM] IBM. "IBM Rational PurifyPlus". Available in: <http://www.ibm.com/software/awdtools/purifyplus/>, April 2013.
- [IEE] IEEE. "IEEE Standard for Software and System Test Documentation". Available in: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4578383>, July 2013.
- [JG07] Jeffrey, D.; Gupta, N. "Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction", *IEEE Transactions on Software Engineering*, vol. 33, Feb 2007, pp. 108–123.
- [JTG⁺10] Jiang, B.; Tse, T. H.; Grieskamp, W.; Kicillof, N.; Cao, Y.; Li, X. "Regression Testing Process Improvement for Specification Evolution of Real-World Protocol Software". In: Proceedings of the 2010 10th International Conference on Quality Software, 2010, pp. 62–71.
- [KAK08] Kästner, C.; Apel, S.; Kuhlemann, M. "Granularity in Software Product Lines". In: Proceedings of the 30th international conference on Software engineering, 2008, pp. 311–320.
- [KBB] Kitchenham, B. A.; Budgen, D.; Brereton, P. "Using Mapping Studies as the Basis for Further Research – A Participant-observer Case Study, volume = 53, year = 2011, month = Jun", *Information and Software Technology*, pp. 638 – 651.

- [KC07] Kitchenham, B.; Charters, S. "Guidelines for Performing Systematic Literature Reviews in Software Engineering", Technical Report, Keele University and Durham University Joint Report, 2007, 12p.
- [KCH⁺90] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S. "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, Carnegie-Mellon University Software Engineering Institute, 1990, 148p.
- [KKL⁺98] Kang, K. C.; Kim, S.; Lee, J.; Kim, K.; Shin, E.; Huh, M. "FORM: A Feature-oriented Reuse Method with Domain-specific Reference Architectures", *Annals of Software Engineering*, vol. 5, Jan 1998, pp. 143–168.
- [KKP06] Kandl, S.; Kirner, R.; Puschner, P. "Development of a Framework for Automated Systematic Testing of Safety-Critical Embedded Systems". In: International Workshop on Intelligent Solutions in Embedded Systems '06, 2006, pp. 1–13.
- [KMPK06] Kervinen, A.; Maunumaa, M.; Pääkkönen, T.; Katara, M. "Model-based Testing Through a GUI". In: Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software, 2006, pp. 16–31.
- [Kor90] Korel, B. "Automated Software Test Data Generation", *IEEE Transactions on Software Engineering*, vol. 16, Aug 1990, pp. 870–879.
- [Kru02] Krueger, C. W. "Easing the Transition to Software Mass Customization". In: 4th International Workshop on Software Product-Family Engineering, 2002, pp. 282–293.
- [LCYW11] Lin, M.; Chen, Y.; Yu, K.; Wu, G. "Lazy Symbolic Execution for Test Data Generation", *IET Software*, vol. 5, Mar 2011, pp. 132–141.
- [LG10] Lochau, M.; Goltz, U. "Feature Interaction Aware Test Case Generation for Embedded Control Systems", *Electronic Notes in Theoretical Computer Science*, vol. 264, Dec 2010, pp. 37–52.
- [LHH07] Li, Z.; Harman, M.; Hierons, R. M. "Search Algorithms for Regression Test Case Prioritization", *IEEE Transactions on Software Engineering*, vol. 33, Apr 2007, pp. 225–237.
- [LKL12] Lee, J.; Kang, S.; Lee, D. "A survey on Software Product Line Testing". In: Proceedings of the 16th International Software Product Line Conference - Volume 1, 2012, pp. 31–40.
- [LMG10] Löffler, R.; Meyer, M.; Gottschalk, M. "Formal Scenario-based Requirements Specification and Test Case Generation in Healthcare Applications". In: Proceedings of the Workshop on Software Engineering in Health Care, 2010, pp. 57–67.

- [LSR07] Linden, F. J.; Schmid, K.; Rommes, E. "Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering". Springer-Verlag New York, Inc., 2007, 333p.
- [MBC09] Mendonca, M.; Branco, M.; Cowan, D. "S.P.L.O.T.: Software Product Lines Online Tools". In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems, 2009, pp. 761–762.
- [Mem07] Memon, A. M. "An Event-flow Model of GUI-based Applications for Testing: Research Articles", *Software Testing, Verification & Reliability*, vol. 17, Sep 2007, pp. 137–157.
- [Men02] Menasce, D. "TPC-W: a Benchmark for E-commerce", *Internet Computing, IEEE*, vol. 6, May-Jun 2002, pp. 83–87.
- [MOSHL09] Mussa, M.; Ouchani, S.; Sammane, W. A.; Hamou-Lhadj, A. "A Survey of Model-Driven Testing Techniques". In: Proceedings of the 9th International Conference on Quality Software, 2009, pp. 167–172.
- [MS04] Myers, G. J.; Sandler, C. "The Art of Software Testing". John Wiley & Sons, 2004, 256p.
- [MSK⁺07] Mathaikutty, D. A.; Shukla, S. K.; Kodakara, S. V.; Lilja, D.; Dingankar, A. "Design Fault Directed Test Generation for Microprocessor Validation". In: Proceedings of the Conference on Design, Automation and Test in Europe, 2007, pp. 761–766.
- [Nor02] Northrop, L. M. "SEI's Software Product Line Tenets", *IEEE Software*, vol. 19, Jul-Aug 2002, pp. 32–40.
- [NSS10] Nguyen, D. H.; Strooper, P.; Suess, J. G. "Model-based Testing of Multiple GUI Variants Using the GUI Test Generator". In: Proceedings of the 5th Workshop on Automation of Software Test, 2010, pp. 24–30.
- [NT11] Naik, S.; Tripathy, P. "Software Testing and Quality Assurance: Theory and Practice". Wiley, 2011, 648p.
- [NZR10] Naslavsky, L.; Ziv, H.; Richardson, D. J. "MbSRT2: Model-based Selective Regression Testing with Traceability". In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, 2010, pp. 89–98.
- [OG05] Olimpiew, E. M.; Gomaa, H. "Model-based Testing for Applications Derived from Software Product Lines". In: Proceedings of the 1st international workshop on Advances in model-based testing, 2005, pp. 1–7.
- [OGM10] Oliveira, E. A.; Gimenes, I. M. S.; Maldonado, J. C. "Systematic Management of Variability in UML-based Software Product Lines", *Journal of Universal Computer Science*, vol. 16, Aug-Sep 2010, pp. 2374–2393.

- [Oli08] Olimpiew, E. M. "Model-based Testing for Software Product Lines", Ph.D. Thesis, George Mason University, Washington, USA, 2008, 296p.
- [OMGa] OMG. "UML Profile for Schedulability, Performance, and Time Specification - OMG Adopted Specification Version 1.1". Available in: <http://www.omg.org/spec/SPTP/1.1/>, May 2013.
- [OMGb] OMG. "Unified Modeling Language - UML". Available in: <http://www.uml.org/>, April 2013.
- [OMG12] OMG. "UML Testing Profile (UTP) - Version 1.1", Technical Report, Object Management Group, 2012, 104p.
- [Par06] Paradkar, A. "A Quest for Appropriate Software Fault Models: Case Studies on Fault Detection Effectiveness of Model-based Test Generation Techniques", *Information and Software Technology*, vol. 48, Oct 2006, pp. 949–959.
- [PBL05] Pohl, K.; Böckle, G.; Linden, F. J. v. d. "Software Product Line Engineering: Foundations, Principles and Techniques". Springer-Verlag New York, Inc., 2005, 494p.
- [PFMM08] Petersen, K.; Feldt, R.; Mujtaba, S.; Mattsson, M. "Systematic Mapping Studies in Software Engineering", *12th International Conference on Evaluation and Assessment in Software Engineering*, vol. 17, Jun 2008, pp. 1–10.
- [PG06] Perez, J.; Guckenheimer, S. "Software Engineering with Microsoft Visual Studio Team System". Pearson Education, 2006, 304p.
- [PMBF05] Pelliccione, P.; Muccini, H.; Bucchiarone, A.; Facchini, F. "TeStor: Deriving Test Sequences from Model-Based Specifications". In: *Component-Based Software Engineering*, Heineman, G.; Crnkovic, I.; Schmidt, H.; Stafford, J.; Szyperski, C.; Wallnau, K. (Editors), Springer Berlin Heidelberg, 2005, pp. 267–282.
- [Por] Portal Action. "System Action Statistical Package". Available in: <http://www.portalaction.com.br/en>, April 2013.
- [Puo08] Puolitaival, O.-P. "Adapting Model-based Testing to Agile Context", *VTT Publications*, vol. 694, Sep 2008, pp. 1–80.
- [R P] R Project. "The R Project for Statistical Computing". Available in: <http://www.r-project.org/>, May 2013.
- [Rou] Roubtsov, V. "EMMA: a Free Java Code Coverage Tool". Available in: <http://emma.sourceforge.net>, April 2013.
- [RVZ10] Rodrigues, E. M.; Viccari, L. D.; Zorzo, A. F. "PLeTs - Test Automation using Software Product Lines and Model Based Testing". In: *Proceedings of the 22th International Conference on Software Engineering and Knowledge Engineering*, 2010, pp. 483–488.

- [RW85] Rapps, S.; Weyuker, E. J. "Selecting Software Test Data Using Data Flow Information", *Transactions on Software Engineering*, vol. 11, Apr 1985, pp. 367–375.
- [SD10] Saifan, A.; Dingel, J. "A Survey of Using Model-Based Testing to Improve Quality Attributes in Distributed Systems". In: *Advanced Techniques in Computing Sciences and Software Engineering*, Elleithy, K. (Editor), Springer Netherlands, 2010, pp. 283–288.
- [SEI] SEI. "Software Engineering Institute - Case Studies". Available in: <http://www.sei.cmu.edu/productlines/casestudies/>, April 2013.
- [Sem] Semantic Designs. "Semantic Designs Test Coverage". Available in: <http://www.semdesigns.com>, April 2013.
- [SHH07] Schulz, S.; Honkola, J.; Huima, A. "Towards Model-Based Testing with Architecture Models". In: 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2007, pp. 495–502.
- [SHTB07] Schobbens, P.-Y.; Heymans, P.; Trigaux, J.-C.; Bontemps, Y. "Generic Semantics of Feature Diagrams", *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 51, Feb 2007, pp. 456–479.
- [Sil12] Silveira, M. B. "Conjunto de características para teste de desempenho : uma visão a partir de modelos", Master's Thesis, Pontificia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil, 2012, 105p.
- [SL10] Shafique, M.; Labiche, Y. "A Systematic Review of Model-based Testing Tool Support", Technical Report, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 2010, 21p.
- [SMJU10] Sarma, M.; Murthy, P. V. R.; Jell, S.; Ulrich, A. "Model-based Testing in Industry: A Case Study with Two MBT Tools". In: Proceedings of the 5th Workshop on Automation of Software Test, 2010, pp. 87–90.
- [Som11] Sommerville, I. "Software Engineering". Pearson/Addison–Wesley, 2011, 792p.
- [SRZ⁺11] Silveira, M. B.; Rodrigues, E. M.; Zorzo, A. F.; Vieira, H.; Oliveira, F. "Model-based Automatic Generation of Performance Test Scripts". In: Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering, 2011, pp. 1–6.
- [SWK09] Stefanescu, A.; Wieczorek, S.; Kirshin, A. "MBT4Chor: A Model-based Testing Approach for Service Choreographies". In: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, 2009, pp. 313–324.

- [SWW10] Stefanescu, A.; Wieczorek, S.; Wendland, M.-F. "Using the UML Testing Profile for Enterprise Service Choreographies". In: Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications, 2010, pp. 12–19.
- [TAR] TARGET. "TaRGeT Product Line". Available in: <http://twiki.cin.ufpe.br/twiki/bin/view/TestProductLines/TaRGeTProductLine>, July 2013.
- [TKB⁺12] Thüm, T.; Kästner, C.; Benduhn, F.; Meinicke, J.; Saake, G.; Leich, T. "FeatureIDE: An Extensible Framework for Feature-oriented Software Development", *Science of Computer Programming*, vol. 1, Jun 2012, pp. 1–16.
- [TKES11] Thum, T.; Kastner, C.; Erdweg, S.; Siegmund, N. "Abstract Features in Feature Modeling". In: Proceedings of the 2011 15th International Software Product Line Conference, 2011, pp. 191–200.
- [UL06] Utting, M.; Legeard, B. "Practical Model-Based Testing: A Tools Approach". Morgan Kaufmann, 2006, 456p.
- [UPL12] Utting, M.; Pretschner, A.; Legeard, B. "A Taxonomy of Model-based Testing Approaches", *Software Testing, Verification and Reliability*, vol. 22, Aug 2012, pp. 297–312.
- [VCG⁺08] Veanes, M.; Campbell, C.; Grieskamp, W.; Schulte, W.; Tillmann, N.; Nachmanson, L. "Model-based Testing of Object-oriented Reactive Systems with Spec Explorer". In: *Formal methods and testing*, Hierons, R. M.; Bowen, J. P.; Harman, M. (Editors), Springer-Verlag, 2008, pp. 39–76.
- [VMWD05] Vincenzi, A.; Maldonado, J.; Wong, W.; Delamaro, M. "Coverage Testing of Java Programs and Components", *Science of Computer Programming*, vol. 56, Apr 2005, pp. 211–230.
- [VSM⁺08] Vieira, M.; Song, X.; Matos, G.; Storck, S.; Tanikella, R.; Hasling, B. "Applying Model-Based Testing to Healthcare Products: Preliminary Experiences". In: ACM/IEEE 30th International Conference on Software Engineering, 2008, pp. 669–672.
- [WL99] Weiss, D. M.; Lai, C. T. R. "Software Product-line Engineering: A Family-based Software Development Process". Addison-Wesley Longman Publishing Co., Inc., 1999, 426p.
- [WRH⁺00] Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M. C.; Regnell, B.; Wesslén, A. "Experimentation in Software Engineering: An Introduction". Kluwer Academic Publishers, 2000, 259p.
- [YLS12] Yuan, Y.; Li, Z.; Sun, W. "A Graph-Search Based Approach to BPEL4WS Test Generation". In: International Conference on Software Engineering Advances, 2012, pp. 1–9.

- [YX10] Yu, G.; Xu, Z. W. "Model-based Safety Test Automation of Safety-Critical Software". In: International Conference on Computational Intelligence and Software Engineering, 2010, pp. 1–3.
- [Zha06] Zhang, M. "ArgoUML", *Journal of Computing Sciences in Colleges*, vol. 21, Feb-Jun 2006, pp. 6–7.
- [ZLLW09] Zeng, F.; Li, L.; Li, J.; Wang, X. "Vulnerability Testing of Software Using Extended EAI Model". In: Software Engineering, 2009. WCSE '09. WRI World Congress on, 2009, pp. 261–265.
- [ZSH09] Zhao, H.; Sun, J.; Hu, G. "Study of Methodology of Testing Mobile Games Based on TTCN-3". In: Proceedings of the 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009, pp. 579–584.