PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

LEANDRO TEODORO COSTA

SPLIT-MBT: A MODEL-BASED TESTING METHOD FOR SOFTWARE PRODUCT LINES

Porto Alegre
2017

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
GRADUATE PROGRAM IN COMPUTER SCIENCE

SPLIT-MBT: A MODEL-BASED TESTING METHOD FOR SOFTWARE PRODUCT LINES

LEANDRO TEODORO COSTA

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

Porto Alegre
2017

Ficha Catalográfica

C837s Costa, Leandro Teodoro

SPLiT-MBt : A Model-based Testing Method for Software Poduct Lines / Leandro Teodoro Costa . – 2017.
194.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Avelino Francisco Zorzo.

1. Software Product Line. 2. Software Product Line Testing. 3. Functional Testing. 4. Model-based Testing. 5. Test Case Generation Method. I. Zorzo, Avelino Francisco. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Salete Maria Sartori CRB-10/1363

Leandro Teodoro Costa

SPLiT-MBt: A Model-based Testing Method for Software Product Lines

This Thesis has been submitted in partial fulfillment of the requirements for the degree of Doctor of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on August 22th, 2017.

COMMITTEE MEMBERS:

Prof. Dr. Edson Alves de Oliveira Junior (UEM)

Prof. Dr. Fabian Luis Vargas (PPGEE/PUCRS)

Prof. Rafael Prikladnicki (PPGCC/PUCRS)

Prof. Dr. Avelino Francisco Zorzo (PPGCC/PUCRS - Advisor)

To God, Holy Mary,
my family and friends
who supported me and
have been on my side
whenever I needed.

"*Do you wish to rise? Begin by descending. You plan a tower that will pierce the clouds? Lay first the foundation of humility.*"
(St. Augustine)

ACKNOWLEDGMENTS

I thank God for supporting me at all times during these five years. I am aware that without Your help I would not have been able to reach my objectives. Thank you Lord Jesus, I know you have been so patient with me. I promise that I will repay You for all the graces You have poured into my life. I am also thank Virgin Holy Blessed Mary, Mother of God, by your glorious intercession.

A special thank to Prof. Dr. Avelino Francisco Zorzo for believing in my work, for his all supporting and patience during the moments that I thought I would fall. Thank you for the advice and teachings, for being my supervisor, my mentor and my friend in the last 10 years.

I thank Dell for the financial support provided me. I also thank the opportunity to develop my research as a member of the CePES, a project from the Development and Information Technology Program (PDTI) related to the DELL/PUCRS partnership.

I would like to thank my colleagues and friends from CePES: Aline Zanin, Elder Rodrigues, Flávio Oliveira, Juliana Damasio, Maicon Bernardino. Thank you all for the collaboration, discussion, support and laugh moments.

Thank my friends from Church: Father Pedro for being my spiritual director, for always giving me good advice, for praying for me and, above all, for being a beloved friend. Thank my friends Umberto Guarise, Bruno Pires, Cristian Nunes, Father Kauê, Cássia Feijó, Tainá Martins, Josilene Cristina and Rodrigo Naimayer.

I especially thank my father, mother, sister and my beloved fiancee, Ana Luíza Vicentini for always being by my side in every difficult moment. Thank you, mother, for the magnificent person you are. You always believed and supported me in countless ways.

SPLIT-MBT: UM MÉTODO DE TESTE BASEADO EM MODELOS PARA LINHAS DE PRODUTO DE SOFTWARE

RESUMO

Linhas de Produtos de Software (LPS) tem como objetivo auxiliar no desenvolvimento de sistemas com base na reutilização de componentes de software. Através deste conceito, é possível criar um conjunto de sistemas similares, reduzindo assim o tempo de comercialização e custo e, consequentemente, obter maior produtividade e melhorias na qualidade do software. Embora o reuso seja a base para o desenvolvimento de sistemas para LPS, a atividade de teste ainda não se beneficia totalmente desse conceito. Isto se deve a um importante fator inerente a LPS, *i.e.*, Variabilidade. A variabilidade diz respeito a como os membros/componentes que compõem os produtos de uma LPS diferem entre si. Além disso, a variabilidade representa diferentes tipos de variação sob diferentes níveis com diferentes tipos de dependência. O problema de lidar com a variabilidade no contexto do teste não é uma tarefa trivial, uma vez que quando a variabilidade em LPSs cresce, a quantidade de testes necessários para avaliar a qualidade do produto pode aumentar exponencialmente. Portanto, esta tese apresenta um método chamado SPLiT-MBt para gerar casos de teste funcional e scripts para testar produtos derivados de LPSs. Assim, os casos de teste para testar funcionalidades comuns entre os produtos são gerados com base nesse reuso inerente às LPSs. Para fornecer esse reuso, o SPLiT-MBt é aplicado em duas etapas. Na primeira, as informações de variabilidade e teste anotadas em modelos de sistema são utilizadas para gerar seqüências de teste usando diferentes métodos, *e.g.*, HSI, UIO, DS ou TT. Esses métodos são aplicados a modelos formais, *e.g.*, Máquinas de Estado Finitos (FSMs), as quais são estendidas para lidar com informações de variabilidade. Na segunda etapa, os modelos de teste e as seqüências geradas são reutilizados para gerar scripts de teste, os quais podem ser executados por diferentes ferramentas de teste funcional com o objetivo de avaliar a qualidade dos produtos. Finalmente, para demonstrar a aplicabilidade deste trabalho, utilizamos nosso método para testar produtos de duas LPSs, *i.e.*, uma LPS real chamada PLeTs e uma LPS acadêmica chamada AGM. Além disso, realizamos um estudo experimental com o intuito de avaliar o esforço de gerar casos de teste para produtos de uma LPS. O objetivo foi comparar o nosso SPLiT-MBt com outras duas metodologias/abordagens de teste de LPSs. Ao final, os resultados apontam

que o esforço para gerar casos de teste usando nosso método foi reduzido consideravelmente quando comparado com as outras metodologias.

Palavras Chave: Linha de Produto de software, Teste de Linha de Produto de Software, Teste Funcional, Teste Baseado em Modelos, Métodos de Geração de Casos de Teste.

SPLIT-MBT: A MODEL-BASED TESTING METHOD FOR SOFTWARE PRODUCT LINES

ABSTRACT

Software Product Lines (SPL) aim to develop systems based on reuse of software components. Through this concept it is possible to create a set of similar systems, thus reducing time to market and cost and thus obtaining greater productivity and improve software quality. Although reuse is the basis for developing systems from SPLs, the testing activity does not yet fully benefit from this concept. This is due to an important aspect inherent to SPLs, *i.e.*, variability. The variability refers to how the members/components that compose the products of an SPL are different from each other. It represents different types of variation on different levels with different types of dependencies. The problem of dealing with variability in the test context is not a trivial task, since when variability in SPLs grows, the amount of tests needed to assess the product quality can increase exponentially. This thesis presents a method called SPLiT-MBt to generate functional test cases and scripts to test products derived from SPLs. Thus, test cases to test products common functionalities are generated based on the reuse inherent to SPLs. In order to provide this reuse, SPLiT-MBt is applied in two steps. In the first step, variability and test information annotated in system models are used to generate test sequences using different methods, *e.g.*, HSI, UIO, DS or TT. These methods are applied to formal models, *e.g.*, Finite State Machines (FSMs) that are extended to deal with variability information. In the second step, test models and sequences are reused to generate test scripts, which could be executed by different functional testing tools with the aim of evaluating the quality of products. Finally, in order to demonstrate the applicability of this work, we apply our method to test products of two SPLs, *i.e.*, an actual SPL named PLeTs and an academic SPL named AGM. Moreover, we also performed an experimental study to evaluate the effort to generate test cases for SPL products. The main goal was to make a comparison between our SPLiT-MBt and two other methodologies/approaches. Thus, the results point out that the effort to generate test cases using our method was reduced considerably when compared to the other methodologies.

Keywords: Software Product Line, Software Product Line Testing, Functional Testing, Model-based Testing, Test Case Generation Method.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ANOVA - Analysis of Variance

AGM - Arcade Game Maker

CFG - Control Flow Graph

CADeT - Customizable Activity Diagrams, Decision tables and Test specifications

DF - Degree of Freedom

DS - Distinguishing Sequence

FSM - Finite State Machine

HSI - Harmonized State Identification

HSD - Honestly Significant Difference

QTP - HP Quick Test Professional

RFT - IBM Rational Functional Tester

IDE - Integrated Development Environment

MTM - Microsoft Test Manager

VS - Microsoft Visual Studio

MBT - Model-based Testing

OVM - Orthogonal Variability Modele

PLeTs - Product Line of Model-based Testing Tools

PLUS - Product Line UML-based Software Engineering

PLUC - Product Line Use Case

PLUTO - Product Lines Use Case Test Optimization

RQ - Research Question

ScenTED - Scenario based TEst Case Derivation

SEI - Software Engineering Institute

SPL - Software Product Line

SPLiT-MBt - Software Product Line Testing Method Based on System Models

SDL - Specification and Description Language

SMarty - Stereotype-based Management of Variability

SUT- System Under Test

TDL - Technology Development Laboratory

TT - Transition Tour

UIO - Unique Input/Output

CONTENTS

1. INTRODUCTION

"There is no excuse for those who could be scholars and are not."

St. Josemaría Escrivá

1.1 Problem Statement and Rationale for the Research

Currently, several companies have investigated ways to obtain a higher productivity through reusing software artifacts in order to reduce time and cost in the development of new versions of the software they produce. In this context, Software Product Lines (SPLs) [CN01] have been used as a valuable approach. SPL is defined as a set of software assets sharing common and variable features in order to meet the needs of a specific domain, which may be a market segment or mission [CN01] [Ins16a]. By means of SPL, we can obtain reuse of the software components; thereby reducing cost, time to market, and increasing the products quality. Despite these benefits, the adoption of SPL concepts has also brought some challenges, *e.g.* the complexity to test products derived from them.

A concern related to SPL testing is that, differently from testing single applications, an SPL requires testing functionalities shared by several products (Domain Engineering, oriented to the development of reusable artifacts), as well as testing functionalities of specific products (Application Engineering, oriented to the reuse of developed artifacts) [Lin02]. Therefore, a fault[1] not found during Domain Engineering may result in the generation of several products that may fail [ER11].

In order to overcome these issues, several authors have been focusing their efforts on the development of methodologies [BG04] [OG09] [NFTJ04], techniques [CGSE12] [HVR04], methods [RRKP06] [ER11] [NdCMM$^+$11] [LPP13] and approaches [MSM04] [McG01] [KLKL07] to apply SPL testing. Although these works are valuable to support the generation of test sequences regarding the variability, being able to adapt single application testing techniques into an SPL context, they do not address some relevant issues. For example, these works do not systematically optimize the test case generation, they do not provide a systematic way to extend these techniques, and just some of these works address, in a strict aspect, the problem of test automation for SPL products.

Moreover, to the best of our knowledge, those works do not address the adoption of any prioritization and minimisation technique to generate test cases. Therefore, as consequence of applying these techniques, a bunch of test cases are generated and some of them are useless, *i.e.* repetitive and irrelevant. Therefore, would be useful to define a way to test SPL products considering variability, as well as reducing the amount of test cases and at the same time selecting

[1] We used the concepts of fault, error and failure according to [ALRL04].

the most relevant ones. These test cases could be also used to, automatically, generate test scripts and then provide test automation in both Domain Engineering and Application Engineering.

1.2 Objectives and Thesis Contributions

In order to address the issues introduced earlier, we propose a method named **S**oftware **P**roduct **Li**ne **T**esting **M**ethod **B**ased on Sys**t**em Models (SPLiT-MBt), which provides the reuse of test artifacts based on adapting Model-based Testing (MBT) [Kri04] for automatic generation of functional test cases and scripts from models/notations that represent the SPL functionalities and variability information.

To provide reuse of test artifacts, SPLiT-MBt is applied in two steps. The first one occurs during Domain Engineering, when test and variability information are extracted from SPL models. We assume that these models were previously designed by the SPL analyst using a variability management approach. For example, if the models were designed using UML, then SMarty [Jun10] [OGM10] could be used. SMarty aims to manage variability in UML models supported by a profile and a set of guidelines. This approach could be applied to manage variability present in Use Cases, Classes, Components, Activities, and Sequence Diagrams, as well as to Packages. Therefore, a test analyst uses SPLiT-MBt to add test information[2] on two UML diagrams, *i.e.* Use Case and Activity Diagrams. The test information is added, by the test analyst, on these two diagrams in the form of stereotypes and tags. Then, once the Use Case and Activity Diagrams are annotated with test information, the test analyst uses SPLiT-MBt to generate Finite State Machines (FSMs) [Gil62] from these UML diagrams.

These FSMs are extended in an SPL context and are used as input, in a specific step of our SPLiT-MBt method, to generate test sequences with variability information. These test sequences are generated through extending conventional test sequence generation methods in an SPL context, *e.g.*, Transition Tour (TT) [NT81], Unique Input/Output (UIO) [SD88], Distinguishing Sequence (DS) [Gon70], W [Cho78] or Harmonized State Identification (HSI) [PYLD93].

SPLiT-MBt supports extended versions of these methods, which are modified to generate test sequences considering variability information present in FSMs. An advantage of extending these methods to handle variability in a SPL context is they provide some benefits, such as prioritization and minimisation of test cases. The test sequences generated through applying these modified methods are stored in a test repository and the variability present in these sequences is resolved by our SPLiT-MBt during Application Engineering.

The second step of SPLiT-MBt takes place during Application Engineering, when the variability present in those test sequences is resolved to meet the specific requirements of each system. We assume that the variability is resolved at design time by the SPL analyst from a Traceability Model containing information about the resolved variability. Thus, the Traceability

[2]It corresponds to test data for functional testing, *e.g.* test input data or expected results.

Model is the main artifact to resolve variability present in the test sequences. Once the variability is resolved, the test sequences are reused to test the specific functionalities of several products. Moreover, at this phase, models that represent specific functionalities of each product are generated and they are annotated with test information by the test analyst. Similar as occurs during Domain Engineering, these models are converted into FSMs and the conventional methods of test sequence generation are applied in order to generate specific test sequences for each product. Finally, all these sequences (from Domain Engineering and Application Engineering) are converted into a description equivalent to test cases in natural language, *i.e.*, abstract test cases.

An abstract test case is a text file describing the interaction of the user with the system. By having a generic format, the abstract test cases can easily be used as a reference for generating scripts for different functional testing tools, *e.g.* HP Quick Test Professional (QTP) [Mal09], IBM Rational Functional Tester (RFT) [DCG+09], Selenium [HK06], Microsoft Visual Studio (VS) [Lev11] and Microsoft Test Manager (MTM) [Man16]. SPLiT-MBt allows that test data used to generate test scripts are chosen through functional testing criteria, *e.g.*, Boundary Value Analysis and Equivalence Partitioning [DMJ07] [MS04]. These criteria contribute to a testing process more systematic and effective, since they avoid testing all system inputs, which would make the testing process impractical. The idea is that test artifacts developed during Domain Engineering are reused to test products during Application Engineering.

Therefore, the adoption of our method presents several benefits from the reuse inherent to SPLs. For instance, through the set of test sequence generation methods that are extended using SPLiT-MBt, it is possible to reduce the amount of test cases, provide a full coverage of the product functionalities and contribute to select relevant test cases (prioritization and minimization of test cases). Another advantage of our method is that it is based on MBT, since all test information is annotated on systems models, for example, UML Use Case and Activity Diagrams. Thereby, for being based on MBT, SPLiT-MBt contributes to reduce the likelihood of misinterpretation of the system requirements by a test engineer and decreasing testing time in SPL projects.

Furthermore, SPLiT-MBt saves testing effort, since we reuse the test sequences generated in Domain Engineering to test several products derived from Application Engineering. Therefore, if we have, for example, 10 products sharing several functionalities, our method will generate test sequences to test the common functionalities for these 10 products just once. Therefore, the set of test sequences generated during Domain Engineering is used to test just those products sharing common functionalities.

In a nutshell, SPLiT-MBt aims to answer the following questions: i) How to test the SPL products with the benefits of reusing test artifacts? ii) How to test, in a systematic manner, the products of an SPL? iii) How to reduce the number of test cases and still find the same amount of system failures? iv). How to generate test scripts to be executed by different functional testing tools?

In order to answer these questions, this thesis presents the SPLiT-MBt method, which proposes the following contributions:

- To produce test cases that are reused to generate scripts based on system models. Thus, test cases to test common functionalities for different products are generated just once (Question i).
- To develop a tool named SPLiT-MBt Tool, which supports the activities of the SPLiT-MBt method. Thus, the testing process of products derived from SPLs can be performed automatically (Question ii).
- To adapt test sequence generation methods in an SPL context. Thus, it is possible to reduce the number of test cases through applying prioritization and minimization of test cases, which contributes to select the relevant ones (Question iii).
- To define a generic structure representing test cases in pseudo-natural language. Thus, it is possible to generate test scripts that can be executed by different functional testing tools (Question iv).

This work was developed in the context of a academy-industry collaboration, in which, our research group has worked closely to a Technology Development Laboratory (TDL) of Dell Computer Brazil. It is a global IT company whose development and testing teams are located in different regions worldwide to develop and test in-house solutions in order to attend their own demand systems on a global scale of sales of computer assets. The aim of this cooperation is to experiment and develop new strategies and approaches for software testing. In this collaboration we have developed new strategies and approaches for software testing, *e.g.*, an SPL [CCO+12][RVZG10] to generate testing products for different testing techniques, *i.e.* performance testing [MRMdOTC+15][RBC+15], structural testing [CZR+14], functional testing [LRD+15]. In this context, we have also developed a Domain Specific Language, called Canopus, for performance testing [SRZ16][SZR16].

1.3 Thesis Organization

This thesis has seven chapters and it is organized as follows. **Chapter 2** presents some background on SPL concepts, MBT technique, SPL testing and Variability Management and related work. **Chapter 3** presents a detailed description of the SPLiT-MBt method. In **Chapter 4**, we present two case studies in which SPLiT-MBt is applied to test the product functionalities of an academic SPL: Arcade Game Maker (AGM) [Ins16b] and; an actual SPL: Product Line of Model-Based Testing Tools (PLeTs) [RVZG10]. In **Chapter 5** we also present how we conducted an empirical experiment to analyse three different methods: our SPLiT-MbT, Customizable Activity Diagrams, Decision tables and Test specifications (CADeT) [OG09] and Microsoft Test Manager (MTM) [Man16]. Finally, in **Chapter 6** we present the contribution of this thesis and some conclusions and future work. At the end of this document, we also present an appendix with the UML models of the SPLs we have used to evaluate this work.

2. BACKGROUND

"Humility is the first stage of wisdom."

St. Thomas Aquinas

In this chapter we present the concepts related to Software Testing, Variability management approaches, Software Product Lines and Software Product Line Testing strategies. The goal is to introduce the main concepts related to Software Testing and Software Product Lines. For a more detailed study we suggest the reading of the following works [Web16] [AO08] [LCYW11] [CN01] [Som11] [PBL05] [LSR07] [Gom05] [FGMO12] [RRKP06] [TTK04] [KKB^{+}17].

2.1 Test Concepts and Terminology

In this section, we present some relevant test concepts and terminology that will be used throughout this thesis. Currently, there are some divergences among authors when defining the concepts of fault , error, and failure. Therefore, in order to provide the understanding of these concepts, we will use a terminology defined by authors from fault tolerance area [MM98]. We chose this terminology since it is used as reference by several scientific community researchers and also because it is used by the members of our research project.

During the development of a software system, the minimum expected is that it is in compliance with the software requirements specification. When the software does not reach this goal, we can state that the system has a failure, *i.e.*, it is not in compliance with that specified during the first development stages [Web16] [ALRL04]. This can occur, for example, when the system does not meet some of its functional requirement specifications.

An error is a state of the system in which a processing from that state results in a failure. It is well known that the execution of a functionality may have a set of states, when any of these states diverges from the correct state (what is expected for a specific functionality), we can say that the system is in an error state. Finally, a fault is defined as the initial, physical, or algorithmic cause of an error, usually a consequence of a human action [ALRL04]. Figure 2.1 shows the definition of fault, error and failure concepts, which we have just described.

2.2 Software Testing

The evolution and increased complexity of computer systems have made the testing process an activity so complex as the development process itself. This situation may become more complicated depending on the size and complexity of the software that is being developed. For this reason,

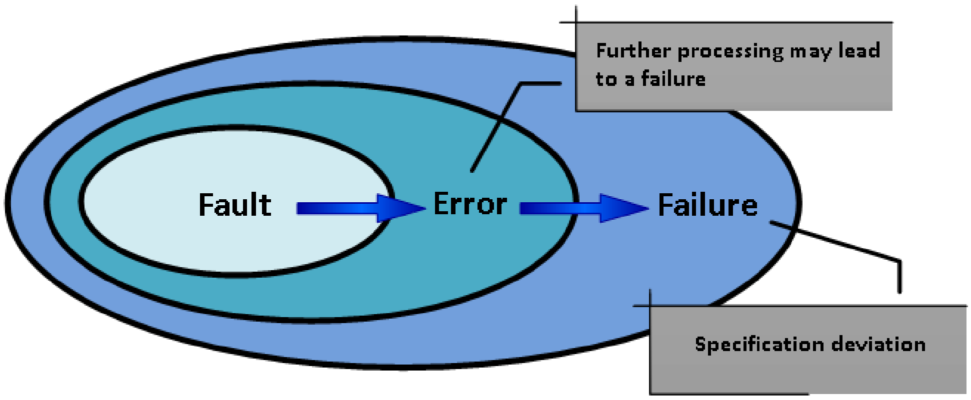


Figure 2.1: Model for fault, error and failure [Web16]

the development of a system may have several problems and, consequently, a product different from that specified during the software requirements stage could be generated [AO08] [MS04].

Currently, there are many factors that contribute to the incidence of errors. However, the main cause of errors is related to the human influence. It is well known that the system development depends on people's notion and their interpretation. Hence, the existence of faults and errors is almost inevitable. For instance, during a software development, the programmers will, inevitably, make some mistake. In this case, we can say that the developers caused a fault and then, a system error was generated; resulting in a set of failures. A system with faults may not generate failures during its execution. In this context, it is better that a system failure occurs frequently than only sometimes [MS04].

The cost of repairing system failures can change according to the moment a fault is found. The cost of a fault found during the first development stages may be very small. On the other hand, a system failure caused by a fault found during the final development steps can cause a huge cost. The damage can be even bigger when a system failure is detected when the client (final user) is using the software. Therefore, when a failure is detected in the early development stages, the cost to repair system faults will be lower as well.

Although find out system faults as soon as possible are very expected/desirable, that is not a trivial task. In order to ensure that these faults will be found before delivering the system to the client, the software must pass by several validation and verification processes [ALRL04]. Validation is a process that aims to evaluate software in order to ensure compliance with the system requirements and; verification is the process that aims evaluating whether the software has met the specified requirements during all software development stages. Validation and verification processes must be present during the first development stages (during the software requirements specification step) and not only in the final stages. In this context, there are several validation and verification techniques that can be applied during the system development, such as: model checking, symbolic execution and software testing [ALRL04].

Software testing is one of the most used techniques to obtain systems reliability. In addition, according to [AO08] software testing is defined as the process of systematically evaluating systems through their controlled execution and observation. The main goal is to identify faults, errors and failures in order to ensure the consistency of the system functionalities.

2.2.1 Testing Techniques

Since we have presented the testing concepts and terminology, we will, in this section, introduce and discuss the concepts of two testing techniques used to assist testers and test analysts to detect and remove faults in software systems, *i.e.* Functional and Structural testing. This two techniques are referenced by several authors and the most of the testing literature as a way to reveal as many failures as possible, which is the purpose of the software testing [MS04] [AO08].

Functional Testing

Functional testing is a technique to derive test cases from the program specification. Therefore, the test case generation is based on the software functional requirements. Moreover, this technique aims to evaluate the external behavior of a program and not only its internal details (source code). For this reason, it is also called a specification test or black-box testing [MS04] [AO08].

The functional test assess a set of outputs from the inputs and checks whether the obtained result corresponds to the expected result [MS04]. Moreover, the functional test is able to identify defects in a program or application, provided that all possible program inputs are applied; when this occurs the test is called *exhaustive test* [AO08]. The problem is that the set of test inputs can be very large or even infinite as well, what could make the test process impracticable or unfeasible. On the other hand, if we do not define some test inputs (or define a small test input size), we could not ensure that all program functionalities are working correctly.

In order to overcome this issue/limitation, there are functional test criteria that were created to make this testing process more systematic, *e.g.*, Equivalence partitioning, Boundary-value analysis and Cause-effect graphing [MS04]. This test criteria can be used by the tester to assist him to reduce the test input size and, at the same time, generate test cases with a higher probability to find faults [MS04] [AO08].

- Equivalence partitioning: this criterion is based on identifying the program input data from the specification and divide this data input domain into valid and invalid equivalence classes [EM07]. Then, assuming that a subset of input data values of a given class (valid or invalid class) is representative of the whole class, we select the smallest amount of test cases based on that subset of input data. Therefore, whether a input value of a given equivalence class reveals a failure, then it is reasonable that all other input values from that equivalence class will reveal the same failure. Therefore, whether a input value of a given equivalence class reveals a failure, then it is reasonable that all other input values from that equivalence class will reveal the same failure. Therefore, whether a input value of a given equivalence class reveals a failure, then it is reasonable that all other input values from that equivalence class will reveal the same failure. Hence, using this criterion, the tester could systematically assess the software requirements and also reduce the amount of test cases. These factors contribute to decrease the effort and time spent when testing a software system.

- Boundary-value analysis: it is known as a complement to the Equivalence Partitioning criterion. However, different from the former criterion, Boundary-value analysis is more rigorous. It means that, instead of randomically selecting any input value from a given class, the Boundary-value analysis criterion aims to generate tests based on the input values associated to a class boundaries (*i.e.*, input values that correspond to the data boundaries of a specific software system functionality). According to [MS04] and [EM07], this criterion can assist the testers, since a large number of failures is usually concentrated in the classes boundaries or close to them.

- Cause-effect graphing: the Equivalence partitioning and Boundary-value analysis criteria have been widely used by several authors from scientific community and testers for many years to assist them to reduce the amount of test cases and the test effort as a whole [MS04]. However, those criteria do not exploit combinations of the input conditions. In order to overcome this issue, Cause-effect graphing criterion aims to define test requirements based on the possible combination of input conditions. Therefore, first, the tester must identify the possible input conditions (causes) and possible actions (effects) of the program. Then, a graph, linking the identified causes and effects, is generated. This graph is converted into a decision table from which test cases are derived.

Structural Testing

Usually, software testing uses functional test cases that are derived from system requirements. However, to test only functional aspects of the system does not guarantee that the program will not present failures during its lifetime, because some part of the program source code may not have been covered. In this context, the execution of a program that contains faults, unfortunately, may not result in the generation of failures noticed by the user [MS04]. Therefore, structural testing is essential, since it aims to verify whether all parts of the source code were covered or not.

Structural testing is a technique for generating test cases from the analysis of source code. It seeks to evaluate the program internal details, such as test conditions and logical paths. For this reason, it is also called test oriented to logic or white-box testing. In general, most of the criteria based on structural analysis use a graph notation named Control Flow Graph (CFG) [VMWD05], which represents all the paths that might be traversed during the program execution. These criteria are based on different program elements that can be connected to the control-flow and data-flow in the program. Control-flow uses the control features of a program to generate test cases, *i.e.*, loops, deviations or conditions. Criteria based on data flow use data flow analysis of the program to generate test cases. The main criteria based on control-flow are:

- All-nodes: this criterion defines that each CFG node must be visited at least once, *i.e.* each command must be executed at least once during the program execution;
- All-edges: this criterion defines that each CFG edge must be traversed at least once, *i.e.* each possible outcome of each decision point must be exercised at least once during the program execution;
- All-paths: this criterion defines that all possible program paths must be executed.

Although it is desirable to execute all paths of a program, this task is impracticable in the most cases. The reason for that is due to the presence of program loops, which might generate an infinite number of paths. This was one of the factors for introducing the criteria based on data flow. Criteria based on data flow use data flow analysis of the program to generate test cases. The test cases are derived from associations between variable definitions and the use of these variables [LVMM07]. One reason for introducing criteria based on data flow was due to, even for small programs, tests based only on the control flow may not reveal faults in some situations, for example, a variable that was declared but never used, or a variable that was declared but was not initialized.

Structural test case generation consists of selecting values from an input domain of a program that satisfies specific criteria. For instance, the All-nodes criterion groups in a domain all the input values that execute a specific node. The selecting input values task could be made using data generation techniques, *e.g.*, random [HT90], based on symbolic execution [LCYW11] or dynamic execution [DLL+09].

Currently, there is a diversity of commercial (*e.g.*, Quick Test Professional [Mal09], IBM Rational PurifyPlus [IBM17]), academic (*e.g.*, JaBUTi [VDMW06], EMMA [Rou17]), and open source (*e.g.*, Semantic Designs Test Coverage [Sem16]) code coverage tools to assist the testing process. However, most of these tools have been individually and independently developed from scratch based on a single architecture. Thus, they face difficulties of integration, evolution, maintenance, and reuse. In order to reduce these difficulties, it would be interesting to have a strategy to automatically generate specific products, *i.e.*, tools that perform tests based on the reuse of assets and the core architecture. This is one of the main ideas behind SPLs [CN01]. SPL and testing are the main concepts addressed on this thesis.

2.2.2 Levels of Testing

In this section, we present the levels of testing that are performed by testers and test analysts during the software development process. These levels of testing is used as reference to assist testers on generating test cases from the software requirements specification to the system acceptance by the customer/client. It is well known that there are many testing level models proposed by several authors in the literature that discuss the relation of each testing level with the software development process [MS04] [AO08] [Som11]. According to these authors and many others, there are four levels of testing (see Figure 2.2), *i.e.*, Unit Testing, Integration Testing, System Testing and Acceptance Testing.

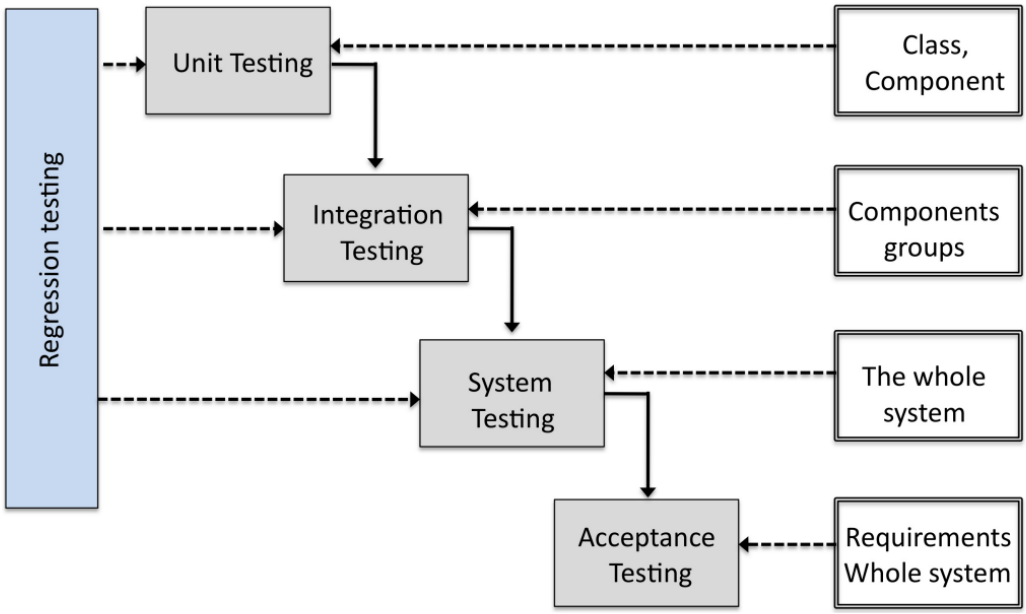


Figure 2.2: Levels of Testing [Utt06]

Unit Testing

The Unit Testing, also known as module test, aims to test the smallest units of the software, *i.e.* the most basic software components, such as, functions, methods and classes. It is usually performed by the developer, since he has a higher knowledge about the source code. Therefore, usually, the Unit Testing aims to detect algorithm and/or logic faults and even minor programming faults. Furthermore, a good set of unit testing contributes to find and remove faults in the early stages of development. Hence, it is possible to save time and reduce costs, since it is cheaper to remove faults during unit testing than at any other testing level [AO08].

Integration Testing

The purpose of the integration testing is to find errors generated from the integration of internal software components that were already tested during the unit testing. Different from the unit testing, integration testing aims to individually test each system component/module. Furthermore, it seeks to find faults when these components are combined, since a software component working properly during unit testing could not work when combined/integrated to another software components. The integration testing can also be applied to verify the compatibility among software components, since there is no guarantee that these modules wont have connectivity problems when integrated. That may occurs, mainly, when developer teams uses different versions of the development IDE to build software components. For instance, a development team A uses an older Eclipse version and a development team B uses the most recent one [Som11].

System Testing

During the system testing level, all software components were already integrated and successfully tested. Hence, this level aims to verify whether the system software is in accordance with defined in the software requirements specification. Therefore, during system testing, testing teams generate a set of test cases based on the software specification document. In this context, the main goal is verify if the software functionalities are working as expected; if the the software still working when it is submitted to a high load. For instance, the functionalities of an e-commerce system may work when one user is accessing the application. However, it could not work as expected when one hundred (high load) users are accessing the application simultaneously. Actually, there are several types of system testing that can be performed by different testing teams with specific and high skills. Next, we present some of them:

- Functional Testing: to perform functional testing, the software requirements specification is analyzed to derive a set of functional test cases. Therefore, it aims to verify whether the system behavior meets its software requirements. Thereunto, the testers submit the software system to a set of input data and then, the output is analyzed. If, for a specific functionality, the output is equal to the expected result, it means that the analyzed functionality is working properly (for more details, see Section 2.2.1).

- Performance Testing: the performance testing aims to analyze the system behaviour when it is submitted to a given users load in a specific test environment. In an nutshell, it seeks to evaluate how much a system or system component is able to meet the performance requirements, such as, response time or throughput. For example, a performance requirement might define that "response time" for a specific functionality must be less than three seconds when one hundred users are accessing the system simultaneously. Hence, the performance tester can simulate that environment and analyze the results. If the "response time" is higher than 3 seconds, for example, then the system must be optimized. An example of system optimization could be increase some system resources, *e.g.* memory and processor. In addition, the performance test aims to identify potential bottlenecks that cause performance degradation in the system [SW02].

- Stress Testing: stress testing aims to assist performance testers on analyzing the system behavior and determine whether the system meets its performance requirements when it is submitted to beyond the normal load conditions. Therefore, the performance testers may verify if the system still working even under the worst load.

- Security Testing: it aims to ensure that the software system meets the security requirements. It also verifies if the software system works as expected when submitted to the most diverse access illegal attempts, aiming to identify possible vulnerabilities. In order to perform this, it tests whether the system protection mechanisms, actually protect it from improper access. It is very common that software systems being target from people who seek to perform actions that could harm or even benefit others. Due to situations like that, the security test aims to demonstrate whether the system performs exactly what it should do or not. Furthermore, it can also assist testers to define a contingency plan and then, determine what precaution should be taken against possible attacks.

Regression Testing

Regression testing cannot be defined as testing level, since it is applied along all software testing process. It is most like a sub step performed during the other testing levels (Unit, Integration and System levels). Actually, regression testing is a test technique applied when a change in the software system is made, *i.e.*, when a specific software functionality is added or removed; when the system is migrated to another platform. Therefore, in an nutshell, regression testing is applied to each new version of the software [MS04].

When the software system could not maintain its functionalities working properly in its new versions, we can say that the software system "has regressed". In this context, the regression testing aims to contribute to the "non-regression" of the new system versions. Hence, any testing process that seeks to avoid problems related to systems regression is called "non-regression" testing. By convention, the "no" is omitted and it is usually called regression testing.

The use of this technique is essential, since each system modification can generate faults and failures in the software components (that in previous versions were working properly). Therefore, tests performed in previous software versions should be repeated in their new versions and then, to ensure that the current software components will work and stay valid. A great idea is that these tests are automated, because, consequently, the time spent to perform them again will be smaller. The test automation process can be essential, especially whether the software system is constantly in maintenance process.

Acceptance Testing

The acceptance testing is usually performed by the customer/end user and, in general, it is a testing level that is not considered as a responsibility of the company who developed the system. Therefore, the customer aims to check the system behaviour in order to verify whether the software meets the requirements specification specified in the contract [MS04]. In an nutshell, the customer aims to validate the expected results from the software system. Although the customer has the responsibility for running the tests, the professional from the company (developers and testers) could assist him preparing the test environment and execution. Finally, the customer could accept the software and then, the company prepares the software delivery or the customer may not accept the software as well. In the second case may occur when the customer identify some inconsistency or nonconformity with the software requirements. Therefore, the software must be revised and corrected.

2.2.3 Model-based Testing

Model-based Testing (MBT) is a technique to automate the generation of test artifacts based on system models [Kri04]. Using MBT, it is possible to model the structure and the behavior of a system, hence they can be shared and reused by test team members. Furthermore, it is possible to extract the test information from those models to generate new test artifacts, such as, test cases, scripts and test scenarios [EFW01].

In order to generate test artifacts, the MBT adoption requires the creation of models based on system requirements specified by software engineers and test analysts. One approach that can be applied to better represent the system requirements is the use of UML models [BRJ05]. UML models can improve the system specification through stereotypes and tag definitions. The use of stereotypes is one of the UML extensibility mechanisms in which properties are described using tags. That is, when a stereotype is applied to a model element, the values of the properties are referred to as tagged values. Hence, all the information added to the model through stereotypes and tagged values can be used to derive new artifacts, such as, test cases and/or test scenarios.

Although MBT can bring several advantages, as already mentioned, and test artifacts can be generated using models based on UML, for example, the MBT technique has been widely used to test products generated from single system applications. The benefits from the adoption of this technique can also be applied to test SPL products.

2.3 Software Product Line

In last decade, Software Product Line has emerged as a promising technique to achieve systematic reuse and at the same time to decrease development costs and time-to-market. An SPL can be defined as a group of similar software that were developed from a common set of requirements, that share common core assets and it is focused on obtaining a high degree of reuse [CN01] [PBL05].

In past years, many successful cases studies have been reported [Ins16a] [LSR07] [SPL17] the benefits from the reuse inherent to SPLs. For instance, Nokia reported that the SPL adoption resulted in a significant increase in production of new models of mobile phones. Therefore, SPL allows to optimize time and resources due to the possibility to build a new software from the management of products variability and reuse of existing components.

According to Software Engineering Institute (SEI) [Ins16a], the SPL engineering has three main concepts from which it is possible to provide that reuse of components. The first one, is called *Core Assets Development*, also known as Domain Engineering. The second concept is named

Product Development and it is known as Application Engineering. Finally, the third concept is named *Management of Product Line*.

During Domain Engineering, the common and variable SPL artifacts are defined, while during Application Engineering, the main goal is to derive specific products (applications) exploring the variability of an SPL. Variability management is one of the main activities performed during the development of an SPL, since that is the way to differentiate a product from each other in an SPL [CN01]. A variability can be represented by variants and variation points. A variation point represent artifact places that were not resolved during the development of the core assets and; variants represent a feature/functionality that will be chosen to resolve a variation point. For each variation point, one or more variants can be associated. Variability (*i.e.* variants and variation points) can be introduced in all artifacts, which will be explored during the Application Engineering to derive products that meet the specific requirements of different customers [PBL05]. Some of these artifacts refers to requirements, architecture, components, test cases and feature model.

The feature model represents the aspects related to the variability in an SPL. Moreover, it presents all the characteristics that are inherent to an SPL and the relation between the components as well. According to [KCH+90], a feature is an important/relevant system feature that is visible to the end user. A feature can be optional, common/mandatory or be part of alternative inclusive variants (*alternative_OR*) or exclusive ones (*alternative_XOR*) [KCH+90]. An optional feature does not need to be present in a product derived from the SPL. On the other hand, a mandatory feature must be present in all products derived from the SPL. Furthermore, it is possible to exist *alternative_XOR* features in a feature model, which means that: when one feature is chosen from a list of possible features, then the other features of the same list wont be present in a specific product. In turn, *Alternative_OR* features are used to represent a situation where one or more features of the same variation point can be present in a specific product. Moreover, features may have some relation to each other and some restrictions can be determined, such as: dependency relationships (*depends/requires*) and; mutually exclusive ones (*mutex/excludes*).

Figure 2.3 presents a feature model of a mobile phone, which has four features in the first level. The *Call* and *Screen* features are *mandatory* and; *GPS* e *Media* are *optional*. The *Screen* feature has three *alternative* sub features: *Basic*, *Colored* and *High Resolution*, which means that these three sub features wont be present in the same product, simultaneously. The *Basic* sub feature has a *exclusion* relationship with the feature named *GPS*, *i.e.* if one of them is chosen to resolve the variability, then the other one wont be present in the product. Therefore, whether a product screen is basic, thus, the product will not be the *GPS* feature and vice versa. The *Media* feature has two *or* sub features: *Camera* and *MP3*, where the *Camera* feature has a dependency relationship with the *High Resolution* feature. In this context, if a product has a *Camera* media, then, the screen must be in *High Resolution*.

The SPL features model is responsible for representing aspects related to variability, which may be associated to different abstraction levels, such as, source code and documentation. As described earlier, variability defines how members of a products family differentiate each other and

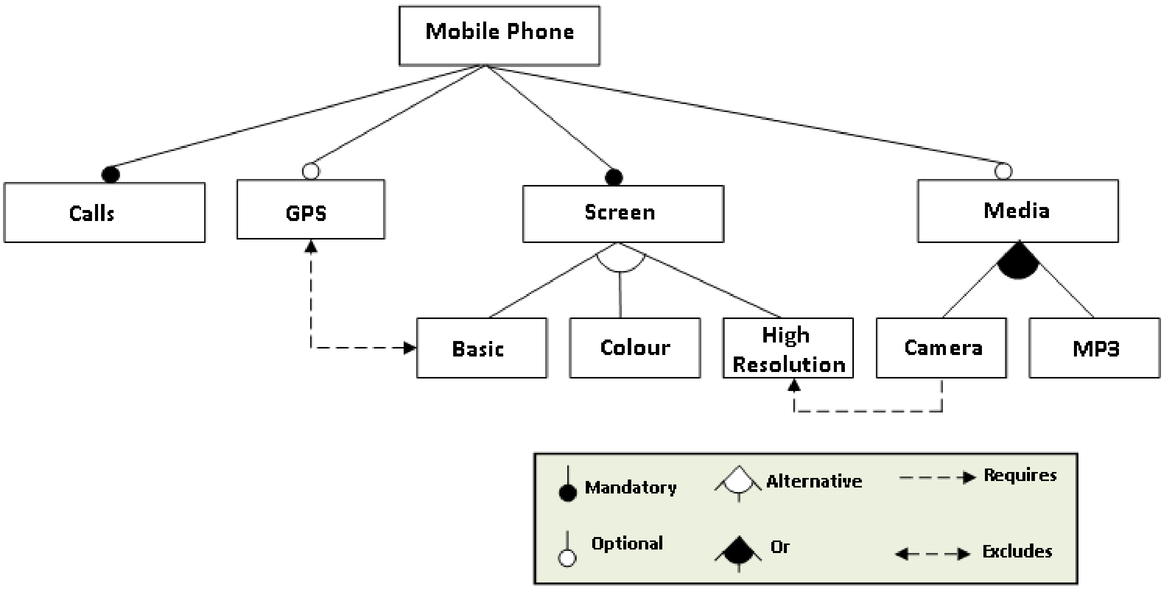


Figure 2.3: Feature Model of a Mobile Phone [Bro16]

also are represented by variation points and variants, where a variation point may contain one or more variants. In a mobile phone SPL, for example, the variation point could be the communication protocol and the variants of this variation point could be GSM, UMTS, CDMA.

The reuse inherent to SPLs is one of its main benefits due to the variability introduced during Domain Engineering. Next, we present some works describing how the variability management could be integrated to system models and the MBT technique.

2.3.1 Variability management in SPLs from System Models

The adoption of MBT can also be applied in the context of SPLs through an adaptation of this technique. However, to adapt MBT in an SPL context it is necessary to consider an important aspect inherent to SPLs, *i.e.* variability. Variability determines the differences among products derived from an SPL and is defined in the Domain Engineering, where the variability management is performed. Variability management is a responsibility of the SPL analyst and includes several tasks, such as, variability identification, analysis and resolution. In order to resolve a specific variability it is necessary to take one or more design decisions, which were postponed at some point during development of an SPL [PBL05] [LSR07].

Variability management is related to development stages of SPLs, which include at least the following activities [PBL05] [LSR07]: **variability identification**, identify differences in SPL products and where they are located in the core artifacts; **variability delimitation**, specify the binding time and the multiplicity of a variability. Binding time defines the time that a variability is resolved, while the multiplicity of variability determines the amount of variants to be chosen to resolve a variability; **variability implementation**, consists of selecting mechanisms to realize variabilities, *e.g.*, plugins are variability mechanisms at runtime. For instance, using Eclipse Java

Integrated Development Environment (IDE) [Gro09], it is possible to add plugins during program execution; **variant management**, controls the variants and their variation points. This task also consists of analyzing products that can be derived from an SPL.

Currently, there are works that aim to manage variabilities using different modeling notations, such as, Specification and Description Language (SDL) [KJG99] and UML [BRJ05]. In [HP03], the authors extend the UML metamodel and propose the triangle notation, in which a triangle represents the relationship between a variation point and its variants. This triangle is included by an Use Case (variation point) and connected to a set of Use Cases (variants). Similarly, [Gom05] presents an approach named *Product Line UML-based Software Engineering* (PLUS), which uses the concept of extension points in Use Case diagrams to represent the relationship between a variation point (extended Use Case) and its variants (Use Case of extension).

Another work describing a variability management approach in SPLs is proposed by [OGM10], in which the author presents an approach named *Stereotype-based Management of Variability* (SMarty). This approach was adopted and used to represent variability information for SPLiT-MBt. The motivation for choosing SMarty among other variability management approaches based on UML notation, is that it can be easily extended, it has a low learning curve, it supports many models, it is able to represent variability information in UML elements by using tags and stereotypes and, different from other approaches, it defines a stereotype to represent inclusive variants. Moreover, through its profile, SMarty represents cardinality information of variants as a meta-attribute, making the variability representation fully compatible with the UML metamodel, which makes SMarty supported by existing UML modeling tools [OGM10] [FGMO12]. Although we have chosen SMarty, the SPLiT-MBt can also be integrated with other similar approaches, *e.g.*, the triangle notation [HP03] or PLUS [Gom05], since they represent variability information in models describing the systems' functionalities.

Those variability management approaches can also be integrated with testing strategies and benefit from the reuse inherent to SPLs to generate reusable test artifacts to test products derived from SPLs.

2.3.2 Software Product Line Testing

The testing activity does not yet fully benefits from the concept of reuse, which is the core of the systems development from SPLs. To mitigate this problem, there are three strategies to test products derived from SPLs [RRKP06] [TTK04]: test each product individually; opportunistic reuse or; test case generation oriented to reuse.

The first strategy (see Figure 2.4) does not benefit from the advantages of reusing test artifacts, since each product generated in the Application Engineering is individually tested. This strategy is very expensive because there is no reuse. Therefore, the effort to test products derived from an SPL is the same to test applications based on the development of single applications. Thus, test cases that are common to many products are derived several times.

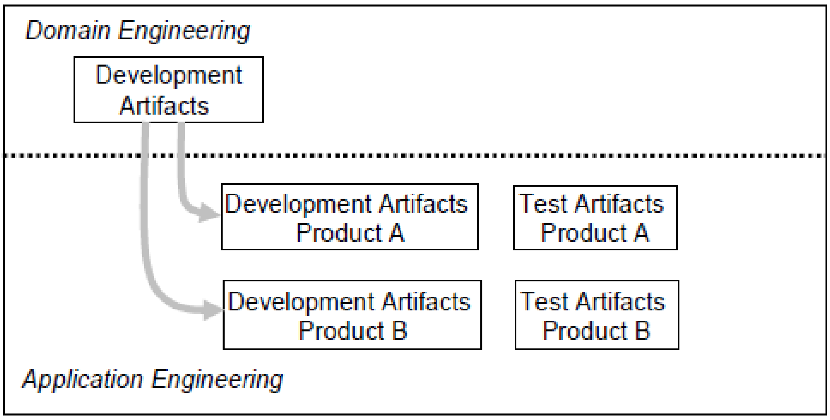


Figure 2.4: Separate test case development [RRKP06]

Differently from the first strategy, the second one (see Figure 2.5) applies the concept of reuse of test artifacts. For this strategy, the test cases generated for the first application derived from an SPL are reused to test new applications. This type of reuse is not systematically performed, *i.e.*, methods to support the selection of test cases are not used. Another problem occurs when selecting test cases from the first application is not correctly performed. Thus, the functionalities of the new applications could not be completely tested.

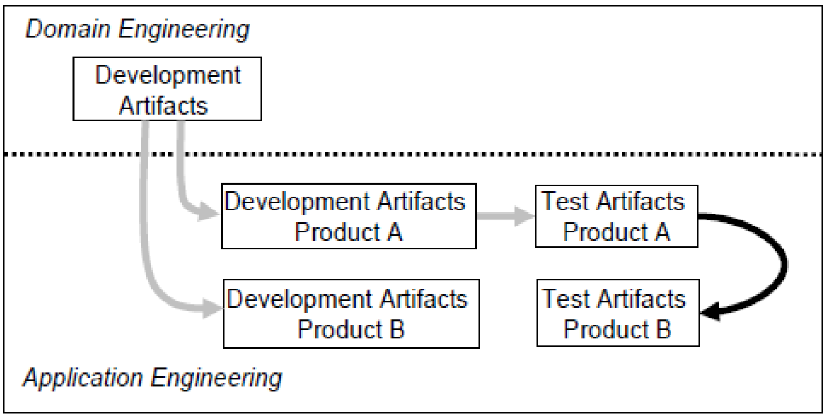


Figure 2.5: Opportunistic reuse of existing test cases [RRKP06]

The third strategy (see Figure 2.6) generates test cases similar to the product development from SPLs, *i.e.*, it takes place in two sub-processes. In the first (*Domain Testing*), reusable test artifacts are generated, while in the second one (*Application Testing*) the test artifacts are reused to test specific products. Thus, test cases to test functionalities common to several applications are generated once. This strategy is used as reference to generate test artifacts using our SPLiT-MBt and some SPL testing approaches, which we discuss in the next section .

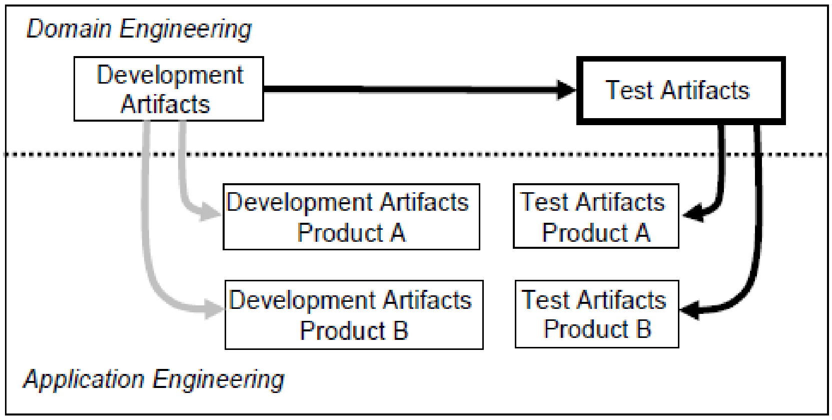


Figure 2.6: Design test cases for reuse [RRKP06]

2.4 Related Work

Currently, there has been an increasing interest related to SPL testing. In this context, several authors have presented approaches and methodologies to test SPLs using testing techniques, such as, structural or functional testing or variability and architecture testing for SPLs. Especially, functional testing based on UML models are the most discussed topic.

Several studies present approaches and methodologies deriving functional test cases from UML models, which are adapted to represent variability information in SPLs. For example, McGregor *et al.* [McG01], Nebut *et al.* [NFTJ04], Bertolino *et al.* [BG04], Kang *et al.* [KLKL07], Hartmann *et al.* [HVR04], Olimpiew *et al.* [OG09], Reuys *et al.* [RRKP06], Capellari [CGSE12] and most recently Kang *et al.* [KKB+17]. A brief description of the majority of these works is found in three systematic mappings [ER11] [NdCMM+11] [LPP13] and a comparison with our method is described below.

In the approach proposed by McGregor *et al.* [McG01], test cases with variability information are derived in Domain Engineering and reused in Application Engineering. Similarly to the approach presented in this thesis, the author applies the proactive reuse concept to derive test cases. However, the approach proposed by the author is not based on MBT, but it focuses on deriving test cases from requirement documents in natural language. Therefore, that approach does not benefit from the advantages of adopting MBT, such as requirements validation through testing models, systematic generation and automation of test cases.

Nebut *et al.* [NFTJ04] present a method for generating test cases from sequence diagrams, which represents scenarios describing the test requirements. These scenarios cannot be directly used to test an application, since they are generic and incomplete, *i.e.*, these are high-level sequences that are used to automatically generate test cases for each product. Similar to SPLiT-MBt, this method applies the proactive reuse concept, and it derives test cases from the MBT technique. However, the method proposed by Nebut *et al.* focuses on reducing the testing effort only through reusing test artifacts developed during Domain Engineering. On the other hand, SPLiT-MBt reduces the testing effort through reusing artifacts and through defining methods to generate test sequences that are applied in both Domain Engineering and Application Engineering.

Bertolino *et al.* [BG04] present a methodology called *Product Lines Use Case Test Optimization* (PLUTO), in which UML use cases are modified to represent variability information in SPLs (*Product Line Use Cases* - PLUCs). This information is explicitly described through defining three tags: *Optional*, *Alternative* and *Parametric*. Based on this methodology, test cases are manually derived using the *Category Partition* method. This methodology extends the *Category Partition* method to deal with the variability in SPLs and then instantiate test cases for a specific product. However, the derivation of test cases is manually performed, since the approach specifies the test requirements in natural language. Therefore, this lack of automation contributes to a greater effort on the testing process as a whole.

Kang *et al.* [KLKL07] present an approach to derive test scenarios from the "merge" of adapted sequence diagrams to represent variability information in Orthogonal Variability Model (OVM), which represents the SPL architecture. However, it is necessary to generate a sequence diagram and combine it with the corresponding architecture model for each derived test scenario. On the other hand, SPLiT-MBt integrated to SMarty derives test cases from a single model, *i.e.*, activity diagrams representing variability and test information.

Hartmann *et al.* [HVR04] use activity diagrams as testing and variability model. However, test cases are derived only during Application Engineering. Therefore, it does not consider the reuse of test cases.

Olimpiew *et al.* [OG09] present a method called Customizable Activity Diagrams, Decision Tables and Test Specifications (CADeT), which is based on the PLUS method to generate test cases from use case diagrams and feature model. Activity diagrams are generated from use case diagrams, from which decision tables describing test data are derived. Although this method could be similar to SPLiT-MBt, it does not present explicitly how the coverage of the test sequences generated from the activity diagram is performed. Moreover, the method proposed by Olimpiew *et al.* does not present precisely how constraints among variants influence the amount of test cases.

Another work similar to SPLiT-MBt is presented by Reuys *et al.* [RRKP06] that proposes a method called Scenario based TEst Case Derivation (ScenTED) to derive test cases from Use Case and Activity Diagrams. These test cases are generated using an extended version of the Branch Coverage criterion to the SPL context. These test cases are represented by sequence diagrams that contain concrete test data. Although this method allows the reuse of test cases preserving variability information in Domain Engineering (as well as the SPLiT-MBt), it is not clear how to generate test cases considering dependency relationships, such as, *requires* or *excludes*. Moreover, for each test sequence generated by the *Branch Coverage*, it is necessary to generate the corresponding sequence diagram, as well as to adapt these diagrams in Application Engineering. On the other hand, SPLiT-MBt provides methods of generating sequences applied to Domain Engineering and Application Engineering, and it presents a systematic way to adapt the test sequences generated in Domain Engineering to test products.

Capellari [CGSE12], presents the FSM-based Testing of Software Product Line FSM-TSPL method to generate HSI Test Sequences for SPLs. Different from SPLiT-MBt, this work is only used to generate test sequences during Application Engineering and it is just based on an opportunistic reuse [RRKP06]. This type of reuse is not systematically performed, *i.e.*, methods to support the selection of test cases are not used. Another problem occurs when selecting test cases from the first application is not correctly performed. Thus, the functionalities of the new applications could not be completely tested.

The majority of approaches, methods and techniques presented in these papers apply MBT to derive functional test cases and reuse these artifacts to test specific products. Although these works present alternatives to reduce the testing effort, most of them focus only on the reuse of developed artifacts. On the other hand, SPLiT-MBt aims to minimize this effort through adapting test sequences generation methods applied to FSMs, *e.g.*, HSI, UIO, TT or DS. These methods allow greater coverage of failures and to generate smaller test sequences. Furthermore, these works do not present clear and structured steps to generate test scripts. In general, most of these studies leave many gaps when referring to traceability; they do not describe precisely how test information present in models are instantiated to generate scripts for testing tools. Moreover, these works have few details on the empirical validation, and do not describe clearly how the test automation process is performed.

It is important to highlight that, recently, Kang *et al.* [KKB+17] presented a comparison among the major Software Product Line Testing methods. In the paper, the author addressed the strategies defined by those methods, *e.g.* opportunities for reuse depending on how variability is represented in the domain test artifacts and the SPL aspects that are essential for reuse in SPL development, *e.g.*, variability representation and test data determination time. The main methods presented by the author are also addressed in this thesis, which demonstrates that our research is valid and is focus of interest for the scientific community.

Table 2.1 presents a comparative summary of the main SPL testing approaches/methodologies regarding to the main concepts and characteristics considered by our SPLiT-MBt.

These comparisons provide some indicators related to the effectiveness and advantages of our SPLiT-MBt when comparing to other SPL testing methodologies. However, we know that an experimental approach and a deep study must be considered to prove these supposed advantages.

Table 2.1: Comparison among SPL Testing Approaches/Methodologies

Features	CADeT	ScenTED	PLUTO	SPLiT-MBt
Variability Management Approach	PLUS	Own simple approach	No	Support SMarty and other approaches
System Models	Use Case and Activity Diagrams	Use Case and Activity Diagrams	No (Requirements are specified in natural language)	Use Case and Activity Diagrams
Test Case Generation Method	Unclear	Extended Branch Coverage criterion	Category Partition	Extended HSI, UIO, W,Wi
Test Automation	Partial (do not generate test scrips)	Partial (do not generate test scrips)	Manual	Script generation from UML models
Functional Testing Techniques	No	No	No	Boundary-value Analysis
Test Script Generator	No	No	No	MTM script

2.5 Research Methodology

It is well known that it is possible to define a structured method as a way to reach the objectives of a scientific research. Moreover, we also know that, in order to achieve these objectives, we can structure or systematize our research through defining a set of technical strategies, *i.e.* a research methodology [Yin13]. In an nutshell, the research methodology corresponds to a process and operations that can be applied in a scientific research [Yin13]. In this section, we present the research methodology we have planned and applied during the development of this thesis.

We classified our research as exploratory. The exploratory research aims to develop or modify concepts and ideas through defining new theories and hypothesis that was not discussed in previously studies [Gil95]. It is important to highlight that we have defined two methods for our exploratory research, *i.e.*, literature review, and empirical experiment. Next, we present the research design we have defined to organize our research.

2.5.1 Research Design

During the research development of this thesis we choose a quantitative strategy regarding to the exploratory research, *i.e* controlled experiment [WRH^{+}00]. An experimental research can be used to test hypothesis under managed conditions through comparing, for example, approaches, methods or models. We have applied an controlled experiment, since the research of our thesis aims to answer "how" and "why" questions and also because we verified the need to compare our SPLiT-MBt against other two methods to generate test cases for SPL products.

In order to develop our research, we defined a research methodology organized in three phases: Conception, Validation and Knowledge (see Figure 2.7). The phases description is as follows:

- Conception: the first phase is divided in two blocks, *i.e.* `Theoretical Base` and `Development`. The former is related to the research objectives, literature review, research question. In an nutshell, it refers to the theoretical basis definition regarding the to main research topics of our thesis, *i.e.* Software Product Line Testing, Model-based Testing, Testing types. The later corresponds to the analysis of the SPL testing methods existing in the literature, *e.g.*, CADeT, ScenTED, PLUTO, which are used as reference to define our SPLiT-MBt. Moreover, we studied different methods to generate test sequences, *e.g.* HSI, UIO, W, TT and how to extend them to an SPL context.

- Validation: this phase is divided in two blocks, *i.e.*, `Utilization` and `Experiment`. The former is used to illustrate how we have applied our SPLiT-MBt to generate test cases and test scripts for two SPLs, *i.e.*, AGM and PLeTs. The later is used to illustrate how we compare our SPLiT-MBt against two other methods through defining an controlled experiment. It refers to the planning, execution, data collection, and the result analysis of the experiment.

- Knowledge: the last phase aims to contribute to the scientific community by publishing our research results.

It is important to highlight that the data analysis we have defined to assess our research corresponds to: statistical methods*e.g.* average, median, and standard deviation. Moreover, during the experiment, we performed advanced statistical techniques to assess the behaviour of our SPLiT-MBt, *i.e.*, Analysis of Variance (ANOVA) test [Lev12] and the Tukey test, which aims to determine the differences among means (averages) in terms of standard error [Lev12] [Por17].

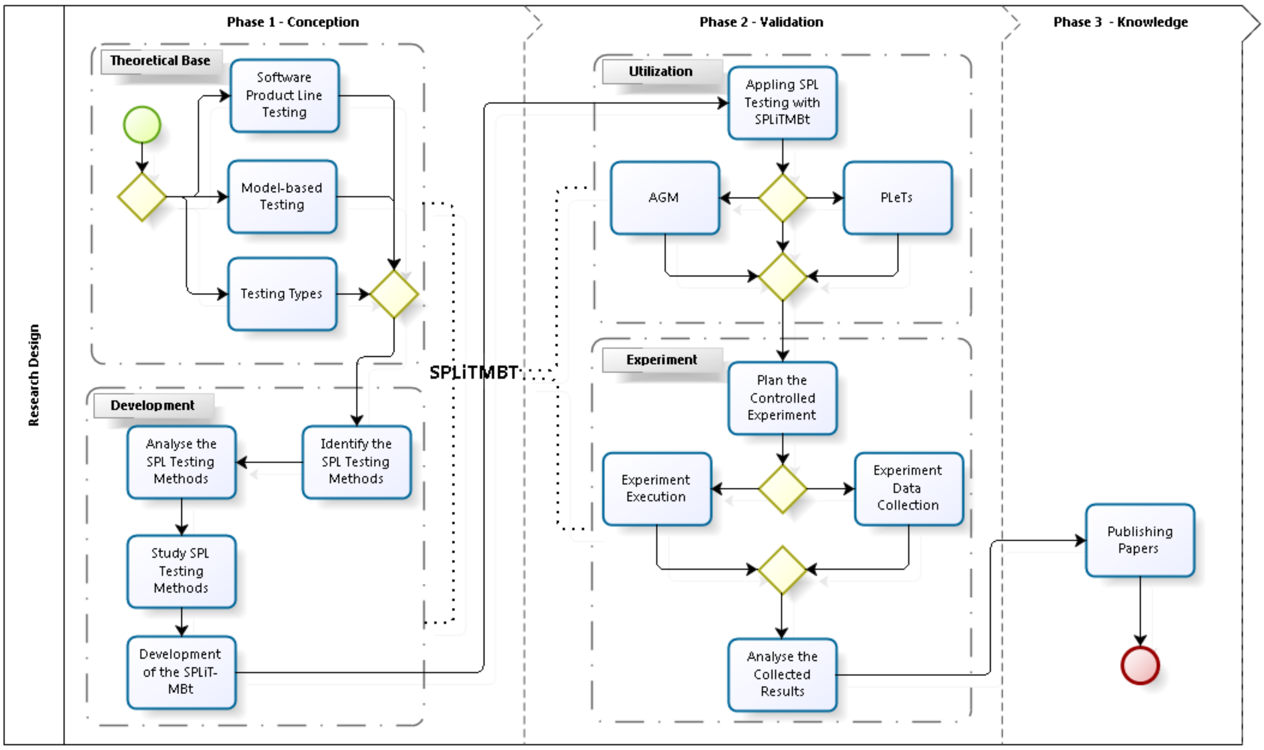


Figure 2.7: Research Design

2.6 Chapter Summary

Due to the capability to model and represent variabilities and thus obtain greater reuse of features, functionalities and components, the use of SPLs can bring several advantages and benefits to customers and consumers. According to [LSR07], after the advent of programming languages, SPLs can represent the most "exciting" and significant changing in the development paradigm, due to the ease and efficiency in developing systems with SPLs. The author also emphasizes that in no other software engineering area are evidenced improvements such as, those provided by SPLs. The most important claims is related to the benefits from the use of SPL. In this context, we highlight the product quality, smaller time to market and higher productivity in product development. Moreover, many companies, such as, Philips and Nokya, have found that, when a strategy for using SPLs is well implemented, it can bring several improvements, such as [Ins16a]: higher productivity in large scale; increasing of products quality; higher customer satisfaction; higher efficiency in the use of human resources; higher company's capability to keep itself in the market; cost reduction; ability to migrate, in months, to new markets (not in years). Despite these benefits, one of the problems with the SPL adoption is related to the product testing. This is due to the variability introduced during Domain Engineering, which makes the SPLs' testing activity a great challenge. In the next section, we present our MBT method to generate functional test cases for SPL products and how it is able to assist testers to deal with variability information during all SPL testing process .

3. SPLIT-MBT: A MODEL-BASED TESTING METHOD FOR SOFTWARE PRODUCT LINES

"Those who are not good to others are bad to themselves."

St. Leo the Great

In the previous chapter, we focused on describing the main concepts related to the Testing, SPL Testing and MBT. We also presented the main works (methods, approaches and techniques) that have characteristics in common with our SPLiT-MBt. In this chapter, we introduce the motivation and discuss the main features of our method, which was developed from our collaboration with a Technology Development Lab (TDL) of Dell Computers Brazil. Furthermore, an example (*i.e.* an UML model) is used as a reference to detail the steps of our SPLiT-MBt. At the end, we present, based on the the steps of our method, how we could perform SPL Testing using an SPL Testing Tool named SPLiT-MBt Tool, which is able to generate test scripts to test products during Application Engineering.

3.1 Contextualization

The adoption of SPL concepts to develop a family of related products leads to the reduction of cost and effort. Although the adoption of these concepts may provide several advantages during the requirements identification, design and coding of software artifacts, the testing phase must be carefully planned and executed. It is well known that software testing in a stand-alone development process is a complex and challenging activity. Therefore, the testing activity in an SPL context is even more complex, since common and specific artifacts must be tested, as well as the interaction among these artifacts. Consequently, in order to test SPL products it is necessary to develop and apply approaches and methods differently from those used in a stand-alone development process, where single applications must be tested. With the purpose of improving the testing of applications derived from SPLs, we propose a method to test SPL products named **S**oftware **P**roduct **Li**ne **T**esting **M**ethod **B**ased on Sys**t**em Models (SPLiT-MBt).

Our method supports automatic functional test case generation from UML models, but could be expanded for other models. As mentioned before, the idea is to generate test artifacts during Domain Engineering and reuse them during Application Engineering. In order to make this possible, SPLiT-MBt is applied in two steps: first, we add test information on UML models, that were previously designed using a variability management approach, to generate test sequences during Domain Engineering. Second, we add test information on UML models (during Application Engineering) and resolve variability present in the test sequences. Finally, test scripts are generated to be executed for a specific testing tool. Next, we present the main topics that encompasses this steps:

- Testing activity during Domain Engineering:
  - extracting test information and variability from extended UML models based on SPL requirements;
  - generating Finite State Machines (FSMs) with test and variability information from UML models;
  - generating test sequences with variability through applying a chosen test sequence generation method (over FSMs), which is adapted/extended to SPL context;
- Testing activity during Application Engineering:
  - resolving variability present in test sequences;
  - extracting test information from UML models based on software requirements of a specific product, since it is possible to exist functionalities of products that are represented only in Application Engineering;
  - generating FSMs from information annotated on UML models of a specific product;
  - generating test sequences to a specific product through applying a chosen test sequence generation method (over FSMs);
  - generating test cases to test SPL products;
  - generating test scripts for a chosen functional testing tools based on test sequences generated on Domain Engineering and Application Engineering.

These steps are performed by a prototype tool supporting the SPLiT-MBt activities, *i.e.* SPLiT-MBt Tool. It is a plugin-based tool that provides automatic test sequence generation using different test sequence generation methods, *e.g.*, W, Wp and HSI. Furthermore, in order to automate test scripts generation, as well as the test execution, our tool can also be integrated with different functional testing tools *e.g.*, QTP, RFT, Selenium, VS, MTM. In the next sections, we present in details the steps to generate test cases and scripts based on the SPLiT-MBt (see Figure 3.1).

3.2 SPLiT-MBt during Domain Engineering

Functional testing at the system level could not be applied during Domain Engineering. The reason is related to the presence of variability and also because complete systems are not derived during Domain Engineering [PBL05]. However, during Domain Engineering it is possible to generate test artifacts that can be reused to test systems generated during Application Engineering. In this context, the SPLiT-MBt supports automatic generation of reusable functional test artifacts

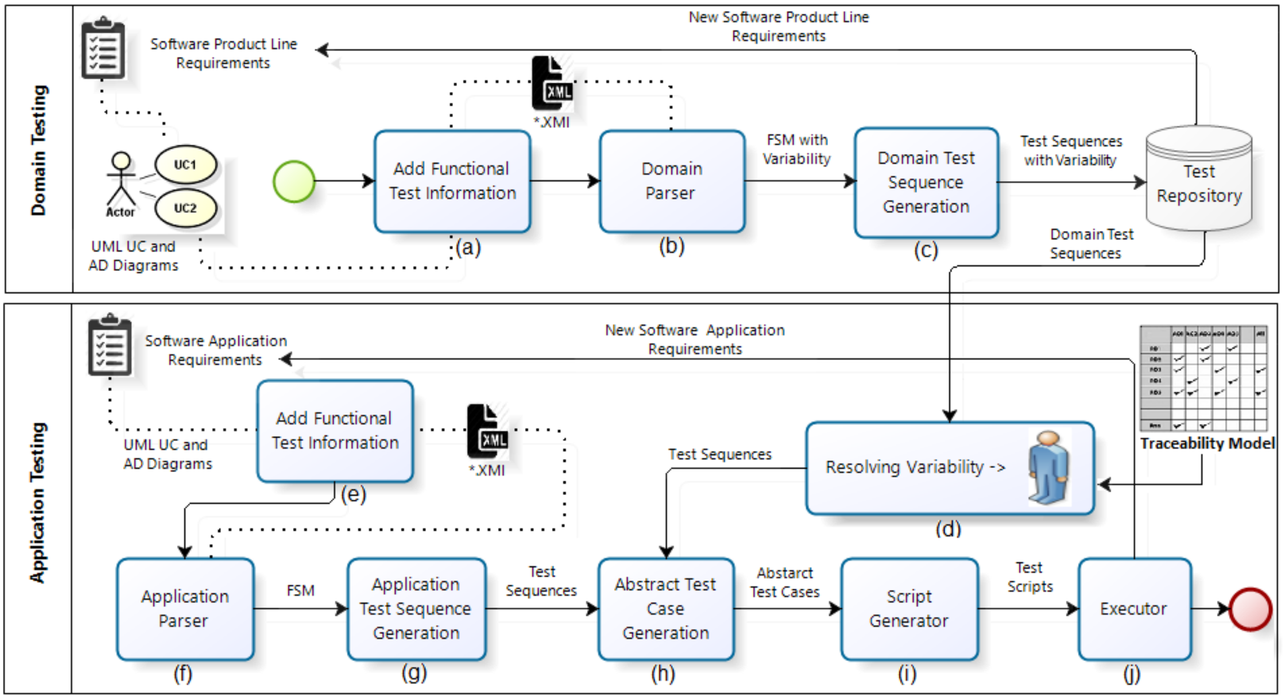


Figure 3.1: Split-MBT steps for generating functional test cases

in three steps: (see Figure 3.1): (a) `Add Functional Test Information`: consists of annotating, with test information, UML models describing variability information; (b) `Domain Parser`: extracts test and variability information from Activity Diagrams to generate FSMs, and; (c) `Domain Test Sequence Generation`: generates test sequences through applying test sequence generation methods that are extended to SPL context to deal with variability.

3.2.1 Add Functional Test Information

The first step of our method (see Figure 3.1 (a)) consists of manually annotating, with test information, an UML model previously designed with a variability management approach. This UML model is generated by an SPL analyst, that using a specific variability management approach, is responsible to extract information from the `SPL requirements` specification to design the model. SPLiT-MBt can be integrated to several variability management approaches, *e.g. Product Line UML-based Software Engineering* (PLUS) [Gom05], *Triangle Notation* [HP03] or *Stereotype-based Management of Variability* (SMarty) [OGM10].

As described in Section 2.3.1, we have adopted the SMarty approach, since it can be easily extended, it has a low learning curve, it supports the variability management for several UML models and, different from other approaches, it define a stereotype to represent inclusive variants. Figure 3.2 presents an example of UML Use Case and Activity Diagram describing variability information that was previously designed by the SPL analyst using the SMarty approach. Variation points in Activity Diagrams are identified in *DecisionNode* elements with the stereotype ≪`variationPoint`≫, as illustrated in the Figure 3.2. In that example, the variation point represents the variability Var1 (*name*), which requires the selection of at least one (*minSelection*) and up to three (*maxSelection*) variants, and that must be done at design time (*bindingTime*). When an SPL grows, this variability allows the addition of new variants (*allowAddingVar*). The variation point in question has three variants, which are represented by the *Action* elements S9, S2 and S8 (variants). They are annotated with the stereotype `alternative_OR`, indicating that a derived product may have at least one or even three of these variants representing features in its architecture. The other diagram elements are annotated with the stereotype `mandatory`, indicating that the features associated with these elements must be present in the configuration of all derived products.

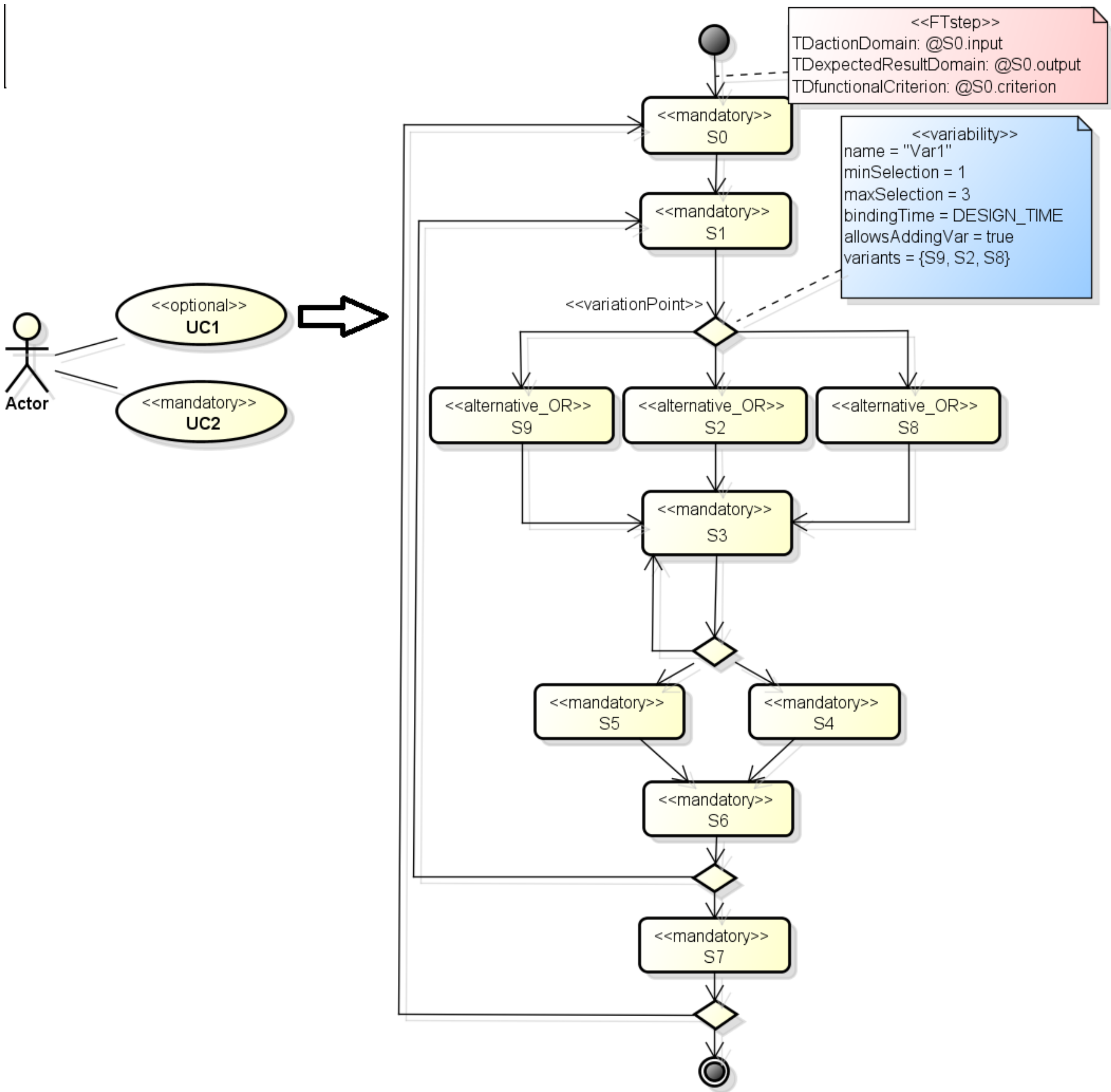


Figure 3.2: Domain Engineering testing model

Although SMarty approach could manage variability in different UML models, it is not able to represent test information, *e.g.*, input/output data to test the application functionalities. Therefore, to generate functional test cases for products derived from SPLs using SPLiT-MBt integrated to SMarty, it is still necessary to add functional test information on models representing the SPL functionalities, *i.e.* UML Activity Diagrams.

We have chosen UML Activity Diagrams as functional testing model, since they are the most widely used models to represent the behavior and functionalities of a system [BRJ05]. Basically, stereotypes and tagged values are, manually, added to these diagrams by the test analyst, and some specific UML elements are annotated with test information [RVZG10] [SRZ+11]. The use of stereotypes and tags allows to describe functional test information necessary to generate test scripts. Moreover, stereotypes and tagged values can be used to enhance the specification documents, improving the quality of models and test artifacts [AD97].

Based on the analysis of several scientific papers (see Chapter 2), as well as the *ad hoc* experience, observations and practices developed in our research group (we experimented some functional testing tools, *e.g.* QTP, RFT, Selenium, VS and MTM), we have defined one stereotype (*FTstep*) that has three tags (*TDactionDomain*, *TDexpectedResultDomain* and *TDfunctionalCriterion*) where test information is annotated. Their description is as follows: *FTstep*, stereotype annotated in the *ControlFlow* (transition) elements of an Activity Diagram. It has three associated tags: *TDactionDomain*, that specifies the action data to be performed by the user to test a specific functionality. Test information present in this tag is used to perform a specific system functionality; *TDexpectedResultDomain*, that specifies the expected result data used to check whether a specific functionality is working as described in the functional requirements. The information present in this tag is compared to the result obtained from the system execution for the data described in *TDactionDomain* tag; and *TDfunctionalCriterion*, that specifies information about the functional test criterion used to test some system functionality, *e.g.* Boundary Value Analysis or Equivalence Partitioning [MS04].

For instance, it is possible to use different functional testing criteria to test different functionalities of the same application. In a specific example, it would be possible to apply the Boundary Value Analysis criterion to test a specific functionality and to apply another criterion, such as Equivalence Partitioning, to test another functionality. Therefore, the tester/test analyst who is using SPLiT-MBt Tool could choose among different functional criteria to test different functionalities of the same system.

The test analyst could annotate the actual test data directly in a specific tag or use an external XMI file (for great volume of data) named *Functional Test Data* to describe the test information. In the case of using an external file, the tag describes only a reference to this file. For instance, `TDactionDomain` tag is annotated with a value corresponding the concatenation of two pieces of information: activity name and tag name (see Figure 3.2)[1]. On the other hand, the *Functional Test Data* file has the names of all activities of the diagram, the name of all tags and actual test data as well. Thus, it is possible to correlate the information described in the tags of the diagram with the actual data described in this file for later generation of test cases.

[1]All other transitions of the model have their transitions annotated with similar test information.

We consider the option of using an external file with actual test data information rather than only add them directly in the model, since the addition of such information directly in the tags is feasible up to a certain limit. Representing a great volume of data in a single tag may be difficult to visualize. Furthermore, the use of a structured file as an XML, provides greater clarity and understanding of the actual test data.

It is important to highlight that when the SPL evolves or changes, the test information annotated in these stereotypes and tags must be modified as well. An advantage is that the test information is updated in the SPL models through rewriting test information described in the tags and stereotypes. Thus, the new test cases are automatically generated based on the changes performed in the models. Therefore, it is possible to save time during the evolution of the SPL, since all test modification is earlier performed at the beginning of the modeling process. Hence, all test information is updated just once and there is no need to change the test information for each product of the SPL in a individually manner.

The process of adding tags, stereotypes and test information on the Activity Diagram is required to apply the other steps of the SPLiT-MBt method. At the end, the test analyst must export the models describing all test and variability information to an `XMI` file, which is the input of the next step of our method.

3.2.2 Domain Parser

This step (see Figure 3.1 (b)) consists of automatically extracting variability and test information from Activity Diagrams (`XMI` file) to generate a formal model, *e.g.* Finite State Machines (FSM). We have chosen FSMs, since they are among the most used formal models applied in MBT [CSV10]. Furthermore, FSMs are a good alternative to design software testing components, since they may be applicable in any specification model describing a finite number of states [EFW01] [Cho78] and also because FSMs are the most suitable models to generate sequences used as testing data input [Cho78]. The reason to generate FSMs is to apply test sequence generation methods (*e.g.*, UIO, HSI, DS) that are able to generate less amount of tests. The idea is to test products with less effort (time spent) as well as generating less test cases when comparing with other approaches. The problem is that FSMs were essentially designed to test software based on single system paradigm, and only few works [Gom05] [MRKN13] extend FSMs to an SPL context. Therefore, an extension of FSMs to represent variability information was required.

In order to deal with this issue, we have developed an approach that is integrated to SPLiT-MBt, which is able to generate extended FSMs from Activity Diagrams. Basically, that approach consists of converting information present in Activity Diagrams (`XMI` file) into FSMs with test and variability information (an example of an FSM generated from the Activity Diagram of the Figure 3.2 is shown Figure 3.3). Thus, all variability and test information from an Activity Diagram

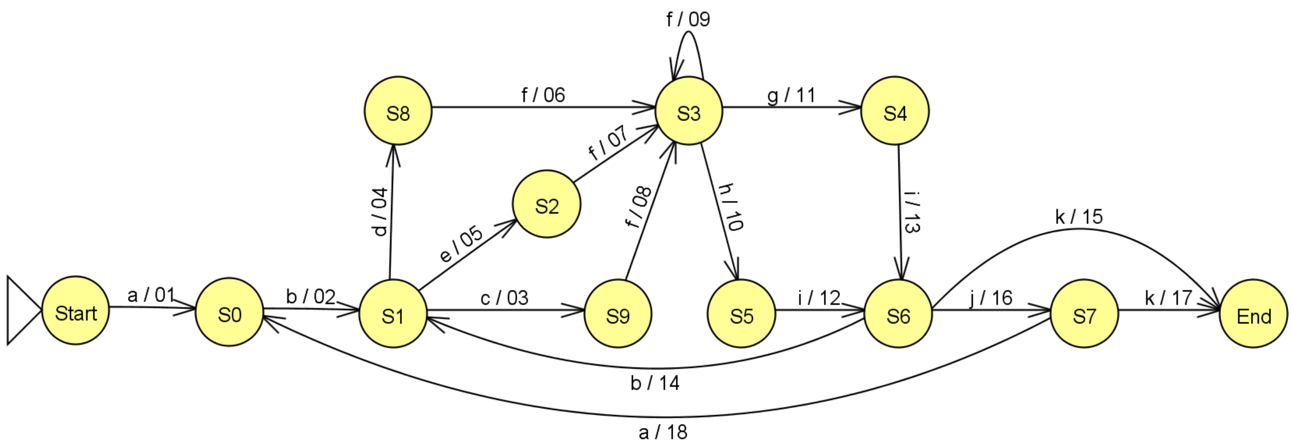


Figure 3.3: FSM generated from Activity Diagram of the Figure 3.2

is forwarded to the FSM. The conversion process just mentioned occurs according to the following criteria:

- The *InitialNode* and *ActivityFinal* elements from the Activity Diagram are converted into *Start* and *End* states;
- *Action* elements from the Activity Diagram are converted into corresponding states in the FSM;
- *ControlFlow* elements from the Activity Diagram are converted into corresponding transitions in the FSM;
- *DecisionNode* elements are not converted to a specific element in the FSM, since FSMs do not have a corresponding element. However, the transitions associated to this element are connected directly to the state (activity/element *Action*) that yielded the deviation.
- *DecisionNode* elements tagged with the ≪`VariationPoint`≫ stereotype are associated to the corresponding transitions in the FSM, *i.e.*, a variation point is represented by a numeric identifier (VPid) associated to its variants in the FSM (see Table 3.1 (b));
- Variability information (*e.g.*, `alternative_XOR`, `alternative_OR`, `optional`, `mandatory`) annotated in *Action* elements from the Activity Diagram are represented in the FSM transitions;
- Input information from the FSM transitions corresponds to the test input data and functional criterion in the Activity Diagram (see Table 3.1 (a));
- Output information from the FSM transitions describes several information, *i.e.*, source state, target state, expected result and variability information (see Table 3.1 (b)).

Based on the criteria described above, the Activity Diagram is converted to an FSM supporting variability information (see Table 3.1). This table presents input and output data information of the FSMs. The input data are described in Table 3.1 (a) and has three types of information: ID, which corresponds to the FSM input identifier (see Figure 3.3); Input, which corresponds to the input data used to test the system functionalities and; Functional Criterion, which corresponds to the functional criteria used to select test data. Similarly, the output data are represented in Table 3.1 (b) by a set of information: ID, which corresponds to the FSM output identifier; Output, which corresponds to the expected result related to a test performed to assess a specific system functionality; Variability, which corresponds to the variant type associated with a given state (Target State) and; information regarding the source and target states of the FSM. FSMs are the outcome of *Domain Parser* and it is used as input in the next steps of our SLPiT-MBt.

(a) FSM Input			(b) FSM Output				
ID	**Input**	**Functional Criterion**	**ID**	**Output**	**State Source**	**State Target**	**Variability**
a	S0.input	S0.criterion	01	S0.output	Start	S0	mandatory
			18		S7	S0	
b	S1.input	S1.criterion	02	S1.output	S0	S1	mandatory
			14		S6	S1	
c	S9.input	S9.criterion	03	S9.output	S1	S9	alternative_OR (VP1)
d	S8.input	S8.criterion	04	S8.output	S1	S8	alternative_OR (VP1)
e	S2.input	S2.criterion	05	S2.output	S1	S2	alternative_OR (VP1)
f	S3.input	S3.criterion	06	S3.output	S8	S3	mandatory
			07		S2	S3	
			08		S9	S3	
			09		S3	S3	
g	S4.input	S4.criterion	11	S4.output	S3	S4	mandatory
h	S5.input	S5.criterion	10	S5.output	S3	S5	mandatory
i	S6.input	S6.criterion	12	S6.output	S5	S6	mandatory
			13		S4	S6	
j	S7.input	S7.criterion	16	S7.output	S6	S7	mandatory
k	End.input	End.criterion	15	End.output	S6	End	mandatory
			17		S7	End	

Table 3.1: Input and Output information of the FSM from Figure 3.3

3.2.3 Domain Test Sequence Generation

Once the conversion from an Activity Diagram to an FSM containing variability and test information is performed, the next step (see Figure 3.1 (c)) consists of generating test sequences (*Domain Test Sequences*). Therefore, the FSM previously generated is used to create test sequences with variability information. These sequences are produced through the use of a specific test sequence generation method, *e.g.*, TT, UIO, DS, W or HSI. However, these methods are applied only in FSMs and are able to generate less test sequences. The idea is to test products with less effort (time spent), as well as to generate less test cases when comparing with other approaches. However, they had to be extended in an SPL context to handle variability information present in FSMs, since they were originally created to test applications developed from the single system paradigm, which make them inefficient to reduce the number of test cases to test products derived from SPLs. Therefore, one of the goals of this work is to investigate how different methods of generating test sequences existing in the literature can be adapted and/or applied to reduce the testing effort for applications derived from SPLs.

In a nutshell, in an SPL context, these methods must be able to determine the location of a variation point and then generate distinct sequences to test all variants associated to a particular variation point. Furthermore, these methods must be able to handle all variability information present in the FSMs, *e.g.* *Optional* and *Mandatory* variants as well dependency relationship (*requires*); mutually exclusive relationship (*mutex*) and; inclusive (*OR*)/exclusive (*XOR*) variants.

After performing a deep investigation based on the comparison of the HSI, UIO, TT, DS and W methods as well as identifying the characteristics that these methods have in common, we could realize that test sequence generation methods have several characteristics in common. For example, they work with a set of *partial test sequences*: *State Cover* (Q), *Transition Cover* (P), *Characterized Set* (W), *Identification Set* (Wi), *Harmonized Identifiers* (HI), *Unique sequence of input and output* (UIO). Some of these *partial test sequences* are gather together and the result corresponds to the final test sequence for each method. For instance, joining the Q, P and HI (partial sequences) produces as result the HSI final test sequence.

Therefore, based on that investigation, we concluded that the HSI method is the most suitable for our purpose. The reason for choosing this method was due to the fact that it is one of the least restrictive methods regarding the properties that FSMs must have. For instance, the HSI is able to interpret complete and partial FSMs [PYLD93]. Moreover, the HSI method allows full coverage of existing failures and it generates smaller test sequences than other methods, which contributes to a testing process optimization. These factors are very relevant in SPL context, because when an SPL grows, the number of test cases necessary to test SPL products could increase exponentially [ER11].

In order to generate test sequences from an extended version of the HSI, it is necessary to apply it considering variability information present in the FSM (see Figure 3.4). We have adopted the HSI to be integrated to SPLiT-MBt with the purpose of generating test sequences with variability information. However, as we mentioned earlier, we had to extend it to an SPL context. This adaptation of the HSI method is described as follows:

1. Variants can also be a variation point. In this case, FSM's states (variants) associated to a variant (that is also a variation point) are "separated" from the original FSM. In this context, a new FSM is produced, *i.e.* a sub FSM. Sub FSMs can be generated in another situation. For instance, they can also be created when exist "Nested Activities" in UML Activity Diagrams, *i.e.* when an action element (activity) of an Activity Diagram makes a reference to another Activity Diagram. It is important to highlight that a sub FSM will be replaced by a state that represents the sub FSM in the main FSM;

2. Variants associated to the same variation point are assumed to be a single state in the FSM. Furthermore, the input transition of this single state must have the input/output information

of all states (variants). This state is a concatenation of: *VP_* plus *a unique identifier*, *e.g.* VP_S1. This state represents a specific variation point and has information about its variants. This occurs because it is not possible to determine which variants associated to a particular variation point will be resolved, since a variation point is not resolved during Domain Engineering. Thus, a solution was to increase the abstraction level through defining a unique state representing the variants associated to a variation point. Therefore, the test sequence generation methods can be applied and still preserve variability, which will be resolved only in Application Engineering. A concrete example representing this situation is shown in Chapter 4;

3. After executing the criteria 1 and 2, it is necessary to generate a set of *partial test sequences*, which is performed by the application of a test sequence generation method under an FSM, *i.e.* HSI. As described earlier, the *partial test sequences* correspond to a set of sequences that are joined together to form the *final set of test sequence* of a specific test sequence generation method. For instance, the final test sequence generated by the HSI method is composed by the combination of three *partial test sequences*, *i.e.* *State Cover* (Q), *Transition Cover* (P) and *Harmonized Identifier* (HI). It is important to highlight that different from the traditional way to generate test sequences (that using the original version of HSI, for example), SPLiT-MBt uses the extended version of HSI, since this new version is able to handle variability information present in the FSMs.

4. In Domain Engineering the goal is to preserve variability. Therefore, variants having dependency relationship (*depends/requires*) or mutually exclusive ones (*excludes/mutex*) among themselves, as well as optional variants and variants that are part of a group of alternative inclusive variants (`alternative_OR`) or exclusive ones (`alternative_XOR`) are not considered when performing the methods, but will be resolved in Application Engineering, in which concrete test cases are derived to test specific products [CN01].

5. The output of this SPLiT-MBt step is a set of test sequences generated from the use of a chosen method, *i.e.* HSI. These sequences still contain variability information, which is represented by the following alphabet:

   - Op = represents optional variants;
   - VP_or = represents a variant that is part of a group of inclusive alternative variants (`alternative_OR`);
   - VP_xor = represents a variant that is part of a group of exclusive alternative variants (`alternative_XOR`);
   - {} = defines the set of variants associated to a variation point;
   - () = defines the set of test sequences generated by the application of a test sequence generation method under an FSM;
   - [] = defines the set of test sequences generated by the application of a test sequence generation method under sub FSMs;

- $\text{Req}_{->}$ = represents the dependency relationship (*depends/requires*) among variants;
- $\text{Ex}_{->}$ = represents the mutually exclusive relationship (*excludes/mutex*) among variants.

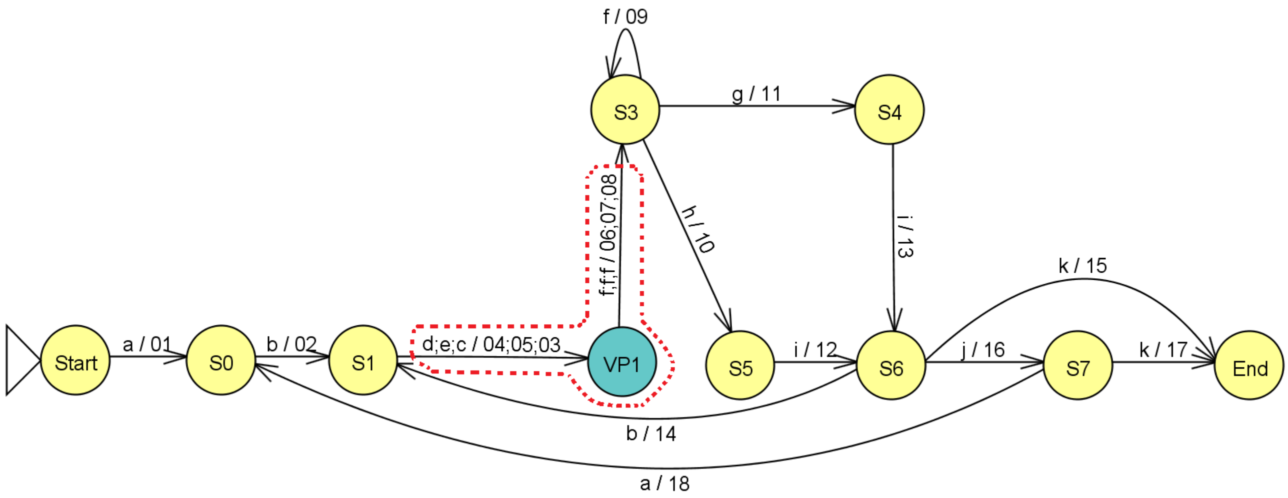


Figure 3.4: Adapted FSM

Although, we have adopted the HSI to generate test sequences with variability information, the process to extend other test sequence generation methods (*e.g.* W or UIO) in an SPL context are similar. Furthermore, it is important to highlight that through the mentioned process steps, it was possible to extend the HSI. This extended version is able to handle variability information present in the FSMs and then, generate test sequences with variability information[2].

Considering the extended version of HSI, which was based on these criteria, the process of generating test sequences for the FSM of Figure 3.4 resulted in the set of HSI test cases described in Table 3.2. As it is possible to realize, 7 test sequences preserving variability information were generated. An example of test sequences in which variability was resolved would be: *(abdfff, abdfgib, abdfgik, abdfhib, abdfhik, abdfgijk, abdfgijab)*. In order to generate these sequences, the variation point VP1 was resolved by selecting the input 'd' associated to the variant (state) S8.

ID	Set of Test Sequences
Sequence 1	ab$\{$d;e;c$\}_{VP_or}$fff,
Sequence 2	ab$\{$d;e;c$\}_{VP_or}$fgib,
Sequence 3	ab$\{$d;e;c$\}_{VP_or}$fgik,
Sequence 4	ab$\{$d;e;c$\}_{VP_or}$fhib,
Sequence 5	ab$\{$d;e;c$\}_{VP_or}$fhik,
Sequence 6	ab$\{$d;e;c$\}_{VP_or}$fgijk,
Sequence 7	ab$\{$d;e;c$\}_{VP_or}$fgijab

Table 3.2: Generated test sequences

Once the test sequences are generated, they are stored in a repository (`Test Repository`) to be later reused (when variability was already resolved) to test specific products during Application

[2]It is not the focus of our thesis to describe how those test sequence generation methods are adapted for an SPL context. A detailed explanation about how this adaptation process occurs can be found in [Zan16].

Engineering (see Figure 3.1). This repository stores test sequences generated based on information from several Activity Diagrams. Therefore, SPLiT-MBt allows to extract test information from Domain Engineering through applying the MBT technique in an SPL context for later generation of test cases and scripts during Application Engineering.

3.3 SPLiT-MBt during Application Engineering

During Domain Engineering (see Section 3.2), a set of test sequences that include variability information were generated. Therefore, during Application Engineering it is necessary to resolve the variability present in those test sequences and reuse them to generate test scripts to test specific products. These tasks are automatically performed in seven steps (see Figure 3.1): (d) `Resolving Variability`: resolve variability present in the test sequences that are stored in the `Test Repository`; (e) `Add Functional Test Information`: consists of annotating, with test information, UML models representing functionalities of a specific product; (f) `Application Parser`: extracts test information from Activity Diagrams to generate FSMs; (g) `Application Test Sequence Generation`: generates test sequences through applying conventional test sequence generation methods; (h) `Abstract Test Case Generation`: based on generated test sequences, a set of abstract test cases is derived; (i) `Script Generator`: generates scripts from the abstract test cases to be executed by a functional testing tool; (j) `Executor`: executes the test using a specific functional testing tool. These last three steps are similar to features from a product line called PLeTs - (Product Line of Model-Based Testing Tools [RVZG10].

3.3.1 Resolving Variability

This step (see Figure 3.1 (d)) describes how to resolve the variability present in the test sequences generated during Domain Engineering in order to reuse them to test specific products. To provide that, this step receives as input two types of artifacts, *i.e.* the test sequences stored in the `Test Repository` and a `Traceability Model` [3]. We assumed that this model was previously designed by the SPL analyst based on the SMarty approach.

The `Traceability Model` is described in a tabular format [MJG14], *i.e.* a matrix that makes a correlation between an SPL feature model and UML model elements, such as, use case and action elements from Activity Diagrams (see Table 3.3a). The lines of the matrix have information about the name of UML elements (*e.g.* UML use case and action elements), and the columns have information about the SPL features. The inter relationships between UML elements and features are marked in the matrix with a "blob" to determine the configuration information, *i.e.* the selected features for resolving variability. Thus, based on the association between the features and UML

[3] An example of a Traceability Model can be found in [Jun10] - Page 84.

elements it is possible to determine the UML elements that have their variability resolved, *i.e.* the models that represent actual products of an SPL.

Table 3.3: Traceability Model and Test Sequences

(a) Traceability Model

Activity Diagram (Action Elements/Input Data)	Features (From Feature Diagram)	
	Navigate	Shopping
Shopping Cart/e		*
Perform Search/y	*	
Order Status/w	*	

(b) Test Sequences

Test Sequences (From Test Repository)
ab{d;e;c}$_{VP_or}$fff
hi{j;k;l}V P _orppp
pq{r;s;t}V P _orvvv

When we have the information about the models with the resolved variability (provided by the `Traceability Model`) the variability present in the test sequences is resolved by crossing information present in the `Traceability Model` with the test sequences present in the `Test Repository` (see Table 3.3b). For instance, when considering the following hypothetical test sequence: (ab{d;e;c}$_{VP_or}$fff). It is possible to realize three inputs (d;e;c) associated to variants type `OR`. When this sequence is compared with information present in a `Traceability Model` (*e.g.* activity *shopping cart* with data input = 'e') it is possible to determine which variant (d;e;c) will be selected to resolve the variability present in that sequence. In this case, the selected variant was 'e' and as result we have the corresponding test sequence: *abefff*.

Moreover, we can also apply this approach to resolve variability present in the sequence "ab{d;e;c}$_{VP_or}$fff" from Table 3.2. When resolving the variability present in that sequence, two other test sequences are generated: abdfff and abcfff. After resolving the variability present in the sequences generated during Domain Engineering, eight test sequences were generated (see Table 3.4). These resolved test sequences can be converted into equivalent test cases in natural language, *i.e.*, abstract test cases. This activity is performed by the `Abstract Test Case Generation` step.

Finally, it is important to notice that, although it has not been addressed in the above example, this step is also responsible for resolving dependency (*depends/requires*) and exclusion (*excludes*) relationships among variants, as well as optional variants. This resolution is also performed by crossing variability information present in the `Traceability Model` with the test sequences present in the `Test Repository`. More details on how that situation can be applied is shown in Chapter 4.

ID	Test Sequences
Sequence 1	abdfff,
Sequence 2	abcff,
Sequence 3	abdfgib,
Sequence 4	abdfgik,
Sequence 5	abdfhib,
Sequence 6	abdfhik,
Sequence 7	abdfgijk
Sequence 8	abdfgijab

Table 3.4: Generated test sequences

3.3.2 Add Functional Test Information

This step (see Figure 3.1 (e)) has some similarities when comparing to its equivalent step from Domain Engineering, *i.e.* test information are added to UML models. However, the model depicts functionalities of a specific product. For example, during Application Engineering could be necessary to add a specific functionality to an existing product when a new version of this product is required. As this functionality is specific to a single product, it is, usually, not added to models during Domain Engineering. In this context, a new model representing the specific functionalities for a particular product must be designed from scratch during Application Engineering. In this SPLiT-MBt step, we assumed that these models (Activity Diagrams) were previously designed based on information extracted from `Software Application Requirements`. Thus, a test analyst is only responsible for annotating, with test information, those UML models. This annotation process is almost the same as that performed during Domain Engineering. Therefore, the same tags used to annotate *ControlFlow elements* (transitions) during Domain Engineering are used at this step, *i.e. TDactionDomain*, *TDexpectedResultDomain* and *TDfunctionalCriterion*. The difference is that, during Application Engineering, the Activity Diagrams have no variability information.

When the Activity Diagrams is fully annotated, with test information, the test analyst must export the models to an `XMI` file, which is the input of the next step of our method.

3.3.3 Application Parser and Application Test Sequence Generation

The `Application Parser` and `Application Test Sequence Generation` steps (see Figure 3.1 (f-g)) are very similar to their equivalent ones from Domain Engineering, just differing in some aspects. For instance, the `Application Parser` is applied to models that contain only test information, since the variability has been previously resolved. Figure 3.5 depicts an Activity Diagram describing the functionalities of *Use Case 3* (UC3), *i.e.* functionalities of a specific product.

Therefore, the `Application Parser` receives, as input, an `XMI` file describing test information related to the Activity Diagram from Figure 3.5 and then one FSM is generated (see

Figure 3.6). This formal model is the input of the `Application Test Sequence Generation` step, from which are applied a specific test sequence generation method, *i.e.* traditional HSI. In this step, there is no need to adapt the HSI, since there is no variability information to be handled. Therefore, a set of test sequences is, automatically, generated through applying the HSI under FSMs describing functionalities of a specific product. Table 3.5 shows the generated test sequences, using HSI, for the FSM from Figure 3.6. These sequences are the input of the next step of SPLiT-MBt.

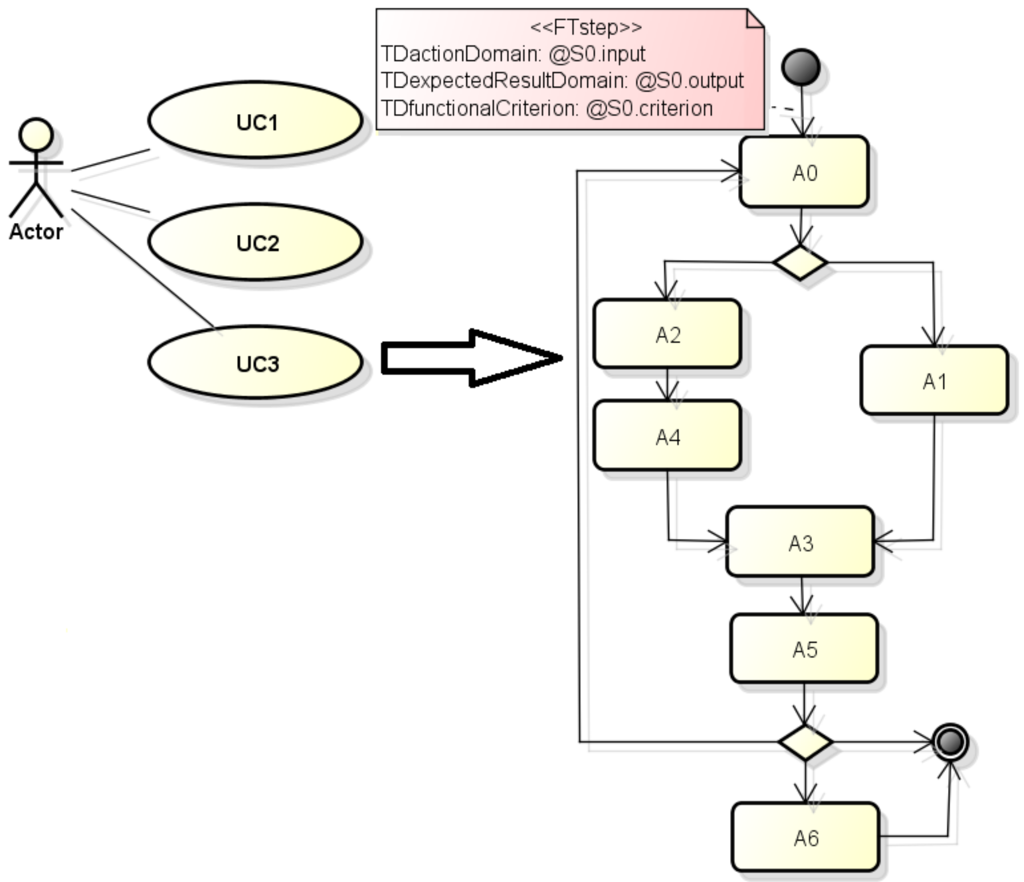


Figure 3.5: Test model of a specific product

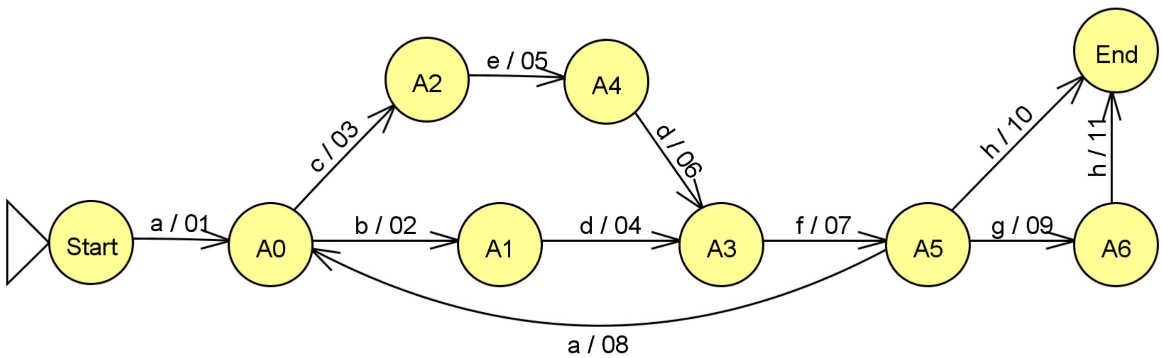


Figure 3.6: FSM generated from the Activity Diagram of Figure 3.5

ID	Test Sequences
Sequence 1	aced,
Sequence 2	abdfh,
Sequence 3	abdfa,
Sequence 4	abdfgh

Table 3.5: Generated test sequences

3.3.4 Abstract Test Case Generation

This step (Figure 3.1 (h)) aims to, automatically, convert the test sequences, generated in `Application Test Sequence Generation` and `Resolving Variability` steps (see Figure 3.1)), into abstract test cases. For each test sequence, the corresponding abstract test case is generated. An abstract test case is a text file structured in a technology independent format that describes the activities to be performed by the user (or a tool) during the interaction with the System Under Test (SUT). It uses the test data to define the user or, their actual data inputs/outputs and the functional criteria. These data input/output and functional criteria present in the abstract test case corresponds to the test information, previously, added to the Activity Diagrams and propagated during the steps of SPLiT-MBt.

The motivation for generating abstract test cases is that they can be reused to, automatically, produce scripts to several functional testing tools, *e.g.* MTM, VS, QTP or RFT. Thus, SPLiT-MBt provides greater flexibility by allowing that products derived from an SPL can be tested using different testing tools. For instance, consider an IT company that has adopted SPLiT-MBt to test the products of its SPL. This company can be motivated by technical or managerial decision to easily migrate from a testing tool A to a testing tool B without the need to manually create new scripts. Thus, all test cases and scripts previously created can be reused. Furthermore, the abstract test case has a clear representation where the test data are presented in a high level language.

Figure 3.7 presents an abstract test case generated from a set of test sequences in the previous step, *i.e.*, abdfgik. Each element of this sequence has information related to the input data (`TDactionDomain`) and output (`TDexpectedResultDomain`), as well as definition of the functional test criterion (`TDfunctionalCriterion`) used for selecting test data. The input/output information and functional criteria present in the abstract test case have actual test data, which were extracted from the file *Functional Test Data* mentioned in Section 3.2.1.

In the example presented in Figure 3.7, the Boundary Value Analysis criterion was used in all functionalities that will be tested. The motivation for choosing this criterion is because it is one of the most known in the literature and can be easily automated. This criterion defines test data to the limits of a range of values and data preceding and succeeding this interval. For instance, considering the range of values [21; 100] (Figure 3.7 - 1. S0), the following data to test a specific functionality must be set: 20; 21; 100; 101. Although the Boundary Value Analysis criterion has been used as an example, other criteria, such as Equivalence Partitioning, could be applied.

```
#Test Case: Sequence 4 - abdfgik
1. S0
<<TDactionDomain = [21; 100] >>
<<TDexpectedResultDomain = "Age must be in the range between 21 and 100 years old">>
<<TDfunctionalCriterion = Boundary Value Analysis>>
2. S1
<<TDactionDomain = ["a"; "abcdefghij"]>>
<<TDexpectedResultDomain = "The field must have at least 1 and at most 10 characters">>
<<TDfunctionalCriterion = Boundary Value Analysis>>
3. S8
<<TDactionDomain = [1; 1638]>>
<<TDexpectedResultDomain = "The font size must be a value between 1 and 1638">>
<<TDfunctionalCriterion = Boundary Value Analysis>>
4. S3
<<TDactionDomain = [1930; 2005]>>
<<TDexpectedResultDomain = "Birth date accepts values from 1930 to 2005">>
<<TDfunctionalCriterion = Boundary Value Analysis>>
5. S4
<<TDactionDomain = [R$ 1.000,00; R$ 85.000,00]>>
<<TDexpectedResultDomain = "Authorized lending to values between
R$ 1.000,00 and R$ 85.000,00">>
<<TDfunctionalCriterion = Boundary Value Analysis >>
6. S6
<<TDactionDomain = [0; 5] >>
<<TDexpectedResultDomain = "Age must be in the range between 0 and 5 years old">>
<<TDfunctionalCriterion = Boundary Value Analysis>>
7. End
<<TDactionDomain = [12; 24]>>
<<TDexpectedResultDomain = "Contract period from 12 to 24 months">>
<<TDfunctionalCriterion = Boundary Value Analysis>>
```

Figure 3.7: Abstract test case generated from the test sequence 4 of Table 3.4

Finally, once the abstract test cases were generated, they are instantiated to concrete test cases, *i.e.*, test scripts.

3.3.5 Script Generator and Executor

This step (`Script Generator` - see Figure 3.1 (i)) consists of, automatically, creating scripts based on the abstract test cases generated in previous step. It is a tool-dependent step, since the scripts are "strongly" associated to a specific functional testing tool, *i.e.* MTM. In this context, SPLiT-MBt supports the instantiation of abstract test cases into test scripts to be used to the test execution. Although we have chosen the MTM, SPLiT-MBt allows the integration with other functional testing tools, *e.g.* VS and QTP and RFT.

The test scripts generated by SPLiT-MBt Tool have a tabular format. These scripts are imported by a testing tool, *e.g.* MTM, for the test execution. Figure 3.8 shows a script with test information generated from the abstract test case illustrated in Figure 3.7. In this example, it is possible to see that the values of the fields `TDactionDomain`, `TDexpectedResultDomain` correspond respectively to the input data and expected results present in cells `Action/Description` and `Expected Results`. Some data of the `Action/Description` cell were set based on functional

	A	B	C	D	E	F
1	**Test case #**	**Work Item ID**	**Test Title**	**Test Step**	**Action/Description**	**Expected Result**
2	TC001	Test Case 259121	Sequence 4	1	S0 - 20, 21, 100, 101	"Age must be in the range between 21 and 100 years old"
3				2	S1 - "", "a", "abcdefghij", "abcdefghijk"	" The field must have at least 1 and at most 10 characters"
4				3	S8 - 0, 1, 1638, 1639	"The font size must be a value between 1 and 1638"
5				4	S3 - 1929, 1930, 2005, 2006	"Birth date accepts values from 1930 to 2005"
6				5	S4 - R$ 999,99, R$ 1.000,00, R$ 85.000,00, 85.000,01	"Authorized lending to values between R$ 1.000,00 and R$ 85.000,00"
7				6	S6 - -1, 0, 5, 6	"Age must be in the range between 21 and 5 years old"
8				7	End - 11, 12, 24, 25	"Contract period from 12 to 24 months"

Figure 3.8: Script generated from the abstract test case of Figure 3.7

criteria Boundary Value Analysis. The other information set in `Test Case #`, `Work Item ID` and `Test Title` cells are generated automatically and correspond respectively to the test case name, test case identifier and test title.

Finally, once the test scripts are generated, the SPLiT-MBt prototype tool performs its last functionality (`Executor` - see Figure 3.1 (j)), which aims to launch the testing tool and to start the test execution. This initialization consists of an internal system call, where through the SPLiT-MBt prototype tool interface the user (tester) provides the testing tool installation path and the scripts path. Thus, the testing tool test environment is initialized and the test can be finally performed.

3.4 Chapter Summary

In this chapter, we presented the SPLiT-MBT method, which is based on the MBT technique to generate functional test cases and scripts for products derived from SPLs. In the SPLiT-MBT, test artifacts developed during Domain Engineering are reused to test products during Application Engineering. In order to provide this reuse, SPLiT-MBt is applied in two phases. First, test information are annotated in UML models to generate test sequences with variability during Domain Engineering. During Application Engineering, the variability in these sequences is resolved and then, they are converted into abstract test cases, from which test scripts are generated. In Chapter 4, we demonstrate how SPLiT-MBt Tool could be used to provide test automation of SPL products in two example of uses.

4. EXAMPLES OF USE

"In a word, never let go on these three things: faith, hope and love. And know that the greatest of these will always be love."

St. Paul

Based on our method SPLiT-MBt, we developed a tool named SPLiT-MBt Tool. This chapter describes how SPLIt-MB Tool was applied to generate test cases for products that could be derived from two SPLs, *i.e.* an academic SPL named Arcade Game Maker (AGM) SPL [Ins16b] and an actual one named Product Line of Testing Tools (PLeTs) [RVZG10]. PLeTs was developed in the context of a collaboration project between our university and a global IT company. Therefore, the main goal is to demonstrate that our method is able to generate reusable test artifacts to assess both PLeTs' and AGM's products.

4.1 Arcade Game Maker (AGM) - AGM

AGM was developed by the Software Engineering Institute (SEI) with the purpose of assisting the learning of the SPL concepts through a practical approach. This SPL could be used to derive three different electronic games, *i.e.* *Bowling*, *Brickles* and *Pong*, which are used by the scientific community to assess and to validate their approaches [OGM10] [FGMO12] [MSM04]. In order to test the functionalities of the AGM's products, we have generated test artifacts (using SPLiT-MBt Tool) that were reused to generate scripts to test the common functionalities among these products.

4.1.1 Modeling the UML Diagrams of AGM

In order to generate test artifacts (using our SPLiT-MBt Tool) to test the AGM's products, it is necessary to annotate, with test information, Use Case and Activity Diagrams that were previously designed by the SPL analyst with the purpose to describe the AGM's functionalities.

As described in Chapter 3, the the SPL analyst also annotates variability information in these models using the SMarty approach, while test information was annotated (using SPLiT-MBt) through the `TDactionDomain`, `TDexpectedResultDomain` and `TDfunctionalCriterion` tags. Figure 4.1 shows the Use Case Diagram of AGM [Jun10], which represents the user's actions and the product functionalities of this SPL. This diagram has two actors and twelve use cases describing several operations that can be performed by the user/player, such as install and uninstall games

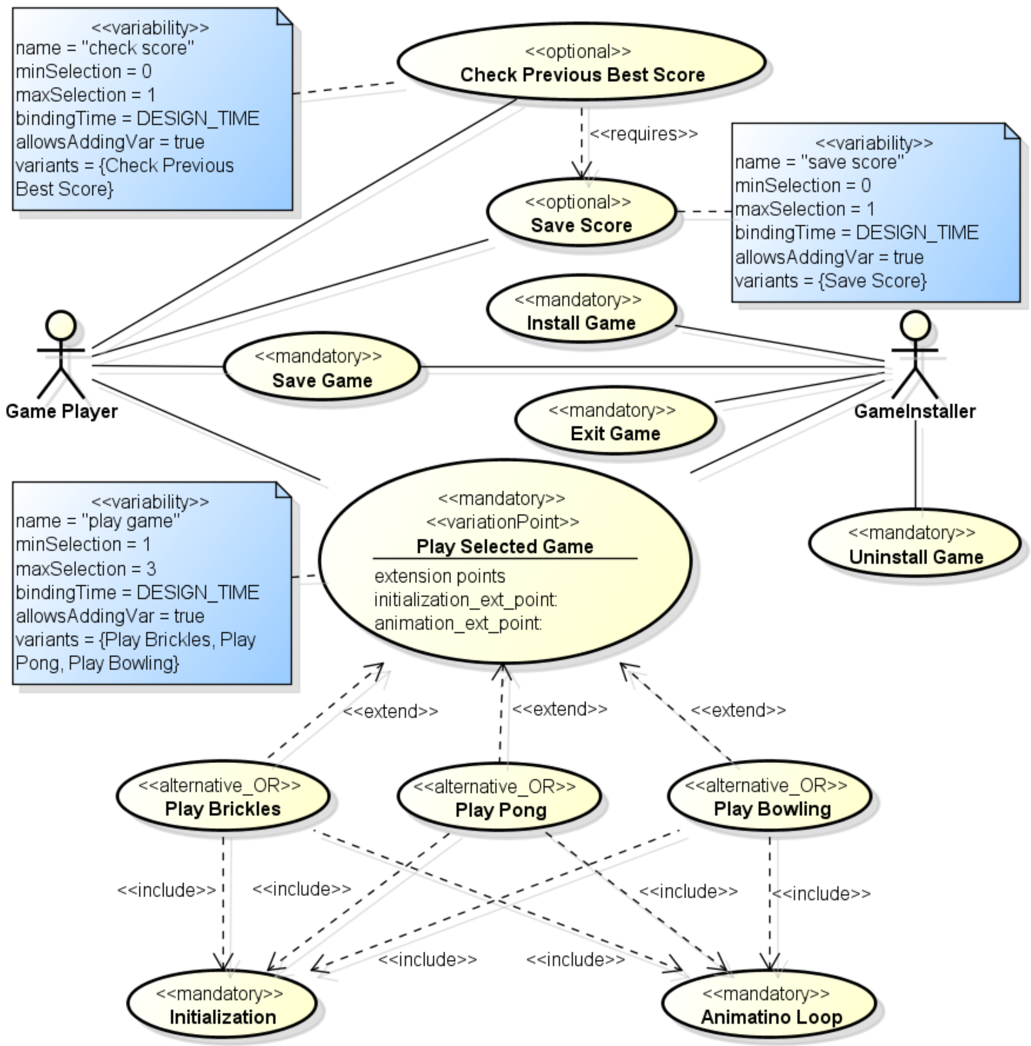


Figure 4.1: Use Case diagram of AGM [Jun10]

(*Install Game* and *Uninstall Game*), to select the game to be played (*Play Selected Game*), to save the current score of a player (*Save Score*), check the recorded best score (*Check Previous Best Score*) and end an ongoing game (*Exit Game*).

The *Play Selected Game* use case was chosen as an example to show how the test case generation is performed using SPLiT-MBt and how these test artifacts are reused to test the product functionalities. The motivation for choosing this use case is that it is what best represents user interactions with the system, even showing the player actions during a game session. From this use case, an Activity Diagram was derived (see Figure 4.2), which describes the player interactions with the AGM products. First, the player selects the *Play* option in the menu (*Select Play*) in order to initialize a game. Then the player clicks the left mouse button and start a game session (*Initialize the game*). Next, the player clicks the left mouse button or use the keyboard to send commands to the selected game (*Brickles Moves*, *Pong Moves* or *Bowling Moves*). At the end of each game session, the player answers a dialog box (*Responds to Won/Lost/Tied Dialog*) and decides whether to restart or end the game.

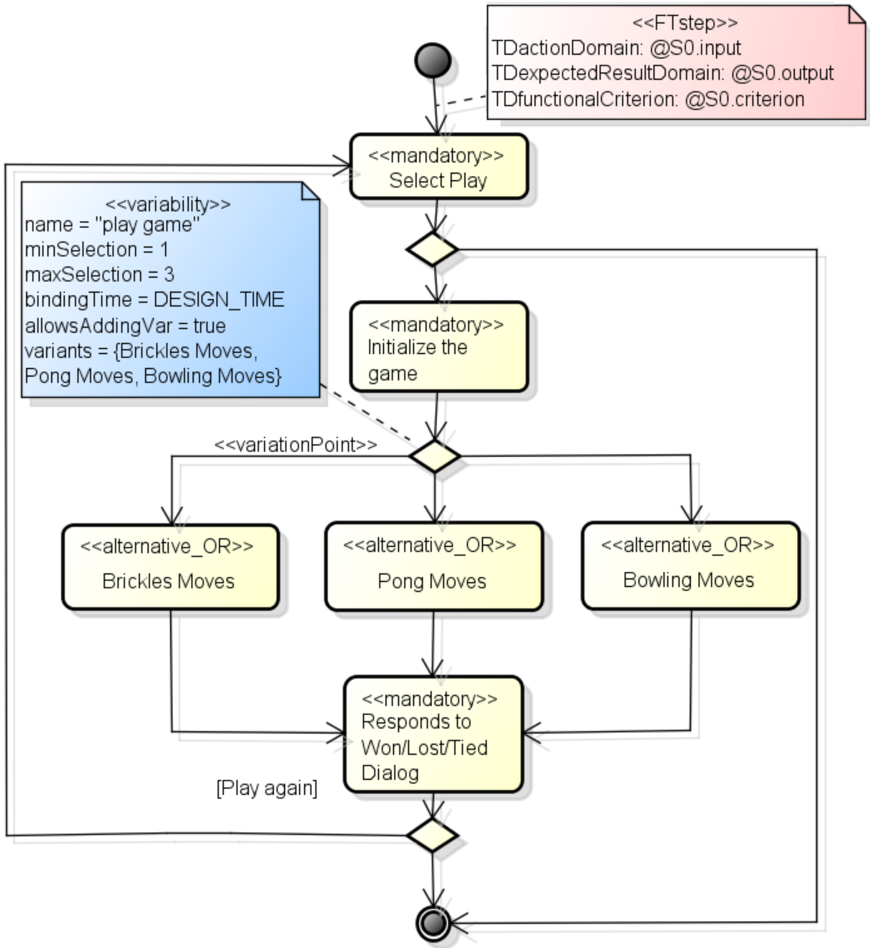


Figure 4.2: Activity Diagram of the use case *Play Selected Game*

The *Select Play*, *Initialize the game* and *Responds to Won/Lost/Tied Dialog* activities represent mandatory variants (`mandatory`), *i.e.*, they have to be present in all generated products. The *Brickles Moves*, *Pong Moves* and *Bowling Moves* activities, on the other hand, correspond to inclusive variants (`alternative_OR`), since a product derived from AGM can have the combination of one, two or even three games. Each one of these three activities has a link to another corresponding diagram, which describes, in details, the player actions during a game session. These other three Activity Diagrams can be seen in Figure 4.3.

4.1.2 Deriving Test Scripts Using the SPLiT-MBt Tool

In order to generate test scripts to test the AGM products, first, test and variability information annotated in the models were exported to an XMI file. This step was performed using the Astah Professional modeling tool [Pro16]. SPLiT-MBt Tool parses this file to convert it into an FSM, in which the adapted version of the HSI method was applied. When applying this method twelve sequences containing test and variability information were produced, and ten of them are sequences generated from sub FSMs, which represent the Activity Diagrams depicted in Figure 4.3.

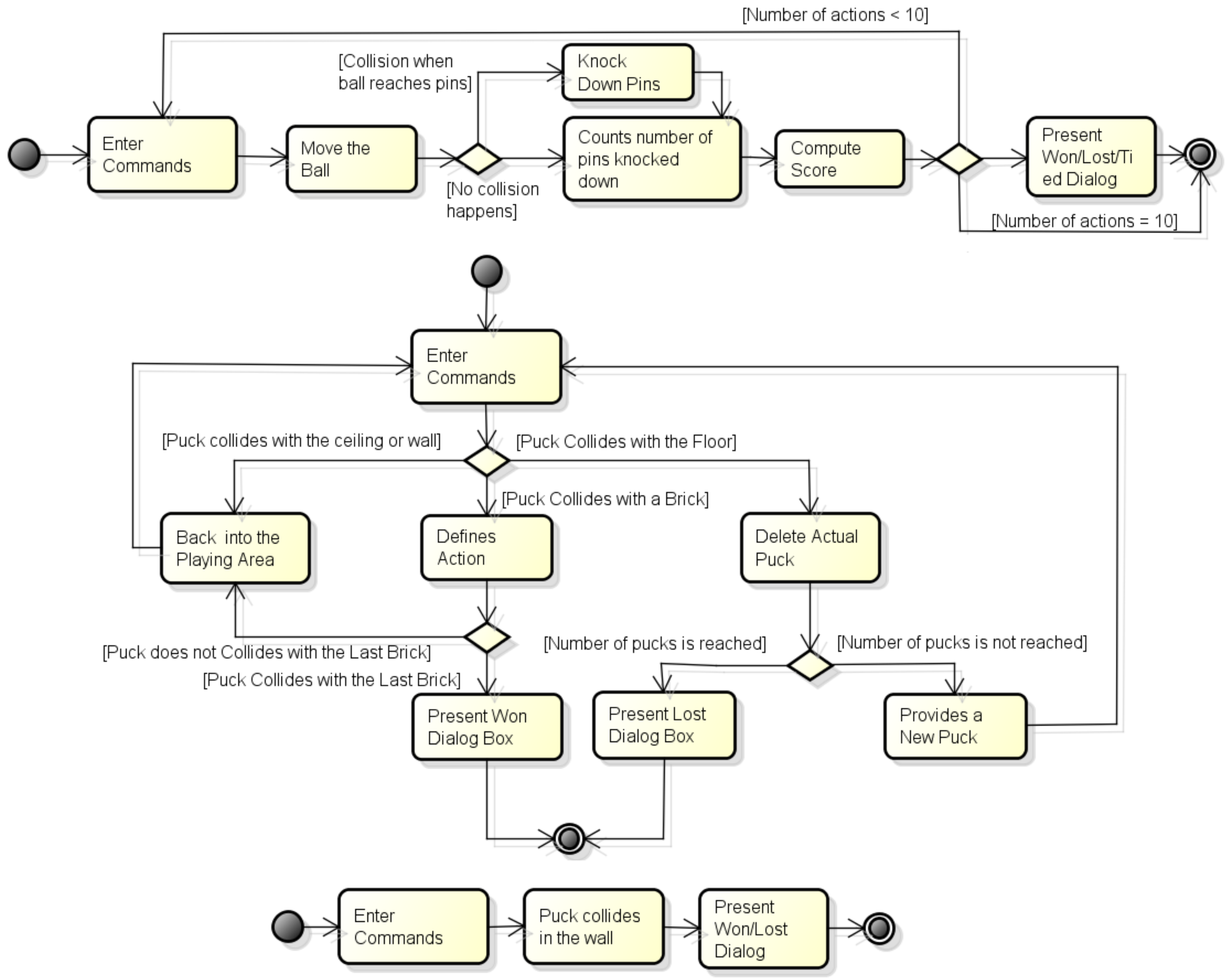


Figure 4.3: Activity Diagrams of the activities *Bowling Moves*, *Brickles Moves* and *Pong Moves*

Finally, these sequences are stored in the repository for later script generation during Application Engineering.

During Application Engineering, seven products were derived. These products represent a combination of one, two and three AGM games and they were generated when variability was resolved based on information present in `Traceability Model`. The information present in `Traceability Model` is used as reference to resolve the variability present in the test sequences stored in the `Test Repository`. Then, during Application Engineering, a set of test sequences is reused and converted into an equivalent description to test cases in natural language, *i.e.*, abstract test cases.

Finally, the abstract test cases were instantiated to concrete test cases, *i.e.*, test scripts to be executed by the functional testing tool MTM (see Section 3.3.5). Figure 4.4 shows a script describing the test cases used to test some functionalities of the *Brickles* game. It is important to note that no functional testing criteria was used, since for each tested functionality, only one input was set and not a data domain as presented in Section 3.3.4.

	A	B	C	D	E	F
1	**Test case #**	**Work Item ID**	**Test Title**	**Test Step**	**Action/Description**	**Expected Result**
2	TC001	Test Case 2514	Play Brickles Sequence 3	1	Select Play - "Select Play from menu"	"System starts the game"
3				2	Initialize the game - "The player clicks the left mouse button and the game is started"	"System starts the game action"
4				3	Enter Commands - "Player clicks the left mouse button or uses the keyboard to enter commands"	"System moves the paddle horizontally to follow the mouse track"
5				4	Defines Action Puck collides with a brick	"Puck is absorbed or changes direction according to the laws of physics"
6				5	Present Won Dialog Box - "Puck collides with the last brick"	"The Won dialog is presented"
7				6	Responds to Won/Lost/Tied Dialog - "Responds to Won/Lost/Tied dialog"	"Dialog to play again is presented"

Figure 4.4: Generated script to test the functionalities of *Brickles* game

4.1.3 Analysis

When variability was resolved, 20 test sequences were generated. During Application Engineering, these sequences were reused to test the functionalities of 7 products. These products represent the combination from one to three games and the number of test sequences reused for these products are depicted in Table 4.1. As shown in this table, the number of test sequences reused among those products was equal to 80. Based on this number, the reuse percentage was obtained through a metric called *Size and Frequency metric* (R_{sf}) [DKMT96]. Considering this metric, the reuse percentage of generated test sequences is given by:

$$R_{sf} = \frac{Size_{sf} - Size_{act}}{Size_{sf}} = \frac{80 - 20}{80} = 0.75$$

Product ID	**Games that Compose the Products**	**Number of Derived Test Sequences**
Product 1	*Brickles*	10
Product 2	*Bowling*	8
Product 3	*Pong*	2
Product 4	*Brickles* and *Bowling*	18
Product 5	*Brickles* and *Pong*	12
Product 6	*Bowling* and *Pong*	10
Product 7	*Brickles*, *Bowling* and *Pong*	20
		Total = 80

Table 4.1: Generated sequences for each product

This value determines that 75% of test sequences[1] generated during Domain Engineering are reused to test the functionalities of 7 products derived from AGM. Therefore, for this specific example, SPLiT-MBt allowed the test artifacts generation with a considerable reuse percentage.

[1]The test sequences with variability, test sequences with variability already resolved, abstract test cases and test scripts can be found in the appendix of this thesis.

Thus, it demonstrates a possible gain when comparing with approaches that do not consider reuse as a strategy for generating test artifacts, where test cases are individually generated for each product. Furthermore, the use of test sequence generation methods, such as HSI contributes to an effort reduction in the test activity, since they allow full coverage of the failures. These features are essential in the SPL context, because an SPL growing (the increasing of variabilities) has influence in the amount of tests needed to validate the product quality. Finally, SPLiT-MBt allows the test script generation for several functional testing tools through the abstract test cases. This feature contributes to a greater flexibility, allowing test execution independent of technology.

4.2 Product Line of Testing Tools - PLeTs

PLeTs was designed and developed to automate the generation of MBT Tools (products) [RVZG10] [SRZ+11] [CCO+12]. These testing tools automate the generation of test cases based on the system models, *i.e.* products derived from PLeTs accept a system model as an input and generate test cases. Actually, PLeTs could be used to generate MBT tools that perform three type of tests, *i.e.* Performance, Functional or Structural Testing. Table 4.2 summarizes the main functionalities of PLeTs products, which are used to explain our method.

Table 4.2: PLeTs SPL Requirements

ID	Requirement	Description
RF-01	Choose Type of Test	The system should allow the user to select the type of testing that will be performed.
RF-02	Functional	The MBT tools must support automatic generation of testing data for functional testing. These tools must support integration with other functional testing tools.
RF-03	Performance	The MBT tools must support automatic generation of testing data for performance testing. These tools must support integration with other performance testing tools.
RF-04	Structural	The MBT tools must support automatic generation of testing data for structural testing. These tools must support integration with other structural testing tools.
RF-05	Functionalities Functional	The MBT tools must allow users to: create a log file, edit an configuration file and close the system interface.
RF-06	Functionalities Performance	The MBT tools must allow users to: set performance test environment (scripts and scenarios), and test case generation.

4.2.1 Add Test Information to SPL Models

Based on PLeTs requirements presented in Table 4.2, the SPL analyst has to design the Use Case and Activity Diagrams that describe the functionalities of the PLeTs products. Thus, the SPL analyst has to build these diagrams and add variability information in accordance with the

SMarty approach. Figure 4.5 presents an Use Case model with one actor and six use case elements describing several operations that can be performed by the user, such as: select the type of testing to be executed (*Choose Type of Test*); to perform one of three types of test (*Functional*, *Performance* or *Structural*).

The PLeTs *Structural* use case element (see Figure 4.5) was chosen as an example to demonstrate how the test case generation is performed using SPLiT-MBt and how these test artifacts are reused to test the product functionalities. The reason for choosing this use case element is that it is decomposed into an Activity Diagram (see Figure 4.6) that presents more variability elements, *e.g. optional* and *mandatory* variants; dependency relationship (*requires*); mutually exclusive relationship (*mutex*) and; inclusive variants (*alternative_OR*)[2].

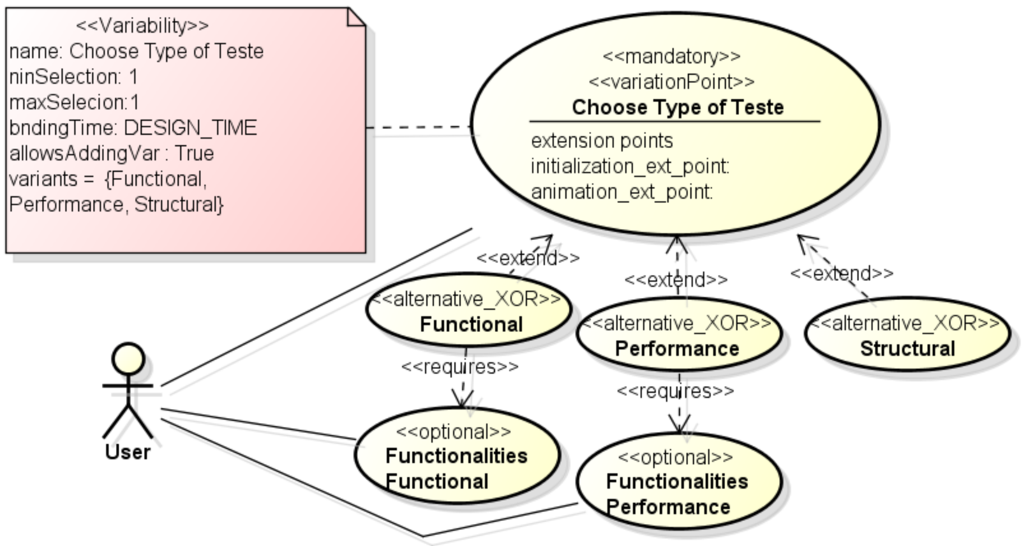


Figure 4.5: PLeTs Use Case model

This PLeTs Activity Diagram describes the user interactions with some PLeTs products, specifically, those used to perform structural testing. First, the user must import an XMI file (*Load XMI File*) that has information related to structural testing. Next, the user click on "Parser button" (*Submit the XMI File to a Parser*) in order to generate test cases. Then, the user chooses the path (*Type the Path to Save Abstract Structure and Data File*) where the test cases will be saved (*Saving the Abstract Structure and Data File*), and selects the tool that will perform the structural testing (*Informing the Tool Path*). Next, the user generates test scripts for one of the three available tools (*Export to JaBUTi*, *Export to PokeTool*, *Export to Emma*). Finally, the user chooses a directory to save the scripts (*Save Scripts*), executes the test and presents the test results (*Test Results*). If the user generates test cases for JaBUTi, a project file will be created (*Save Project File .jbt*), since this tool needs to create an additional configuration file.

The *Load XMI File*, *Submit the XMI File to a Parser*, *Type the Path to Save Abstract Structure and Data File*, *Saving the Abstract Structure and Data File*, *Informing the Tool Path*,

[2]We have generated test cases for all Activity Diagrams related to the other use case elements. These diagrams and all test artifacts generated in this Example of Use (*e.g.* test sequences with variability, test sequences with variability already resolved, abstract test cases and test scripts) can be found in the appendix of this thesis.

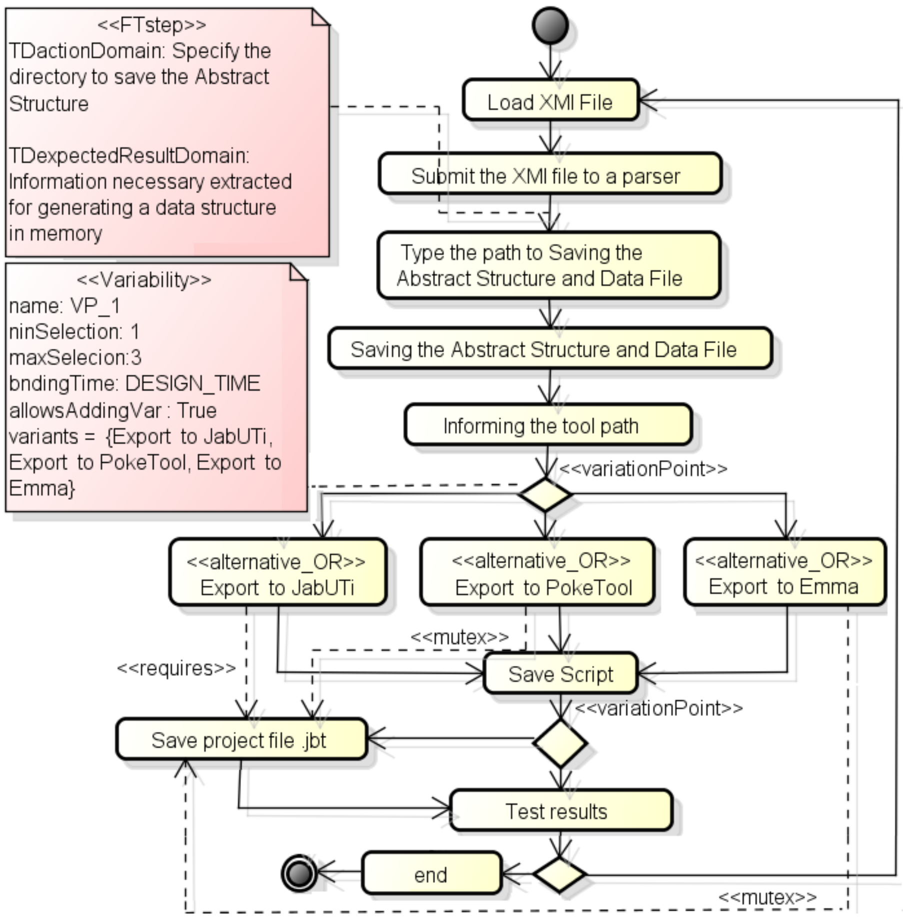


Figure 4.6: Activity Diagram of PLeTs Structural Tools

Save Scripts and *Test Results* activities represent mandatory variants (`mandatory`), *i.e.* they have to be present in all generated products. The *Export to JaBUTi*, *Export to PokeTool*, and *Export to Emma* activities, on the other hand, correspond to inclusive variants (`alternative_OR`), since a product derived from PLeTs may have the combination of one, two or even three structural testing tools. This Activity Diagram has also dependency and mutually exclusive relationships, *i.e. Export to JaBUTi* requires the *Save Project File .jbt* activity, while the *Export to PokeTool* and *Export to Emma* will exist in a product configuration only whether the *Save Project File .jbt* activity will not be selected to make part of a specific product.

It is important to highlight that when the Use Case and Activity Diagrams have been modeled and the variability information has been added to these models, our method can be applied. As described in Chapter 3, the first SPLit-MBt step consists of annotating *ControlFlow* elements in Activity Diagrams, with test information, through the use of `TDactionDomain` and `TDexpectedResultDomain` tags and their respective tagged values.

Therefore, to make it clearer, we inserted a note element, in the Activity Diagram depicted in Figure 4.6, to show some tagged values annotated in a *ControlFlow* element. In this example, all tags we have defined for our method and their corresponding tagged values are shown. Each of these tags has a value that is bounded to the transition between the *Load XMI File* and *Submit the XMI File and Save* activities.

4.2.2 Generate Test Sequences with Variability

After the SPL analyst modeled and exported the PLeTs models to an XMI file[3], this file must be loaded using the SPLiT-MBt Tool and then, seven FSMs, with variability information, are generated. Figure 4.7 shows an example of FSM, which depicts information related to input/output information. Input (*e.g.* bs, bt, bu) corresponds to the input data used to test the system functionalities, and output (*e.g.* 72, 73, 74) corresponds to the expected result. This information is just a set of identifiers and the actual test data is described in a table that can be found in the appendix of this thesis. This table also presents information regarding the source/target states of the FSM, as well as variability, which corresponds to the variant type associated with a given state (Target State).

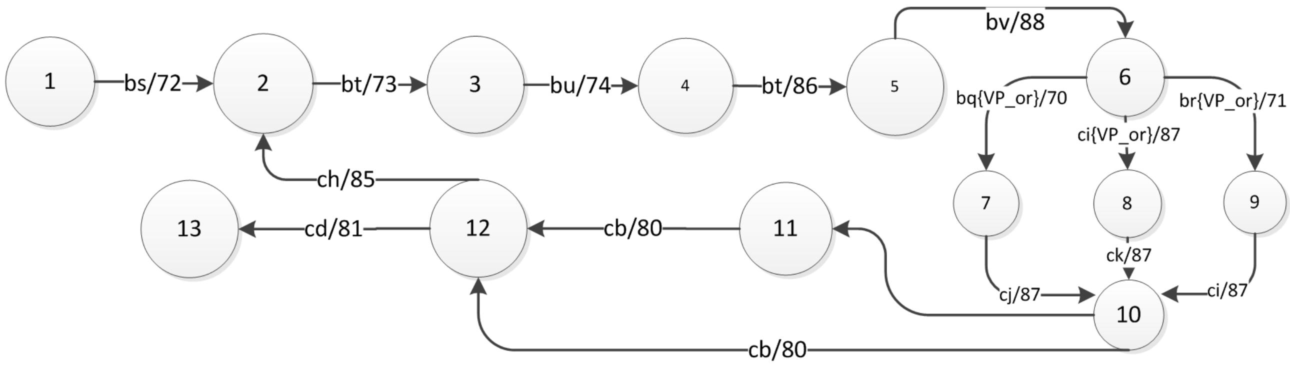


Figure 4.7: FSM with variability information

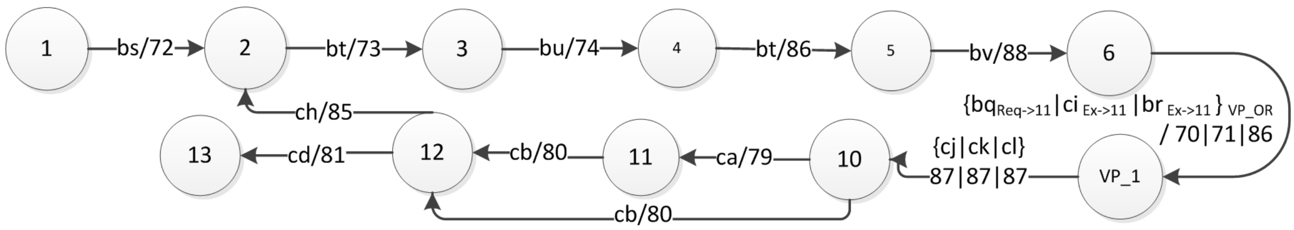


Figure 4.8: FSM with a state representing a variation point

Considering that FSM with variability, the HSI method must generate test sequences during Domain Engineering. In order to make it possible, the states (variants) associated to the same variation point are assumed to be a single state in the FSM (VP_1), in which the input transition of this state must have the input/output information of all states (variants).

As a result of this step, the FSM depicted in Figure 4.7 is converted into the FSM in Figure 4.8. It is possible to notice, in this FSM, that the input transition of **VP_1** has input and output information from the states (variants) *Export to JaBUTi*, *Export to PokeTool*, and *Export to Emma* as well as information related to dependency ($bq_{req->11}$) and mutually exclusive ($ci_{ex->11}$, $br_{ex->11}$) relationship.

[3]Most of the UML modeling environments export models to an XMI file. Here we used Astah Professional modeling tool [Pro16]

Table 4.3: Sample of test sequences Q, P, HI and HSI

State	**State Cover (P)**
3	ϵ, bs, bt
VP_1	ϵ, bs, bt, bu, bv{bq;ci;br\}$_{VP\_or}$
11	ϵ, bs, bt, bu, bv{bq;ci;br\}$_{VP\_or}$, {bs;bs;bs},ca
State	**Transition Cover (Q)**
3	ϵ, bs, bt, bu
VP_1	ϵ, bs, bt, bu, bv,{bq;ci;br\}$_{VP\_or}$, {bs;bs;bs}
11	ϵ, bs, bt, bu, bv,{bq;ci;br\}$_{VP\_or}$, {bs;bs;bs},cb
State	**Harmonized Identifier (HI)**
2	bt
4	bt
11	null
HSI Final Test Sequence: bs,bt,bu ,bt,bv,{bq$_{req->11}$; ci$_{ex->11}$; br$_{ex->11}$\}$_{VP\_or}$,{bs;bs;bs},ca, cb	

Once the FSM is modified, the HSI method will generate test sequences for FSMs with transitions containing a set of inputs instead of transitions with just one input, as those used to test single applications. In order to apply the modified HSI, we must consider and adapt the three steps used by this method to generate partial test sequences, *i.e.* Q, P and HI. These steps produce, as result, a set of partial test sequences that are combined with each other to compose the HSI final test sequence. Table 4.3 presents a sample of test sequences with variability information, in which we describe the partial test sequences generated through applying P, Q and HI as well as HSI final test sequence. The test sequences are stored in a repository to be resolved during Application Engineering.

4.2.3 Resolving Variability

Through applying the adapted HSI for all FSMs (seven FSMs) that correspond to the entire PLeTs functionalities in Domain Engineering, 18 test sequences with variability information were generated. During Application Engineering, when the variability is resolved through the use of the `Traceability Model`, 3,257 test sequences were produced. In order to illustrate an example of test sequence with variability already resolved, we consider the following test sequences: "*bu bt bv ci bs cb cd*" and "*bu bt bv br bs cb cd*". These sequences were generated when the variability present in the test sequence "*bs bu bt bv* $\{bq_{req->11};ci_{ex->11};br_{ex->11}\}_{VP\_or}$ *{bs;bs;bs} cb cd*" was resolved. All the 3,257 test sequences were reused to test the functionalities of 336 products derived from PLeTs and based on these numbers, the reuse percentage was obtained by a metric called *Size and Frequency* (R$_{sf}$) [DKMT96]. Considering this metric, the reuse percentage for the generated test sequences is given by:

$$R_{sf} = \frac{Size_{sf} - Size_{act}}{Size_{sf}} = \frac{3,257 - 18}{3,257} = 0.99$$

This value determines that 99% of test sequences generated in Domain Engineering were reused to test the functionalities of all products derived from PLeTs. Therefore, the SPLiT-MBt allowed the test artifacts generation with a considerable reuse percentage for this example. Thus, it demonstrates a possible gain when comparing with approaches that do not consider reuse as a strategy for generating test artifacts, where test cases are individually generated for each product. Furthermore, the use of methods for generating test sequences, such as HSI contributes to an effort reduction in the test activity, since they allow full failures coverage. These features are essential in the SPL context, because the increasing of variabilities has influenced the amount of tests needed to validate the quality of products. SPLiT-MBt can also be useful to adapt several test sequence generation methods, from which test sequences are converted into abstract test cases and test scripts.

4.2.4 Abstract Test Case Generation

The test sequences generated in the last step were converted into an equivalent description to test cases in natural language, *i.e.* abstract test cases. Figure 4.9 presents an abstract test case generated from a set of test sequences in the previous step, *i.e.*, "*bu bt bv ci bs cb cd*". Each element of this sequence has information related to the input data (`TDactionDomain`) and output (`TDexpectedResultDomain`), as well as definition of the functional test criterion (`TDfunctionalCriterion`) used for selecting test data. The input/output information and functional criteria present in the abstract test case have actual test data, which corresponds to the test information present in the Activity Diagram from Figure 4.6. This abstract test case represents user activities (*e.g.* 1. Load XMI File and 2. Submit the XMI file to a Parser) and its related tagged values are presented between double angle quotation marks (*e.g.* ≪`TDactionDomain`≫: "Press Enter").

In the example presented in Figure 4.9, no functional testing criteria was used, since for each tested functionality, only one input was set and not data domain (data set). It is important that SPLiT-MBt supports the use of the Boundary Value Analysis criterion to select a set of test data, since it is one of the most-known criteria in the literature and can be easily automated. It is important to highlight that SPLiT-MBt allows the use of other criteria, such as Equivalence Partitioning. Finally, once the abstract test cases are generated, they are instantiated to concrete test cases, *i.e.*, test scripts.

```
#Abstract Test Case: Structural - bu bt bv ci ck cb cd
1. Load XMI File
<<TDactionDomain = "Type the path of the XMI file on
console" >>
<<TDexpectedResultDomain = "File XMI loaded">>
2. Submit the XMI file to a Parser
<<TDactionDomain = "Press Enter">>
<<TDexpectedResultDomain = "Information necessary
 extracted for generating a data structure in memory">>
3. Type the path to Saving the Abstract Structure and
Data File
<<TDactionDomain = "Specify the directory to save
the Abstract Structure">>
<<TDexpectedResultDomain = "Directory where the
abstract data is saved is displayed on console">>
4. Saving the Abstract Structure and Data File
<<TDactionDomain = "Press Enter">>
<<TDexpectedResultDomain = "Data File and Abstract
Structure saved">>
5. Informing the tool path
<<TDactionDomain = "Inform the launcher path of
Jabuti, EMMA or Poketool">>
<<TDexpectedResultDomain = "Path informed">>
6. Export  to PokeTool
<<TDactionDomain = "Click on Poke-Tool application
located on c:/Poketool.exe">>
<<TDexpectedResultDomain = "PokeTool application
opened">>
7. Save Script
<<TDactionDomain = Select directory to save java class
for poketool">>
<<TDexpectedResultDomain = "Java class is saved">>
8. Test results
<<TDactionDomain = "Application will open on screen">>
<<TDexpectedResultDomain = "Tests results on screen">>
9. end
<<TDactionDomain = "Press on Close">>
<<TDexpectedResultDomain = "Application is closed">>
```

Figure 4.9: Snippet of an abstract test case

4.2.5 Test Script Generation and Test Execution

The next step to be performed by SPLiT-MBt is to instantiate scripts to MTM from the abstract test cases previously generated. As described in Chapter 3, the scripts generated by the SPLiT-MBt Tool have a tabular format and for this Example of Use 3,257 scripts were automatically generated. Figure 4.10 shows an snippet script with test information generated from the abstract test case illustrated in Figure 4.9. In this example, it is possible to notice that the values of the fields `TDactionDomain`, `TDexpectedResultDomain` correspond respectively to the input data and expected results present in cells `Action/Description` and `Expected Results`. The information present in `Action/Description` cells corresponds to the test data.

Action/Description	Expected Results
Load XMI File "Type the path of the XMI file on console"	File XMI loaded
Submit the XMI file to a Parser "Press Enter"	Information necessary extracted for generating a data structure in memory
Type the path to Saving the Abstract Structure and Data File "Specify the directory to save the Abstract Structure"	Directory where is saved the abstract data structure is displayed on the console
Saving the Abstract Structure and Data File "Press Enter"	Data File and Abstract Structure saved
Informing the tool path "Inform the launcher path of Jabuti, EMMA or Poketool"	Path informed
Export to PokeTool "Click on Poke-Tool application located on c:/Poketool.exe""	PokeTool application opened
Save Script "Select directory to save java class for poketool"	Java class is saved
Test results "Application will open on screen"	Tests results on screen
End "Press on Close"	Application is closed

Figure 4.10: Script to test the functionalities of a PLeTs product

Finally, once the test scripts were generated, we use the SPLiT-MBt Tool to perform the test execution. Thereunto, we have used our tool to launch the interface of the MTM, load the scripts previously generated and start the test execution. This initialization consists of an internal system call, where through the SPLiT-MBt Tool interface[4] we provided the scripts and the MTM installation path. Thus, the MTM's environment is initialized and the test is performed.

4.3 Chapter Summary

In this chapter, we presented two SPLs (*i.e.* an academic SPL named AGM and a actual one named PLeTs) from which we generate reusable functional test cases and scripts using our tool, *i.e.* SPLiT-MBt Tool. This tool was developed from the concepts and features created for our method, *i.e.* it is an instance of the SPLiT-MBt. We also showed that our method is able to generate, during Domain Engineering, test sequences with variability information from UML models previously designed by the SPL analyst. Then, when variability is resolved, these test sequences are reused, during Application Engineering, to generate scripts to test products derived from AGM and PLets.

One of the main advantages of our method is related to the possibility of reusing test information described in SPL models to generate, during Domain Engineering, test sequences using an extended version of a test sequence generation method, *i.e.* HSI. This extended version is able to handle variability information present in FSMs and then, generate test sequences with variability information. Moreover, using the extended version of the HSI it is possible to reduce the amount

[4]A picture of SPLiT-MBt Tool interface can be found in the appendix of this thesis.

of test cases (since it is a feature inherent to HSI and one of the purposes of this method has been created) providing full coverage of the product functionalities.

Another advantage of SPLiT-MBt is related to the possibility of generating, during Application Engineering, an abstract structure that can be used to generate test scripts to different functional test technologies, such as QTP, RFT, Selenium, VS and MTM. Therefore, a company that is using tool A can, motivated by a technical or managerial decision, easily change to a testing tool B without having to create new test cases. Hence, SPLiT-MBt provides benefits not only during Domain Engineering, but also during Application Engineering when the SPL products can be tested using the functional test technology available for a specific company.

Finally, our method provides a considerable reuse percentage of the test artifacts generated for both Examples of Use, *i.e.* AGM and PLeTs. Therefore, we can claim that SPLiT-MBt is, in these specific contexts, a useful method to provide: reusable test artifacts, full functional test coverage, and flexibility to generate test scripts for different test technologies. In Chapter 5, we demonstrate how we apply an controlled experimental study to compare our SPLiT-MBt with other similar approaches.

5. EMPIRICAL EXPERIMENT

"The most beautiful act of faith is the one made in darkness, in sacrifice, and with extreme effort."

St. Pio of Pietrelcina

It is well known that software testing is a costly, time-consuming and critical activity to the success of software projects. Therefore, all applications must pass to a rigorous validation process to ensure the desired quality level. This statement is also true when considering products derived from a Software Product Line (SPL). Hence, the test of applications based on SPLs is even more important, since, as mentioned in Chapter 1, a fault not found in a given software component can generate hundreds of products with failures. The problem is that software testing approaches found in the literature define techniques, processes and methods to test applications individually, and the are no evidence of studies comparing the approaches and methods to test products derived from SPLs. In order to overcome this issue, we conducted a controlled experiment with the purpose of demonstrating the performance of our SPLiT-MBt against two other methods, one is similar to ours and the other one refers to the conventional way to test single applications.

This empirical experiment is organized as follows. In Section 5.1 the experiment definition. Section 5.2 describes the instruments used for this experimental study. Section 5.3 presents the experiment planning, the research question, the hypotheses and variables. Moreover, we present in this section, the design of our experiment and the threats to the experiment validity. In Section 5.4 we present the preparation and the experiment execution. Section 5.5 we describe the results of this study. Finally, Sections 5.6 and 5.7 we present the analysis of our findings and the conclusions we have drawn from the experiment results.

5.1 Definition of the Experimental Study

The motivation of our controlled experiment is to evaluate the effort when applying functional testing (at System Level) to verify the functionalities of products derived from an specific SPL. For this purpose, we analyzed three different methods: SPLiT-MbT (see Chapter 3); Customizable Activity Diagrams, Decision tables and Test specifications (CADeT) [OG09]; and Microsoft Test Manager (MTM) [Man16]. The first two methods are oriented to reuse of test cases; and the third one consists of testing each product individually, *i.e.* the conventional way to test single applications. In a nutshell, our main goal with this study is to answer the following question:

"What is the effort to apply functional testing for products derived from SPLs when using SPLiT-MBt, MTM and CADeT?"

5.2 Experiment Instruments

In order to perform our experimental study, we defined four experiment instruments, *i.e.* SPLiT-MBt, CADeT, MTM and System Under Test (SUT). Next, we briefly introduce them.

SPLiT-MBt: our method to generate test cases for SPLs. Chapter 3 has the description of this instrument;

CADeT: is a functional test design method for SPLs that applies feature-based test coverage criteria together with a variability management approach named *Product Line UML-based Software Engineering* (PLUS) [Gom05] to create reusable test cases for an SPL. PLUS supports the variability management in several UML models, *e.g.* Use Case and Activity Diagrams. Reusable test information are annotated in Activity Diagrams and mapped to decision tables. Moreover, information regarding variability and all variability relationships are also described in decision tables, which will be later analyzed to apply a specific feature-based test coverage criterion to the SPL. Based on this coverage criteria, representative products configurations are generated to cover all features, all relevant feature combinations of an SPL, and then test information are generated to cover the functionalities of each product;

Microsoft Test Manager (MTM): is a functional testing tool used to perform manual or automated functional testing. The use of MTM can bring several advantages, *e.g.* centralize the project, write and execute tests and facilitate the execution of user interactions with the system under test. It allows testers to perform testing using the Visual Studio interface or command line. Furthermore, it is also possible to execute tests using Team Foundation Build. In order to perform a manual or automated testing, it is necessary to execute the tests using a test plan. The test plan has test data information and could be described in a excel file, which is imported by the MTM that perform the test execution. For this experimental study, we compare the three methods through analyzing the time spent by a tester when using each method.

We defined two SPLs as SUT: an academic and an actual SPL, *i.e.* Arcade Game Maker (AGM) [Ins16b] and Product Line of Model-Based Testing Tools (PLeTs) [RVZG10]. A full description of these two SPLs can be found in Sections 4.1 and 4.2.

5.3 Experiment Planning

In this section, we describe how we plan our experiment, as well as introduce the research question, its hypotheses and variables. Moreover, we present how we selected the subjects, the experiment design and the threats to the experiment validity.

5.3.1 Context Selection

The context of our experiment was characterized according to four dimensions:

Process: in our experiment, we used an *in-vitro* approach, since it refers to the experiment in the laboratory under controlled conditions. Our experiment is not an industrial SPL testing, *i.e.*, it is off-line;

Participants: we invited doctoral, master and undergraduate students from Computer Science courses;

Reality: our experiment addresses a real problem, *i.e.*, the differences in individual effort to create functional test cases to test products derived from AGM and PLeTs when using SPLiT-MBt, CADeT and MTM;

Generality: it is a specific context, since the tool we have used during the experiment is a testing tool generated from the concepts of our method, *i.e.* SPLiT-MBt Tool.

5.3.2 Hypothesis Formulation

In this section, we present our hypothesis and also define the measures used to evaluate them. Informally, we define our hypotheses as follows:

1. To apply functional testing for products derived from SPLs using MTM (conventional way to test single applications) needs more effort when compared to approaches/methods that provide reuse of test artifacts when performing that type of testing, *e.g.*, SPLiT-MBt and CADeT.

2. Moreover, our SPLiT-MBt also provides some advantages related to effort when compared to the CADeT method.

Based on those informal hypothesis, we can formally state them. Furthermore, we also define the measures we have used to evaluate the hypotheses. For each hypothesis, we have defined the following notation:

ϕ_{spt}: it represents the effort when using SPLiT-MbT to apply functional testing for products derived from SPLs.

ϕ_{cad}: it represents the effort when using CADeT to apply functional testing for products derived from SPLs.

ϕ_{mtm}: it represents the effort when using MTM to apply functional testing for products derived from SPLs.

Our Research Question (RQ) is **"What is the effort to apply functional testing for products derived from SPLs when using SPLiT-MBt, MTM and CADeT?"**. And our hypotheses are:

H_0: the effort is the same when using SPLiT-MbT, MTM and CADeT to apply functional testing for products derived from SPLs.

H_0: $\phi_{spt} = \phi_{cad} = \phi_{mtm}$

H_1: the effort is different when using SPLiT-MbT, MTM or CADeT to apply functional testing for products derived from SPLs for at least one pair of these methods.

H_1: $\phi_{spt} \neq \phi_{cad}$ or

$\phi_{spt} \neq \phi_{mtm}$ or

$\phi_{cad} \neq \phi_{mtm}$

5.3.3 Variables Selection

In this section, we present the dependent and independent variables used to represent the treatments of our experiment and their measured values as well (see Table 5.1). The independent variables represents the treatments and correspond to the variables whose results and behavior must be evaluated. The dependent variable represents the effort (time spent) when using the three SPL testing methods. This measured value is used to describe the effectiveness of the methods.

Table 5.1: Scales of Experiment Variables

Experiment variables		
Variable type	**Variable name**	**Scale type**
Independent	SPL Testing Method	Nominal
Control	Subjects Experience	Ordinal
Control	Degree of Formal Education	Ordinal
Dependent	Effort	Ratio

Independent and Dependent Variables

The independent variable of interest in our experiment corresponds to the choice of a SPL testing method. It has a nominal scale and can assume one of three values: SPLiT-MBt, CADeT or MTM. The dependent variable we have defined for this experiment is **effort**. It is measured as the amount of time spend by SPL testers to generate functional test cases for products derived from SPLs.

Control Variables

For our experiment, we have defined two control variables, *i.e.*, degree of formal education and the subjects experience in functional testing, UML notation and Software Product Lines. The degree of formal education and the subjects experience corresponds to blocking variables and they were defined to reduce sources of variability, which contributes to improve the experiment precision.

5.3.4 Selection of Subjects

The subjects selection was defined in accordance with the availability of academic students. We invited doctoral, master and undergraduate students to participate in our experiment as subjects. The undergraduate students were third year, or later, students from Computer Science. They are students from the PUCRS[1] university and each subject has different experience knowledge, *e.g.*, experience in the industry as software analyst, software tester or as developer, or just experience developing software in an Computer Science undergraduate course.

5.3.5 Experiment Design

The experiment design addressed the following general principles:

Randomization: The subjects were randomly allocated to each SPL testing method, *i.e.* SPLiT-MBt, or CADeT or MTM. Moreover, as all subjects would execute all treatments (SPLiT-MBt, CADeT or MTM - *Randomized complete block design*), we always randomly defined their execution sequence.

Blocking: as we mentioned earlier, the subjects we have selected for our experiment have different background in functional testing, UML notation and Software Product Lines. Therefore, in order to minimize the effect of those differences, we classified the subjects into three groups according to their skills (Beginner, Intermediate and Advanced groups - see Table 5.2). In order to define whether a subject is beginner, intermediate or advanced, we have applied a characterization questionnaire, prior to the experiment, to quantify the subject background on functional testing, UML modeling notation and Software Product Lines.

Balancing: the subjects were randomly grouped into each group (randomized block design). Therefore, each SPL method is performed by the same number of subjects (SPLiT-MBt, CADeT or MTM).

Standard design types: The design type aims to evaluate whether the values of ϕ_{spt}, ϕ_{cad} and ϕ_{mtm} are different for similar values of μ_{spt}, μ_{cad} and μ_{mtm}. Therefore, it is necessary to

[1]http://www.pucrs.br

compare the three treatments against each other. According to [WRH+00], the *One Factor with more than Two Treatments* design type must be applied. The factor corresponds to the SPL testing method we have defined for this experiment and; the treatments corresponds to the SPLiT-MBt, CADeT and MTM methods. The response variable is measured on a ratio scale, which allow us to rank the measured items and to quantify and compare the differences among them.

5.3.6 Instrumentation

According to [WRH+00], there are three types of instruments, *i.e.*, objects, guidelines and measurement instruments. Next, we describe each one of these instruments:

Objects: the experiment objects are test artifacts generated by the subjects to test PLeTs' and AGM's products. We also provide other documents for the experiment execution, *e.g.*, SPL requirements, UML models (previously designed) describing SPL functional requirements and variability information and test specification. Moreover, we provided a UML modeling tool named Astah Professional to assist the subjects for adding test information on the use cases and activity diagrams;

Guidelines: the methods and the tools were presented to the subjects through a printed manual. It presents an overview of the methods and detailed instructions on how to apply them to add test information to test products derived from SPLs (SUTs). Moreover, we performed a training phase in a laboratory room for all the experiment subjects. During the training, the subjects could ask open questions about the methods and the processes to add test information using the three methods described in the manual. It is important to mention that in the training phase, a SPL different from the experiment was used when adding test information. Furthermore, questions and answers were shared among all the subjects in the training room. During the experiment execution, a printed guide was used by the subjects. It includes the steps to add test and the related information about the processes for generating test cases using the three methods, *i.e.*, SPLiT-MBt (a guideline for modeling SPL functional using Astah UML tool was also included), CADeT and MTM.

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

5.3.7 Threats to Validity

An experimental process must clearly identify the concerns about the different types of threats to the experiment validity [WRH+00]. It is important to mention that there are some factors that contribute to mitigate the threats in an experiment process. According to [CC79], these factors helps to a further experiment analysis by researchers, and also contributes to simplify the experiment replication. The author relates/associates these factors to different classification schemes for several types of threats to validate the experiment. For our experiment, we adopted a classification scheme that is divided in four types of threats:

Conclusion validity: This is a threat that may affect the experiment when we are drawing conclusions related to the treatment and their outcomes. For instance, the small number of subjects (30 subjects) to perform our experiment can be considered as a threat. This threat may, in some degree, contributed to affect the experiment results. However, when performing this experiment, we have achieved some important results and feedback on how SPL testing methods could bring advantages for the scientific community and also because there is no evidence, in the literature, of an empirical experiment related to a comparison among different SPL testing methods. The threats to the experiment conclusion validity are the following:

- Measures reliability: this type of threat may depend on many different factors, *e.g.*, bad instrument layout, bad instrumentation. Moreover, we can state, for this type of threat, that objective measures is more reliable when compared to subjective ones (those related to human judgement). In our experiment, the effort (time spent) is a objective measure and then, do not depend on human judgement;
- Treatment reliability implementation: although we have used the same SPLs (PLeTs and AGM) for the three methods and the guidelines (printed manual) delivered to the subjects contain the same set of SPL test information, it is possible that treatment implementation is not similar among the subjects who perform the experiment. This risk could not be completely avoided, since we cannot interfere with the subjects when they are applying functional testing for SPL products. In order to mitigate the influence of this threat, we defined the same starting time for each one of the three treatments, *i.e.*, a guideline describing the test information to be added when using the SPLiT-MBt, CADeT or MTM;
- Random irrelevancies in the experimental setting: in order to mitigate this threat, our experiment execution was conducted in an isolated laboratory. The main goal was to avoid external interaction, *e.g.*, the use of mobile phones, interruptions, and other factors;
- Random heterogeneity of subjects: the variation related to the selection of heterogeneous subjects with different experiences and academic degrees may be a threat to the experiment results. In order to mitigate this threat, we have defined both academic degree and experience in functional testing, UML notation and Software Product Lines as blocking variables.

Internal validity: it refers to the threats related to the internal validity of our experiment.

- History: the schedule of our experiment execution was planned to avoid periods in which subjects could be exposed to external influences. Therefore, we have intended to avoid performing the experiment during the student (subjects) exam period;
- Maturation: the training phase and the experiment as well were applied, in general, during the morning, since the subjects are more motivated and less tired;
- Selection: we have applied a characterization questionnaire to assess the subjects knowledge/experience. We used that information to select and group (block) the subjects.

External validity:

- Subjects: a threat for our experiment external validity was selecting subjects that may not be representative for the SPL and functional testing community. Therefore, in order to mitigate this issue, the students we have selected to participate of our experiment should be in the masters and doctoral computer science courses or be students close to end their undergraduate course (in computer science as well). Thus, all subjects have a basic/sufficient functional testing and UML modeling knowledge. Moreover, we categorized the subjects into three groups, *i.e.*, Advanced, Intermediate and Beginner. Therefore, we were able to obtain a balanced group of subjects. Although we have a balanced group, the number of Advanced (6 subjects) and Beginner (9 subjects) subjects were less than Intermediate (15 subjects). This is a factor that could have some influence in the experiment results. In order to overcome this issue, we design our experiment so that each subject executes the three treatments. Hence, we were able to make a fairer comparison among the three methods and then, obtain more precise results. Moreover, we also random divided the subjects of each block into three groups and, each subjects group started the experiment with a different treatment;
- Tasks: another threat for our experiment is related to the tasks we have defined to apply functional testing to test products derived for PLeTs' and AGM's SPL. It is possible that, when performing the experiment, the activities executed by the subjects may not reflect the activities performed by an actual SPL tester. In order to mitigate this issue, we defined (based on the experience obtained from the work developed between our research group and a Technology Development Laboratory (TDL) of a large IT company) for our experiment a representative set of test cases with a reasonable task size (defined by the number of test cases). Moreover, we validated this set of test cases with an SPL professional who had no contact with the subjects during the experiment execution;
- Experiment effects: one of the threats is related to the fact that some subjects could know the empirical experiment's author and then, this may have influenced the results/outcomes. In order to mitigate this threat, we do not reveal the author's identity till the end of the experiment. Moreover, we invited another researcher to assist us when applying the experiment.

Construct validity: a threat that may occur in any empirical experiment is that the subjects could wrongly conclude that their performance is measured with the purpose of verifying who is "the best". In order to mitigate this issue, we explained to the subjects (before the training phase and also before the experiment execution) that we are only assess and evaluate the SPL methods and not their behaviour or personal performance. As we said earlier, another threat is that the test cases we have defined may not be representative. This threat was minimized by evaluating the complexity of the test cases with an SPL professional and also because the experience obtained from the collaboration between our research group and the TDL we have work with, enabled us to define a representative set of test cases.

5.4 Operation of the Experimental Study

In this section, we present the preparation phase of our experiment and also show the details about its execution as well.

5.4.1 Preparation

An important factor that must be considered when designing an experiment is to have a balanced sample of subjects [WRH^{+}00]. Therefore, we present in this section: how we conducted the experiment; the documentation we have prepared for the participants and; how we configured the whole experiment environment.

First, we made a personal contact with the subjects through a presentation, in which the experiment and its purpose was explained. This presentation was divided into two moments. First, we gave to the subjects information about the general idea of the experiment and we told them that they could use that moment to ask any question. Then, we explained the purpose of our research and how the results would be published. Next, we provided them a profile form/characterization questionnaire (it can be found in the appendix of this thesis) that all subjects should answer. The information obtained from this questionnaire was used to characterize and distribute the subjects through the blocks before performing the experiment. At the end, we explained to the subjects that their personal data would be kept confidential.

An important issue we have considered during the experiment preparation was defining how we should collect the data related to the independent variable *effort* (time spent). In order to avoid any human mistake, we used a chronometer application to measure the time spent by each subject when performing the experiment. Thus, after every experiment session, all test artifacts generated by the subjects were further analyzed (to verify any inconsistency) and then, stored in a cloud application.

5.4.2 Execution

The experiment execution took place in March of 2016 and was organized in two phases: training and experiment execution. The training phase was divided into two sessions: one used to train thirty (30) subjects in modeling functional testing using UML Use Case and Activity Diagrams and; another one to use SPLiT-MBt Tool, CADeT Tool, and MTM. During the training phase, the subjects were oriented to generate functional test cases for AGM's SPL (SUT).

On the other hand, the experiment execution phase was divided into three sessions, in which each session was related to a specific method (SPLiT-MBt, CADeT or MTM) to generate functional test cases. Thus, Session 1 was performed by subjects using SPLiT-MbT Tool; Session 2 was performed by subjects using CADeT Tool and Session 3 was performed by subjects using MTM. Moreover, each session was designed to analyze the effort when using the mentioned methods. At this phase, the SUT we have defined was PLeTs SPL. The synthesis of the experiment execution is as follows:

1. **Training Phase (two days to perform this phase):**

   Session 1: UML Use Case and Activity Diagrams;

   Session 2: SPLiT-MBt Tool, CADeT Tool, and MTM.

2. **Experiment Execution Phase (three days to perform this phase):**

   Session 1: Generate test cases using SPLiT-MbT;

   Session 2: Generate test cases using CADeT;

   Session 3: Generate test cases using MTM.

As we mentioned earlier, the subjects were divided into three groups (blocks), according to information obtained from the profile form, *i.e.*, Advanced, Intermediate and Beginner. Each group of subjects performed the experiment as follows:

> **Advanced Group**: the Advanced Group (composed by 6 subjects) was equally divided and distributed among the three treatments (SPLiT-MBt, CADeT and MTM methods). It is important to highlight that, although all the Advanced subjects have executed the experiment using the three methods, two of them started the experiment using a different method (*Randomized complete block design*). Therefore, two started with SPLiT-MBt; two started with CADeT and two started with MTM. Then, when two subjects ended the experiment with their first method, they performed the experiment using a second method and finally performed the experiment using the third method. Thus, all subjects executed the experiment using the three methods;

Intermediate Group: the Intermediate Group (composed by 15 subjects) was equally divided and distributed among the three treatments. Similarly as done for the Advanced Group, all the Intermediate subjects executed the experiment using the three methods and five of them started the experiment using a different method. Therefore, five started with SPLiT-MBt; five started with CADeT and five started with MTM. Then, when five subjects ended the experiment with their first method, they performed the experiment using a second method and finally performed the experiment using the third method. Thus, all subjects executed the experiment using the three methods;

Beginner Group: exactly as done for the other groups, the Beginner Group (composed by 9 subjects) was equally divided and distributed among the three treatments. Therefore, all the Beginner subjects executed the experiment using the three methods and three of them started the experiment using a different method. Hence, three started with SPLiT-MBt; three started with CADeT and three started with MTM. Then, when three subjects ended the experiment with their first method, they performed the experiment using a second method and finally performed the experiment using the third method. Thus, all subjects executed the experiment using the three methods;

Table 5.2 shows how the subjects were distributed and the number of subjects in each block. As we previously mentioned, the subjects were randomly selected to start the experiment using one of three treatments (*Randomized complete block design*). Therefore, we had a concern on characterize and distribute the subjects according to their profile, experience and knowledge. It is important to highlight that we also defined a break of one day for each group of subjects (Advanced, Intermediate and Beginner) and a break of two days among the training and experiment phases. The main goal was to avoid that some exhaustion of the participants could influence the results.

Table 5.2: Assigning Subjects to the Treatments for a Randomized Complete Block Design

Subjects assignment		
Treatments	**Blocks**	**Num. of subjects**
	Beginner	9
SPLiT-MBt	Intermediate	15
	Advanced	6
	Beginner	9
CADeT	Intermediate	15
	Advanced	6
	Beginner	9
MTM	Intermediate	15
	Advanced	6

5.5 Results

In this section, we present the effort (dependent variable) data collected from our experiment. Table 5.3 shows the summarized effort data (time spent) by subjects to perform the experiment using SPLiT-MBt, CADeT and MTM methods. In the table, column *Block Average Time* presents the average time per block, while column *Method Average Time* presents the average time per method. For instance, the Beginner block that applied the SPLIt-MBt spent an average time of 38m20s, while the Advanced block that applied the SPLIt-MBt spent 33m00s.

To better summarize the results, we also present the average time spent per method, *i.e.*, average time spent by Beginner, Intermediate and Advanced subjects to apply each method (SPLiT-MBt, CADet, and MTM). For instance, the subjects who applied the CADeT method spent an average time of 38m30s, while the subjects who applied the MTM method spent an average time of 45m10s.

Based on the results summarized in the table, the average effort using SPLiT-MBt was less than with CADeT and MTM (36m16s vs 38m30s vs 45m10s). Otherwise, considering the time spent per block, the average effort using SPLiT-MBt was more balanced than CADeT and MTM, since the Beginner, Intermediate and Advanced blocks who applied SPLiT-MBt had similar effort (38m20s, 36m41s and 33m00s, respectively). Therefore, it is possible to claim, in this particular situation, that SPLiT-MBt is suitable and intuitive, since even Beginner or Intermediate subjects could generate test cases without advanced skills.

Table 5.3: Summarized data of the effort

		Effort (minutes/sec)	
Treatments	**Blocks**	**Block Average Time**	**Method Average Time**
	Beginner	38m20s	
SPLiT-MBt	Intermediate	36m41s	36m16s
	Advanced	33m00s	
	Beginner	49m33s	
CADeT	Intermediate	36m56s	38m30s
	Advanced	27m50s	
	Beginner	58m13s	
MTM	Intermediate	42m56s	45m10s
	Advanced	31m20s	

5.6 Analysis and Interpretation

In this section, we summarize our general results and also describe, in details, the effort data collected from our experiment. Next, we present how we performed hypothesis testing for all data sets using as reference the PortalAction statistical package [Por17], [WRH^{+}00] and Levin's *et al* [Lev12].

5.6.1 Priori Test

The priori test has been used to verify whether there is difference among treatments in an empirical experiment [Lev12]. In our experiment, we have applied the Analysis of Variance (ANOVA) test [Lev12], since it is a test to be applied when there are more than two treatments [WRH+00]. Therefore, we have used the ANOVA test because we defined three treatments for our experiment, *i.e.*, SPLiT-MBt, CADeT and MTM. This test aims to verify, in a first moment, whether the averages of at least two methods differ significantly. In a nutshell, the ANOVA test is applied to reject H_0 or not. If this test rejects H_0, a posteriori test must be applied to identify where the difference exists.

We performed this ANOVA test with a single factor (effort) and a significance level $\alpha = 0.05$ (see Table 5.4 and Table 5.5). The main goal was to detect whether there was a significant difference between the average time spent by the subjects using each one of the methods. Then, we conclude that there is evidence that those methods have difference among themselves, since the data sets collected (from populations) has F = 4.064; F critical = 3.101 and; P-value = 0.021 (see Table 5.5)[2].

Based on these values, we can state that the test rejected H_0 and we have a 2.1% chance of being wrong. Thus, we have a very small likelihood to be wrong when we claim that at least one of the the three methods (SPLiT-MBt, CADeT or MTM) has an average different from the others. Therefore, when rejecting H_0, a posteriori test must be applied to identify where the difference exists.

Table 5.4: ANOVA Summary

Summary				
Treatments	**Count**	**Sum**	**Average**	**Variance**
SPLiT-MBt	30	1088	36.267	41.444
MTM	30	1355	45.167	221.868
CADeT	30	1155	38.5	211.5

Count: Sample size.
Sum: Sum of time spent by the subjects per treatment.
Average: Average per treatment.
Variance: Square of the standard deviation.

5.6.2 Posteriori Test

In this section, we present how we found the differences, among the treatments of our experiment, regarding to the time spent by the subjects when performing the experiment using SPLiT-MBt, CADeT and MTM.

[2] If F > F critical, we reject the null hypothesis (H_0) [Lev12].

Table 5.5: ANOVA Data Set

ANOVA						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F critical**
Between Groups	1286.422	2	643.211	4.064	0.021	3.101
Within Groups	13769.533	87	158.271			
Total	15055.956	89				

SS: Sum of Squares.
df: Degree of freedom.
MS: Mean Square.
F: $MS_{between} \div MS_{within}$, where $MS_{between}$ is *MS Between Groups* and MS_{within} is *MS Within Groups*.
P-value: Error likelihood when rejecting H_0.
F critical: Value found in a table. This value is what is referred to as the F statistic.

Thereunto, we have applied the Tukey test, which aims to determine the differences among means (averages) in terms of standard error [Lev12] [Por17]. In an nutshell, it is a test used to find where exist a Honestly Significant Difference (HSD) among the averages regarding to the time spent to perform the experiment using the three methods. "Honest" because we adjust for making multiple comparisons. Therefore, we applied the Tukey test to determine where these differences were found. Next, we present the steps to apply the HSD using the Tukey test.

Step 1: to calculate the differences (absolute value/modulus) among the averages (effort) per method. Then, to create a table of differences among ordered averages (see Table 5.6).

Table 5.6: Table of differences among ordered averages

	SPLiT-MBt = 36,267	CADeT = 38,5	MTM = 45,167
SPLiT-MBt = 36,267	-	2.233	8.9
CADeT = 38,5	-	-	6.667
MTM = 45,167	-	-	-

Step 2: to identify in Table H the value of **q**, which depends on three factors:

1. Degree of freedom (df) for MS_{within}. In our case, **df = 30 - 1 = 29**, where "30" is the number of subjects per treatment;
2. Number *k* of tested averages. In our case, **k = 3**, since we have three treatments;
3. Significance level: $\alpha = 0.05$.

After this analysis, we have identified the value of q, which is: **q = 3.49**.

Step 3: to calculate the Honestly Significant Difference (HSD).

$$HSD = q\sqrt{\frac{MS_{within}}{n_{within}}}$$

Where:

q is the value obtained from the Table H.

MS_{within} is the mean square within the methods (SPLiT-MBt, CADeT and MTM).

n_{within} is the sample size (number of subjects).

$$HSD = 3.49\sqrt{\frac{158.271}{30}} = 8.016$$

Step 4: to compare the DSH with the values from the Table of differences among averages described in step 1. When the differences (in absolute value/modulus) are higher than HSD, then, we can state that there is a significant statistical difference between the average times of the methods, with alpha of 5% of significance. Therefore, in the three calculated differences (SPLiT-MBt and CADeT; SPLiT-MBt and MTM; CADeT and MTM), the Tukey test identified, through the HSD, that there is a significant difference between: SPLiT-MBt and MTM, whose difference (in absolute value/modulus) is 8.9, *i.e.*, a value higher than 8.016 (HSD). The test did not identify other significant differences among the other methods, since the difference between SPLiT-MBt and CADeT is 2.233 and; the difference between CADeT and MTM is 6.667, *i.e.* these values are smaller than 8.016 (HSD). Next, we present the conclusions and the result analysis obtained from these data values.

5.7 Conclusions and Result Analysis

The main contribution of our experiment is regarding to comparison of our SPLiT-MBt against a similar method named CADeT and a conventional approach used to test single applications named MTM. This comparison was made by measuring the effort (time spent) to apply functional testing for products derived from SPLs using each one of these three methods, wherein SPLiT-MBt and CADeT are oriented to reuse of test artifacts and MTM provide functional testing without any reuse.

Based on the results we have presented in this chapter, we can draw some conclusions. Our SPLiT-MBt presents a significant statistical difference when compared to MTM. It means that, in this particular situation, our method present advantages when compared to the conventional way to test single applications. However, SPLiT-MBt shown no significant difference regarding to the average times when compared to CADeT, a method similar to SPLiT-MBt and also used to test products derived from SPLs. Although SPLiT-MBt and CADeT did not present a significant difference, the test we have applied (ANOVA test) demonstrates that CADeT has no significant difference when compared to MTM as well. In an nutshell, regarding to the average times, SPLiT-MBt has a significant advantage when comparing to MTM, but CADeT have no difference when compared to the same method.

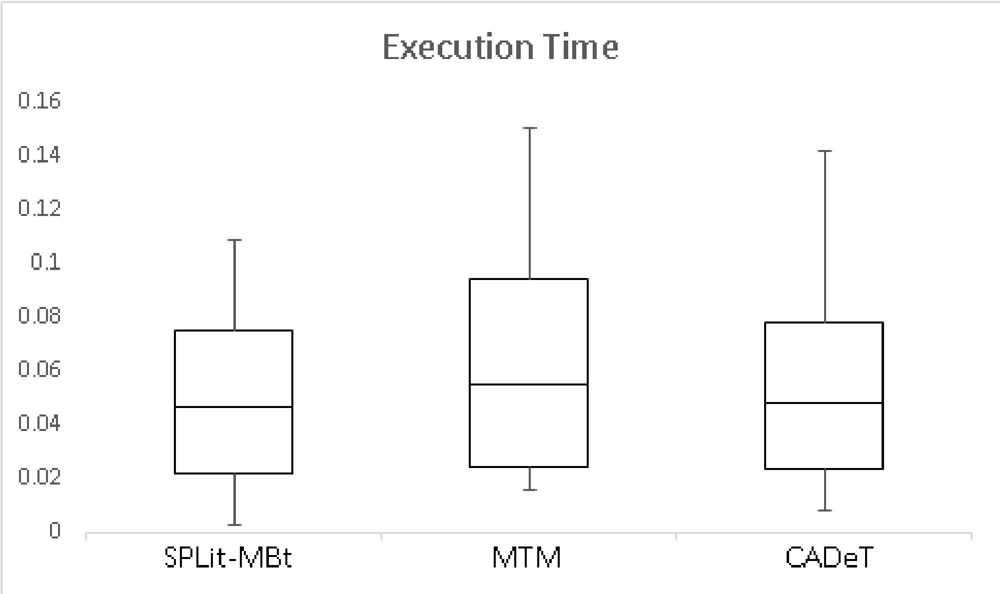


Figure 5.1: Experiment Time

Figure 5.1 presents the box-plot graph of the data set relative to the experiment time. In this data set, the medians of execution time with SPLiT-BMt, MTM and CADeT were, respectively, 35, 44 and 36 minutes. Therefore, we can state that SPLiT-MBt had a substantial advantage against MTM and a slight advantage when compared to CADeT. Although the difference between SPLiT-MBt and CADeT was very small, it is important to highlight that SPLiT-MBt generated automated test scripts that can be executed using a test technology (*e.g.*, MTM or RFT) while CADeT generated only test cases in natural language.

Moreover, when observing the ANOVA Table, it is possible to identify that the variability of the average times obtained by the **Variance** are quite higher for MTM and Cadet when comparing to SPLiT-MBt (221.868, 211.5 and 41.444, respectively - see Table 5.4). Therefore, we can state (again, for this particular situation) that SPLiT-MBt has a more homogeneous behavior when compared to the other two methods.

Therefore, we can claim that through the use of our SPLiT-MBt, users who do not have advanced knowledge about UML and functional testing will have a performance similar to users with advanced skills. This is not the case of the other two methods, since the variance is quite higher for them. In the next chapter, we present the conclusions, the points that can be improved in our work and the thesis summary.

6. THESIS SUMMARY AND FUTURE WORK

"Without love, deeds, even the most brilliant, count as nothing."

St. Therese of Lisieux

In this thesis, we presented the SPLiT-MBT method, which is based on the MBT technique to generate functional test cases and scripts for products derived from SPLs. The proposed method supports the generation of test artifacts developed during Domain Engineering to reuse them to test products during Application Engineering. In order to provide this reuse, SPLiT-MBt is applied in two phases. In the first one, test and variability information annotated in models, *e.g.*, Use Case and Activity Diagrams are used to generate test sequences in Domain Engineering. These sequences are generated through applying test sequences generation methods in FSMs, which are extended to deal with variability information.

In the second phase, where the variability is resolved, test sequences generated during Domain Engineering are reused to test products derived in Application Engineering. Furthermore, in this phase, conventional test sequence generation methods are used to generate specific test sequences. These sequences are converted into abstract test cases, from which scripts are generated. These scripts could be executed by several functional testing tools, *e.g.*, VS, MTM, RFT or QTP. The input data present in these test scripts are generated using a functional criteria, *e.g.*, Boundary Value Analysis and Equivalence Partitioning.

Through these activities, the SPLiT-MBt allows: the reuse of test artifacts through the information present in models and reuse of test sequences generated in Domain Engineering. These features are essential to reduce the test effort, because when the variabilities in SPLs grows, the combination of tests can increase exponentially. Furthermore, SPLiT-MBt also aims to reduce the test effort through applying test sequence generation methods, which are adapted to deal with variability information. Another advantage is that SPLiT-MBt allows the integration of different functional testing tools, which is performed based on a structure representing test cases in natural language, *i.e.*, abstract test cases. Thus, the script generation is performed independent of technology.

6.1 Thesis Contributions

In this work, we have developed a method (SPLiT-MBt) to test products derived from SPLs through the reuse of test artifacts generated during Domain Engineering. SPLiT-MBt also generates test sequences through extending conventional test sequence generation methods (*e.g.* UIO, W, TT, HSI) in an SPL context. Furthermore, our method provide the reuse of test artifacts based on adapting MBT to generate functional test cases and scripts from models/notations that represent the SPL functionalities and variability information.

In this context, we claim that SPLiT-MBt provides several advantages, such as: it generates test cases that are reused to generate scripts based on system models. Therefore, test cases to test common functionalities for different products are generated just once; our method generates a generic structure representing test cases in pseudo-natural language. Thus, it is possible to generate test scripts that can be executed by different functional testing tools; it provides reduction of amount of test cases through applying prioritization and minimization of test cases. Thus, when test sequences are generated, the user could choose the most relevant test cases according to personal criteria.

Based on the concepts of our method, we developed a tool named SPLiT-MBt Tool, which was used demonstrate the applicability of our method through generating test cases for two SPLs, *i.e.*, AGM and PLeTs. At the end, our method provide evidences that is able to generate test cases with reuse of test artifacts. At the end, we conducted a controlled experiment with the purpose of demonstrating the performance of our SPLiT-MBt against two other methods, *i.e.* CADeT and MTM. This study demonstrates that SPLiT-MBt is significantly better than MTM and also has a more homogeneous behavior when compared to the CADeT and MTM.

6.2 Limitations and Future Works

Despite these benefits and contributions we have presented, there are some points in our work that could be improved. For example, we can better evaluate/assess SPLiT-MBt through generating test cases and scripts for another SPLs. Another concern is that SPLiT-MBt automates only the functional test case generation. However, this method could be applied to other types of testing, *e.g.*, performance test for web applications. Although we have demonstrated the applicability of our method using two SPLs and conducted an empirical experiment to evaluate our results, we are aware that we could define a bigger sample size and choose subjects from a company with experience on SPL testing. Furthermore, we can work on providing test automation using testing technologies different from MTM, *e.g.*, Selenium, RFT or QTP. Finally, we have to point out that another limitation of our work concerns the fact that SPLiT-MBt is not able to generate test cases and scripts from dynamic SPLs and the variability is resolved only at design time. However, we are working on extending SPLiT-MBt to address these issues.

6.3 Publications

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

- Costa, L. T.; Zorzo, A. F. ; Rodrigues, E. M. ; Bernardino, M. ; Oliveira, F. M. . Structural Test Case Generation Based on System Models. In: International Conference on Software Engineering Advances (ICSEA), 2014, Nice. International Conference on Software Engineering Advances, 2014.

- Costa, L. T.; Oliveira, F. M. ; Rodrigues, E. M. ; Silveira, M. B. ; Zorzo, A. F. . An approach for Generating Structural Test Cases Based on System Models. In: Workshop de Teste e Tolerância a Falhas, 2012, Ouro Preto - MG. Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC) - WTF 2012, 2012.
- Costa, L. T.; Czekster, R. ; Oliveira, F. M. ; Rodrigues, E. M. ; Silveira, M. B. ; Zorzo, A. F. . Generating Performance Test Scripts and Scenarios Based on Abstract Intermediate Models. In: The 24th International Conference on Software Engineering and Knowledge Engineering, 2012, San Francisco. 24rd International Conference on Software Engineering and Knowledge Engineering, 2012.
- Zanin, A. ; Zorzo, A. F. ; Costa, L. T. . SPLiT-TeSGe - Um Processo para Adaptação de Métodos de Geração de Sequências de Testes para Linha de Produto de Software. In: Workshop de Testes e Tolerância a Falhas, 2015, Vitória. Workshop de Testes e Tolerância a Falhas, 2015.
- Rodrigues, E. ; Bernardino, M. ; Costa, L. T. ; Zorzo, A. F. ; Oliveira, F. . PLeTsPerf - A Model-Based Performance Testing Tool. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), 2015, Graz. 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), 2015.
- Rodrigues, E. M. ; Oliveira, F. M. ; Bernardino, M. ; Saad, R. ; Costa, L. T. ; Zorzo, A. F. . Evaluating Capture and Replay and Model-based Performance Testing Tools: An Empirical Comparison. In: Empirical Software Engineering (2014), 2014, Turin. Empirical Software Engineering, 2014.
- Rodrigues, E. M. ; Oliveira, F. M. ; Costa, L. T. ; Bernardino, M. ; Zorzo, A. F. ; Souza, S. S. ; Saad, R. . An empirical comparison of model-based and capture and replay approaches for performance testing. Empirical Software Engineering, v. 1, p. 1-30, 2014.

REFERENCES

[AD97] Apfelbaum, L.; Doyle, J. "Model Based Testing". In: Proceedings of the 10th Software Quality Week Conference, 1997, pp. 296–300.

[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, 2004, pp. 11–33.

[AO08] Ammann, P.; Offutt, J. "Introduction to Software Testing". Cambridge University Press, 2008.

[BG04] Bertolino, A.; Gnesi, S. "PLUTO: A Test Methodology for Product Families". In: Proceedings of the 5th Workshop on Software Product-Family Engineering, 2004, pp. 181–197.

[BRJ05] Booch, G.; Rumbaugh, J.; Jacobson, I. "The Unified Modeling Language User Guide (2nd Edition)". Addison-Wesley Professional, 2005.

[Bro16] Broek, P. "Extended Feature Models". Available in: http://www.utwente.nl, Jan 2016.

[CC79] Cook, T. D.; Campbell, D. T. "Quasi-experimentation: Design and Analysis Issues for Field Settings". Houghton Mifflin, 1979.

[CCO+12] Costa, L. T.; Czekster, R. M.; Oliveira, F. M.; Rodrigues, E. M.; 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 & Knowledge Engineering, 2012, pp. 112–117.

[CGSE12] Capellari, M. L.; Gimenes, I.; Simão, A.; Endo, A. "Towards Incremental FSM-based Testing of Software Product Line". In: Proceedings of the 6th Brazilian Software Quality Symposium, 2012, pp. 9–23.

[Cho78] Chow, T. S. "Testing Software Design Modeled by Finite-State Machines", *IEEE Transactions on Software Engineering*, vol. 4, 1978, pp. 178–187.

[CN01] Clements, P.; Northrop, L. "Software Product Lines: Practices and Patterns". Addison-Wesley, 2001.

[CSV10] Cristiá, M.; Santiago, V.; Vijaykumar, N. L. "On Comparing and Complementing two MBT Approaches". In: Proceedings of the 11th Latin American Test Workshop, 2010, pp. 1–6.

[CZR+14] Costa, L. T.; Zorzo, A. F.; Rodrigues, E. M.; Silveira, M. B.; Oliveira, F. M. "Structural Test Case Generation Based on System Models". In: Proceedings of the 9th International Conference on Software Engineering Advances, 2014, pp. 276–281.

[DCG+09] Davis, C.; Chirillo, D.; Gouveia, D.; Saracevic, F.; Bocarsley, J. B.; Quesada, L.; Thomas, L. B.; Lint, M. v. "Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource". IBM Press, 2009.

[DKMT96] Devanbu, P.; Karstu, S.; Melo, W.; Thomas, W. "Analytical and Empirical Evaluation of Software Reuse Metrics". In: Proceedings of the 18th International Conference on Software Engineering, 1996, pp. 189–199.

[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 25th IEEE International Conference on Software Maintenance, 2009, pp. 433–436.

[DMJ07] Delamaro, M. E.; Maldonado, J. C.; Jino, M. "Introdução ao Teste de Software". Elsevier, 2007.

[EFW01] El-Far, I. K.; Whittaker, J. A. "Model-based Software Testing". Wiley, 2001.

[EM07] Everett, G. D.; McLeod, R. J. "Software Testing: Testing Across the Entire Software Development Life Cycle". John Wiley & Sons, 2007.

[ER11] Engstrom, E.; Runeson, P. "Software Product Line Testing - A Systematic Mapping Study", *Information and Software Technology*, vol. 53, 2011, pp. 2–13.

[FGMO12] Fiori, D. R.; Gimenes, I. M. S.; Maldonado, J. C.; OliveiraJr, E. A. "Variability Management in Software Product Line Activity Diagrams". In: Proceedings of the 18th International Conference on Distributed Multimedia Systems, 2012, pp. 89–94.

[Gil62] Gill, A. "Introduction to the Theory of Finite State Machines". McGraw-Hill, 1962.

[Gil95] Gil, A. "Métodos e Técnicas de Pesquisa Social". Atlas, 1995.

[Gom05] Gomaa, H. "Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures". Addison-Wesley, 2005.

[Gon70] Gonenc, G. "A Method for the Design of Fault Detection Experiments", *IEEE Transactions on Computer*, vol. 19, 1970, pp. 551–558.

[Gro09] Gronback, R. C. "Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit". Addison-Wesley, 2009.

[HK06] Holmes, A.; Kellogg, M. "Automating Functional Tests Using Selenium". In: Proceedings of the 9th International Conference on Agile, 2006, pp. 270–275.

[HP03] Halmans, G.; Pohl, K. "Communicating the Variability of a Software-Product Family to Customers", *Software and System Modeling*, vol. 2, 2003, pp. 15–36.

[HT90] Hamlet, D.; Taylor, R. "Partition Testing does not Inspire Confidence (Program Testing)", *IEEE Transactions on Software Engineering*, vol. 16, 1990, pp. 1402–1411.

[HVR04] Hartmann, J.; Vieira, M.; Ruder, A. "A UML-based Approach for Validating Product Lines". In: Proceedings of the 3rd International Workshop on Software Product Line Testing, 2004, pp. 58–65.

[IBM17] IBM. "IBM Rational PurifyPlus". Available in: http://www.ibm.com/software/awdtools/purifyplus/, Jul 2017.

[Ins16a] Institute, S. E. "Software Product Lines (SPL)". Available in: http://www.sei.cmu.edu/productlines/, Oct 2016.

[Ins16b] Institute, S. E. "Software Product Lines (SPL)". Available in: http://www.sei.cmu.edu/productlines/ppl, Oct 2016.

[Jun10] Junior, E. A. O. "SystEM-PLA: um Método para Avaliação de Arquitetura de Linha de Produto de Software Baseada em UML", Ph.D. Thesis, Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo, Brasil, 2010.

[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, 1990.

[KJG99] Kerbrat, A.; Jéron, T.; Groz, R. "Automated Test Generation from SDL Specifications". In: Proceedings of the 9th SDL Forum, 1999, pp. 135–152.

[KKB+17] Kang, S.; Kim, J.; Baek, H.; Ahn, H.; Jung, P.; Lee, J. "Comparison of Software Product Line Test Derivation Methods from the Reuse Viewpoint". In: Proceedings of the 6th International Conference on Software and Computer Applications, 2017, pp. 1–8.

[KLKL07] Kang, S.; Lee, J.; Kim, M.; Lee, W. "Towards a Formal Framework for Product Line Test Development". In: Proceedings of the 7th IEEE International Conference on Computer and Information Technology, 2007, pp. 921–926.

[Kri04] Krishnan, P. “Uniform Descriptions for Model Based Testing”. In: Proceedings of the 15th Australian Software Engineering Conference, 2004, pp. 96–105.

[LCYW11] Lin, M.; Chen, Y.; Yu, K.; Wu, G. “Lazy Symbolic Execution for Test Data Generation”, *IET Software*, vol. 5, 2011, pp. 132–141.

[Lev11] Levinson, J. “Software Testing With Visual Studio 2010”. Pearson Education, 2011.

[Lev12] Levin, Jack; Fox, J. A. F. D. R. “Estatística para Ciências Humanas”. Pearson Education do Brasil, 2012.

[Lin02] Linden, F. “Software Product Families in Europe: the Esaps Cafe Projects”, *IEEE Software*, vol. 19, 2002, pp. 41–49.

[LPP13] Lamancha, B. P.; Polo, M.; Piattini, M. “Systematic Review on Software Product Line Testing”. In: Proceedings of the 10th International Software and Data Technologies, 2013, pp. 58–71.

[LRD+15] Laser, M.; Rodrigues, E. M.; Domingues, A. R.; de Oliveira, F. M.; Zorzo, A. F. “Research Notes on the Architectural Evolution of a Software Product Line”, *International Journal of Software Engineering and Knowledge Engineering*, vol. 20, 2015, pp. 1753–1758.

[LSR07] Linden, F. J. v. d.; Schmid, K.; Rommes, E. “Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering”. Springer-Verlag, 2007.

[LVMM07] Lemos, O. A. L.; Vincenzi, A. M. R.; Maldonado, J. C.; Masiero, P. C. “Control and Data Flow Structural Testing Criteria for Aspect-oriented Programs”, *Journal of Systems and Software*, vol. 80, 2007, pp. 862–882.

[Mal09] Mallepally, S. R. “Quick Test Professional (QTP) Interview Questions and Guidelines: A Quick Reference Guide to QuickTest Professional”. Parishta, 2009.

[Man16] Manager, M. T. “Running Tests in Microsoft Test Manager”. Available in: http://www.msdn.microsoft.com/en-us/library/dd286680, Jan 2016.

[McG01] McGregor, J. D. “Testing a Software Product Line”, Technical Report, Clemson University, 2001.

[MJG14] Marcolino, A.; Jr., E. A. O.; Gimenes, I. M. S. “Towards the Effectiveness of the SMarty Approach for Variability Management at Sequence Diagram Level”. In: Proceedings of the 16th International Conference on Enterprise Information Systems, 2014, pp. 249–256.

[MM98] Meadows, C.; McLean, J. "Security and Dependability: Then and Now". In: Proceedings of the 7th Computer Security, Dependability and Assurance: From Needs to Solutions, 1998, pp. 166–170.

[MRKN13] Millo, J. V.; Ramesh, S.; Krishna, S. N.; Narwane, G. "Compositional Verification of Software Product Lines". In: Proceedings of the 10th Integrated Formal Methods, 2013, pp. 109–123.

[MRMdOTC+15] Macedo Rodrigues, E.; Moreira de Oliveira, F.; Teodoro Costa, L.; Bernardino, M.; Zorzo, A. F.; do Rocio Senger Souza, S.; Saad, R. "An Empirical Comparison of Model-based and Capture and Replay Approaches for Performance Testing", *Empirical Software Engineering*, vol. 20, 2015, pp. 1831–1860.

[MS04] Myers, G. J.; Sandler, C. "The Art of Software Testing". John Wiley & Sons, 2004.

[MSM04] McGregor, J.; Sodhani, P.; Madhavapeddi, S. "Testing Variability in a Software Product Line". In: Proceedings of the 3rd International Workshop on Software Product Line Testing, 2004, pp. 45–50.

[NdCMM+11] Neto, P. A. M. S.; do Carmo Machado, I.; McGregor, J. D.; de Almeida, E. S.; de Lemos Meira, S. R. "A Systematic Mapping Study of Software Product Lines Testing", *Information and Software Technology*, vol. 53, 2011, pp. 407–423.

[NFTJ04] Nebut, C.; Fleurey, F.; Traon, Y.; Jezequel, J. M. "A Requirement-Based Approach to Test Product Families". In: Proceedings of the 5th Workshop on Software Product-Family Engineering, 2004, pp. 198–210.

[NT81] Naito, S.; Tsunoyama, M. "Fault Detection for Sequential Machines by Transitions Tours". In: Proceedings of the 11th IEEE Fault Tolerant Computing Conferece, 1981, pp. 283–243.

[OG09] Olimpiew, E.; Gomaa, H. "Reusable model-based testing". In: Proceedings of the 3rd Formal Foundations of Reuse and Domain Engineering, 2009, pp. 76–85.

[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, 2010, pp. 2374–2393.

[PBL05] Pohl, K.; Buckle, G.; Linden, F. J. v. d. "Software Product Line Engineering: Foundations, Principles and Techniques". Springer-Verlag, 2005.

[Por17] Portal Action. "System Action Statistical Package". Available in: http://www.portalaction.com.br/en, Apr 2017.

[Pro16] Professional, A. "Astah Professional". Available in: http://www.astah.net/editions/professional, Jan 2016.

[PYLD93] Petrenko, A.; Yevtushenko, N.; Lebedev, A.; Das, A. "Nondeterministic State Machines in Protocol Conformance Testing". In: Proceedings of the 6th International Workshop on Protocol Test Systems, 1993, pp. 363–378.

[RBC+15] Rodrigues, E. M.; Bernardino, M.; Costa, L.; Zorzo, A.; Oliveira, F. "Pletsperf - a model-based performance testing tool". In: Proceedings of 8th IEEE International Conference on the Software Testing, Verification and Validation, 2015, 2015, pp. 1–8.

[Rou17] Roubtsov, V. "EMMA: a Free Java Code Coverage Tool". Available in: http://emma.sourceforge.net, Jul 2017.

[RRKP06] Reuys, A.; Reis, S.; Kamsties, E.; Pohl, K. "The ScenTED Method for Testing Software Product Lines". In: Proceedings of the 10th Software Product Lines, 2006, pp. 479–520.

[RVZG10] Rodrigues, E. M.; Viccari, L. D.; Zorzo, A. F.; Gimenes, I. M. S. "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.

[SD88] Sabnani, K.; Dahbura, A. "A Protocol Test Generation Procedure", *Computer Networks and ISDN Systems*, vol. 15, 1988, pp. 285–297.

[Sem16] Semantic Designs. "Semantic Designs Test Coverage". Available in: http://www.semdesigns.com, Jul 2016.

[Som11] Sommerville, I. "Software Engineering". Pearson/Addison–Wesley, 2011.

[SPL17] SPLHF. "Software Product Line Hall of Fame". Available in: http://splc.net/hall-of-fame/, Jan 2017.

[SRZ+11] Silveira, M. B.; Rodrigues, E. M.; Zorzo, A. F.; Costa, L. T.; Vieira, H. V.; de Flavio Moreira Oliveira. "Generation of Scripts for Performance Testing Based on UML Models". In: Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering, 2011, pp. 258–263.

[SRZ16] Silveira, M. B.; Rodrigues, E. M.; Zorzo, A. F. "Performance Testing Modeling: an Empirical Evaluation of DSL and UML-based Approaches". In: Proceedings of the 31st ACM Symposium on Applied Computing, 2016, pp. 1660–1665.

[SW02] Smith, C. U.; Williams, L. G. "Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software". Addison-Wesley, 2002.

[SZR16] Silveira, M. B.; Zorzo, A. F.; Rodrigues, E. M. "Canopus: A Domain- Specific Language for Modeling Performance Testing". In: Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation, 2016, pp. 157–167.

[TTK04] Tevanlinna, A.; Taina, J.; Kauppinen, R. "Product Family Testing: a Survey", *ACM SIGSOFT Software Engineering Notes*, vol. 29, 2004, pp. 12.

[Utt06] Utting, M.; Legeard, B. "Practical Model-Based Testing: A Tools Approach". Morgan Kaufmann, 2006.

[VDMW06] Vincenzi, A. M. R.; Delamaro, M. E.; Maldonado, J. C.; Wong, W. E. "Establishing Structural Testing Criteria for Java Bytecode", *Software: Practice and Experience*, vol. 36, 2006, pp. 1513–1541.

[VMWD05] Vincenzi., A. M. R.; Maldonado, J. C.; Wong, W. E.; Delamaro, M. E. "Coverage Testing of Java Programs and Components", *Science of Computer Programming*, vol. 56, 2005, pp. 211–230.

[Web16] Weber, T. S. "Tolerância a Falhas: Conceitos e Exemplos". Available in: http://www.inf.ufrgs.br/~taisy/disciplinas/textos/ConceitosDependabilidade.PDF, Oct 2016.

[WRH+00] Wohlin, C.; Runeson, P.; Host, M.; Ohlsson, M. C.; Regnell, B.; Wesslen, A. "Experimentation in Software Engineering: An Introduction". Kluwer Academic Publishers, 2000.

[Yin13] Yin, R. "Case Study Research: Design and Methods". SAGE Publications, 2013.

[Zan16] Zanin, A. "Split-Tesge: um Processo para Adaptação de Métodos de Geração de Sequências de Testes para Linha de Produto de Software", Master's Thesis, Programa de Pós-Graduação em Ciência da Computação, Pontifícia Universidade Católica do Rio Grande do Sul, Brasil, 2016.

APPENDIX A – SUB SEQUENCES GENERATED PER PRODUCT

Table A.1: State Cover per Product 1

PRODUCT S8		PRODUCT S2		PRODUCT S9	
Start	-	Start	-	Start	-
S0	a	S0	a	S0	a
S1	b	S1	b	S1	b
S2	n/a	S2	abe{VP_OR}	S2	n/a
S3	abc{VP_OR}f	S3	abe{VP_OR}f	S3	abc{VP_OR}f
S4	abc{VP_OR}fg	S4	abe{VP_OR}fg	S4	abc{VP_OR}fg
S5	abc{VP_OR}fh	S5	abe{VP_OR}fh	S5	abc{VP_OR}fh
S6	abc{VP_OR}fhi abc{VP_OR}fgi	S6	abe{VP_OR}fhi abe{VP_OR}fgi	S6	abc{VP_OR}fhi abc{VP_OR}fgi
S7	abc{VP_OR}fhj abc{VP_OR}fgj	S7	abe{VP_OR}fhij abe{VP_OR}fgij	S7	abc{VP_OR}fhij abc{VP_OR}fgij
S8	abc{VP_OR}	S8	n/a	S8	n/a
S9	n/a	S9	n/a	S9	abc{VP_OR}f
END	abd{VP_OR}fhjk abd{VP_OR}fgjk	END	abe{VP_OR}fhjk abe{VP_OR}fgjk	END	abc{VP_OR}fhjk abc{VP_OR}fgjk

Table A.2: State Cover per Product 2

PRODUCT S2-S8		PRODUCT S9-S8		PRODUCT S9-S2		PRODUCT S2,S8,S9	
Start	-	Start	-	Start	-	Start	-
S0	a	S0	a	S0	a	S0	a
S1	b	S1	b	S1	b	S1	b
S2	abe{VP_OR}	S2	n/a	S2	abe{VP_OR}	S2	abe{VP_OR}
S3	abe{VP_OR}f abd{VP_OR}f	S3	abc{VP_OR}f abd{VP_OR}f	S3	abc{VP_OR}f abe{VP_OR}f	S3	abd{VP_OR}f; abe{VP_OR}f abc{VP_OR}f
S4	abe{VP_OR}fg abd{VP_OR}fg	S4	abc{VP_OR}fg abd{VP_OR}fg	S4	abc{VP_OR}fg abe{VP_OR}fg	S4	abc{VP_OR}fg; abe{VP_OR}fg abd{VP_OR}fg
S5	abe{VP_OR}fh abd{VP_OR}fh	S5	abc{VP_OR}fh abd{VP_OR}fh	S5	abc{VP_OR}fh abe{VP_OR}fh	S5	abc{VP_OR}fh
S6	abe{VP_OR}fgi abe{VP_OR}fhi abd{VP_OR}fgi abd{VP_OR}fhi	S6	abc{VP_OR}fgi abc{VP_OR}fhi abd{VP_OR}fhi abd{VP_OR}fgi	S6	abc{VP_OR}fgi abc{VP_OR}fhi abe{VP_OR}fgi abe{VP_OR}fhi	S6	abe{VP_OR}fh abc{VP_OR}fh abc{VP_OR}fhi abe{VP_OR}fhi abd{VP_OR}fhi abc{VP_OR}fgi abe{VP_OR}fgi abd{VP_OR}fgi
S7	abe{VP_OR}fgij abd{VP_OR}fgij abe{VP_OR}fhij abd{VP_OR}fhij	S7	abc{VP_OR}fgij abd{VP_OR}fgij abc{VP_OR}fhij abd{VP_OR}fhij	S7	abc{VP_OR}fgij abe{VP_OR}fgij abc{VP_OR}fhij abe{VP_OR}fhij	S7	abc{VP_OR}fhij abc{VP_OR}fgij abe{VP_OR}fhij abe{VP_OR}fgij abd{VP_OR}fhij abd{VP_OR}fgij
S8	abc{VP_OR}	S8	abc{VP_OR}	S8	n/a	S8	abc{VP_OR}
S9	n/a	S9	abc{VP_OR}f	S9	abc{VP_OR}f	S9	abc{VP_OR}f
END	abe{VP_OR}fhjk abe{VP_OR}fgjk abd{VP_OR}fhjk abd{VP_OR}fgjk	END	abc{VP_OR}fhjk abc{VP_OR}fgjk abd{VP_OR}fhjk abd{VP_OR}fgjk	END	abc{VP_OR}fhjk abc{VP_OR}fgjk abe{VP_OR}fhjk abe{VP_OR}fgjk	END	abc{VP_OR}fhjk abc{VP_OR}fgjk abe{VP_OR}fhjk abe{VP_OR}fgjk abd{VP_OR}fhjk abd{VP_OR}fgjk

Table A.3: Transition Cover per Product 1

PRODUCT S8		PRODUCT S2		PRODUCT S9	
Start	ϵ a	Start	ϵ a	Start	ϵ a
S0	ϵab	S0	ϵab	S0	ϵab
S1	ϵabe{VP_OR}	S1	ϵabd{VP_OR}	S1	ϵabe{VP_OR}
S2	ϵabe{VP_OR}f	S2	n/a	S2	n/a
	abe{VP_OR}ff		abd{VP_OR}ff		abc{VP_OR}ff
S3	abe{VP_OR}fg	S3	abd{VP_OR}fg	S3	abe{VP_OR}fg
	abe{VP_OR}fh		abd{VP_OR}fh		abc{VP_OR}fh
S4	abe{VP_OR}fgi	S4	abd{VP_OR}fgi	S4	abc{VP_OR}fgi
S5	abe{VP_OR}fhi	S5	abd{VP_OR}fhi	S5	abc{VP_OR}fhi
	abe{VP_OR}fhik		abd{VP_OR}fhik		abc{VP_OR}fhik
	abe{VP_OR}fhij		abd{VP_OR}fhij		abc{VP_OR}fhij
S6	abe{VP_OR}fhib	S6	abd{VP_OR}fhib	S6	abc{VP_OR}fhib
	abe{VP_OR}fgik		abd{VP_OR}fgik		abc{VP_OR}fgik
	abe{VP_OR}fgij		abd{VP_OR}fgij		abc{VP_OR}fgij
	abe{VP_OR}fgib		abd{VP_OR}fgib		abc{VP_OR}fgib
	abe{VP_OR}fhjk		abd{VP_OR}fhjk		abc{VP_OR}fhjk
S7	abe{VP_OR}fhja	S7	abd{VP_OR}fhja	S7	abc{VP_OR}fhja
	abe{VP_OR}fgjk		abd{VP_OR}fgjk		abc{VP_OR}fgjk
	abe{VP_OR}fgja		abd{VP_OR}fgja		abc{VP_OR}fgja
S8	n/a	S8	abd{VP_OR}f	S8	n/a
S9	n/a	S9	n/a	S9	abc{VP_OR}f
END	abe{VP_OR}fhjk	END	abd{VP_OR}fhjk	END	abc{VP_OR}fhjk

Table A.4: Transition Cover per Product 2

PRODUCT S2,S8,S9		PRODUCT S9-S2		PRODUCT S8-S2		PRODUCT S8-S9	
Start	ϵ a	Start	ϵ a	Start	ϵ a	Start	ϵ a
S0	ϵab	S0	ϵab	S0	ϵab	S0	ϵab
	ϵabe{VP_OR}	S1	ϵabe{VP_OR}	S1	ϵabe{VP_OR}	S1	ϵabd{VP_OR}
S1	ϵabd{VP_OR}		ϵabe{VP_OR}		ϵabd{VP_OR}		ϵabe{VP_OR}
	ϵabe{VP_OR}	S2	ϵabe{VP_OR}f	S2	ϵabe{VP_OR}f	S2	n/a
S2	ϵabe{VP_OR}f		abc{VP_OR}ff		abe{VP_OR}ff		abd{VP_OR}ff
	abe{VP_OR}ff		abe{VP_OR}fg		abe{VP_OR}fg		abd{VP_OR}fg
	abe{VP_OR}fg	S3	abc{VP_OR}fh	S3	abe{VP_OR}fh	S3	abd{VP_OR}fh
	abe{VP_OR}fh		abe{VP_OR}ff		abd{VP_OR}ff		abc{VP_OR}ff
	abd{VP_OR}ff		abe{VP_OR}fg		abd{VP_OR}fg		abe{VP_OR}fg
S3	abd{VP_OR}fg		abe{VP_OR}fh		abd{VP_OR}fh		abc{VP_OR}fh
	abd{VP_OR}fh	S4	abc{VP_OR}fgi	S4	abe{VP_OR}fgi	S4	abd{VP_OR}fgi
	abc{VP_OR}ff		abe{VP_OR}fgi		abd{VP_OR}fgi		abc{VP_OR}fgi
	abe{VP_OR}fg	S5	abc{VP_OR}fhi	S5	abe{VP_OR}fhi	S5	abd{VP_OR}fhi
	abc{VP_OR}fh		abe{VP_OR}fhi		abd{VP_OR}fhi		abc{VP_OR}fhi
	abe{VP_OR}fgi		abc{VP_OR}fhik		abe{VP_OR}fhik		abd{VP_OR}fhik
S4	abd{VP_OR}fgi		abc{VP_OR}fhij		abe{VP_OR}fhij		abd{VP_OR}fhij
	abc{VP_OR}fgi		abc{VP_OR}fhib		abe{VP_OR}fhib		abd{VP_OR}fhib
	abe{VP_OR}fhi		abc{VP_OR}fgik		abe{VP_OR}fgik		abd{VP_OR}fgik
S5	abd{VP_OR}fhi		abc{VP_OR}fgij		abe{VP_OR}fgij		abd{VP_OR}fgij
	abc{VP_OR}fhi	S6	abc{VP_OR}fgib	S6	abe{VP_OR}fgib	S6	abd{VP_OR}fgib
	abe{VP_OR}fhik		abe{VP_OR}fhik		abd{VP_OR}fhik		abc{VP_OR}fhik
	abe{VP_OR}fhij		abe{VP_OR}fhij		abd{VP_OR}fhij		abc{VP_OR}fhij
	abe{VP_OR}fhib		abe{VP_OR}fhib		abd{VP_OR}fhib		abc{VP_OR}fhib
	abe{VP_OR}fgik		abe{VP_OR}fgik		abd{VP_OR}fgik		abc{VP_OR}fgik
	abe{VP_OR}fgij		abe{VP_OR}fgij		abd{VP_OR}fgij		abc{VP_OR}fgij
	abe{VP_OR}fgib		abe{VP_OR}fgib		abd{VP_OR}fgib		abc{VP_OR}fgib
	abd{VP_OR}fhik		abc{VP_OR}fhjk		abe{VP_OR}fhjk		abd{VP_OR}fhjk
	abd{VP_OR}fhij		abc{VP_OR}fhja		abe{VP_OR}fhja		abd{VP_OR}fhja
S6	abd{VP_OR}fhib		abc{VP_OR}fgjk		abe{VP_OR}fgjk		abd{VP_OR}fgjk
	abd{VP_OR}fgik		abc{VP_OR}fgja		abe{VP_OR}fgja		abd{VP_OR}fgja
	abd{VP_OR}fgij	S7	abe{VP_OR}fhjk	S7	abd{VP_OR}fhjk	S7	abc{VP_OR}fhjk
	abd{VP_OR}fgib		abe{VP_OR}fhja		abd{VP_OR}fhja		abc{VP_OR}fhja
	abc{VP_OR}fhik		abe{VP_OR}fgjk		abd{VP_OR}fgjk		abc{VP_OR}fgjk
	abc{VP_OR}fhij		abe{VP_OR}fgja		abd{VP_OR}fgja		abc{VP_OR}fgja
	abc{VP_OR}fhib	S8	n/a	S8	abd{VP_OR}f	S8	abd{VP_OR}f
	abc{VP_OR}fgik	S9	abc{VP_OR}f	S9	n/a	S9	abc{VP_OR}f
	abc{VP_OR}fgij		abc{VP_OR}fhjk		abe{VP_OR}fhjk		abd{VP_OR}fhjk
	abc{VP_OR}fgib	END	abc{VP_OR}fgjk	END	abe{VP_OR}fgjk	END	abd{VP_OR}fgjk
	abe{VP_OR}fhjk		abe{VP_OR}fhjk		abd{VP_OR}fhjk		abc{VP_OR}fhjk
	abe{VP_OR}fhja		abe{VP_OR}fgjk		abd{VP_OR}fgjk		abc{VP_OR}fgjk
	abe{VP_OR}fgjk						
	abe{VP_OR}fgja						
	abd{VP_OR}fhjk						
S7	abd{VP_OR}fhja						
	abd{VP_OR}fgjk						
	abd{VP_OR}fgja						
	abc{VP_OR}fhjk						
	abc{VP_OR}fhja						
	abc{VP_OR}fgjk						
	abc{VP_OR}fgja						
S8	abd{VP_OR}f						
S9	abc{VP_OR}f						
	abe{VP_OR}fhjk						
	abe{VP_OR}fgjk						
END	abd{VP_OR}fhjk						
	abd{VP_OR}fgjk						
	abc{VP_OR}fhjk						
	abc{VP_OR}fgjk						

Table A.5: Wi Sequences of State Pairs per Product

PRODUCT S2			PRODUCT S8			PRODUCT S9		
STATE PAIR		Wi	STATE PAIR		Wi	STATE PAIR		Wi
Start	S0	a	Start	S0	a	Start	S0	a
Start	S1	a	Start	S1	a	Start	S1	a
Start	S2	a	Start	S3	a	Start	S3	a
Start	S3	a	Start	S4	a	Start	S4	a
Start	S4	a	Start	S5	a	Start	S5	a
Start	S5	a	Start	S6	a	Start	S6	a
Start	S6	a	Start	S7	a	Start	S7	a
Start	S7	a	Start	S8	a	Start	S9	a
Start	End	a	Start	End	a	Start	End	a
S0	S1	a	S0	S1	a	S0	S1	a
S0	S2	a	S0	S3	a	S0	S3	a
S0	S3	a	S0	S4	a	S0	S4	a
S0	S4	a	S0	S5	a	S0	S5	a
S0	S5	a	S0	S6	a	S0	S6	a
S0	S6	a	S0	S7	a	S0	S7	a
S0	S7	a	S0	S8	a	S0	S9	a
S0	End	a	S0	End	a	S0	End	a
S1	S2	b	S1	S3	b	S1	S3	b
S1	S3	b	S1	S4	b	S1	S4	b
S1	S4	b	S1	S5	b	S1	S5	b
S1	S5	b	S1	S6	b	S1	S6	b
S1	S6	b	S1	S7	b	S1	S7	b
S1	S7	b	S1	S8	b	S1	S9	b
S1	End	b	S1	End	b	S1	End	b
S2	S3	e{VP_OR}	S3	S4	f	S3	S4	f
S2	S4	e{VP_OR}	S3	S5	f	S3	S5	f
S2	S5	e{VP_OR}	S3	S6	f	S3	S6	f
S2	S6	e{VP_OR}	S3	S7	f	S3	S7	f
S2	S7	e{VP_OR}	S3	S8	f	S3	S9	f
S2	End	e{VP_OR}	S3	End	f	S3	End	f
S3	S4	f	S4	S5	g	S4	S5	g
S3	S5	f	S4	S6	g	S4	S6	g
S3	S6	f	S4	S7	g	S4	S7	g
S3	S7	f	S4	S8	g	S4	S9	g
S3	End	f	S4	End	g	S4	End	g
S4	S5	g	S5	S6	h	S5	S6	h
S4	S6	g	S5	S7	h	S5	S7	h
S4	S7	g	S5	S8	h	S5	S9	h
S4	End	g	S5	End	h	S5	End	h
S5	S6	h	S6	S7	i	S6	S7	i
S5	S7	h	S6	S8	i	S6	S9	i
S5	End	h	S6	End	i	S6	End	i
S6	S7	i	S7	S8	j	S7	S9	j
S6	End	i	S7	End	j	S7	End	j
S7	End	j	S8	End	d{VP_OR}	S9	End	c{VP_OR}

Table A.6: Wi Sequences of States Pairs per Product

PRODUCT S2 - WI PER STATE		PRODUCT S8 - WI PER STATE		PRODUCT S9 - WI PER STATE	
Start	a	Start	a	Start	a
S0	a	S0	a	S0	a
S1	ab	S1	ab	S1	ab
S2	abe{VP_OR}	S3	abf	S3	abf
S3	abe{VP_OR}f	S4	abfg	S4	abfg
S4	abe{VP_OR}fg	S5	abfgh	S5	abfgh
S5	abe{VP_OR}fgh	S6	abfghi	S6	abfghi
S6	abe{VP_OR}fghi	S7	abfghij	S7	abfghij
S7	abe{VP_OR}fghij	S8	abfghijd{VP_OR}	S9	abfghijc{VP_OR}
End	abe{VP_OR}fghij	End	abfghijd{VP_OR}	End	abfghijc{VP_OR}

Table A.7: Wi Sequences of States Pairs per Product 2

PRODUCT S2,S8,S9

STATE PAIR		Wi||
Start	S0	a
Start	S1	a
Start	S2	a
Start	S3	a
Start	S4	a
Start	S5	a
Start	S6	a
Start	S7	a
Start	S8	a
Start	S9	a
Start	End	a
S0	S1	a
S0	S2	a
S0	S3	a
S0	S4	a
S0	S5	a
S0	S6	a
S0	S7	a
S0	S8	a
S0	S9	a
S0	End	a
S1	S2	b
S1	S3	b
S1	S4	b
S1	S5	b
S1	S6	b
S1	S7	b
S1	S8	b
S1	S9	b
S1	End	b
S2	S3	e{VP_OR}
S2	S4	e{VP_OR}
S2	S5	e{VP_OR}
S2	S6	e{VP_OR}
S2	S7	e{VP_OR}
S2	S8	e{VP_OR}
S2	S9	e{VP_OR}
S2	End	e{VP_OR}
S3	S4	f
S3	S5	f
S3	S6	f
S3	S7	f
S3	S8	f
S3	S9	f
S3	End	f
S4	S5	g
S4	S6	g
S4	S7	g
S4	S8	g
S4	S9	g
S4	End	g
S5	S6	h
S5	S7	h
S5	S8	h
S5	S9	h
S5	End	h
S6	S7	i
S6	S8	i
S6	S9	i
S6	End	i
S7	S8	j
S7	S9	j
S7	End	j
S8	S9	d{VP_OR}
S8	End	d{VP_OR}
S9	End	c{VP_OR}

PRODUCT S9,S2

STATE PAIR		Wi
Start	S0	a
Start	S1	a
Start	S2	a
Start	S3	a
Start	S4	a
Start	S5	a
Start	S6	a
Start	S7	a
Start	S9	a
Start	End	a
S0	S1	a
S0	S2	a
S0	S3	a
S0	S4	a
S0	S5	a
S0	S6	a
S0	S7	a
S0	S9	a
S0	End	a
S1	S2	b
S1	S3	b
S1	S4	b
S1	S5	b
S1	S6	b
S1	S7	b
S1	S9	b
S1	End	b
S2	S3	e{VP_OR}
S2	S4	e{VP_OR}
S2	S5	e{VP_OR}
S2	S6	e{VP_OR}
S2	S7	e{VP_OR}
S2	S9	e{VP_OR}
S2	End	e{VP_OR}
S3	S4	f
S3	S5	f
S3	S6	f
S3	S7	f
S3	S9	f
S3	End	f
S4	S5	g
S4	S6	g
S4	S7	g
S4	S9	g
S4	End	g
S5	S6	h
S5	S7	h
S5	S9	h
S5	End	h
S6	S7	i
S6	S9	i
S6	End	i
S7	S9	j
S7	End	j
S9	End	d{VP_OR}

PRODUCT S8-S2

STATE PAIR		Wi
Start	S0	a
Start	S1	a
Start	S2	a
Start	S3	a
Start	S4	a
Start	S5	a
Start	S6	a
Start	S7	a
Start	S8	a
Start	End	a
S0	S1	a
S0	S2	a
S0	S3	a
S0	S4	a
S0	S5	a
S0	S6	a
S0	S7	a
S0	S8	a
S0	End	a
S1	S2	b
S1	S3	b
S1	S4	b
S1	S5	b
S1	S6	b
S1	S7	b
S1	S8	b
S1	End	b
S2	S3	e{VP_OR}
S2	S4	e{VP_OR}
S2	S5	e{VP_OR}
S2	S6	e{VP_OR}
S2	S7	e{VP_OR}
S2	S8	e{VP_OR}
S2	End	e{VP_OR}
S3	S4	f
S3	S5	f
S3	S6	f
S3	S7	f
S3	S8	f
S3	End	f
S4	S5	g
S4	S6	g
S4	S7	g
S4	S9	g
S4	End	g
S5	S6	h
S5	S7	h
S5	S8	h
S5	End	h
S6	S7	i
S6	S8	i
S6	End	i
S7	S8	j
S7	End	j
S9	End	c{VP_OR}

PRODUCT S8-S9

STATE PAIR		Wi
Start	S0	a
Start	S1	a
Start	S3	a
Start	S4	a
Start	S5	a
Start	S6	a
Start	S7	a
Start	S8	a
Start	S9	a
Start	End	a
S0	S1	a
S0	S3	a
S0	S4	a
S0	S5	a
S0	S6	a
S0	S7	a
S0	S8	a
S0	S9	a
S0	End	a
S1	S3	b
S1	S4	b
S1	S5	b
S1	S6	b
S1	S7	b
S1	S8	b
S1	S9	b
S1	End	b
S3	S4	f
S3	S5	f
S3	S6	f
S3	S7	f
S3	S8	f
S3	S9	f
S3	End	f
S4	S5	g
S4	S6	g
S4	S7	g
S4	S8	g
S4	S9	g
S4	End	g
S5	S6	h
S5	S7	h
S5	S8	h
S5	S9	h
S5	End	h
S6	S7	i
S6	S8	i
S6	S9	i
S6	End	i
S7	S8	j
S7	S9	j
S7	End	j
S8	S9	d{VP_OR}
S8	End	d{VP_OR}
S9	End	c{VP_OR}

Table A.8: Wi Sequences of States Pairs per Product 2

Wi PER STATE - PRODUCT S2,S8,S9	
Start	a
S0	a
S1	ab
S2	abe{VP_OR}
S3	abe{VP_OR}f
S4	abe{VP_OR}fg
S5	abe{VP_OR}fgh
S6	abe{VP_OR}fghi
S7	abe{VP_OR}fghij
S8	abe{VP_OR}fghijd{VP_OR}
S9	abe{VP_OR}fghijd{VP_OR}
End	abe{VP_OR}fghijd{VP_OR}c{VP_OR}

Wi PER STATE - PRODUCT S9,S2	
Start	a
S0	a
S1	ab
S2	abe{VP_OR}
S3	abe{VP_OR}f
S4	abe{VP_OR}fg
S5	abe{VP_OR}fgh
S6	abe{VP_OR}fghi
S7	abe{VP_OR}fghij
S9	abe{VP_OR}fghijd{VP_OR}
End	abe{VP_OR}fghijd{VP_OR}

Wi PER STATE - PRODUCT S8-S2	
Start	a
S0	a
S1	ab
S2	abe{VP_OR}
S3	abe{VP_OR}f
S4	abe{VP_OR}fg
S5	abe{VP_OR}fgh
S6	abe{VP_OR}fghi
S7	abe{VP_OR}fghij
S8	abe{VP_OR}fghijd{VP_OR}
End	abe{VP_OR}fghijd{VP_OR}

Wi PER STATE - PRODUCT S8-S9	
Start	a
S0	a
S1	ab
S3	abf
S4	abfg
S5	abfgh
S6	abfghi
S7	abfghij
S8	abfghijd{VP_OR}
S9	abfghijd{VP_OR}c{VP_OR}
End	abfghijd{VP_OR}c{VP_OR}

Table A.9: Table to Support the Final Test Sequence Generation for the UIO Method

Source State	Input	Output	Target State	State Cover	UIO
Start	a	1	S0	ϵ	a
Start	b	enpty	Start	ϵ	a
Start	{dec}$_{V}P\_OR$	enpty	Start	ϵ	a
Start	{fff}	enpty	Start	ϵ	a
Start	f	enpty	Start	ϵ	a
Start	g	enpty	Start	ϵ	a
Start	h	enpty	Start	ϵ	a
Start	i	enpty	Start	ϵ	a
Start	j	enpty	Start	ϵ	a
Start	k	enpty	Start	ϵ	a
S0	a	enpty	S0	a	b
S0	b	2	S1	a	b
S0	{dec}$_{V}P\_OR$	enpty	S0	a	b
S0	{fff}	enpty	S1	a	b
S0	f	enpty	S2	a	b
S0	g	enpty	S3	a	b
S0	h	enpty	S4	a	b
S0	i	enpty	S5	a	b
S0	j	enpty	S6	a	b
S0	k	enpty	S7	a	b
S1	a	enpty	S1	a,b	{dec}_VP_OR
S1	b	enpty	S1	a,b	{dec}_VP_OR
S1	{dec}$_{V}P\_OR$	04|04|06	VP_S1	a,b	{dec}_VP_OR
S1	{fff}	enpty	S1	a,b	{dec}_VP_OR
S1	f	enpty	S2	a,b	{dec}_VP_OR
S1	g	enpty	S3	a,b	{dec}_VP_OR
S1	h	enpty	S4	a,b	{dec}_VP_OR
S1	i	enpty	S5	a,b	{dec}_VP_OR
S1	j	enpty	S6	a,b	{dec}_VP_OR
S1	k	enpty	S7	a,b	{dec}_VP_OR
S3	a	enpty	S3	a,b,{dec}$_{VP\_or}${fff}	g
S3	b	enpty	S3	a,b,{dec}$_{VP\_or}${fff}	g
S3	{dec}$_{V}P\_OR$	enpty	S3	a,b,{dec}$_{VP\_or}${fff}	g
S3	{fff}	enpty	S3	a,b,{dec}$_{VP\_or}${fff}	g
S3	f	9	S3	a,b,{dec}$_{VP\_or}${fff}	g
S3	g	11	S4	a,b,{dec}$_{VP\_or}${fff}	g
S3	h	10	S5	a,b,{dec}$_{VP\_or}${fff}	g
S3	i	enpty	S6	a,b,{dec}$_{VP\_or}${fff}	g
S3	j	enpty	S3	a,b,{dec}$_{VP\_or}${fff}	g
S3	k	enpty	S3	a,b,{dec}$_{VP\_or}${fff}	g
S4	a	enpty	S4	a,b,{dec}$_{VP\_or}$,{fff},g	i
S4	b	enpty	S4	a,b,{dec}$_{VP\_or}$,{fff},g	i
S4	{dec}$_{V}P\_OR$	enpty	S4	a,b,{dec}$_{VP\_or}$,{fff},g	i
S4	{fff}	enpty	S4	a,b,{dec}$_{VP\_or}$,{fff},g	i
S4	f	enpty	S4	a,b,{dec}$_{VP\_or}$,{fff},g	i
S4	g	enpty	S4	a,b,{dec}$_{VP\_or}$,{fff},g	i
S4	h	enpty	S4	a,b,{dec}$_{VP\_or}$,{fff},g	i
S4	i	13	S6	a,b,{dec}$_{VP\_or}$,{fff},g	i
S4	j	enpty	S4	a,b,{dec}$_{VP\_or}$,{fff},g	i
S4	k	enpty	S4	a,b,{dec}$_{VP\_or}$,{fff},g	i
S5	a	enpty	S5	a,b,{dec}$_{VP\_or}$,{fff},h	i
S5	b	enpty	S5	a,b,{dec}$_{VP\_or}$,{fff},h	i
S5	{dec}$_{V}P\_OR$	enpty	S5	a,b,{dec}$_{VP\_or}$,{fff},h	i
S5	{fff}	enpty	S5	a,b,{dec}$_{VP\_or}$,{fff},h	i
S5	f	enpty	S5	a,b,{dec}$_{VP\_or}$,{fff},h	i
S5	g	enpty	S5	a,b,{dec}$_{VP\_or}$,{fff},h	i
S5	h	enpty	S5	a,b,{dec}$_{VP\_or}$,{fff},h	i
S5	i	12	S6	a,b,{dec}$_{VP\_or}$,{fff},h	i
S5	j	enpty	S5	a,b,{dec}$_{VP\_or}$,{fff},h	i
S5	k	enpty	S5	a,b,{dec}$_{VP\_or}$,{fff},h	i
S6	a	enpty	S6	a,b,{dec}$_{VP\_or}$,{fff},h,i	k
S6	b	14	S1	a,b,{dec}$_{VP\_or}$,{fff},h,i	k
S6	{dec}$_{V}P\_OR$	enpty	S6	a,b,{dec}$_{VP\_or}$,{fff},h,i	k
S6	{fff}	enpty	S6	a,b,{dec}$_{VP\_or}$,{fff},h,i	k
S6	f	enpty	S6	a,b,{dec}$_{VP\_or}$,{fff},h,i	k
S6	g	enpty	S6	a,b,{dec}$_{VP\_or}$,{fff},h,i	k
S6	h	enpty	S6	a,b,{dec}$_{VP\_or}$,{fff},h,i	k
S6	i	enpty	S6	a,b,{dec}$_{VP\_or}$,{fff},h,i	k
S6	j	16	S7	a,b,{dec}$_{VP\_or}$,{fff},h,i	k
S6	k	15	End	a,b,{dec}$_{VP\_or}$,{fff},h,i	k
S7	a	18	S0	a,b,{dec}$_{VP\_or}$,{fff},h,i,j	k
S7	b	enpty	S7	a,b,{dec}$_{VP\_or}$,{fff},h,i,j	k
S7	{dec}$_{V}P\_OR$	enpty	S7	a,b,{dec}$_{VP\_or}$,{fff},h,i,j	k
S7	{fff}	enpty	S7	a,b,{dec}$_{VP\_or}$,{fff},h,i,j	k
S7	f	enpty	S7	a,b,{dec}$_{VP\_or}$,{fff},h,i,j	k
S7	g	enpty	S7	a,b,{dec}$_{VP\_or}$,{fff},h,i,j	k
S7	h	enpty	S7	a,b,{dec}$_{VP\_or}$,{fff},h,i,j	k
S7	i	enpty	S7	a,b,{dec}$_{VP\_or}$,{fff},h,i,j	k
S7	j	enpty	S7	a,b,{dec}$_{VP\_or}$,{fff},h,i,j	k
S7	k	17	End	a,b,{dec}$_{VP\_or}$,{fff},h,i,j	k
VP_S1	a	enpty	VP_S1	a,b,{dec}$_{VP\_or}$	{fff}
VP_S1	b	enpty	VP_S1	a,b,{dec}$_{VP\_or}$	{fff}
VP_S1	{dec}$_{V}P\_OR$	enpty	VP_S1	a,b,{dec}$_{VP\_or}$	{fff}
VP_S1	{fff}	06|07|08	S3	a,b,{dec}$_{VP\_or}$	{fff}
VP_S1	f	enpty	VP_S1	a,b,{dec}$_{VP\_or}$	{fff}
VP_S1	g	enpty	VP_S1	a,b,{dec}$_{VP\_or}$	{fff}
VP_S1	h	enpty	VP_S1	a,b,{dec}$_{VP\_or}$	{fff}
VP_S1	i	enpty	VP_S1	a,b,{dec}$_{VP\_or}$	{fff}
VP_S1	j	enpty	VP_S1	a,b,{dec}$_{VP\_or}$	{fff}
VP_S1	k	enpty	VP_S1	a,b,{dec}$_{VP\_or}$	{fff}

APPENDIX B – FSM PER PRODUCT FROM AGM

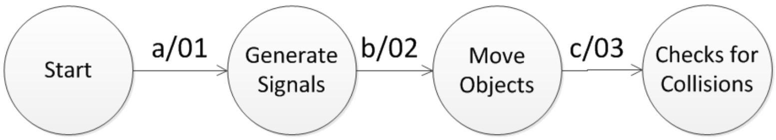


Figure B.1: FSM Animation Loop

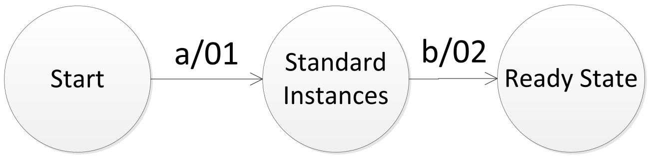


Figure B.2: FSM Initialization

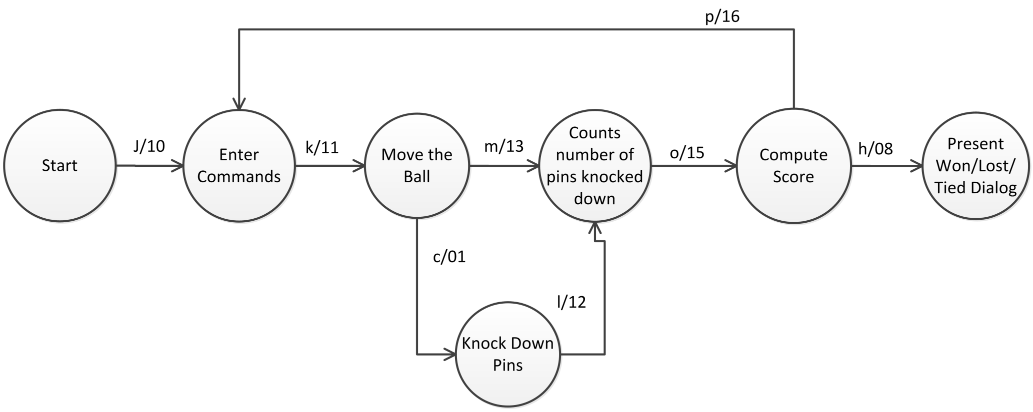


Figure B.3: FSM Bowling Moves

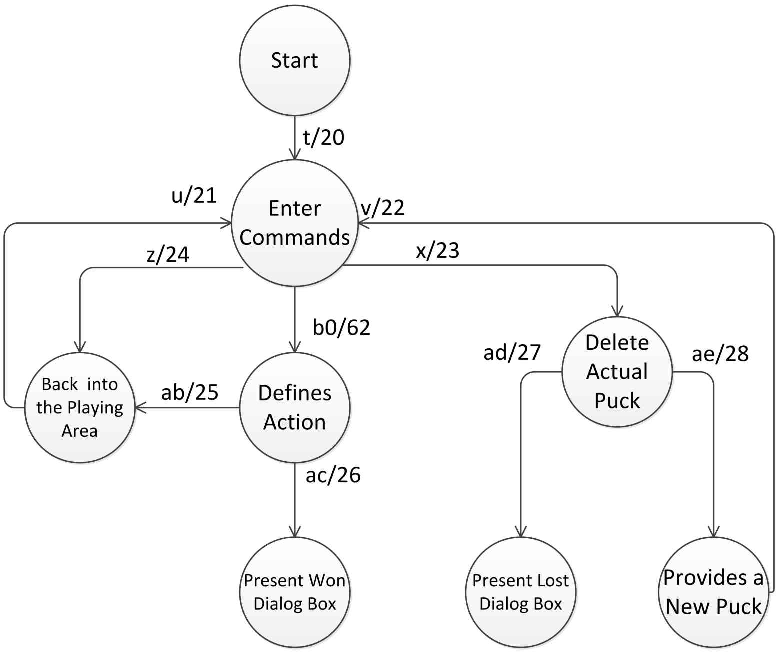


Figure B.4: FSM Brickles Moves

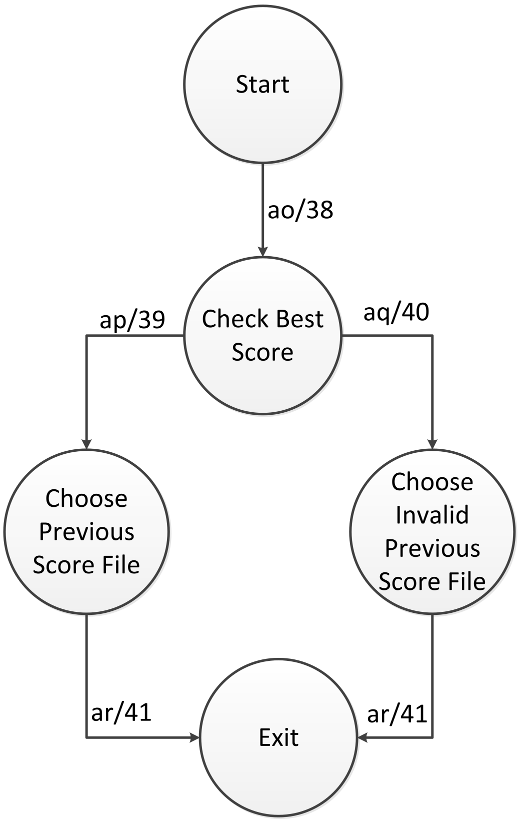


Figure B.5: FSM Check Previous Best Score

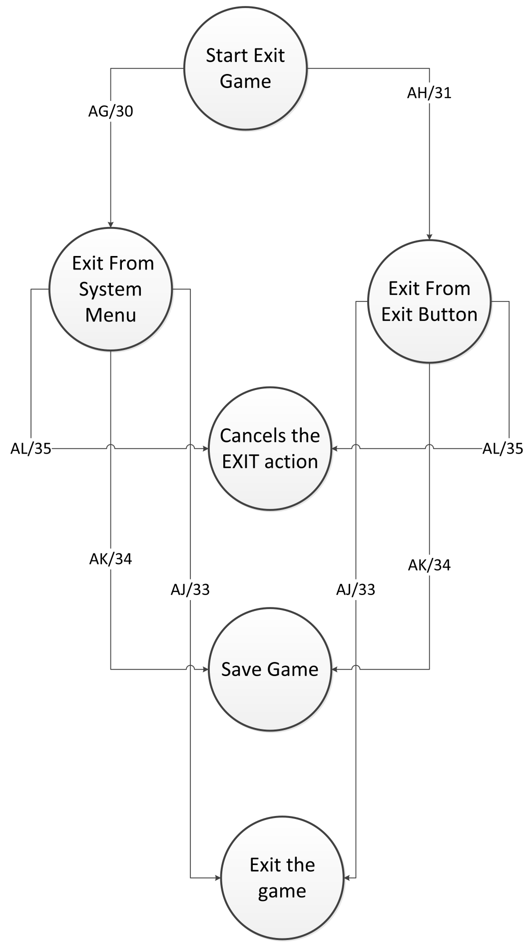


Figure B.6: FSM Exit Game

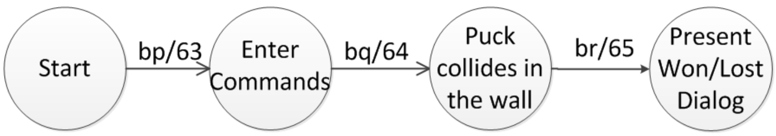


Figure B.7: FSM Pong Moves

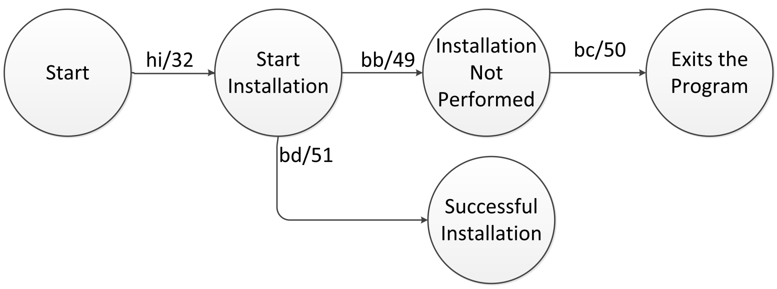


Figure B.8: FSM Install Game

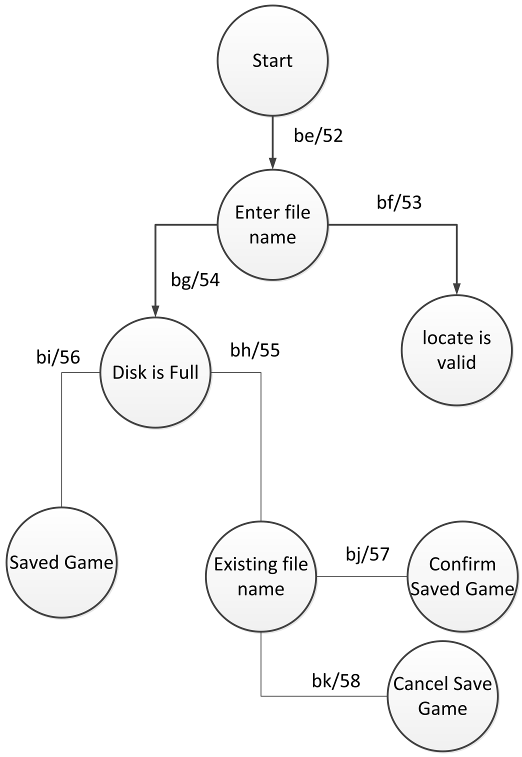


Figure B.9: FSM Save Game

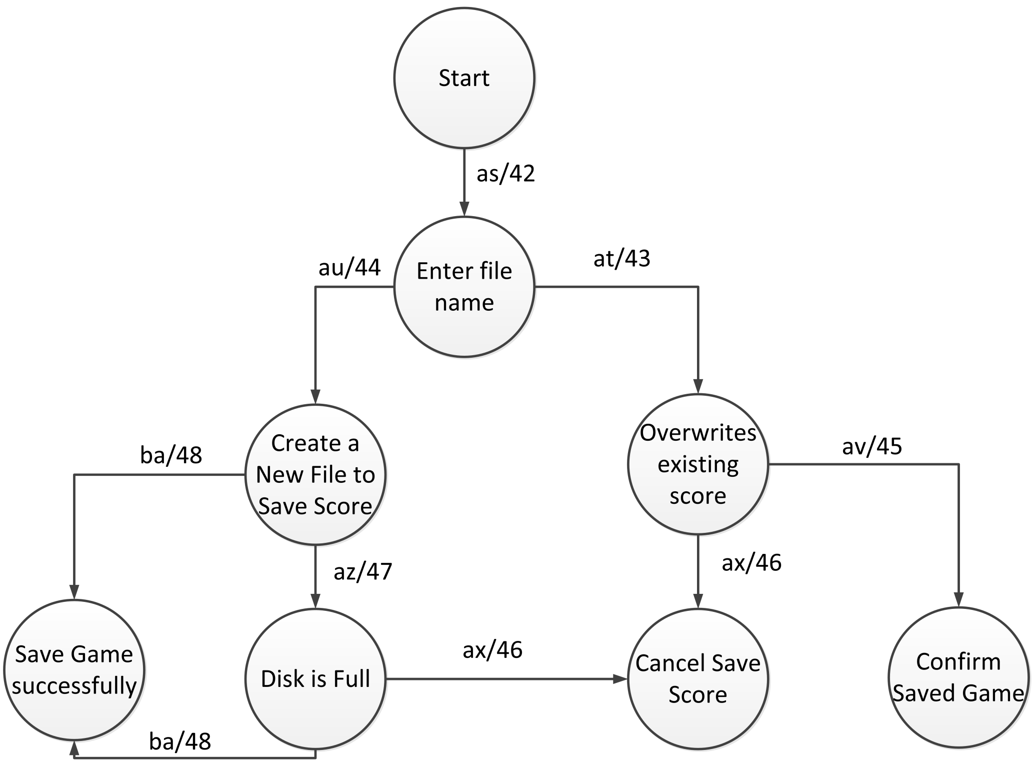


Figure B.10: FSM Save Scores

APPENDIX C – AGM - INPUT, OUTPUT AND VARIABILITY DATA

Table C.1: Actual Input, Output and Variability Information of AGM

ID	Inputs	ID	Outputs	Source States	Target States	Variability
a	Select Play from menu	01	Creates the standard instances of the required classes	Start	Standard Instances	mandatory
b	Enters the READY state	02	The gameboard is displayed	Standard Instances	Ready State	mandatory
c	Left-click Button to begin play	03	Start game action and the animation begins	Ready State	Initialize the game	mandatory
d{VP_or}	Brickles Moves	04	Brickles Moves	Initialize the game	Brickles Moves	alternative_OR
e{VP_or}	Pong Moves	05	Pong Moves	Initialize the game	Pong Moves	alternative_OR
f{VP_or}	Bowling Moves	06	Bowling Moves	Initialize the game	Bowling Moves	alternative_OR
g	Responds to Won/Lost /Tied dialog	07	Dialog to play again is presented	Present Won/Lost/Tied Dialog	Responds to WonLostTied Dialog	mandatory
				Present Won/Lost Dialog	Responds to Won/Lost/Tied Dialog	
				Present Lost Dialog Box	Responds to Won/Lost/Tied Dialog	
				Present Won Dialog Box	Responds to Won/Lost/Tied Dialog	
h	Respond "no" in the dialog to play again	08	Exit the game	Responds to Won/Lost/Tied Dialog	Exit	mandatory
i	Respond "yes" in the dialog to play again	9	Returns the gameboard to its initialized, ready-to-play state	Exit	Standard Instances	mandatory
j	Positions the mouse and left-clicks to send ball down alley	10	Ball starts to move	Bowling Moves	Enter Commands	mandatory (sub-FSM Bowling Moves)
k	Move the ball down the alley using a randomly selected algorithm	11	Ball reaches the pins or not	Enter Commands	Move the Ball	mandatory (sub-FSM Bowling Moves)
l	Ball reaches the pins	12	Pins are knocked down as determined by the physics of the collision	Move the Ball	Knock Down Pins	mandatory (sub-FSM Bowling Moves)
m	Ball NOT reaches the pins	13	Pins are not knocked down as determined by the physics of the collision	Move the Ball	Counts number of pins knocked down	mandatory (sub-FSM Bowling Moves)
n	Counts number of pins knocked down	14	Number of pins knocked down are displayed	Knock Down Pins	Counts number of pins knocked down	mandatory (sub-FSM Bowling Moves)
o	System computes the score	15	The score is displayed and the number of actions are incremented	Counts number of pins knocked down	Compute Score	mandatory (sub-FSM Bowling Moves)
p	Game starts a new action	16	Gameboard in the initial state is presented	Compute Score	Enter Commands	mandatory (sub-FSM Bowling Moves)
q	The last game action is performed	17	The Won/Lost dialog is presented	Compute Score	Present Won/Lost/Tied Dialog	mandatory (sub-FSM Bowling Moves)
r	Left-clicks or uses the keyboard to enter commands	16	Paddles andd puck start to move	Pong Moves	Enter Commands	mandatory (sub-FSM Pong Moves)
s	Let the puck collide into the walls	19	Based on the rules, the puck is absorbed or changes direction according to the laws of physics	Enter Commands	Present Won/Lost Dialog	mandatory (sub-FSM Pong Moves)
t	Left-clicks or uses the keyboard to enter commands	20	Moves the paddle horizontally to follow the mouse track	Brickles Moves	Brickles Enter Commands	mandatory (sub-FSM Brickles Moves)
u	Puck still in movement	21	System checks for a collision with another object	Back into the Playing Area	Brickles Enter Commands	mandatory (sub-FSM Brickles Moves)
v	New puck begins its movement	22	System checks for a collision with another object	Provides a New Puck	Brickles Enter Commands	mandatory (sub-FSM Brickles Moves)
x	Puck collides with the floor	23	Puck ceases to exist	Brickles Enter Commands	Delete Actual Puck	mandatory (sub-FSM Brickles Moves)
z	Puck collides with the ceiling or wall	24	Puck is reflected back into the playing area	Brickles Enter Commands	Back into the Playing Area	mandatory (sub-FSM Brickles Moves)
ab	Puck collides with a brick	25	Puck is reflected back into the playing area	Defines Action	Back into the Playing Area	mandatory (sub-FSM Brickles Moves)
ac	Puck collides with the last brick	26	The Won dialog is presented	Defines Action	Present Won Dialog Box	mandatory (sub-FSM Brickles Moves)
ad	Maximum number of pucks has been reached	27	The Lost dialog is presented	Delete Actual Puck	Present Lost Dialog Box	mandatory (sub-FSM Brickles Moves)
ae	Maximum number of pucks has not been reached	28	A new puck is provided	Delete Actual Puck	Provides a New Puck	mandatory (sub-FSM Brickles Moves)
bo	Puck collides with a brick	62	Puck is absorbed or changes direction according to the laws of physics	Enter Commands	Defines Action	mandatory (sub-FSM Brickles Moves)

af	Let the puck collide into the walls	29	Based on the rules, the puck is absorbed or changes direction according to the laws of physics	Enter Commands	Puck collides in the wall	mandatory (sub-FSM Pong Moves)
ag	Selects exit from system menu	30	Prompts actor to save or exit the game	Start (FSM Exit)	Exit From System Menu	mandatory (sub-FSM exit game)
ah	Click the log out button in the upper right corner of the game window	31	Prompts actor to save or exit the game	Start (FSM Exit)	Exit From Exit Button	mandatory (sub-FSM exit game)
aj	Choose exit game option	33	exit game	Exit From Exit Button	Exit the game	mandatory (sub-FSM exit game)
aj	Choose exit game option	33	exit game	Exit From System Menu	Exit the game	mandatory (sub-FSM exit game)
ak	Choose saves game option	34	Saves the game and exits the program	Exit From Exit Button	Save Game	mandatory (sub-FSM exit game)
ak	Choose saves game option	34	Saves the game and exits the program	Exit From System Menu	Save Game	mandatory (sub-FSM exit game)
al	Choose cancel exit game option	35	Returns to suspended action	Exit From Exit Button	Cancel the EXIT action	mandatory (sub-FSM exit game)
al	Choose cancel exit game option	35	Returns to suspended action	Exit From System Menu	Cancel the EXIT action	mandatory (sub-FSM exit game)
ao	Open Previous Best Score Window	38	Previous Best Score Window is Showed	Start Previous Best Score	Check Best Score	mandatory (sub-FSM Check Best Score)
ap	Prompts Actor to Specify a Filename	39	Sistem Reads the File and Returns Score in a Dialog Box	Check Best Score	Choose Previous Score File	mandatory (sub-FSM Check Best Score)
aq	Prompts actor to specify a filename	40	Finds that file does not exist	Check Best Score	Choose an Invalid Previous Score File	mandatory (sub-FSM Check Best Score)
ar	Selects OK on dialog box to continue	41	Returns to state before select	Choose Previous Score File	Exit Check Best Score Program	mandatory (sub-FSM Check Best Score)
ar	Selects OK on dialog box to continue	41	Returns to state before select	Choose Invalid Previous Score File	Exit Check Best Score Program	mandatory (sub-FSM Check Best Score)
as	Selects SAVE SCORE from system menu	42	Prompts actor to specify a filename	Start Save Score	Enter File Name	mandatory (sub-FSM Save Score)
at	Author enter an existing file name to save score	43	System displays message that the file already exists	Enter File Name	Overwrites existing score	mandatory (sub-FSM Save Score)
au	The author type a unique file name to save score	44	System creates a new file and save the game it	Enter File Name	Create a New File to Save Score	mandatory (sub-FSM Save Score)
av	Click on ok button	45	System overwrites existing score	Overwrites existing score	Confirm Saved Game	mandatory (sub-FSM Save Score)
ax	Click cancel button	46	Score is not saved	Overwrites existing score	Cancel Save Score	mandatory (sub-FSM Save Score)
ax	Click cancel button	46	Score is not saved	Disk is Full	Cancel Save Score	mandatory (sub-FSM Save Score)
az	User selects an invalid location to save the score file	47	System displays message that the disk is full	Create a New File to Save Score	Disk is Full	mandatory (sub-FSM Save Score)
ba	User select a valid location to save the score file	48	System save score	Create a New File to Save Score	Save Game successfully	mandatory (sub-FSM Save Score)
ba	User select a valid location to save the score file	48	System save score	Disk is full	Save Game successfully	mandatory (sub-FSM Save Score)
hi	Selects the installer executable to execute	32	Presents a file chooser to allow selection of a directory in which to place the game files	Start Install Game	Start Installation	mandatory (sub-FSM Install Game)
bb	Selects an invalid location to save the game files	49	System finds insufficient space to which to write files and displays the Out of Space dialog box	Start Installation	Installation Not Performed	mandatory (sub-FSM Install Game)
bc	Click on OK button	50	Exits the program	Installation Not Performed	Exits the Program	mandatory (sub-FSM Install Game)
bd	Selects a directory to save game files	51	Places game files in the directory and install game	Start Installation	Successful Installation	mandatory (sub-FSM Install Game)
be	Selects the SAVE option in the system menu	52	Allows the actor to specify a filename	Start Save Game	Enter File Name	mandatory (sub-FSM Save Game)
bf	Select a valid directory and valid filename to save the file	53	The game is saved	Enter File Name	Locate is valid	mandatory (sub-FSM Save Game)
bg	Reports a file name and a location to save the game	54	Raises exception because the disk is full	Enter File Name	Disk is Full	mandatory (sub-FSM Save Game)
bh	Select another directory to save the file and informs an existing file name	55	System displays message that the file already exists and asks if you want to overwrite	Disk is Full	Existing file name	mandatory (sub-FSM Save Game)
bi	Select a valid directory and valid filename to save the file	56	The game is saved	Disk is Full	Saved Game	mandatory (sub-FSM Save Game)
bj	Click on ok button	57	System overwrites existing game file	Existing file name	Confirm Saved Game	mandatory (sub-FSM Save Game)
bk	Click the cancel button	58	Game is not saved	Existing file name	Cancel Save Game	mandatory (sub-FSM Save Game)
bl	Generate periodic signals	59	Signals are send to the game	Start Animation Loop	Generate Signals	mandatory (sub-FSM Save Game)

bm	Moves all objects one step according to their movement algorithm	60	Objects are moved	Generate Signals	Move Objects	mandatory (sub-FSM Save Game)
bn	Checks for collisions executes the collision algorithms of the objects	61	Collision algorithms are executed	Move Objects	Checks for Collisions	mandatory (sub-FSM Save Game)
bp	Generate periodic signals	63	Signals are send to the game	Start Uninstall Game	Start Uninstall	mandatory (sub-FSM Uninstall Game)
bq	Selects directory where game is stored	64	Erases files in the directory and Presents the Uninstall Completed dialog box	Start Uninstall	Erases Files	mandatory (sub-FSM Uninstall Game)
br	Selects the OK button in the dialog box	65	Closes dialog box	Erases Files	Close	mandatory (sub-FSM Uninstall Game)

APPENDIX D – PLETS ACTIVITY DIAGRAMS

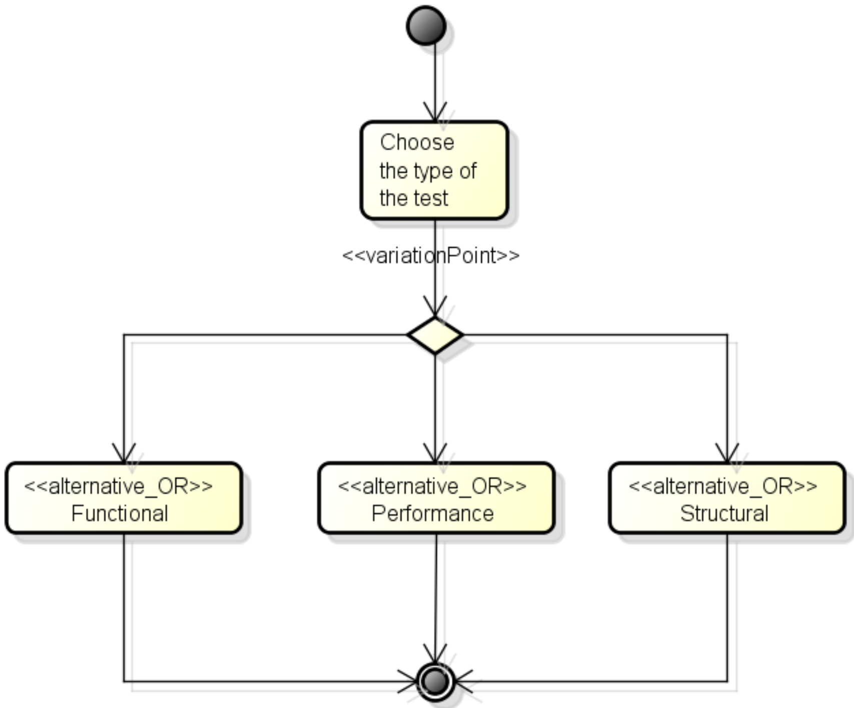


Figure D.1: Activity Diagram of Testing Type

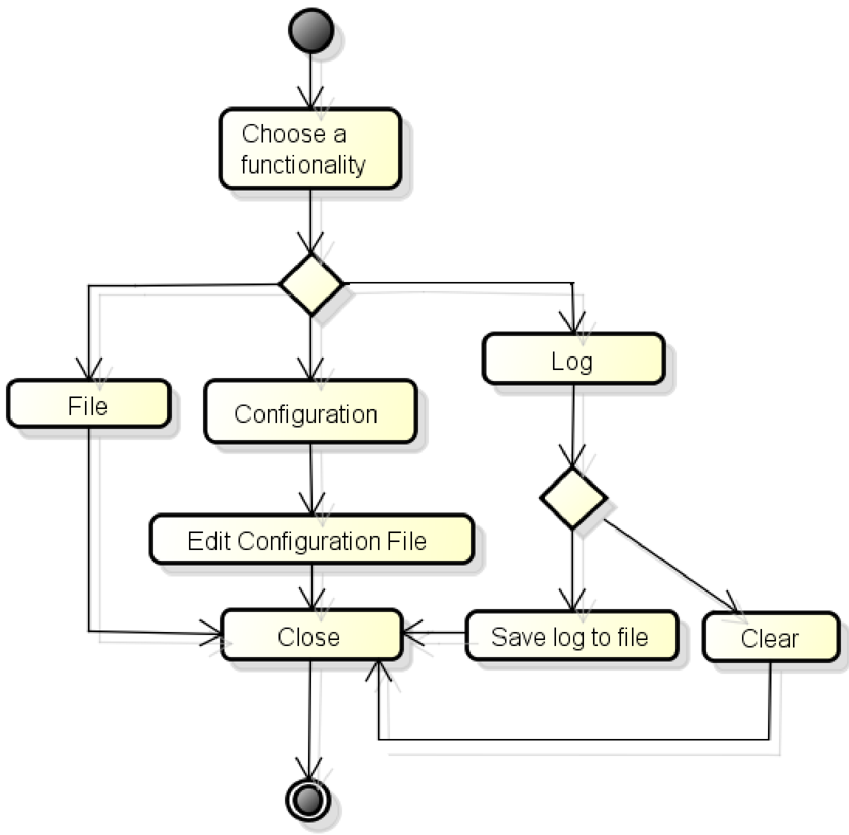


Figure D.2: Activity Diagram of Functional Testing Functionalities

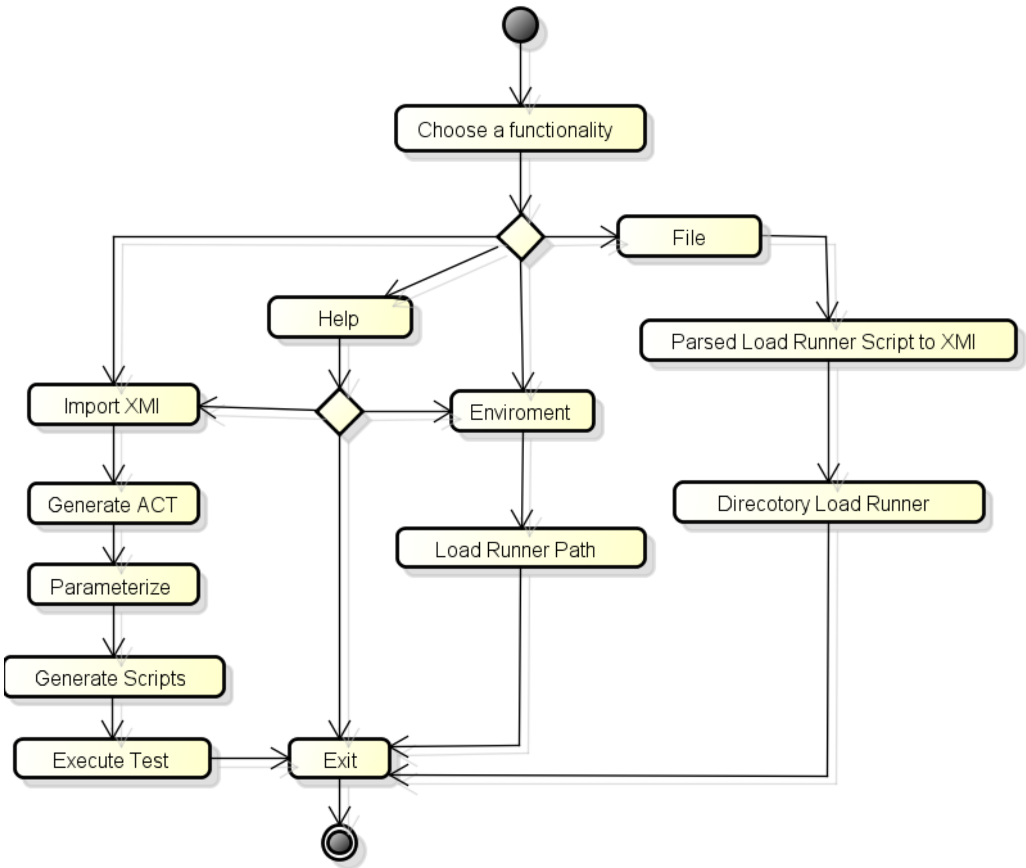


Figure D.3: Activity Diagram of Performance Testing Functionalities

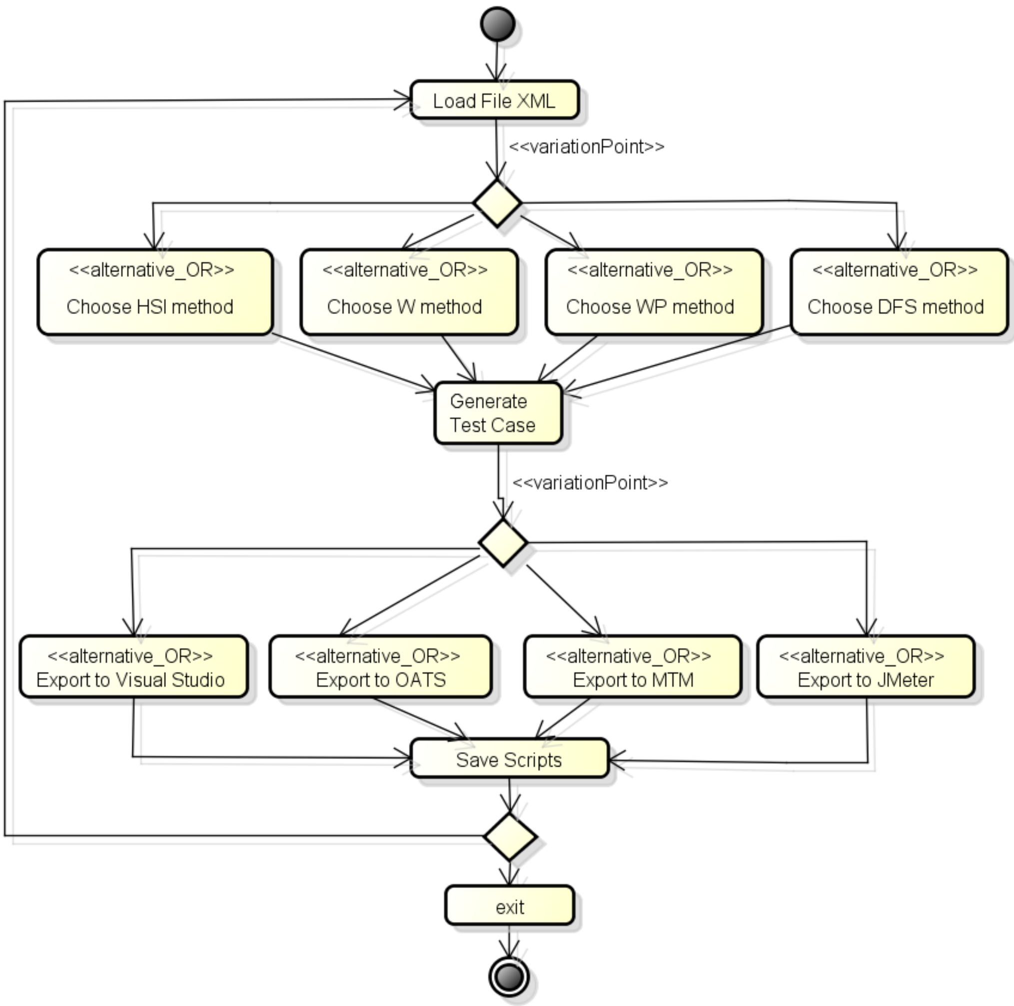


Figure D.4: Activity Diagram of Functional Testing

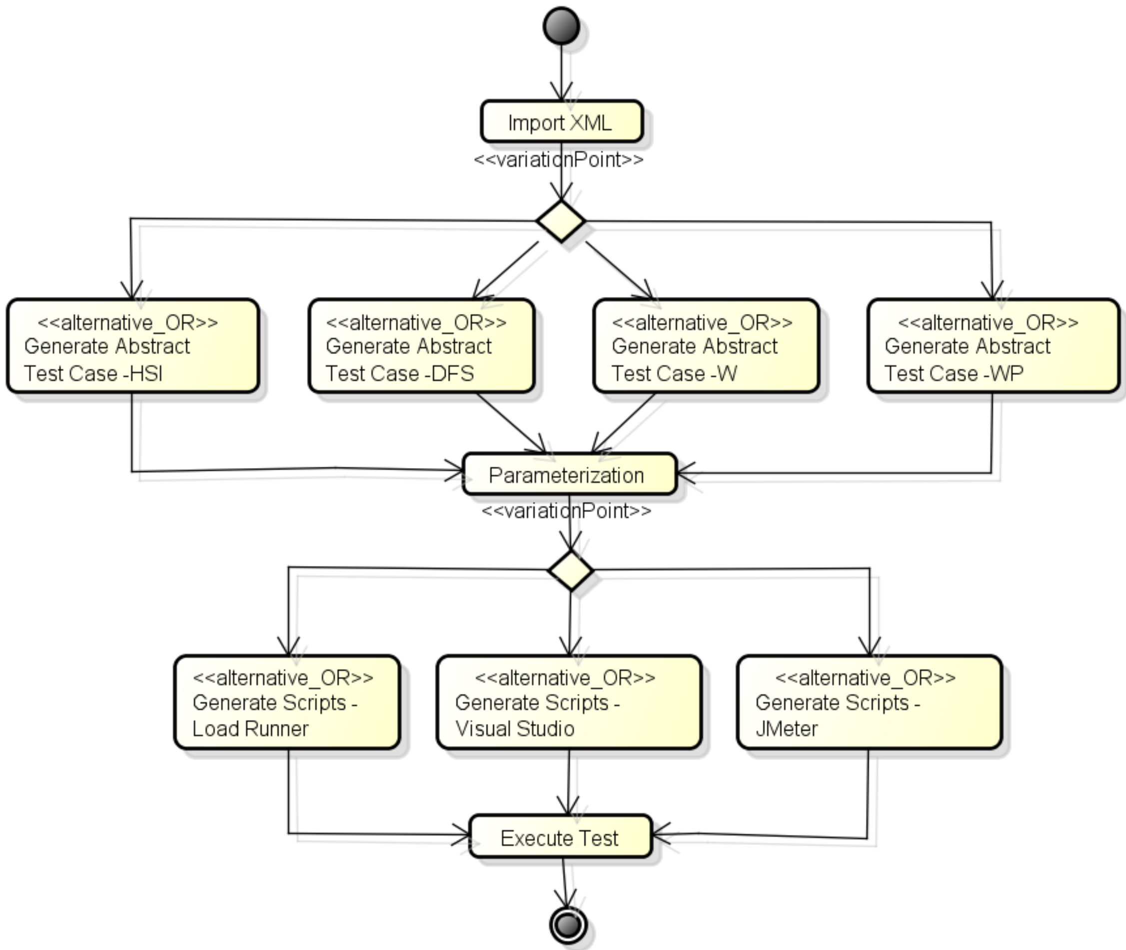


Figure D.5: Activity Diagram of Performance Testing

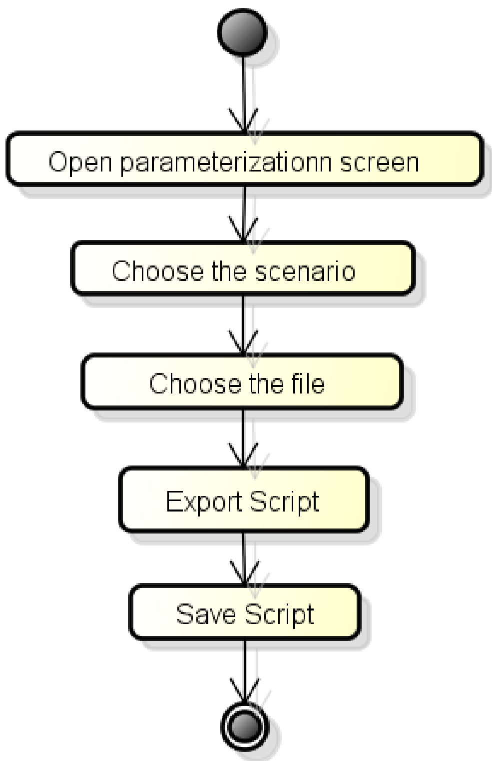


Figure D.6: Activity Diagram of Parameterization

APPENDIX E – FSMS FROM PLETS

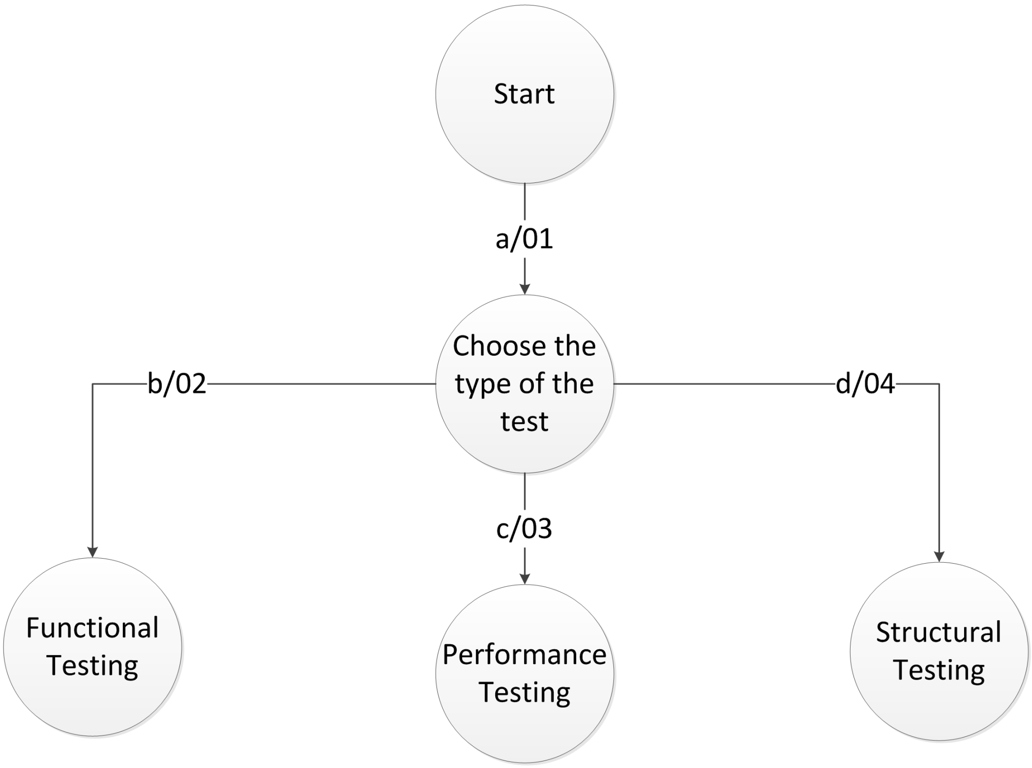


Figure E.1: FSM of Choose the type of the test

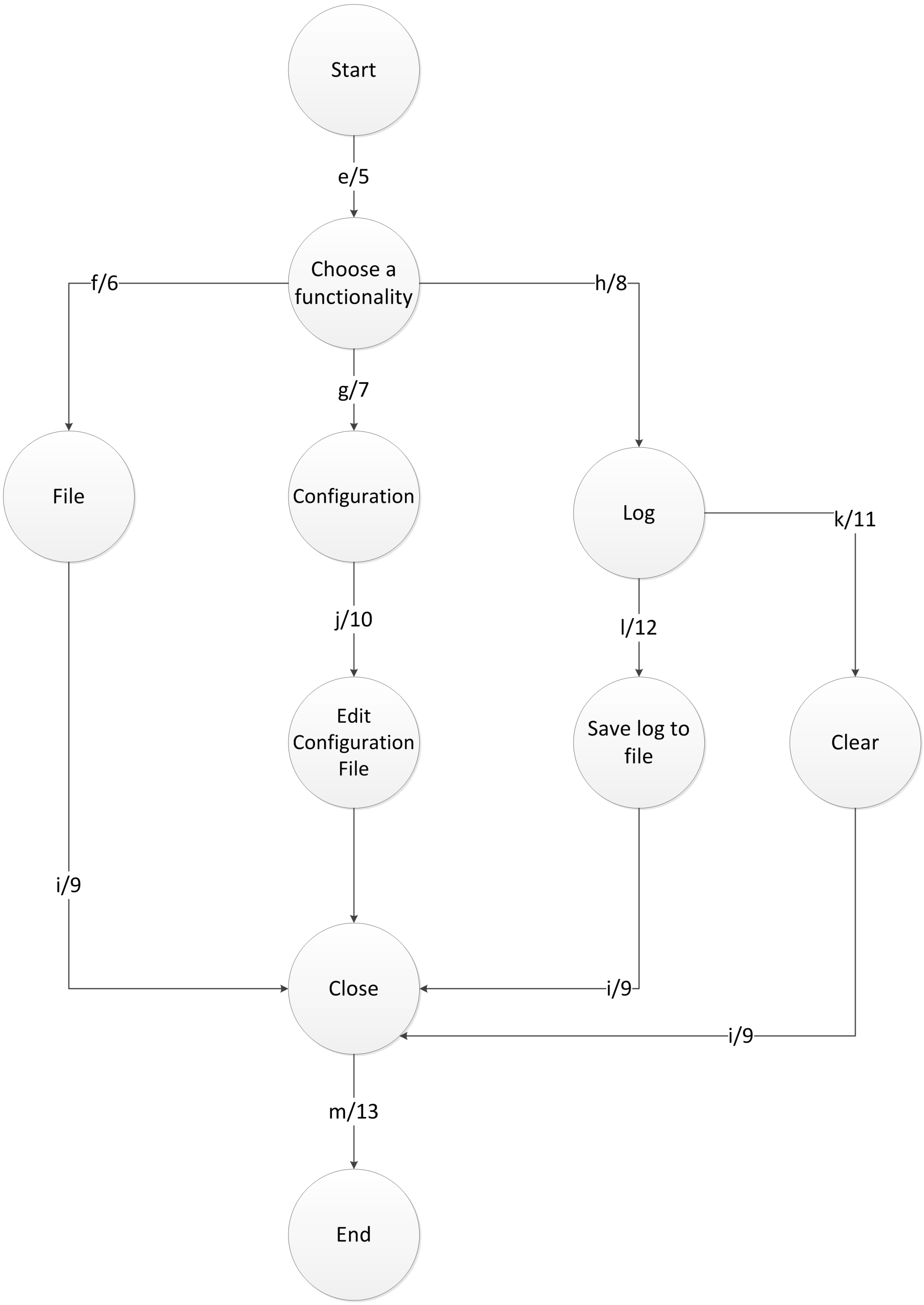


Figure E.2: FSM of Functional Functionalities

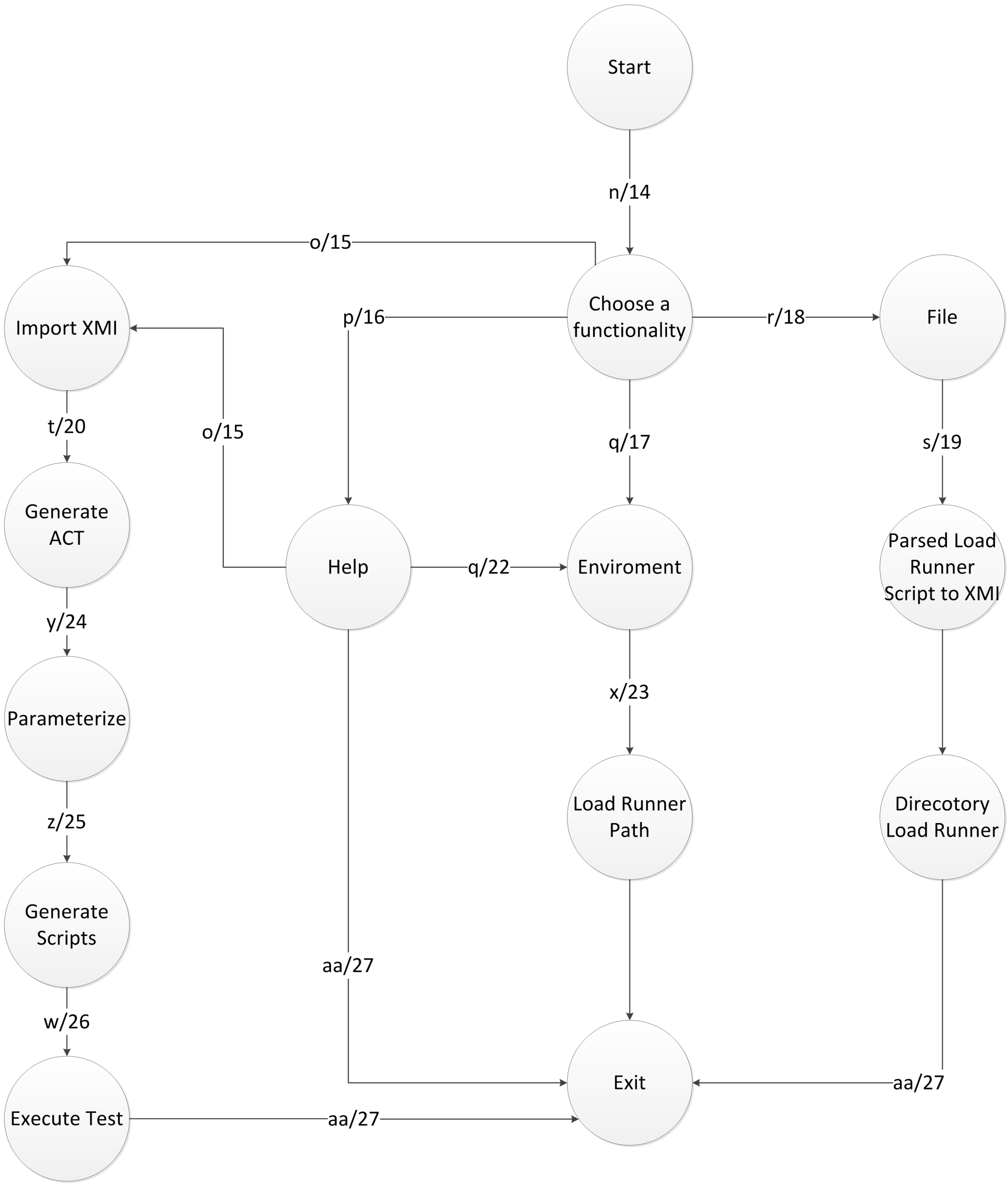


Figure E.3: FSM of Performance Functionalities

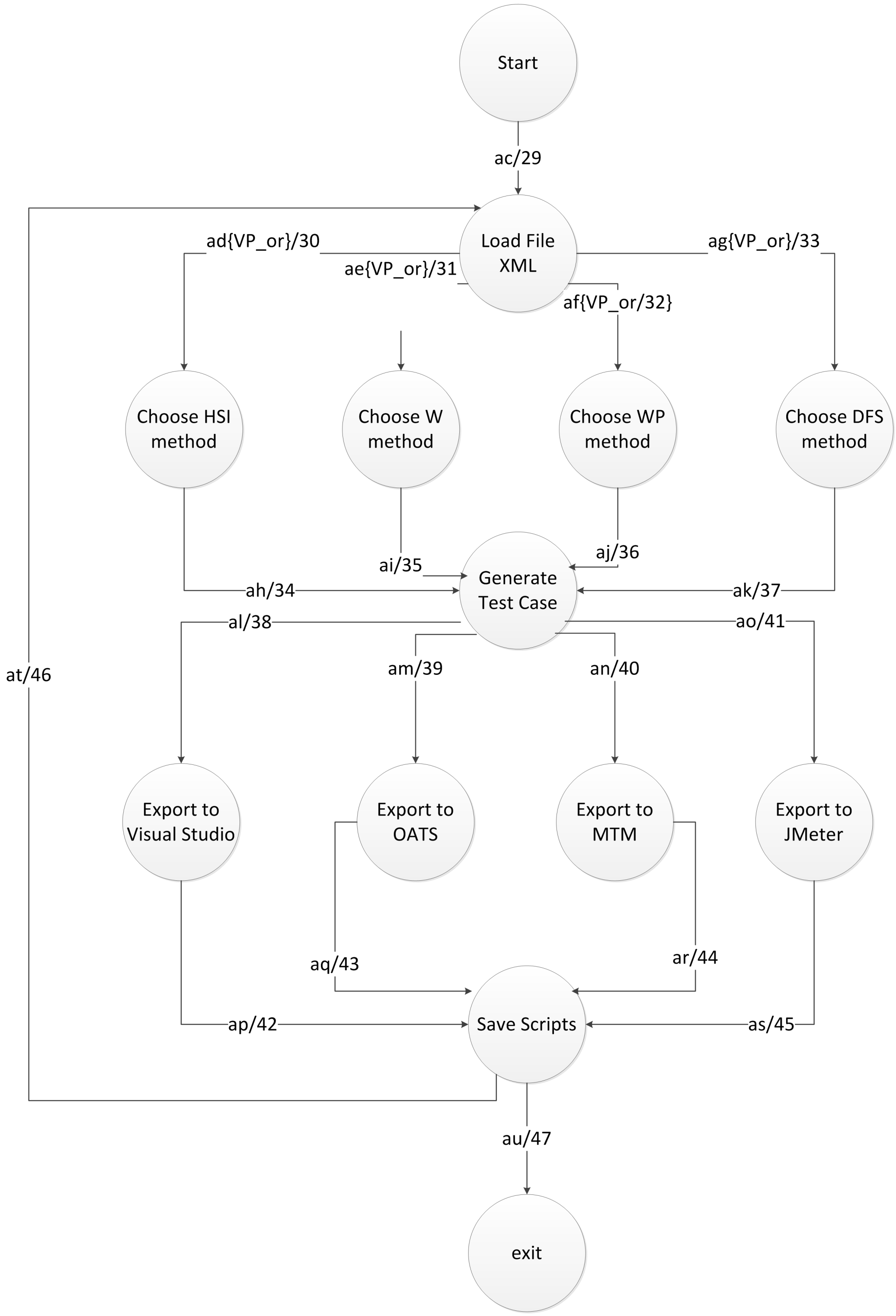


Figure E.4: FSM of Functional Testing

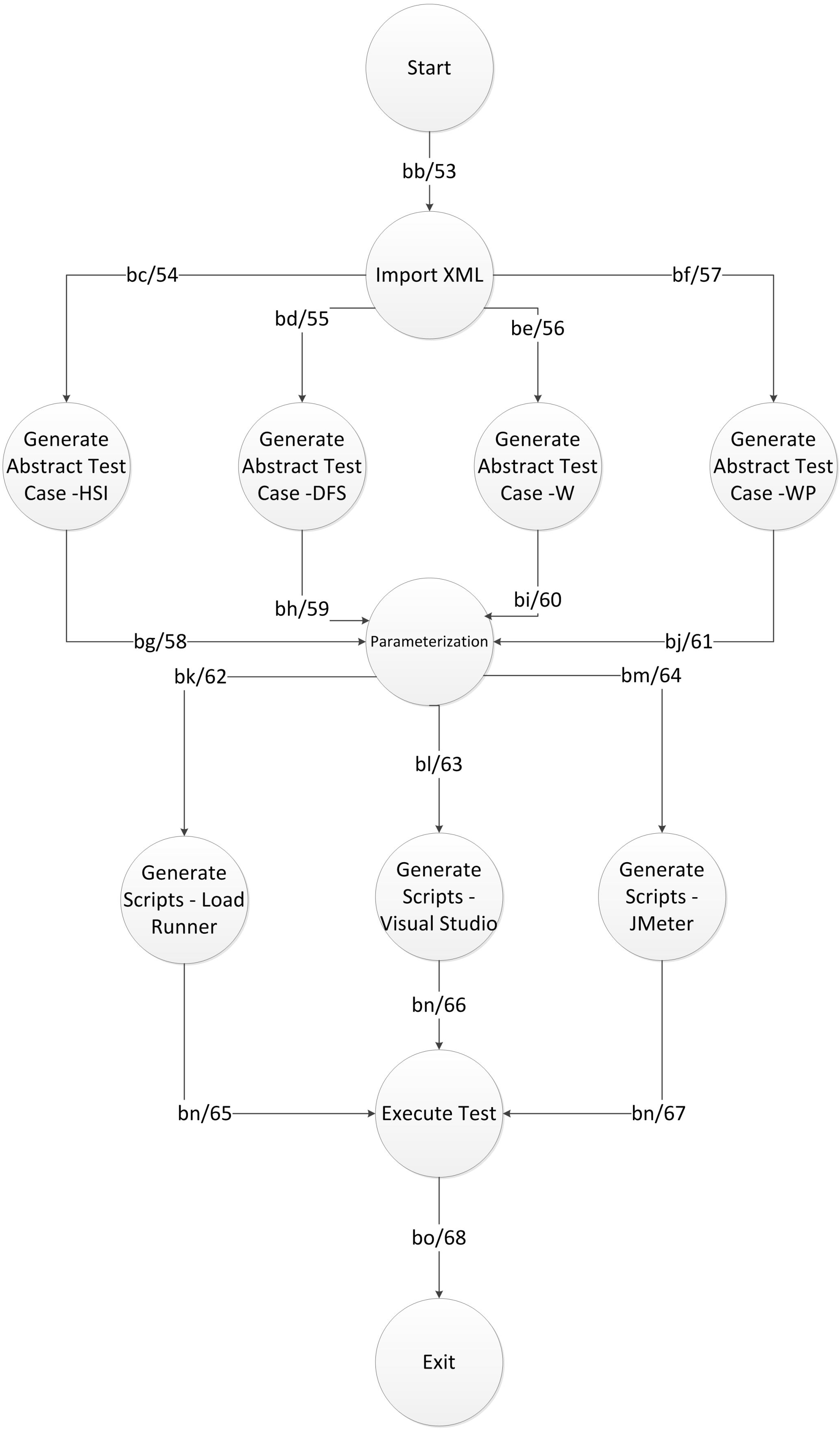


Figure E.5: FSM of Performance Testing

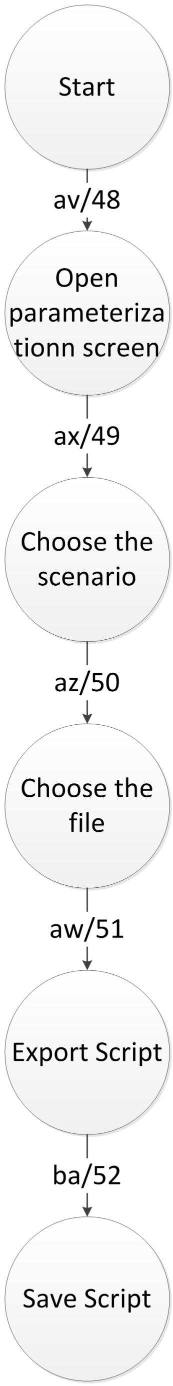


Figure E.6: FSM of Parameterization

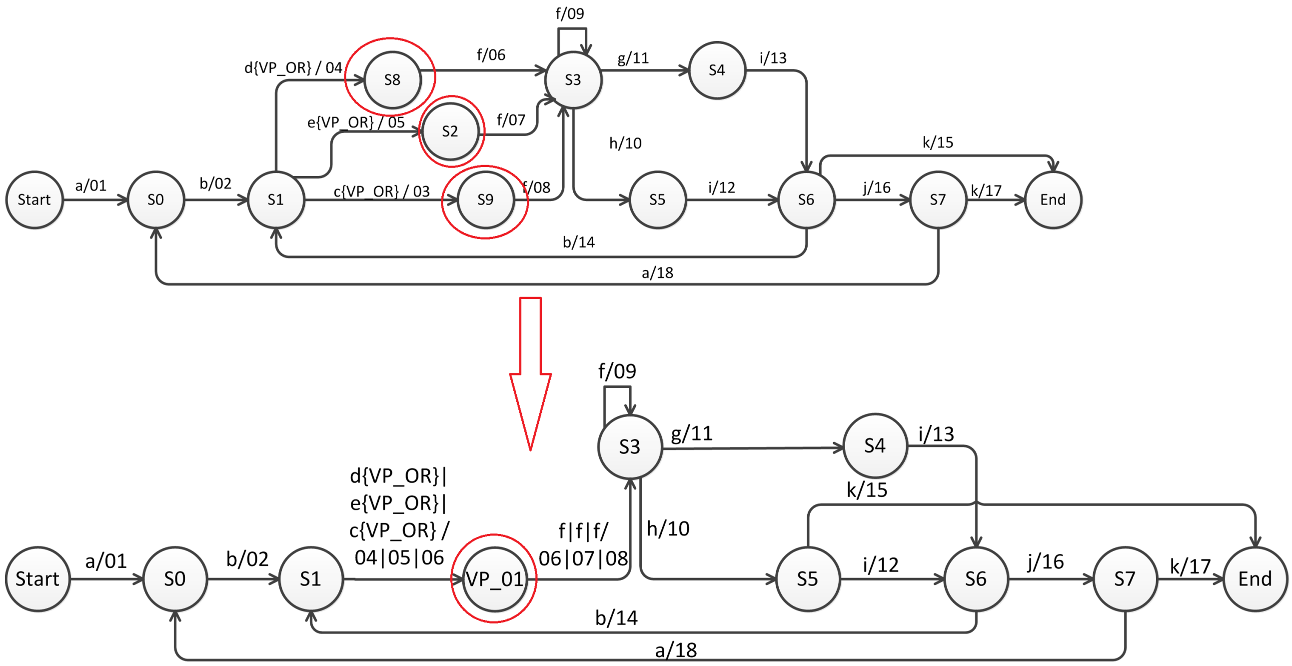


Figure E.7: FSM with Variation Point

APPENDIX F – PLETS - INPUT, OUTPUT AND IDENTIFIERS DATA

Table F.1: Actual Input, Output and Variability Information of PLeTs

Functionalities of Functional Testing			
ID	**Input**	**ID**	**Output**
e	Choose a functionality on the menu	5	Functionality chosen
f	Click on File button	6	File options on the screen
g	Click on Configuration button	7	Configuration options on the screen
h	Click on Log button	8	Will be open two options
i	Click on Close button	9	Program closed
j	Click on Edit Configuration File button	10	File configured
k	Log on the screen cleared	11	Click on Clear button
l	Click on Save log file button	12	Log on the screen saved
m	Application will close	13	Application ended
Functionalities of Performance Testing			
ID	**Input**	**ID**	**Output**
n	Open the tool	14	Functionality chosen
o	Click on import XMI/XML file button	15	File exported
p	Click on Help button	16	Help screen opened
q	Click on Environment button	17	Load Runner Path screen opened
r	Click on file button	18	Will be open two options on the screen
s	Click on Parsed Load Runner Script to XMI button	19	Directory screen opened
t	Click on Generate ATC button	20	Convert test data into generic test scenarios
u	Click on import XMI/XML file button	21	File exported
v	Click on Environment button	22	Environment screen opened
x	Choose a Load Runner path | Click on OK button	23	Path chosen
y	Click on Parameterization button	24	Configure data parameterization
z	Click on Generate Scripts button	25	Convert abstract test cases
w	Click on Execute Test button | Click on OK button	26	Invoke the executable program to run the scripts generated
aa	Click on Exit button	27	Program closed
Functional Testing			
ID	**Input**	**ID**	**Output**
ac	(Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	29	File XML loaded
ad	Select the method of test sequences generation, HSI	30	File XML loaded
ae	Select the method of test sequences generation, W	31	Method of test sequence generation W is selected
af	Select the method of test sequences generation, Wp	32	Method of test sequence generation Wp is selected
ag	Select the method of test sequences generation, DFS	33	Method of test sequence generation DFS is selected
ah	Click on generate test case from load test data button	34	The abstract test case are generated using HSI method
ai	Click on generate test case from load test data button	35	The abstract test case are generated using W method
aj	Click on generate test case from load test data button	36	The abstract test case are generated using Wp method
ak	Click on generate test case from load test data button	37	The abstract test case are generated using DFS method
al	Click on export file to Visual Studio	38	File exported to Visual Studio
am	File exported to OATS	39	Click on export file to OATS
an	File exported to MTM	40	Click on export file to MTM
ao	File exported to JMeter	41	Export file to Jmeter
ap	Select directory to save | Click on OK button	42	Script on VS saved
aq	Select directory to save | Click on OK button	43	Script on OATS saved
ar	Select directory to save | Click on OK button	44	Script on MTM saved
as	Select directory to save | Click on OK button	45	Script on JMeter saved
at	Click on Load File to be parsed | Select file |Click on Open	46	File XML loaded
au	Choose between close the application or run the application again	47	Command chosen
Choose the Testing Type			

ID	Input	ID	Output
a	Select the type of the test	1	Type of the test is selected
b	Start Functional Testing	2	Functional Testing is started
c	Start Performance Testing	3	Performance Testing is started
d	Start Structural Testing	4	Structural Testing is started
Parameterization			
ID	**Input**	**ID**	**Output**
av	Click on parameterization button	48	The screen will open
ax	Choose the scenario on the left	49	The scenario will be marked on the screen
az	Choose the file on the right	50	The file preview will open on the right
aw	Click on Export scripts file	51	Will be open a screen to choose a directory
ba	Click on OK button to save the script	52	Script is saved
Performance Testing			
ID	**Input**	**ID**	**Output**
bb	(Req->Functionalities Performance)Click on import XMI/XML file button	53	File exported
bc	Click on Generate ATC for HSI button	54	Convert test data into generic test scenarios
bd	Click on Generate ATC for DFS button	55	Convert test data into generic test scenarios
be	Click on Generate ATC for W button	56	Convert test data into generic test scenarios
bf	Click on Generate ATC for WP button	57	Convert test data into generic test scenarios
bg	Click on Parameterization button	58	Configure the data of parameterization
bh	Click on Parameterization button	59	Configure the data of parameterization
bi	Click on Parameterization button	60	Configure the data of parameterization
bj	Click on Parameterization button	61	Configure the data of parameterization
bk	Click on Generate Scripts for Load Runner button	62	Script are generated
bl	Click on Generate Scripts for Visual Studio button	63	Script are generated
bm	Click on Generate Scripts for JMeter button	64	Script are generated
bn	Click on Execute Test button	65	Open Load Runner application to run the scripts generated
bn	Click on Execute Test button	66	Open Visual Studio application to run the scripts generated
bn	Click on Execute Test button	67	Open JMeter application to run the scripts generated
bo	Clcik on Close button	68	Application closed
Structural Testing			
ID	**Input**	**ID**	**Output**
bs	Type the path of the XMI file on console	72	File XMI loaded
bt	Press Enter	73	Information necessary extracted for generating a data structure in memory
bu	Specify the directory to save the Abstract Structure	74	Directory where is saved the abstract data structure is displayed on the console
bt	Press Enter	86	Data File and Abstract Structure saved
bv	Inform the launcher path of Jabuti, EMMA or Poketool	88	Path informed
bq	Click on Jabuti application located on“c:/PletsCoverageJabutti.exe” (Req->Specify the directory where the Jabuti.jbt file will be stored)	70	Jabuti application opened
br	Click on Emma application located on“c:/PletsCoverageEmma.exe” (Ex->Specify the directory where the Jabuti.jbt file will be stored)	87	PokeTool application opened
ci	Click on Poke-Tool application located on “c:/Poketool.exe” (Ex->Specify the directory where the Jabuti.jbt file will be stored)	71	Emma application opened
cj	Press Enter to export file to JaBUTi	87	Java class for Jabuti is saved
ck	Java class for Poketool is saved	89	Java class for Poketool is saved
cl	Java class for Emma is saved	90	Java class for Emma is saved
ca	Specify the directory where the Jabuti.jbt file will be stored	79	JaBUTi’s GUI is launched
cb	Application will open on screen	80	Tests results on screen
cd	Press on Close	81	Application closed
ch	Press Y in order to run the tests again	85	Console ready to export XML file
cj	Select directory to save java class for JaBUTi	87	Java class is saved
ck	Select directory to save java class for poketool	87	Java class is saved
cl	Select directory to save java class for Emma	87	Java class is saved

APPENDIX G – SPLIT-MBT TOOL INTERFACE

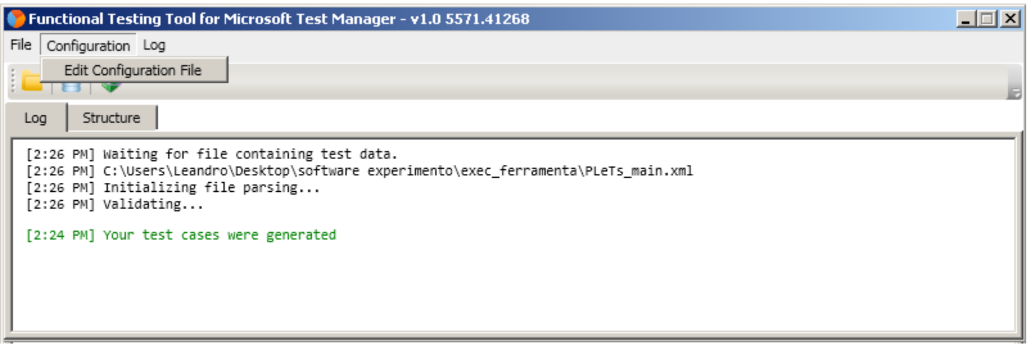
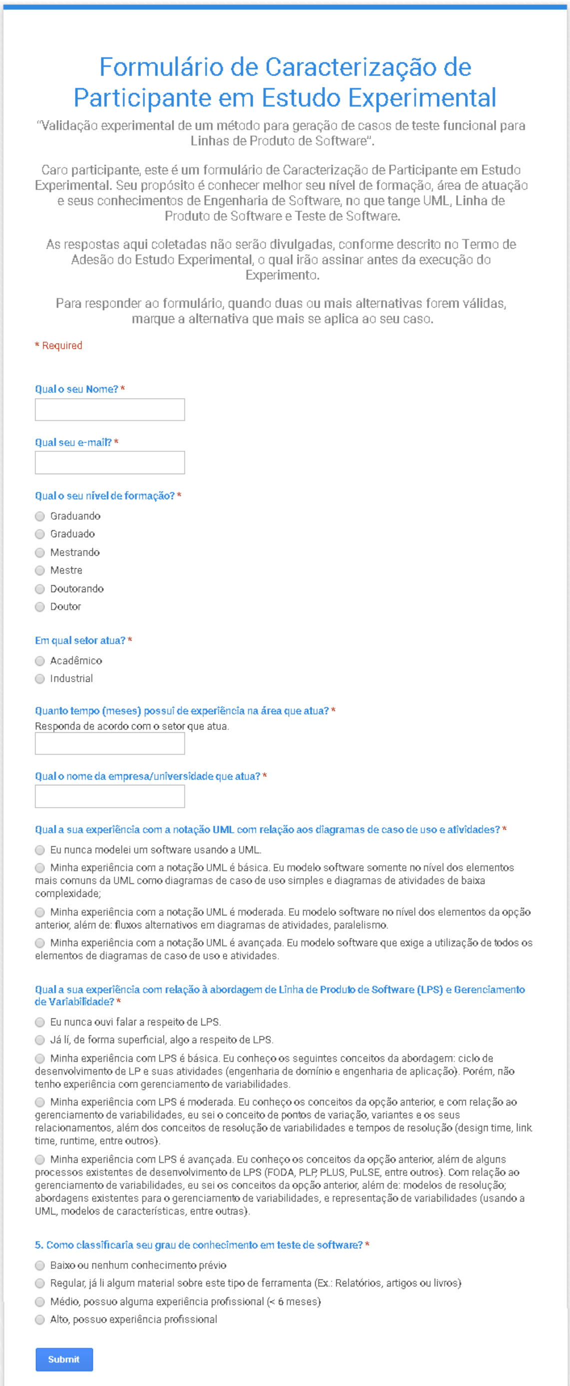


Figure G.1: SPLiT-MBt Tool Interface

APPENDIX H – PROFILE FORM/CHARACTERIZATION QUESTIONNAIRE OF THE EXPERIMENT



Formulário de Caracterização de Participante em Estudo Experimental

"Validação experimental de um método para geração de casos de teste funcional para Linhas de Produto de Software".

Caro participante, este é um formulário de Caracterização de Participante em Estudo Experimental. Seu propósito é conhecer melhor seu nível de formação, área de atuação e seus conhecimentos de Engenharia de Software, no que tange UML, Linha de Produto de Software e Teste de Software.

As respostas aqui coletadas não serão divulgadas, conforme descrito no Termo de Adesão do Estudo Experimental, o qual irão assinar antes da execução do Experimento.

Para responder ao formulário, quando duas ou mais alternativas forem válidas, marque a alternativa que mais se aplica ao seu caso.

* Required

Qual o seu Nome? *

Qual seu e-mail? *

Qual o seu nível de formação? *

- Graduando
- Graduado
- Mestrando
- Mestre
- Doutorando
- Doutor

Em qual setor atua? *

- Acadêmico
- Industrial

Quanto tempo (meses) possui de experiência na área que atua? *

Responda de acordo com o setor que atua.

Qual o nome da empresa/universidade que atua? *

Qual a sua experiência com a notação UML com relação aos diagramas de caso de uso e atividades? *

- Eu nunca modelei um software usando a UML.
- Minha experiência com a notação UML é básica. Eu modelo software somente no nível dos elementos mais comuns da UML como diagramas de caso de uso simples e diagramas de atividades de baixa complexidade;
- Minha experiência com a notação UML é moderada. Eu modelo software no nível dos elementos da opção anterior, além de: fluxos alternativos em diagramas de atividades, paralelismo.
- Minha experiência com a notação UML é avançada. Eu modelo software que exige a utilização de todos os elementos de diagramas de caso de uso e atividades.

Qual a sua experiência com relação à abordagem de Linha de Produto de Software (LPS) e Gerenciamento de Variabilidade? *

- Eu nunca ouvi falar a respeito de LPS.
- Já lí, de forma superficial, algo a respeito de LPS.
- Minha experiência com LPS é básica. Eu conheço os seguintes conceitos da abordagem: ciclo de desenvolvimento de LP e suas atividades (engenharia de domínio e engenharia de aplicação). Porém, não tenho experiência com gerenciamento de variabilidades.
- Minha experiência com LPS é moderada. Eu conheço os conceitos da opção anterior, e com relação ao gerenciamento de variabilidades, eu sei o conceito de pontos de variação, variantes e os seus relacionamentos, além dos conceitos de resolução de variabilidades e tempos de resolução (design time, link time, runtime, entre outros).
- Minha experiência com LPS é avançada. Eu conheço os conceitos da opção anterior, além de alguns processos existentes de desenvolvimento de LPS (FODA, PLP, PLUS, PuLSE, entre outros). Com relação ao gerenciamento de variabilidades, eu sei os conceitos da opção anterior, além de: modelos de resolução; abordagens existentes para o gerenciamento de variabilidades, e representação de variabilidades (usando a UML, modelos de características, entre outras).

5. Como classificaria seu grau de conhecimento em teste de software? *

- Baixo ou nenhum conhecimento prévio
- Regular, já li algum material sobre este tipo de ferramenta (Ex.: Relatórios, artigos ou livros)
- Médio, possuo alguma experiência profissional (< 6 meses)
- Alto, possuo experiência profissional

Submit

Figure H.1: Profile form/characterization questionnaire of the Experiment

APPENDIX I – EXPERIMENT GUIDE WITH SPLIT-MBT TOOL

Roteiro para modelagem de teste de Funcional para SPLs e Geração de Casos de Teste com a Ferramenta SPLiT-MBt Tool

- Anote o horário em que iniciou esta etapa.

Etapa 1 - Anotar os modelos com informações de teste

Esta etapa consiste em anotar os diagramas de atividades com informações necessárias para automatizar a geração de scripts e cenários para teste funcional para SPLs.

Modelo de teste de Performance: com o objetivo de dar início a esta etapa considere:

- Dê um duplo clique com o botão esquerdo do mouse no diagrama com o nome "Performance" para abrir o diagrama de atividades correspondente (ver Figura 1).
- Em seguida, será apresentado o diagrama de atividades "Performance" (ver Figura 2). Este diagrama de atividades possui informações de variabilidade, mas ainda não possui informações de teste. Desta forma, clique nos elementos "Control Flow" (transição de uma atividade para outra atividade ou de uma atividade para um Decision Node e vice-versa) e clique na aba "Tagged Value" (ver Figura 2).
- Clique no botão "Add" para adicionar as tags com os nomes (coluna "name") "TDaction" e TDexpectedResult.
- Insira os valores de teste (coluna "Value") neste diagrama conforme descrito na Tabela 1.

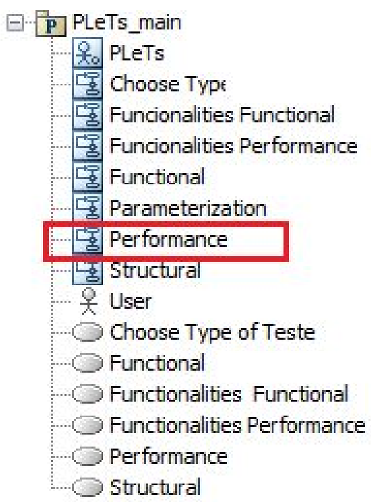


Figura 1

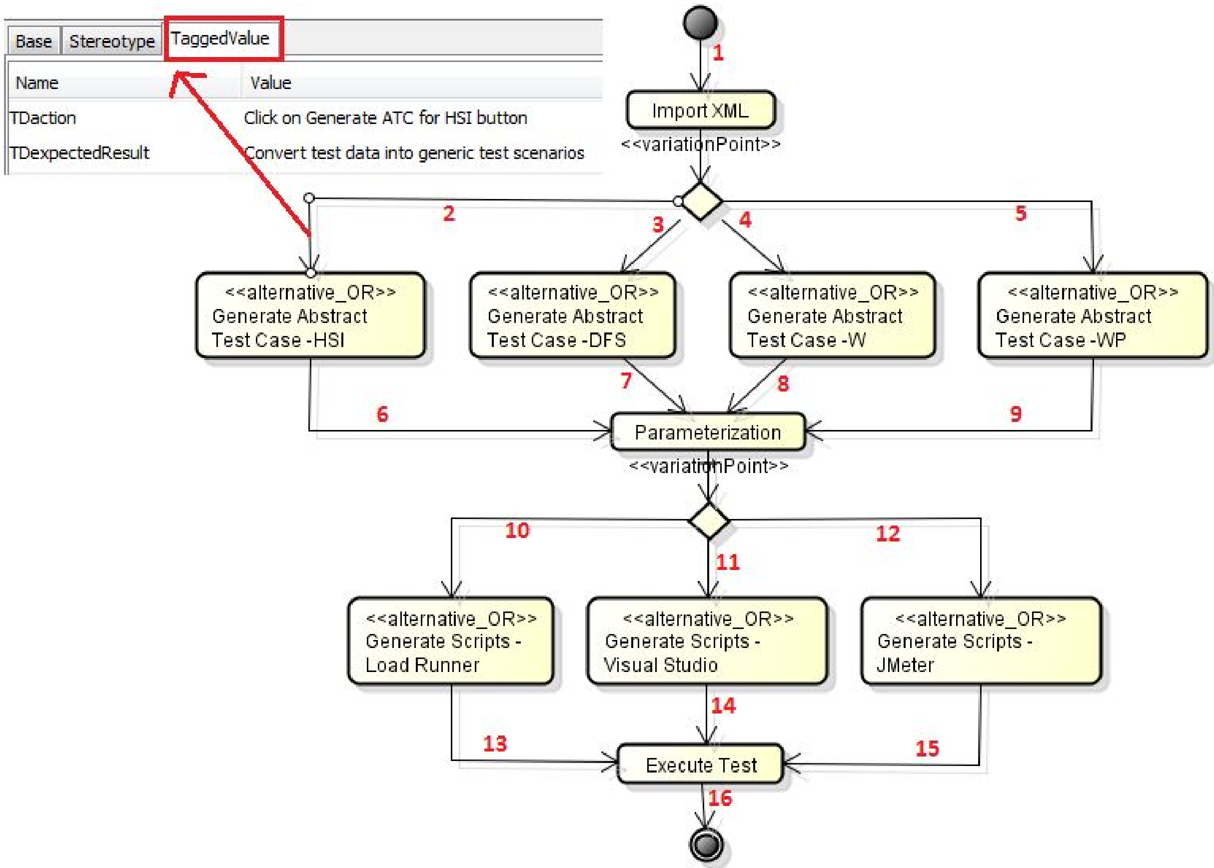


Figura 2

Tabela 1

Transição das Atividades	Tags	Valores
1 - Start State p/ Import XML	TDaction TDexpectedResult	- (Req->Functionalities Performance)Click on import XMI/XML file button - File exported
2 - Decision Node 1 p/ Generate Abstract Test Case -HSI	TDaction TDexpectedResult	- Click on Generate ATC for HSI button - Convert test data into generic test scenarios
3 - Decision Node 1 p/ Generate Abstract Test Case	TDaction TDexpectedResult	- Click on Generate ATC for DFS button - Convert test data into generic test

-DFS		scenarios
4 - Decision Node 1 p/ Generate Abstract Test Case -W	TDaction TDexpectedResult	- Click on Generate ATC for W button - Convert test data into generic test scenarios
5 - Decision Node 1 p/ Generate Abstract Test Case -WP	TDaction TDexpectedResult	- Click on Generate ATC for WP button - Convert test data into generic test scenarios
6 - Generate Abstract Test Case -HSI p/ Parameterization	TDaction TDexpectedResult	- Click on Parameterization button - Configure the data of parameterization
7 - Generate Abstract Test Case -DFS p/ Parameterization	TDaction TDexpectedResult	- Click on Parameterization button - Configure the data of parameterization
8 - Generate Abstract Test Case -W p/ Parameterization	TDaction TDexpectedResult	- Click on Parameterization button - Configure the data of parameterization
9 - Generate Abstract Test Case -WP p/ Parameterization	TDaction TDexpectedResult	- Click on Parameterization button - Configure the data of parameterization
10 - Decision Node 2 p/ Generate Scripts - Load Runner	TDaction TDexpectedResult	- Click on Generate Scripts for Load Runner button - LoadRunner script is generated
11 - Decision Node 2 p/ Generate Scripts - Visual Studio	TDaction TDexpectedResult	- Click on Generate Scripts for Visual Studio button - Visual Studio script is generated
12 - Decision Node 2 p/ Generate Scripts - JMeter	TDaction TDexpectedResult	- Click on Generate Scripts for JMeter button - JMeter script is generated
13 - Generate Scripts - Load Runner p/ Execute Test	TDaction TDexpectedResult	- Click on Execute Test button - Open Load Runner application to run the scripts generated
14 - Generate Scripts - Visual Studio p/ Execute Test	TDaction TDexpectedResult	- Click on Execute Test button - Open Visual Studio application to run the scripts generated
15 - Generate Scripts - JMeter p/ Execute Test	TDaction TDexpectedResult	- Click on Execute Test button - Open JMeter application to run the scripts generated

16 - Execute Test p/ Final State	TDaction TDexpectedResult	- Click on Close button - Application is closed

Modelo de teste Funcional: esta etapa consiste em anotar informações de teste em um outro diagrama de atividades. Para isso, considere os seguintes passos:

- Dê um duplo clique com o botão esquerdo do mouse no diagrama com o nome “Functional” para abrir o diagrama de atividades correspondente (ver Figura 3).
- Em seguida, será apresentado o diagrama de atividades “Functional” (ver Figura 4). Este diagrama de atividades possui informações de variabilidade, mas ainda não possui informações de teste. Desta forma, clique nos elementos “Control Flow” (transição de uma atividade para outra atividade ou de uma atividade para um Decision Node e vice-versa) e clique na aba “Tagged Value” (ver Figura 4).
- Clique no botão “Add” para adicionar as tags com os nomes (coluna “name”) “TDaction” e TDexpectedResult.
- Insira os valores de teste (coluna “Value”) neste diagrama conforme descrito na Tabela 2.

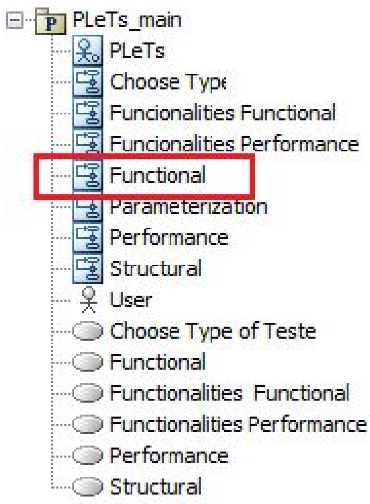


Figura 3

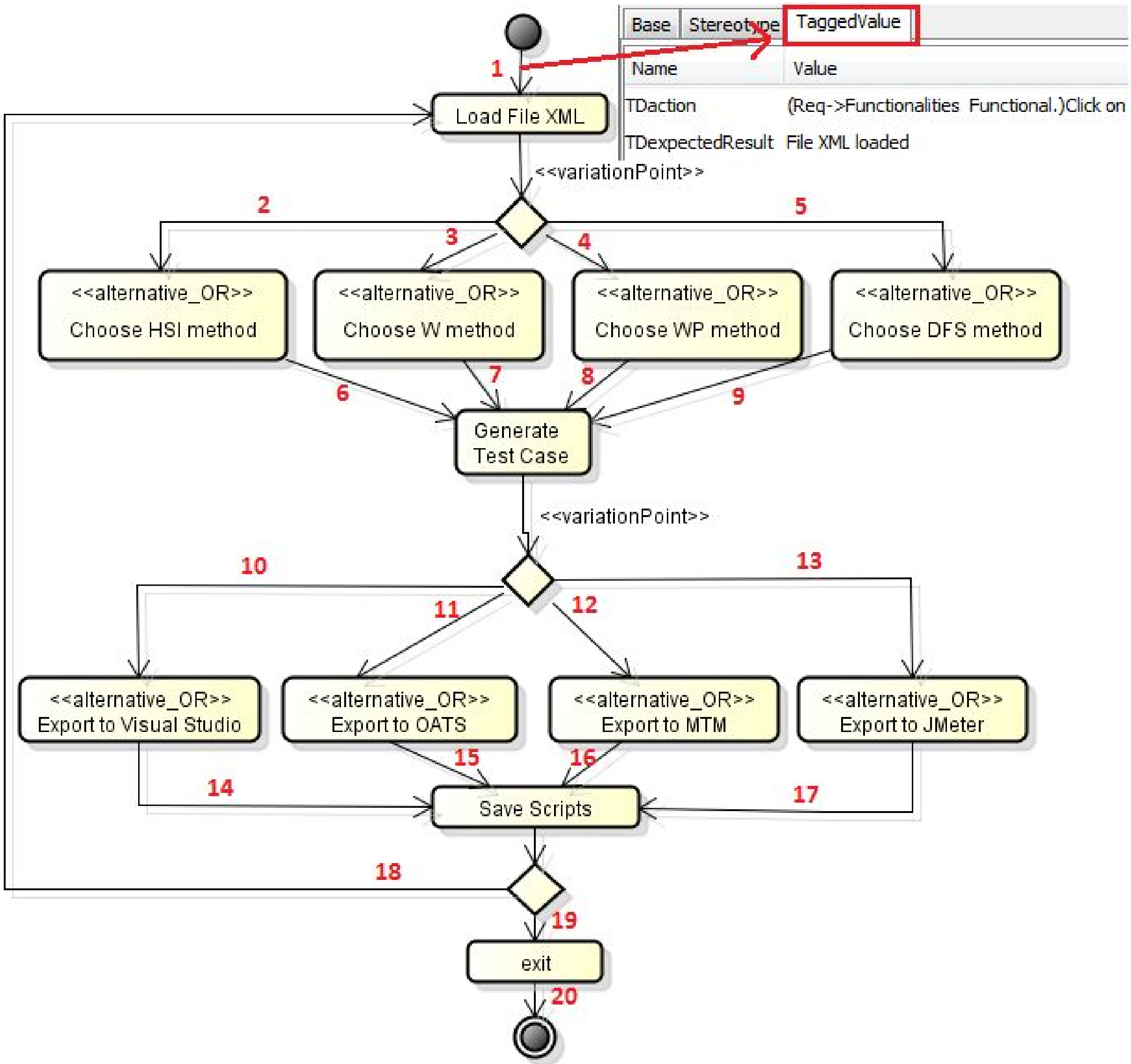


Figura 4

Tabela 2

Transição das Atividades	Tags	Valores
1 - Start State p/ Load File XML	TDaction TDexpectedResult	- (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open - File XML loaded
2 - Decision Node 1 p/ Choose HSI method	TDaction TDexpectedResult	- Select the method of test sequences generation, HSI - Method of test sequence generation HSI is selected

3 - Decision Node 1 p/ Choose W method	TDaction TDexpectedResult	- Select the method of test sequences generation, W - Method of test sequence generation W is selected
4 - Decision Node 1 p/ Choose WP method	TDaction TDexpectedResult	- Select the method of test sequences generation, Wp - Method of test sequence generation Wp is selected
5 - Decision Node 1 p/ Choose DFS method	TDaction TDexpectedResult	- Click on generate test case from load test data button - The abstract test case are generated using DFS method
6 - Choose HSI method p/ Generate Test Case	TDaction TDexpectedResult	- Click on generate test case from load test data button - The abstract test case are generated using HSI method
7 - Choose W method p/ Generate Test Case	TDaction TDexpectedResult	- Click on generate test case from load test data button - The abstract test case are generated using W method
8 - Choose WP method p/ Generate Test Case	TDaction TDexpectedResult	- Click on generate test case from load test data button - The abstract test case are generated using Wp method
9 - Choose DFS method p/ Generate Test Case	TDaction TDexpectedResult	- Click on generate test case from load test data button - The abstract test case are generated using DFS method
10 - Decision Node 2 p/ Export to Visual Studio	TDaction TDexpectedResult	- Click on export file to Visual Studio - File exported to Visual Studio
11 - Decision Node 2 p/ Export to OATS	TDaction TDexpectedResult	- Click on export file to OATS - File exported to OATS
12 - Decision Node 2 p/ Export to MTM	TDaction TDexpectedResult	- Click on export file to MTM - File exported to MTM
13 - Decision Node 2 p/ Export to JMeter	TDaction TDexpectedResult	- Script on JMeter saved - File exported to JMeter

14 - Export to Visual Studio p/ Save Scripts	TDaction TDexpectedResult	- Select directory to save | Click on OK button - Script on VS saved
15 - Export to OATS p/ Save Scripts	TDaction TDexpectedResult	- Select directory to save | Click on OK button - Script on OATS saved
16 - Export to MTM p/ Save Scripts	TDaction TDexpectedResult	- Select directory to save | Click on OK button - Script on MTM saved
17 - Export to JMeter p/ Save Scripts	TDaction TDexpectedResult	- Select directory to save | Click on OK button - Script on JMeter saved
18 - DecisionNode 3 p/ Load File XML	TDaction TDexpectedResult	- Click on Load File to be parsed | Select file | Click on Open - File XML loaded
19 - DecisionNode 3 p/ Exit	TDaction TDexpectedResult	- Choose between close the application or run the application again - Command chosen
20 - exit p/ Final State	TDaction TDexpectedResult	- Click on Close button to end the application - Application is closed

Modelo de teste Estrutural: esta etapa consiste em anotar informações de teste em um outro diagrama de atividades. Para isso, considere os seguintes passos:

- Dê um duplo clique com o botão esquerdo do mouse no diagrama com o nome "Structural" para abrir o diagrama de atividades correspondente (ver Figura 5).
- Em seguida, será apresentado o diagrama de atividades "Structural" (ver Figura 6). Este diagrama de atividades possui informações de variabilidade, mas ainda não possui informações de teste. Desta forma, clique nos elementos "Control Flow" (transição de uma atividade para outra atividade ou de uma atividade para um Decision Node e vice-versa) e clique na aba "Tagged Value" (ver Figura 6).
- Clique no botão "Add" para adicionar as tags com os nomes (coluna "name") "TDaction" e TDexpectedResult.
- Insira os valores de teste (coluna "Value") neste diagrama conforme descrito na Tabela 3.

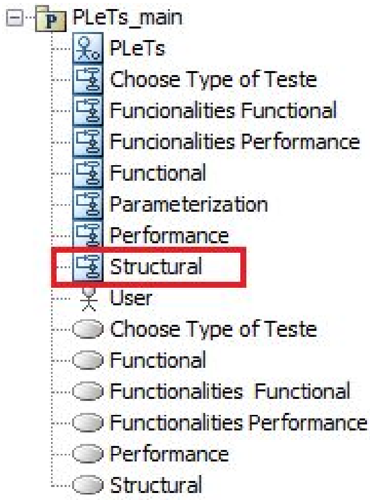
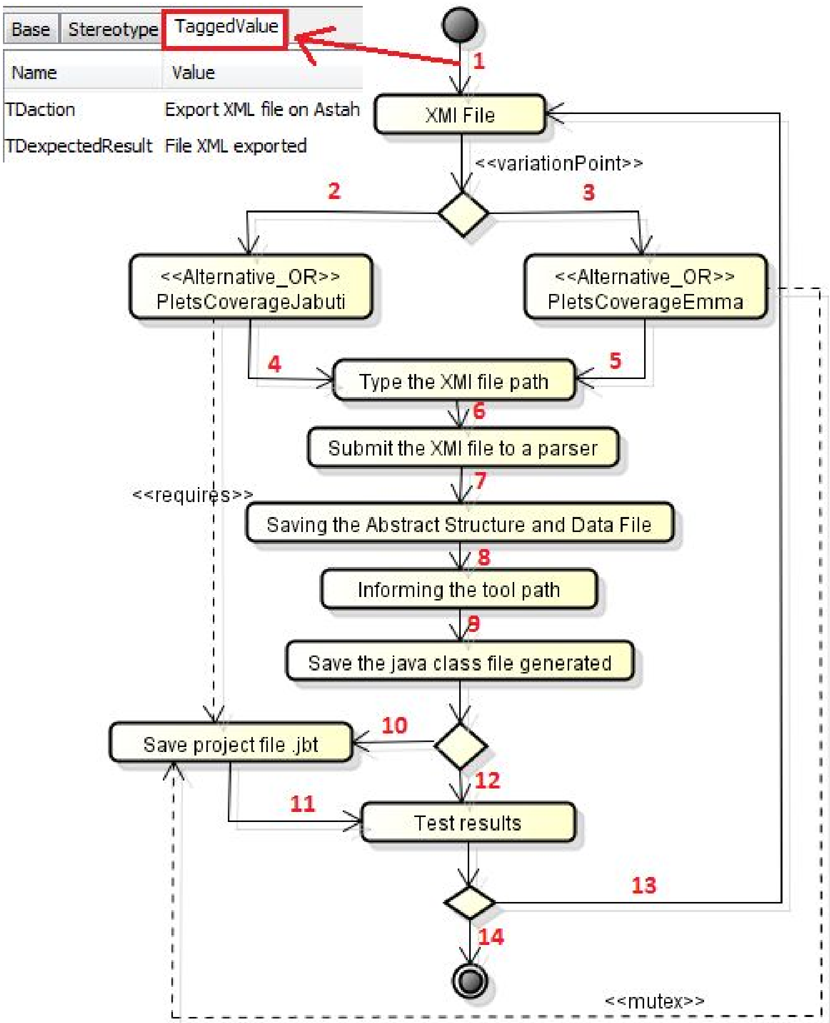


Figura 5



Figuta 6

Tabela 3

Transição das Atividades	Tags	Valores
1 - Start State p/ XMI File	TDaction TDexpectedResult	- Export XML file on Astah - File XML exported
2 - Decision Node 1 p/ PletsCoverageJabuti	TDaction TDexpectedResult	- Click on Jabuti application located on "c:/PletsCoverageJabuti.exe" (Req->Specify the directory where the Jabuti.jbt file will be stored) - Jabuti application opened
3 - Decision Node 1 p/ PletsCoverageEmma	TDaction TDexpectedResult	- Click on Emma application located on "c:/PletsCoverageEmma.exe" (Ex->Specify the directory where the Emma file will be stored) - Emma application opened
4 - PletsCoverageJabuti p/ Type the XMI file path	TDaction TDexpectedResult	- Type the path of the XMI file on console - File XMI loaded
5 - PletsCoverageEmma p/ Type the XMI file path	TDaction TDexpectedResult	- Type the path of the XMI file on console - File XMI loaded
6 - Type the XMI file path p/ Submit the XMI file to a parser	TDaction TDexpectedResult	- Press Enter to Submit the file to a parser - Information necessary extracted for generating a data structure in memory
7 - Submit the XMI file to a parser p/ Saving the Abstract Structure and Data File	TDaction TDexpectedResult	- Specify the directory to save the Abstract Structure | Press Enter - Data File and Abstract Structure saved
8 - Saving the Abstract Structure and Data File p/ Informing the tool path	TDaction TDexpectedResult	- Inform the launcher path of Jabuti or EMMA - Path informed
9 - Informing the tool path p/ Save the java class file generated	TDaction TDexpectedResult	- Press Enter in order to save java class - Java class saved
10 - DecisionNode 2 p/ Save project file .jbt	TDaction TDexpectedResult	- Specify the directory where the Jabuti.jbt file will be stored - JaBUTi's GUI is launched

11 - Save project file .jbt p/ Test results	TDaction TDexpectedResult	- Application will open on screen - Tests results on screen
12 - Decision Node 2 p/ Test results	TDaction TDexpectedResult	- Application will open on screen - Tests results on screen
13 - Decision Node 3 p/ XMI File	TDaction TDexpectedResult	- Press Y in order to run the tests again - Console ready to export XML file
14 - Decision Node 3t p/ Final State	TDaction TDexpectedResult	- Press on Close - Application is closed

Etapa 2 - Gerar arquivo XML dos modelos.

- Esta etapa consiste em exportar um arquivo no formato XMI. Portanto, clique no menu "Tool" -> "XML Input & Output" -> "Save as XML Project".

- Exporte para o arquivo **Desktop** salvando com o nome **PLeTs.xml**.

- Anote o horário em que terminou esta etapa.

Etapa 3 - Execução da ferramenta SPLiT-MBt Tool.

Esta etapa consiste em executar ferramenta **SPLiT-MBt Tool**, a qual foi desenvolvida com base nos conceitos do método SPLiT-MBt. Portanto, execute a ferramenta **TestingTool.exe** localizada no diretório: **C:\....\Desktop__output\plets** (ver as etapas das figuras 2, 3 e 4)

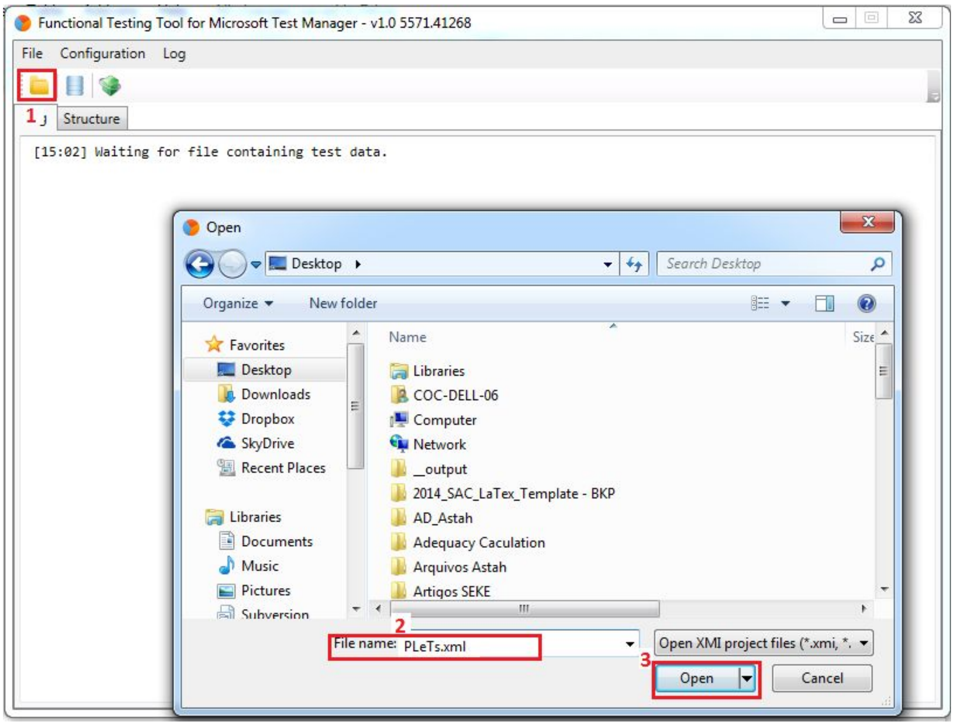


Figura 7

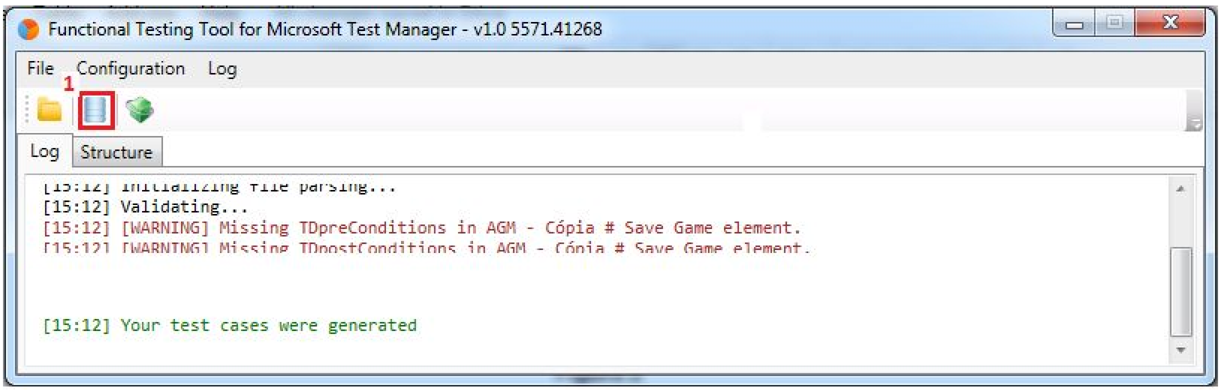


Figura 8

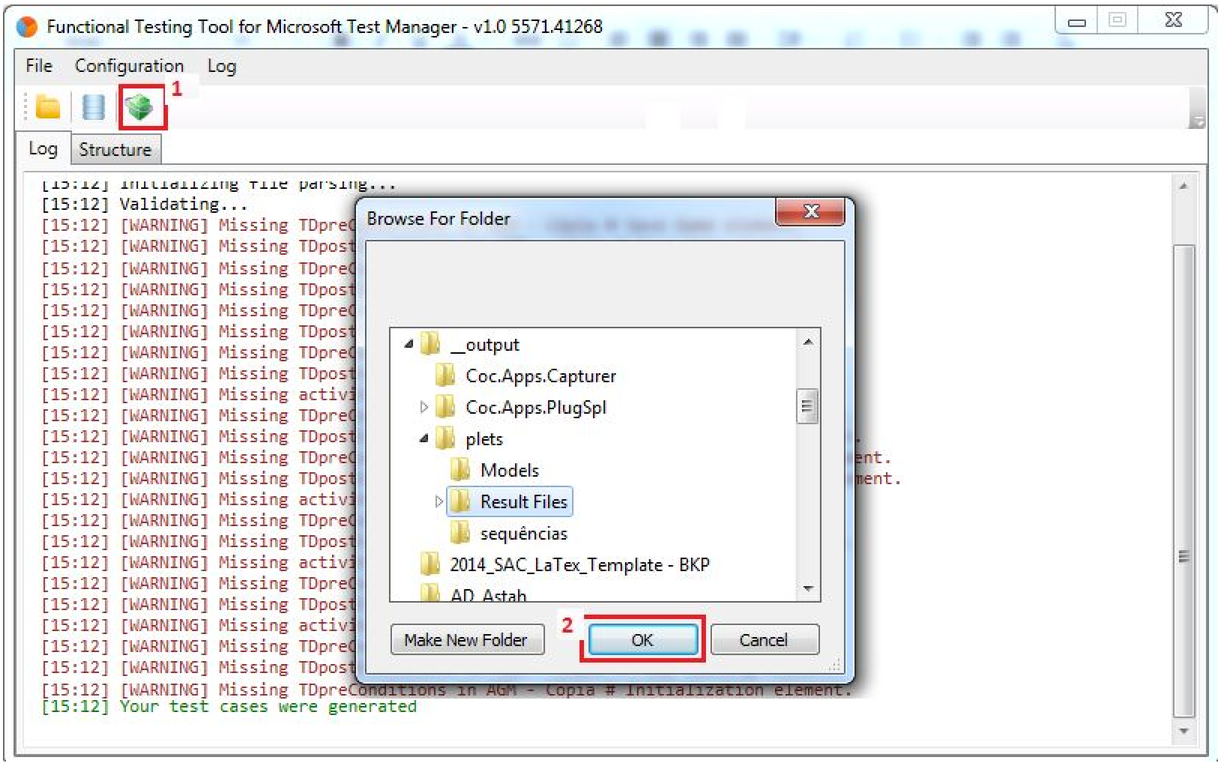


Figura 4

Etapa 4 - Será automaticamente gerado o arquivo "Plan.xls", o qual contém informações dos casos de teste. Abra-o para visualizar estes casos de teste.

- Anote o horário em que terminou esta etapa.

APPENDIX J – EXPERIMENT GUIDE WITH CADET

Roteiro para Geração de Casos de Teste para SPLs - Abordagem CADeT para gerar casos de teste.

Anote o horário em que iniciou esta etapa

Esta etapa consiste em mapear informações do diagrama de atividades da Figura 1 para uma tabela chamada “Tabela de Decisão”. O objetivo é que uma vez que a Tabela de Decisão contém os dados de teste associados com as informações descritas no modelo, casos de teste para testar SPLs podem ser gerados.

Etapa 1 - Teste de Performance

Portanto, complete a “Tabela de Decisão” conforme descrito a seguir: O diagrama de atividades da Figura 1 possui informações de teste, as quais procedem de um identificador especifico (Ex.: Action 2, ExpectedResult 2).

Os dados reais referentes a estes identificadores estão descritos na Tabela 1. Portanto, preencha os dados da “Tabela de Decisão” (Tabela 2 - arquivo excel) associando os identificadores do diagrama de atividades da Figura 1 com os valores reais descritos na Tabela 1.

Por exemplo, a célula que corresponde à linha 6 com a coluna ‘A’ da Tabela 2 que antes possuia apenas o identificador **Action 1** deverá conter o valor: **(Req->Functionalities Performance)Click on import XMI/XML file button**. Enquanto a célula que corresponde a linha 6 com a coluna ‘B’ da Tabela 2 que antes possuia apenas o identificador **TDexpectedResult 1** deverá conter o valor: **File exported**.

Para cada valor real de teste que substitui os identificadores Action e ExpectedResult, considere a coluna C da Tabela de Decisão (arquivo excel) na mesma linha e marque com um ‘X’ se a atividade no diagrama de atividades **NÂO** possui a descrição “Kernel”.

Por exemplo, a primeira atividade do diagrama de atividades da Figura 1 possui o nome **Kernel Import XML**, logo a célula C:6 da Tabela 2 **NÃO** deverá ser marcada com um ‘X’.

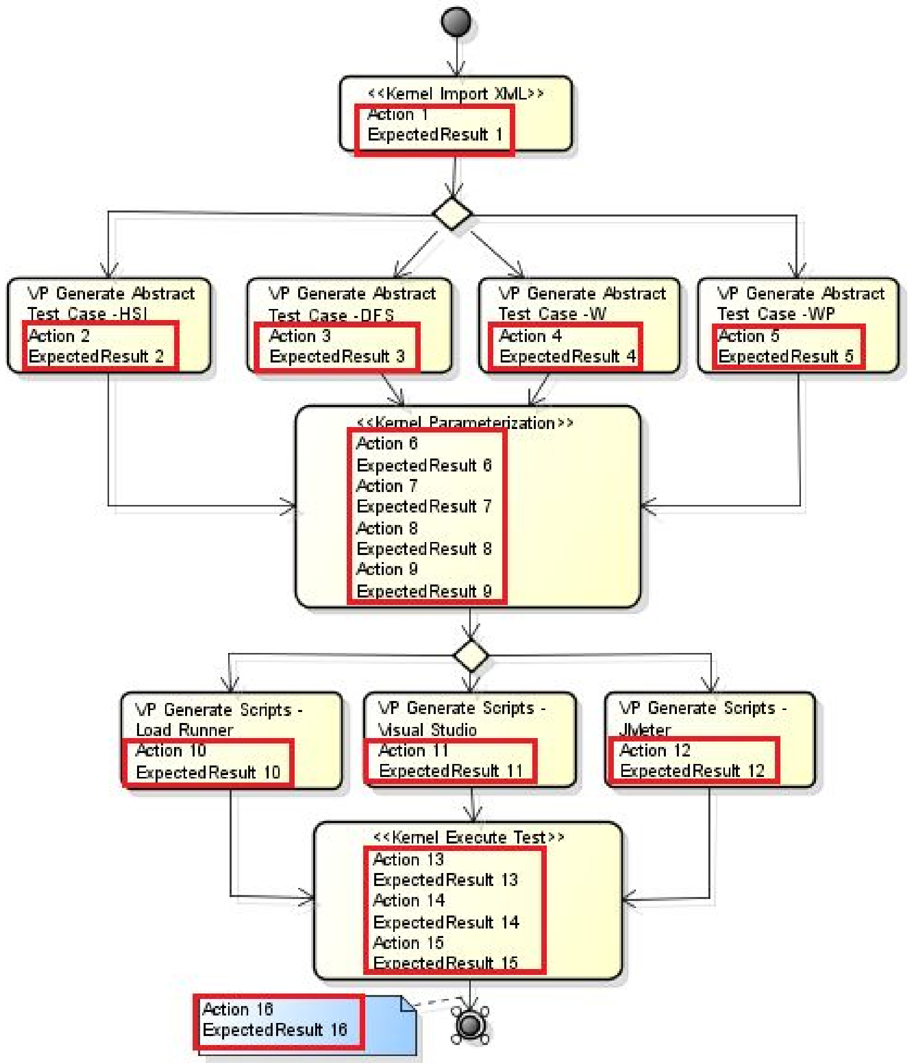


Figura 1

Tabela 1

Identificador	**Valores**
TDaction 1 TDexpectedResult 1	**- 1 (Req->Functionalities Performance)Click on import XMI/XML file button** **- File exported**
TDaction 2 TDexpectedResult 2	- 2 Click on Generate ATC for HSI button - Convert test data into generic test scenarios
TDaction 3 TDexpectedResult 3	- 3 Click on Generate ATC for DFS button - Convert test data into generic test scenarios

TDaction 4 TDexpectedResult 4	- 4 Click on Generate ATC for W button - Convert test data into generic test scenarios
TDaction 5 TDexpectedResult 5	- 5 Click on Generate ATC for WP button - Convert test data into generic test scenarios
TDaction 6 TDexpectedResult 6	- 6 Click on Parameterization button - Configure the data of parameterization
TDaction 7 TDexpectedResult 7	- 7 Click on Parameterization button - Configure the data of parameterization
TDaction 8 TDexpectedResult 8	- 8 Click on Parameterization button - Configure the data of parameterization
TDaction 9 TDexpectedResult 9	- 9 Click on Parameterization button - Configure the data of parameterization
TDaction 10 TDexpectedResult 10	- 10 Click on Generate Scripts for Load Runner button - LoadRunner script is generated
TDaction 11 TDexpectedResult 11	- 11 Click on Generate Scripts for Visual Studio button - Visual Studio script is generated
TDaction 12 TDexpectedResult 12	- 12 Click on Generate Scripts for JMeter button - JMeter script is generated
TDaction 13 TDexpectedResult 13	- 13 Click on Execute Test button - Open Load Runner application to run the scripts generated
TDaction 14 TDexpectedResult 14	- 14 Click on Execute Test button - Open Visual Studio application to run the scripts generated
TDaction 15 TDexpectedResult 15	- 15 Click on Execute Test button - Open JMeter application to run the scripts generated
TDaction 16 TDexpectedResult 16	- 16 Click on Close button - Application is closed

Tabela 2

	A	B	C
1	1 Performance		Method and Testing Tool
2	Feature Conditions		
3	Type of Testing Method		{HSI, DFS, W, Wp}
4	Performance Scripts		{LoadRunner, Visual Studio, Jmeter}
5	Actions	Expected Results	
6	1 (Req->Functionalities Performance)Click on import XMI/XML file button	File exported	
7	TDaction 2	TDexpectedResult 2	
8	TDaction 3	TDexpectedResult 3	
9	TDaction 4	TDexpectedResult 4	
10	TDaction 5	TDexpectedResult 5	
11	TDaction 6	TDexpectedResult 6	
12	TDaction 7	TDexpectedResult 7	
13	TDaction 8	TDexpectedResult 8	
14	TDaction 9	TDexpectedResult 9	
15	TDaction 10	TDexpectedResult 10	
16	TDaction 11	TDexpectedResult 11	
17	TDaction 12	TDexpectedResult 12	
18	TDaction 13	TDexpectedResult 13	
19	TDaction 14	TDexpectedResult 14	
20	TDaction 15	TDexpectedResult 15	
21	TDaction 16	TDexpectedResult 16	
22			

Performance / Functional / Structu

Etapa 2 - Teste Funcional (clicar na aba "Functional" do arquivo excel)

Complete a "Tabela de Decisão" conforme descrito a seguir: O diagrama de atividades da Figura 2 possui informações de teste, as quais procedem de um identificador especifico (Ex.: Action 2, ExpectedResult 2).

Os dados reais referentes a estes identificadores estão descritos na Tabela 3. Portanto, preencha os dados da "Tabela de Decisão" (Tabela 4 - arquivo excel) associando os identificadores do diagrama de atividades da Figura 2 com os valores reais descritos na Tabela 3.

Por exemplo, a célula que corresponde a linha 6 com a coluna 'A' da Tabela 4 que antes possuia apenas o identificador **Action 1** deverá conter o valor: **1 (Req->Functionalities Performance)Click on import XMI/XML file button**. Enquanto a célula que corresponde a linha 6 com a coluna 'B' da Tabela 4 que antes possuia apenas o identificador **TDexpectedResult 1** deverá conter o valor: **File XML loaded**.

Para cada valor real de teste que substitui os identificadores Action e ExpectedResult, considere a coluna C da Tabela de Decisão (arquivo excel) na mesma linha e marque com um 'X' se a atividade no diagrama de atividades **NÂO** possui a descrição "Kernel".

Por exemplo, a primeira atividade do diagrama de atividades da Figura 2 possui o nome **Kernel Load File XML**, logo a célula C:6 da Tabela 2 **NÃO** deverá ser marcada com um 'X'.

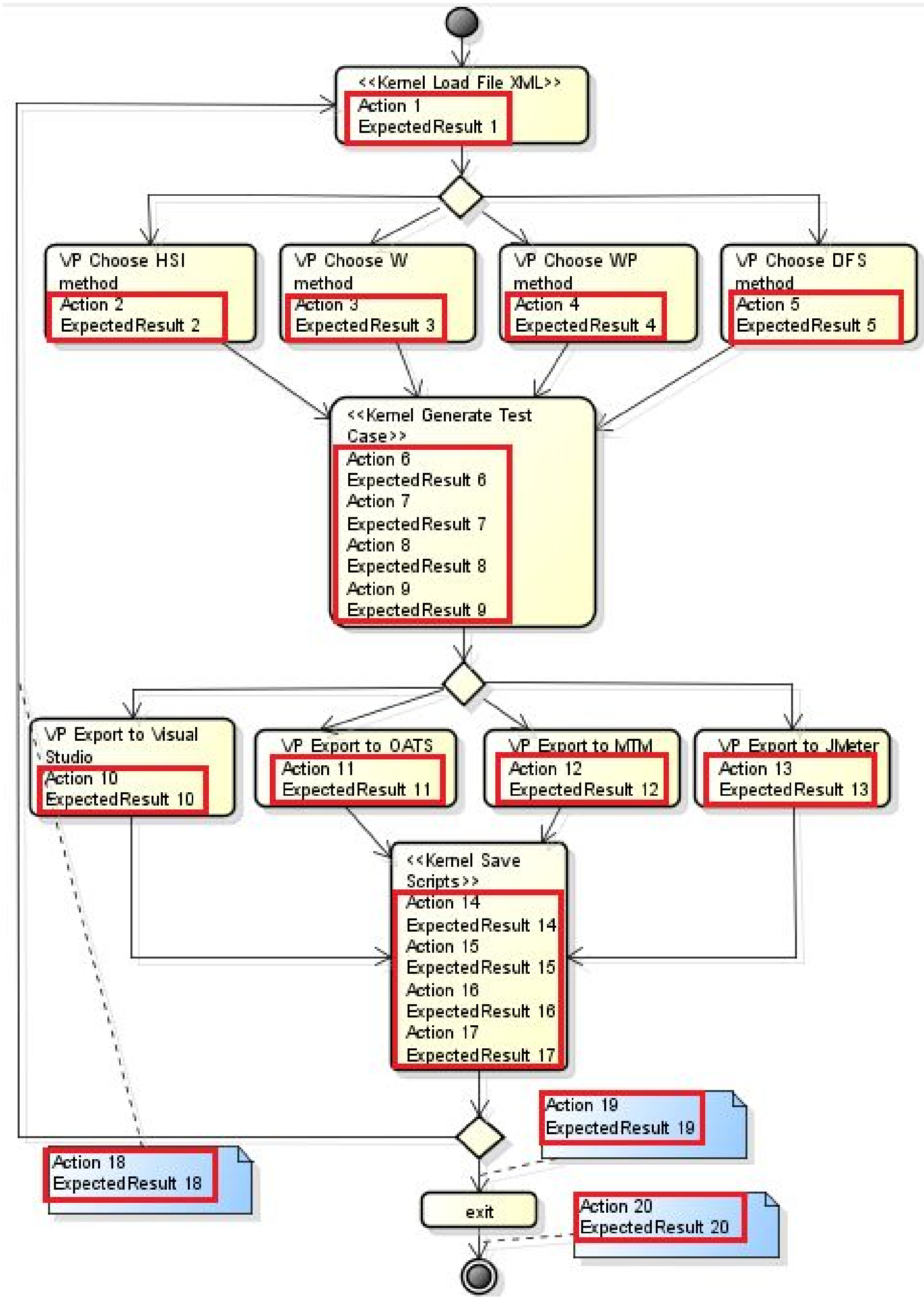

<<Kernel Load File XML>>
Action 1
ExpectedResult 1
VP Choose HSI method
Action 2
ExpectedResult 2
VP Choose W method
Action 3
ExpectedResult 3
VP Choose WP method
Action 4
ExpectedResult 4
VP Choose DFS method
Action 5
ExpectedResult 5
<<Kernel Generate Test Case>>
Action 6
ExpectedResult 6
Action 7
ExpectedResult 7
Action 8
ExpectedResult 8
Action 9
ExpectedResult 9
VP Export to Visual Studio
Action 10
ExpectedResult 10
VP Export to OATS
Action 11
ExpectedResult 11
VP Export to MTM
Action 12
ExpectedResult 12
VP Export to JMeter
Action 13
ExpectedResult 13
<<Kernel Save Scripts>>
Action 14
ExpectedResult 14
Action 15
ExpectedResult 15
Action 16
ExpectedResult 16
Action 17
ExpectedResult 17
Action 19
ExpectedResult 19
Action 18
ExpectedResult 18
exit
Action 20
ExpectedResult 20

Figura 2

Tabela 3

Identificador	Valores
TDaction 1 TDexpectedResult 1	- 1 (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open - File XML loaded
TDaction 2 TDexpectedResult 2	- 2 Select the method of test sequences generation, HSI - Method of test sequence generation HSI is selected
TDaction 3 TDexpectedResult 3	- 3 Select the method of test sequences generation, W - Method of test sequence generation W is selected
TDaction 4 TDexpectedResult 4	- 4 Select the method of test sequences generation, Wp - Method of test sequence generation Wp is selected
TDaction 5 TDexpectedResult 5	- 5 Click on generate test case from load test data button - The abstract test case are generated using DFS method
TDaction 6 TDexpectedResult 6	- 6 Click on generate test case from load test data button - The abstract test case are generated using HSI method
TDaction 7 TDexpectedResult 7	- 7 Click on generate test case from load test data button - The abstract test case are generated using W method
TDaction 8 TDexpectedResult 8	- 8 Click on generate test case from load test data button - The abstract test case are generated using Wp method
TDaction 9 TDexpectedResult 9	- 9 Click on generate test case from load test data button - The abstract test case are generated using DFS method
TDaction 10 TDexpectedResult 10	- 10 Click on export file to Visual Studio - File exported to Visual Studio
TDaction 11 TDexpectedResult 11	- 11 Click on export file to OATS - File exported to OATS
TDaction 12 TDexpectedResult 12	- 12 Click on export file to MTM - File exported to MTM
TDaction 13 TDexpectedResult 13	- 13 Script on JMeter saved - File exported to JMeter
TDaction 14 TDexpectedResult 14	- 14 Select directory to save | Click on OK button - Script on VS saved
TDaction 15 TDexpectedResult 15	- 15 Select directory to save | Click on OK button - Script on OATS saved

TDaction 16 TDexpectedResult 16	- 16 Select directory to save | Click on OK button - Script on MTM saved
TDaction 17 TDexpectedResult 17	- 17 Select directory to save | Click on OK button - Script on JMeter saved
TDaction 18 TDexpectedResult 18	- 18 Click on Load File to be parsed | Select file | Click on Open - File XML loaded
TDaction 19 TDexpectedResult 19	- 19 Choose between close the application or run the application again - Command chosen
TDaction 20 TDexpectedResult 20	- 20 Click on Close button to end the application - Application is closed

Tabela 4

	A	B	C
1	**2 Functional**		**Method and Testing Tool**
2	**Feature Conditions**		
3	Type of Testing Method		{HSI, DFS, W, Wp}
4	Functinal Scripts		{OATS, Visual Studio, Jmeter, MTM}
5	**Actions**	**Expected Results**	
6	**1 (Req->Functionalities Functional.)Click on Load from XMl File button, select file and click on open**	**File XML loaded**	
7	TDaction 2		
8	TDexpectedResult 2		
9	TDaction 3		
10	TDexpectedResult 3		
11	TDaction 4		
12	TDexpectedResult 4		
13	TDaction 5		
14	TDexpectedResult 5		
15	TDaction 6		
16	TDexpectedResult 6		
17	TDaction 7		
18	TDexpectedResult 7		
19	TDaction 8		

Performance | Functional | Structu

Etapa 3 - Teste Estrutural (clicar na aba "Structural" do arquivo excel)

Complete a "Tabela de Decisão" conforme descrito a seguir: O diagrama de atividades da Figura 3 possui informações de teste, as quais procedem de um identificador especifico destacado em vermelho (Ex.: Action 2, ExpectedResult 2).

Os dados reais referentes a estes identificadores estão descritos na Tabela 5. Portanto, preencha os dados da "Tabela de Decisão" (Tabela 6) associando os identificadores do diagrama de atividades da Figura 3 com os valores reais descritos na Tabela 5.

Por exemplo, a célula que corresponde a linha 6 com a coluna 'A' da Tabela 6 que antes possuia apenas o identificador **Action 1** deverá conter o valor: **1 Export XML file on Astah**.

Enquanto a célula que corresponde a linha 6 com a coluna 'B' da Tabela 6 que antes possuia apenas o identificador **TDexpectedResult 1** deverá conter o valor: **File XML exported**.

Para cada valor real de teste que substitui os identificadores Action e ExpectedResult, considere a coluna C da Tabela de Decisão (arquivo excel) na mesma linha e marque com um 'X' se a atividade no diagrama de atividades **NÂO** possui a descrição "Kernel".

Por exemplo, a primeira atividade do diagrama de atividades da Figura 2 possui o nome **Kernel XMI File**, logo a célula C:6 da Tabela 2 **NÃO** deverá ser marcada com um 'X'.

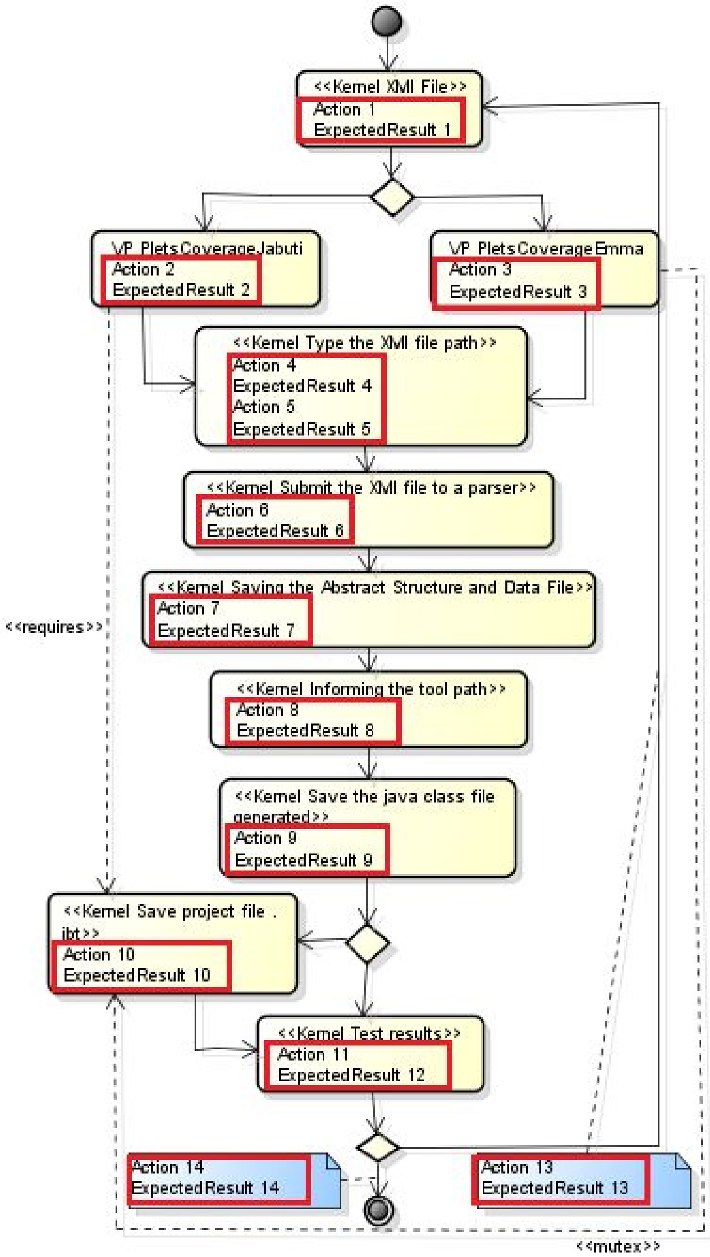


Figura 3

Tabela 5

Identificador	Valores
TDaction 1 TDexpectedResult 1	- 1 Export XML file on Astah - File XML exported
TDaction 2 TDexpectedResult 2	- 2 Click on Jabuti application located on "c:/PletsCoverageJabuti.exe" (Req->Specify the directory where the Jabuti.jbt file will be stored) - Jabuti application opened
TDaction 3 TDexpectedResult 3	- 3 Click on Emma application located on "c:/PletsCoverageEmma.exe" (Ex->Specify the directory where the Emma file will be stored) - Emma application opened
TDaction 4 TDexpectedResult 4	- 4 Type the path of the XMI file on console - File XMI loaded
TDaction 5 TDexpectedResult 5	- 5 Type the path of the XMI file on console - File XMI loaded
TDaction 6 TDexpectedResult 6	- 6 Press Enter to Submit the file to a parser - Information necessary extracted for generating a data structure in memory
TDaction 7 TDexpectedResult 7	- 7 Specify the directory to save the Abstract Structure | Press Enter - Data File and Abstract Structure saved
TDaction 8 TDexpectedResult 8	- 8 Inform the launcher path of Jabuti or EMMA - Path informed
TDaction 9 TDexpectedResult 9	- 9 Press Enter in order to save java class - Java class saved
TDaction 10 TDexpectedResult 10	- 10 Specify the directory where the Jabuti.jbt file will be stored - JaBUTi's GUI is launched
TDaction 11 TDexpectedResult 11	- 11 Application will open on screen - Tests results on screen
TDaction 12 TDexpectedResult 12	- 12 Application will open on screen - Tests results on screen

TDaction 13 TDexpectedResult 13	- 13 Press Y in order to run the tests again - Console ready to export XML file
TDaction 14 TDexpectedResult 14	- 14 Press on Close - Application is closed

Tabela 6

	A	B	C
1	**3 Structural**		**Method and Testing Tool**
2	**Feature Conditions**		
3	Type of Testing Method		{Jabuti, Emma}
4			
5	**Actions**	**Expected Results**	
6	**Export XML file on Astah**	**File XML exported**	
7	TDaction 2		
8	TDexpectedResult 2		
9	TDaction 3		
10	TDexpectedResult 3		
11	TDaction 4		
12	TDexpectedResult 4		
13	TDaction 5		
14	TDexpectedResult 5		
15	TDaction 6		
16	TDexpectedResult 6		
17	TDaction 7		
18	TDexpectedResult 7		
19	TDaction 8		
20	TDexpectedResult 8		
21	TDaction 9		
22	TDexpectedResult 9		
23	TDaction 10		
24	TDexpectedResult 10		
25	TDaction 11		
26	TDexpectedResult 11		
27	TDaction 12		
28	TDexpectedResult 12		
29	TDaction 13		
30	TDexpectedResult 13		
31	TDaction 14		
32	TDexpectedResult 14		

Functional | **Structural**

- **Anote o horário em que terminou esta etapa**

APPENDIX K – EXPERIMENT GUIDE WITH MTM

Roteiro para Geração de Casos de Teste para SPLs - Método sem considerar reuso na geração de casos de teste

- **Anote o horário de inicio desta atividade**

Etapa 1 - Criar casos de teste de forma manual utilizando o Excel

Esta etapa consiste em gerar casos de teste para cada produto individualmente sem considerar o reuso dos casos de teste. Este método corresponde ao modo tradicional de geração de casos de teste para aplicações individuais. Para dar inicio a esta etapa, abra o arquivo **PlanPerf** localizado no **Desktop\Experimento**. Em seguida, complete o arquivo em questão conforme os dados presentes na Tabela 1.

Tabela 1

Test Step	Action/Description	Expected Results
1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -HSI - Click on Generate ATC for HSI button	Convert test data into generic test scenarios
3	Parameterization - Click on Parameterization button	Configure the data of parameterization
4	Generate Scripts - Load Runner - Click on Generate Scripts for Load Runner button	LoadRunner script is generated
5	Execute Test Click on Execute Test button	Open Load Runner application to run the scripts generated
6	Exit - Click on Close button	Application is closed

1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -HSI - Click on Generate ATC for HSI button	Convert test data into generic test scenarios
3	Parameterization - Click on Parameterization button	Configure the data of parameterization
4	Generate Scripts - Visual Studio - Click on Generate Scripts for Visual Studio button	Visual Studio script is generated
5	Execute Test Click on Execute Test button	Open Visual Studio application to run the scripts generated
6	Exit - Click on Close button	Application is closed
1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -HSI - Click on Generate ATC for HSI button	Convert test data into generic test scenarios
3	Parameterization - Click on Parameterization button	Configure the data of parameterization

4	Generate Scripts – Jmeter - Click on Generate Scripts for JMeter button	JMeter script is generated
5	Execute Test Click on Execute Test button	Open JMeter application to run the scripts generated
6	Exit - Click on Close button	Application is closed

Em seguida, clique na aba **GeneralTestCase Perf DFS** e complete o arquivo em questão conforme os dados presentes na Tabela 2:

Tabela 2

Test Step	Action/Description	Expected Results
1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -DFS - Click on Generate ATC for DFS button	Convert test data into generic test scenarios
3	Parameterization - Click on Parameterization button	Configure the data of parameterization
4	Generate Scripts - Load Runner - Click on Generate Scripts for Load Runner button	LoadRunner script is generated
5	Execute Test Click on Execute Test button	Open Load Runner application to run the scripts generated

6	Exit - Click on Close button	Application is closed
1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -DFS - Click on Generate ATC for DFS button	Convert test data into generic test scenarios
3	Parameterization - Click on Parameterization button	Configure the data of parameterization
4	Generate Scripts - Visual Studio - Click on Generate Scripts for Visual Studio button	Visual Studio script is generated
5	Execute Test Click on Execute Test button	Open Visual Studio application to run the scripts generated
6	Exit - Click on Close button	Application is closed
1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -DFS - Click on Generate ATC for DFS button	Convert test data into generic test scenarios

3	Parameterization - Click on Parameterization button	Configure the data of parameterization
4	Generate Scripts – Jmeter - Click on Generate Scripts for JMeter button	JMeter script is generated
5	Execute Test Click on Execute Test button	Open JMeter application to run the scripts generated
6	Exit - Click on Close button	Application is closed

Em seguida, clique na aba **GeneralTestCase Perf W** e complete o arquivo em questão conforme os dados presentes na Tabela 3:

Tabela 3

Test Step	Action/Description	Expected Results
1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -W - Click on Generate ATC for W button	Convert test data into generic test scenarios
3	Parameterization - Click on Parameterization button	Configure the data of parameterization
4	Generate Scripts - Load Runner	LoadRunner script is generated

	- Click on Generate Scripts for Load Runner button	
5	Execute Test Click on Execute Test button	Open Load Runner application to run the scripts generated
6	Exit - Click on Close button	Application is closed
1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -W - Click on Generate ATC for W button	Convert test data into generic test scenarios
3	Parameterization - Click on Parameterization button	Configure the data of parameterization
4	Generate Scripts - Visual Studio - Click on Generate Scripts for Visual Studio button	Visual Studio script is generated
5	Execute Test Click on Execute Test button	Open Visual Studio application to run the scripts generated
6	Exit - Click on Close button	Application is closed
1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -W	Convert test data into generic

	- Click on Generate ATC for W button	test scenarios
3	Parameterization - Click on Parameterization button	Configure the data of parameterization
4	Generate Scripts – Jmeter - Click on Generate Scripts for JMeter button	JMeter script is generated
5	Execute Test Click on Execute Test button	Open JMeter application to run the scripts generated
6	Exit - Click on Close button	Application is closed

Em seguida, clique na aba **GeneralTestCase Perf Wp** e complete o arquivo em questão conforme os dados presentes na Tabela 4:

Tabela 4

Test Step	Action/Description	Expected Results
1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -WP - Click on Generate ATC for WP button	Convert test data into generic test scenarios
3	Parameterization	Configure the data of parameterization

	- Click on Parameterization button	
4	Generate Scripts - Load Runner - Click on Generate Scripts for Load Runner button	LoadRunner script is generated
5	Execute Test Click on Execute Test button	Open Load Runner application to run the scripts generated
6	Exit - Click on Close button	Application is closed
1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -WP - Click on Generate ATC for WP button	Convert test data into generic test scenarios
3	Parameterization - Click on Parameterization button	Configure the data of parameterization
4	Generate Scripts - Visual Studio - Click on Generate Scripts for Visual Studio button	Visual Studio script is generated
5	Execute Test Click on Execute Test button	Open Visual Studio application to run the scripts generated
6	Exit	Application is closed

	- Click on Close button	
1	Import XML - (Req->Functionalities Performance)Click on import XMI/XML file button	File exported
2	Generate Abstract Test Case -WP - Click on Generate ATC for WP button	Convert test data into generic test scenarios
3	Parameterization - Click on Parameterization button	Configure the data of parameterization
4	Generate Scripts – Jmeter - Click on Generate Scripts for JMeter button	JMeter script is generated
5	Execute Test Click on Execute Test button	Open JMeter application to run the scripts generated
6	Exit - Click on Close button	Application is closed

Nesta etapa abra o arquivo excel "PlanFunc". Em seguida, clique na aba **GeneralTestCase Func HSI** e complete o arquivo em questão conforme os dados presentes na Tabela 5:

Tabela 5

Test Step	Action/Description	Expected Results
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose HSI method - Select the method of test sequences generation, HSI	Method of test sequence generation HSI is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using HSI method
4	Export to Visual Studio - Click on export file to Visual Studio	File exported to Visual Studio
5	Save Scripts - Select directory to save | Click on OK button	Script on VS saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed

1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose HSI method - Select the method of test sequences generation, HSI	Method of test sequence generation HSI is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using HSI method
4	Export to OATS - Click on export file to OATS	File exported to OATS
5	Save Scripts - Select directory to save | Click on OK button	Script on OATS saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose HSI method - Select the method of test sequences generation, HSI	Method of test sequence generation HSI is selected

3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using HSI method
4	Export to MTM - Click on export file to MTM	File exported to MTM
5	Save Scripts - Select directory to save | Click on OK button	Script on MTM saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose HSI method - Select the method of test sequences generation, HSI	Method of test sequence generation HSI is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using HSI method
4	Export to JMeter - Click on export file to JMeter	File exported to JMeter
5	Save Scripts	Script on JMeter saved

	- Select directory to save | Click on OK button	
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed

Em seguida, clique na aba **GeneralTestCase Func DFS** e complete o arquivo em questão conforme os dados presentes na Tabela 6:

Tabela 6

Test Step	Action/Description	Expected Results
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose DFS method - Select the method of test sequences generation, DFS	Method of test sequence generation DFS is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using DFS method
4	Export to Visual Studio - Click on export file to Visual Studio	File exported to Visual Studio

5	Save Scripts - Select directory to save | Click on OK button	Script on VS saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose DFS method - Select the method of test sequences generation, DFS	Method of test sequence generation DFS is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using DFS method
4	Export to OATS - Click on export file to OATS	File exported to OATS
5	Save Scripts - Select directory to save | Click on OK button	Script on OATS saved
6	Exit - Choose between close the	Command chosen

	application or run the application again	
7	Close - Click on Close button to end the application	Application is closed
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose DFS method - Select the method of test sequences generation, DFS	Method of test sequence generation DFS is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using DFS method
4	Export to MTM - Click on export file to MTM	File exported to MTM
5	Save Scripts - Select directory to save | Click on OK button	Script on MTM saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed

1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose DFS method - Select the method of test sequences generation, DFS	Method of test sequence generation DFS is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using DFS method
4	Export to JMeter - Click on export file to JMeter	File exported to JMeter
5	Save Scripts - Select directory to save | Click on OK button	Script on JMeter saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed

Em seguida, clique na aba **GeneralTestCase Func W** e complete o arquivo em questão conforme os dados presentes na Tabela 7:

Tabela 7

Test Step	Action/Description	Expected Results
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose W method - Select the method of test sequences generation, W	Method of test sequence generation W is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using W method
4	Export to Visual Studio - Click on export file to Visual Studio	File exported to Visual Studio
5	Save Scripts - Select directory to save | Click on OK button	Script on VS saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed

1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose W method - Select the method of test sequences generation, W	Method of test sequence generation W is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using W method
4	Export to OATS - Click on export file to OATS	File exported to OATS
5	Save Scripts - Select directory to save | Click on OK button	Script on OATS saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose W method - Select the method of test sequences generation, W	Method of test sequence generation W is selected

3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using W method
4	Export to MTM - Click on export file to MTM	File exported to MTM
5	Save Scripts - Select directory to save | Click on OK button	Script on MTM saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose W method - Select the method of test sequences generation, W	Method of test sequence generation W is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using W method
4	Export to JMeter - Click on export file to JMeter	File exported to JMeter

5	Save Scripts - Select directory to save | Click on OK button	Script on JMeter saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed

Em seguida, clique na aba **GeneralTestCase Func WP** e complete o arquivo em questão conforme os dados presentes na Tabela 8:

Tabela 8

Test Step	Action/Description	Expected Results
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose WP method - Select the method of test sequences generation, WP	Method of test sequence generation WP is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using WP method
4	Export to Visual Studio - Click on export file to Visual Studio	File exported to Visual Studio

5	Save Scripts - Select directory to save | Click on OK button	Script on VS saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose WP method - Select the method of test sequences generation, WP	Method of test sequence generation WP is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using WP method
4	Export to OATS - Click on export file to OATS	File exported to OATS
5	Save Scripts - Select directory to save | Click on OK button	Script on OATS saved

6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed
1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose WP method - Select the method of test sequences generation, WP	Method of test sequence generation WP is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using WP method
4	Export to MTM - Click on export file to MTM	File exported to MTM
5	Save Scripts - Select directory to save | Click on OK button	Script on MTM saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed

1	Load File XML - (Req->Functionalities Functional.)Click on Load from XMI File button, select file and click on open	File XML loaded
2	Choose WP method - Select the method of test sequences generation, WP	Method of test sequence generation WP is selected
3	Generate Test Case - Click on generate test case from load test data button	The abstract test case are generated using WP method
4	Export to JMeter - Click on export file to JMeter	File exported to JMeter
5	Save Scripts - Select directory to save | Click on OK button	Script on JMeter saved
6	Exit - Choose between close the application or run the application again	Command chosen
7	Close - Click on Close button to end the application	Application is closed

- **Anote o horário de término desta etapa**

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br