Universidade Estadual de Campinas
Instituto de Computação

Leydi Rocio Erazo Paruma

A method and a tool for model-based testing of software product lines

Um método e uma ferramenta para testes baseados em modelos para linhas de produto software

CAMPINAS
2017

Leydi Rocio Erazo Paruma

A method and a tool for model-based testing of software product lines

Um método e uma ferramenta para testes baseados em modelos para linhas de produto software

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestra em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientadora: Profa. Dra. Eliane Martins

Este exemplar corresponde à versão final da Dissertação defendida por Leydi Rocio Erazo Paruma e orientada pela Profa. Dra. Eliane Martins.

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): CAPES

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Er18m

Erazo, Leydi, 1987-
A method and a tool for model-based testing of software product lines / Leydi Rocio Erazo Paruma. – Campinas, SP : [s.n.], 2017.

Orientador: Eliane Martins.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Linhas de produto de software. 2. Teste baseado em modelos. 3. Software - Testes. I. Martins, Eliane, 1955-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Um método e uma ferramenta para testes baseados em modelos para linhas de produto software
Palavras-chave em inglês:
Software product lines
Model-based testing
Software - Testing
Área de concentração: Ciência da Computação
Titulação: Mestra em Ciência da Computação
Banca examinadora:
Eliane Martins [Orientador]
Leonardo Monteechi
Maria de Fátima Mattiello Francisco
Data de defesa: 11-12-2017
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Leydi Rocio Erazo Paruma

A method and a tool for model-based testing of software product lines

Um método e uma ferramenta para testes baseados em modelos para linhas de produto software

Campinas, 11 de dezembro de 2017

Acknowledgements

First of all, I would like to thank God, for blessing me and allowing me to fulfill one more stage in my life. Thanks for Your Love and Mercy in the hard moments. Secondly, I would like to thank my family, mainly my parents Paulina and Aldugar. Without their encouragement and support, it would not be possible. I know that you are happy with this moment. I cannot forget the support that I had from my dear siblings John, Ivan, and Faber, from my sisters in law Nancy and July, and my beloved nephews and nieces.

My sincere thanks to my advisor Eliane Martins to lead me and help me to face the challenges presented in this process. I admire your way of life as advisor and researcher. Thanks for the talks, discussions, and challenges that we faced together. I also would like to thank Juliana Greghi. Thanks for all your support, the meetings and discussions about MARITACA and its rules.

Institute of Computing of Unicamp, thanks for the wonderful opportunity of to study in the best university of Brazil. Thanks for the financial support to CAPES and thanks to the DEVASSES project for the opportunity to make the scholarships in Department of Computer Science of Coimbra University. Thanks to the professors Marco Vieira, Nuno Laranjeiro, Nahmegh Ivaki and Nuno Antunes.

My dear friends, that I had the opportunity to meet: Pablo, Ricardo, Jael, El Gato and Edson. It is amazing to meet people with a huge heart like yours.

Thanks to all beautiful people that I had the privilege to meet, especially my second family in Brazil: Karina, Leo, Astrid, Tony, Pr Almir, Pra Eliane, Pr Marcos and Pra Ivanisse. I will never forget all the support that you gave me.

To the committee of my defense, my sincere thanks.

Resumo

As linhas de produtos de software (LPS) estão ganhando interesse devido à crescente demanda por produtos personalizáveis. Tal se deve, em parte, por que as LPS são um meio eficiente e efetivo de entregar produtos com maior qualidade a um custo menor. Em uma LPS, produtos têm requisitos em comum e também, características específicas a cada um. Testar se um produto implementa os requisitos comuns e específicos é um importante passo para garantir uma boa qualidade. No entanto, o teste de uma LPS é uma tarefa complexa, uma vez que a variedade de produtos que podem ser derivados a partir da combinação de características comuns e específicas é enorme. Mesmo que se escolha apenas alguns produtos selecionados, o esforço para testá-los ainda assim é grande, dado que os produtos variam em termos das características específicas selecionadas. Portanto, reutilizar casos de teste de um produto para o outro para determinar se satisfazem os requisitos funcionais, pode não ser possível. Os testes baseados em modelos (MBT) podem ser úteis neste caso, nos quais um modelo de comportamento pode ser obtido a partir dos requisitos e este modelo pode ser usado para a geração automática de casos de teste.

Neste trabalho é apresentada uma abordagem em que os requisitos SPL são centrados em casos de uso. Casos de uso (UC) são um formato popular para representar os requisitos. A partir das descrições de casos de uso escritas em um formato semi-estruturado e contendo a especificação de variabilidade, os modelos de comportamento são gerados automaticamente para um produto sob teste, na forma de um modelo de máquina de estado. Construir uma máquina de estado não é trivial para a maioria dos profissionais, que estão mais habituados com descrições textuais e informais dos requisitos. Em geral, a criação manual de modelos de máquinas de estado a partir de UCs pode ser demorado e propenso a erros. O objetivo é fornecer aos engenheiros de teste um método que os guie na criação dos artefatos necessários para que uma versão preliminar de um modelo de estado seja extraída automaticamente dos requisitos. Este modelo preliminar pode ser refinado para tornar-se adequado para uma ferramenta de geração de casos de teste.

Para esse processo de refinamento também são fornecidas algumas diretrizes. Como prova de conceito, desenvolveu-se um protótipo de uma ferramenta, MARITACA, que utiliza técnicas de processamento de língua natural para extrair as máquinas de estado a partir das descrições dos casos de uso. O texto apresenta o uso do método e da ferramenta em um exemplo ilustrativo, obtido da literatura, e em uma família de aplicações distribuídas tolerantes a falhas. Este estudo mostrou a aplicabilidade do método proposto. Uma das preocupações nos testes de SPL é a geração de casos de teste redundantes de um produto para outro. Os resultados, embora preliminares, mostraram que a maioria dos casos de teste gerados para um novo produto não são redundantes, pois envolvem características específicas de cada produto.

Abstract

Software product lines (SPL) are gaining interest because of the increasing demand for customizable products. This is partly because SPLs are an efficient and effective means of delivering products with higher quality at a lower cost. In SPL, products have common requirements and also, specific features for each one. Testing whether a product implements common and specific requirements is an important step to ensure good quality of the derived products. However, testing a SPL is a complex task, since the variety of products that can be derived from the combination of common and specific features is huge. Even if only a few specific products are selected, the effort to test them is still significant, since the products vary in terms of the specific features that are selected. Therefore, reusing test cases from one product to another to determine whether they satisfy the functional requirements may not be possible. Model-based testing (MBT) may be useful in this case, in which a behavior model can be obtained from the requirements and this model can be used for automatic test cases generation.

This work presents model-based product testing approach (MBPTA) for software product lines, in which requirements are centered on use cases. Use Cases (UC) are a popular format for representing requirements. From the use case descriptions written in the form of a semi-structured format and containing the variability specification, the behavior models are automatically generated for a product under test, in the form of a state machine model. Building a state machine is not a trivial task for most practitioners, who are more familiarized with textual and informal descriptions of requirements. In general, the manual creation of state machine models from UCs can be time-consuming and prone to errors. The goal is to provide the test engineers with a method that guides them in the creation of artifacts necessary to extract a preliminary version of a state model from the requirements. This preliminary model can be refined to become suitable for a test case generation tool.

MBPTA also provides guidelines for the refinement process of the preliminary model. As proof of concept, a prototype of a tool was developed, MARITACA, which uses natural language processing techniques to extract state machines from the use case descriptions. The text presents the use of the method and the tool in an illustrative example, obtained from the literature, and in a family of distributed fault-tolerant applications. This study showed the applicability of the proposed method. One of the concerns in SPL testing is the generation of redundant test cases from one product to another. The results, though preliminary, showed that most of the test cases generated for a new product are not redundant because they involve specific features of each product.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

In the last years, the software development on large scale and with different variants that fit the user requirements has made the software development companies recognize the need to introduce the Software Product Line (SPL) paradigm. According to [7], a SPL defined as: “a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way”.

Diverse benefits like cost reduction, decreased time-to-market, and quality improvement can be expected from reuse of domain-specific software assets [26]. Reuse and the maintenance of traceability between the different artifacts in the product line are fundamental requirements in this paradigm [40]. In according to Clements and Northrop [7], the management of the variability among the products derived from a SPL is an important strategy to improve the efficiency and effectiveness when compared to classic software engineering.

As with single-system development, in SPL is necessary to lead an efficient testing process to uncover defects in the derived products of a SPL. In an organization using SPLs, it is even more crucial since increasing the the share of testing costs decreases cost for each product. Testing a software product line is a complex and costly task since the variety of products that can be derived is huge. The major challenge in software product line testing regards the large number of required tests and consequently costs [15]

The use of model-based testing (MBT) is adequate, as it allows test cases generation from abstract models representing system behavior. This technique offers advantages like automatic creation of test cases, early validation of the requirements for test automation and pro-active reuse [42]. Among the different notations that can be used to represent system behavior, state machine models are commonly used for test case generation purposes. However, in practice, testers are not yet used to build models that can be further processed by a tool. Creating and updating models is still a challenge for the widespread use of MBT [6], as practitioners use to write requirements as informal textual specifications. For software product lines there is still another difficulty, which is how to represent variability within models. In this sense, use cases are a good option, in that use cases are generally described using natural language texts. Besides, there are various proposals on how to extend use cases to represent commonalities and variabilities among product line members like in [18], [12] just to name a few.

1.1 Context and scope

The context of this work is the model-based testing (MBT) of end-products that can be instantiated from a software product line. The goal is to determine whether the instantiated products satisfy the requirements. What is distinct with respect to single product testing is that requirements include common and variant functionalities. Therefore, it is not only important to verify whether a new product satisfies the common functional requirements, but also to determine if the product implements the variant features: does the product include the required features? Is there a feature that is omitted? For each new product, it is thus necessary to create test cases to exercise the specific combination of common and variant features. For this reason, reuse of test cases that cover functional requirements is not straightforward [35], as modifications to the previous test cases are required to cover variant features as well as requirements that are specific to a given product. That is why the use of MBT is the focus, in that manually generating test cases for each product requires a great effort for the test team. Another point is that, for the reuse of test cases, these must be maintained and evolved as each new product is created, and also, when new product versions are created. Significant work is necessary for manually maintain and create new test cases, and over time, what eventually happens is that the test case base becomes obsolete and is simply ignored. With MBT, what is maintained is the test model, not the test cases. As existing products change, or new features are created, the test model evolves accordingly. Instead of reusing test cases, new ones are generated based on the updated model. Since models are simpler and described on a more abstract level than test cases, they are easier to maintain. Besides, since test models are based on the requirements, the traceability between requirements, test models and in consequence, the test cases, is straightforward. Also very important, although not focused on our work, is that test model creation helps to validate the requirements and uncover omissions or conflicting feature interactions.

However, two points are to be considered. The first one is that requirements are generally described in natural language. The creation of a test model from these textual requirements, with the formalization necessary for test case generation, is not an easy task for most practitioners. The second one is that, in single product development, it is possible to have one or more test models, according to the different test objectives, for one product. But, what about SPL, in which there are various different products? Some works use a product-by-product strategy (see Section 2.2), in which a test model is built for each and every product. However, this approach becomes intractable for larger SPLs. Others consider describing variability in the test model, creating a so-called 150% model, as they describe more than one possible product (e.g. [42] [49]). Others consider variability in the requirements specification, more specifically, as use cases, and derive test models from the requirements (e.g. [5][35]). The difficulty with these approaches is to cope with constraints that may exist among features. Another point is that the test model is manually produced, or can also be obtained using model transformations (e.g. [35]).

1.2 Proposed solution

This work describes an approach to guide the testing process of specific products in a SPL. The goals of the approach are: *i)* to define the activities and artifacts needed for testing; *ii)* to allow the extraction of a test model from the developed artifacts; and *iii)* to generate test cases from the extracted test model.

The test model is an Extended Finite State Machine (EFSMs) representing the behavior of a specific product. Use case descriptions are the main source for model extraction. The use cases descriptions are also extended with exceptional and variation flows; the former is used to handle exceptional behavior in the software systems whereas the variation flows are used to handle variability when specifying a product family. In order to reduce ambiguities and incompleteness inherent in natural language descriptions, this approach uses a restricted textual form, inspired on [52]. The extracted state-models represent sequence dependencies among use cases.

Many approaches already exist for the generation of test models from use cases (e.g. [45][51]), but they are for single product software, and therefore, must be adapted for SPL context. In special, they need to be adapted for the representation of variability at the use case level. Although some use case specifications were proposed to cope with variation points [4] [36], they were not conceived for automatic test model extraction. The extracted state model is, in fact, a skeleton of the product-specific test model, in that, it may contain redundant states, and the triggers and actions are described in the same abstraction level as the use cases from which they were derived. In this way, the proposed approach is meant to solve the following problems: *i)* how to describe use cases in order to cope with the variability present in SPL requirements? *ii)* how to take into account the constraints among features, as well as the features selected for a specific product? *iii)* how to automatically generate the state model from the textual requirements? *iv)* how to refine the skeleton test model automatically extracted from the use cases in order to obtain a model amenable to a test case generation tool?

1.3 Contributions

The contributions of this work are: *i)* an approach for model-based system testing in software product lines context; *ii)* a structured use case description that allows the representation of the variability as well as exception handling; *iii)* a set of rules and guidelines for the transformation of use case descriptions into a state machine model; *iv)* a prototype tool to support the automatic extraction of models from use case descriptions, which uses natural language processing techniques to extract the initial version of a state model for a specific product. The work also produced two accepted papers:

- Modeling Dependable Product-Families: from Use Cases to State-Machine Models [16].
- MARITACA - from textual use case descriptions to behavior models [17]

1.4 Dissertation structure

This work is structured into eight chapters plus appendices. Chapter 2 provides background concepts on the software product lines, challenges in software product line testing and model-based testing. Chapter 3 presents the related work on Model-Based Product Line Testing and model extraction from textual specifications. The approach for model-based testing in a software product lines is presented in Chapter 4. Chapter 5 details the prototype tool implemented to support the model extraction from textual descriptions. Chapter 6 shows the application of the approach. A discussion of the obtained results is presented in Chapter 7. Finally, is presented conclusions and future works in Chapter 8.

Chapter 2

Fundamental concepts

This chapter defines concepts that will be used in the rest of this text. Section 2.1 presents the definition of software product lines (SPLs) and main concepts such as features and feature models. Strategies for software product line testing and issues of software product line testing are described in Section 2.2. Finally, are shown the concepts and use of model-based testing in the SPL context in Section 2.3.

2.1 Software product line

Software product line engineering (SPLE) is a new software development paradigm, which has been adopted in the last years in software development organizations. SPLE is defined as a paradigm to develop software applications, which is based on two principles: platforms and mass customization [39]. Mass customization is the large-scale production of goods tailored to individual customer's needs [11] and a software platform is a set of generic components that form a common structure, from which a set of derivative products can be developed [32].

The products share features, requirements, code, test artifacts, and so on. The process of deriving a product from a SPL consists of the selection and integrations of core assets. These core assets are shared by all products that belong to a SPL. Differently from conventional single product development, the definition process of a customer-specific application hence is influenced not only by the customer requirements but also by the capabilities of the product line [4]. Therefore, requirement analysis and management methods are needed to ensure that all products derived from a product line satisfy the customer requirements.

The improvement of cost and time to market are strongly correlated in software product line engineering: the approach supports large-scale reuse during software development. Thus, both development cost and time to market can be dramatically reduced by product line engineering [29].

The development of a SPL is usually divided into two processes: *Domain Engineering* (development for reuse) defines the commonality and the variability of the product line, the general concept of a product line is provided as well all the assets that are common to the product line. *Application Engineering* (development with reuse) produces customer-specific software products on top of the product line, by selecting and configuring shared assets defined in the domain engineering [29]. Figure 2.1 presents the two processes of software product line engineering.

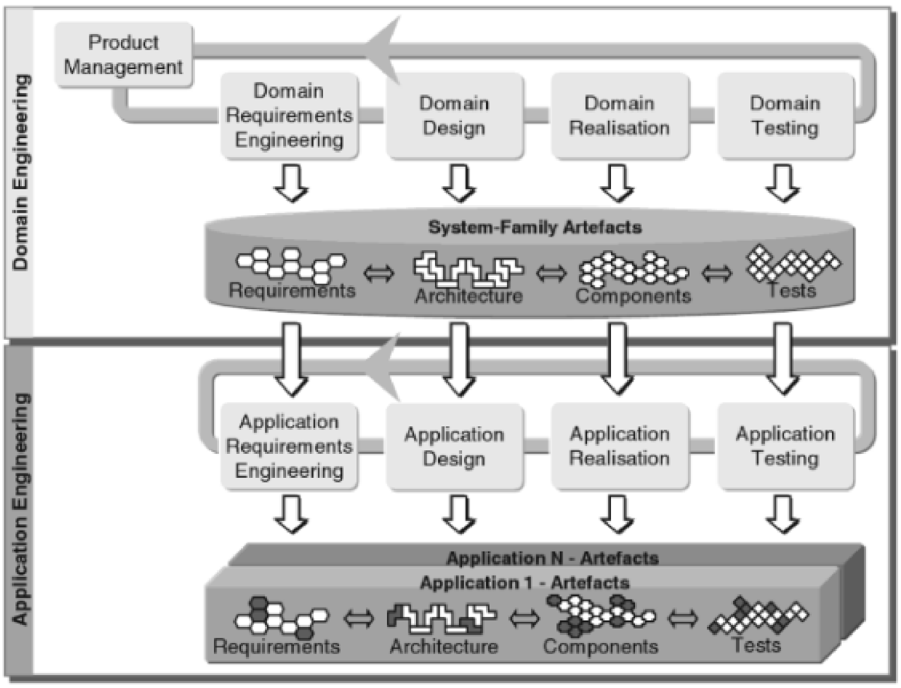


Figure 2.1: The two-life-cycle model of software product line engineering [39]

Variability is a central concept in the SPLs; this defines how different products can be derived from a SPL. Variability allows the derivation of different product family applications by reusing the realized product family assets [22]. According to Metzger [31], variability describes the variation among the applications of a software product line in terms of properties, such as features that are offered or functional and quality requirements that are fulfilled. This variability is defined in the domain engineering by means of variation points. A variation point defines a decision point along with its possible choices (called variants).

The most evident question in this process is how to handle and represent variability [24]. Variability management is the key to introduce software product line concepts in software development organizations. Decision Models [47] and Features Models [25] are commonly used to represent variability within SPLs. Feature models can be used to communicate *Domain Engineering* and *Application Engineering*, these models describe allowed variabilities among products of the same SPL, feature dependencies, guide the selection of features for a construction of a specific product, and so on. To handle variability, use cases are a powerful tool to capture functional requirements in the requirements phase [39].

Feature Models are frequently used to model commonalities and variabilities through features (mandatory, optional, and alternative) and a set of all possible relations between them. According to [25], a feature is defined as a *"prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"*. The features of a SPL are grouped in a feature model to represent the SPL variability, where a specific product can be represented by a unique configuration of features.

The Features are categorized as common, optional and alternative. The common features determine the degree of commonality in the SPL. The optional and alternative features determine the degree of variability in the SPL [18]. Common features are also referred to as *mandatory*, which must be provided by every member of a software product line. Optional features are provided by some members of a software product line. Alternative features, only one feature can be selected from a group (XOR) in a member of software product line. "OR" features, one or more features can be selected from a group in a member of a software product line.

Also, constraints between features can be represented in the feature model, such as, "requires" and "excludes", For example, if a feature to be included necessarily add (or delete) other feature.

Figure 2.2 shows a feature model of a car audio system. The traffic message channel, playback, and basic controls are common to all radios; a radio must allow the selection of channels or titles, the change of the volume, and the search for new channels or new titles. The choice of playback media (CD or cassette player) also influences the behavior. The availability of a USB port, the wheel control, and the navigation system are optional features. Depending on the playback media, the update of map data for the navigation system is an optional feature [49].

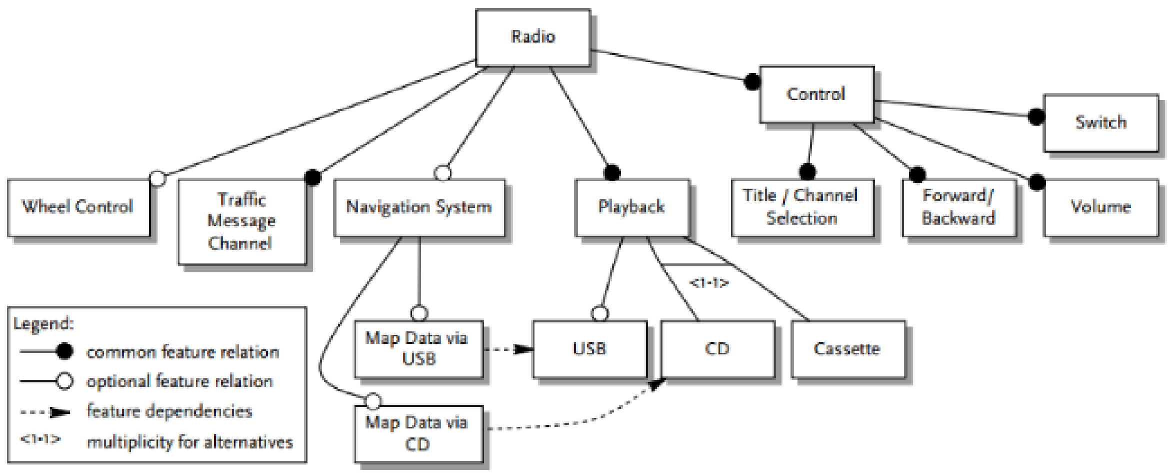


Figure 2.2: Feature model of car audio systems [49]

2.2 Software product lines testing

A persistent problem with software product line is the need to verify that each unique, and legitimate combination of artifacts works correctly. All possible combinations of the assets, each configured in all possible ways, quickly becomes unmanageable [13]. In the literature have been reported a wide variety of strategies for testing product lines, in [9] are presented five testing strategies:

i *Testing product by product:* test cases for each derived product are created independently. This approach offers the best guarantee of product quality but is extremely costly.

ii *Incremental testing of product lines:* the first product is tested individually and the following products are tested using regression testing techniques (features of the product previously tested are re-tested).

iii *Design test assets for reuse:* test cases are created as early as possible in domain engineering. Domain tests aim at testing common parts and preparing for testing variable parts. In application engineering, these test assets are reused, extended and refined to test specific products.

iv *Division of responsibilities:* testing is divided into levels in both domain and application engineering. In Domain Engineering are tested the common components using unit testing. When a product is derived in application engineering, it is responsible for integration, system and acceptance testing.

v *Opportunistic reuse of test assets:* this strategy is applied to reuse application test assets. Assets for one application are developed. Then, the application derived from the product line uses the assets developed for the first application. This form of reuse is not performed systematically, which means that there is no method that supports the activity of selecting the test assets.

Issues in software product line testing are related to the huge amount of products that can be derived from a SPL. Techniques to deal with the problem of the combinatorial explosion are used to reduce the set of possibilities to a reasonable and representative set of product configurations. *Combinatorial interaction testing* (t-wise, pair-wise) is a technique that enables the selection of a small subset of products where the interaction faults are most likely to occur. T-wise is the best-known application of combinatorial testing, this assumes that many errors only arise from the specific interaction of a small number of variables; in Pair-wise the input model is described as combinations of two features, obeying the constraints between them, it might be easier to find inconsistencies, rather than trying to combine all features at once [13].

Other issues in testing process of products derived from a SPL are related with: *i)* test case generation, in which is difficult to assure that test model generated from test artifacts are consistent with requirements; *ii)* test case selection, in which there exists the problem of redundancy of test cases to test a specific product; *iii)* test of absent variants: in domain engineering components can be unavailable to be tested yet, but tests can be executed by adding test code to absent components. In application engineering tests can be executed when variants are available; *iv)* in an application-specific test, there are gaps between what is available and what is required to customer products [27].

The major difference between SPL testing and single product testing comes from variability in domain artifacts. Handling variability is very challenging and the decisions on how to deal with it are the starting point in SPL testing [27]. Insufficient attention has been given to the problem that is how to test product lines, although software organizations recognized that testing takes an amount of development resources.

Product line testing requires a more efficient test approach than those used in single system engineering because the variety of derived products from the product line is huge. Testing consumes up to 50% of the total effort in single system engineering [3], and this percentage increases in SPL because the variability of the product family complicates testing [41]. Reducing cost and increasing productivity in software product line testing is an important challenge for Software Development Companies.

2.3 Model-based testing (MBT)

Model-based testing is a technique used in the automatic generation of test cases from models, these models are generated throughout the development process. MBT provides benefits like the early validation of requirements to find defects and ambiguities, the systematic and automatic creation of test cases, and reuse of the generated models [42].

Model-based testing can also be very useful for software product line testing, in that MBT provides a technique for generation of test cases using models extracted from software artifacts [18]. MBT provides the automation of test case generation, which improves testing productivity. In addition, the models can be reused and are easily adapted to describe variability [38]. There are three important aspects to adopt model-based testing: the notation used to describe the software behavior in the models, the algorithm to generate the test cases and tools to support the test case generation [10].

One key advantage of MBT is that test cases can be systematically generated from requirements represented by the model, in that way, automating test case design. A key element for the use of a MBT approach is the model used for software behavior description (e.g. activity diagrams or finite state machines). MBT requires solid modeling expertise from developers in order to build models that can be processed by a tool. However, developers are more familiar with describing requirements using a narrative text, as in use case descriptions.

Four different techniques in MBT have been described in [46]: *i)* generation of test data from a domain model; *ii)* generation of test cases from an environmental model; *ii)* generation of test cases with a oracle from a behaviour model; *iv)* generation of test scripts from abstract tests. Besides, Utting et al. identified that the models used in MBT must be detailed, as much as possible , but they must be small when compared with the software size under test.

MBT has not yet a great use in software industry, as MBT relies on precise models for test case generation, and developing such models, in special for large systems, requires a solid modeling expertise which the test teams, in general, do not have. Besides, in practice, written specifications are rarely, and when they exist, they are written in natural language.

2.4 Chapter Summary

Software product lines aim to reduce cost, time to market, improving the software quality, and improving the reuse. This paradigm is still in evolution and new challenges in software development life cycle are presented. Especially in the testing area, to verify that each product derived from a SPL works correctly strategies have been proposed. These strategies aim to reduce cost and effort in test tasks, some are focused on test case generation to test product by product and others promote the test case reuse in all derived product. In test case generation, MBT technique is used due to the test cases can be derived from models that describe functional aspects of the system under test. These models can be reused and are easily adapted to describe SPL variability, helping to improve testing productivity.

Next Chapter presents proposed approaches in modeling software product line requirements, test case generation using model-based testing technique and approaches to extract test models from textual specifications.

Chapter 3

Related works

In this chapter are presented approaches to capture SPL requirements and to extract analysis models from textual requirements, more specifically, use case descriptions. Works to model software product line requirements using UML are shown in 3.1. In Section 3.2, approaches to extract test models from textual specifications are presented. Finally, in Section 3.3 the use of a Model-Based Testing in software product line is shown.

3.1 Modeling software product line requirement using UML

As for single systems, it is also possible to represent SPL requirements with Use Cases (UCs), extended in some way to include variability information. Various extensions were proposed, as for example [5], [18], [34], [12], just to name a few.

Bertolino and Gnesi [5] proposed PLUC (Product Line Use Cases), which describes variations in UCs using tags to explicitly indicate the variable parts of SPL requirements. PLUC is the base of a testing methodology, PLUTO (Product Line Use Case Test Optimization). As in the approach proposed in our work, they also start planning tests from the requirements specification, as it is in the requirements phase that the commonalities and variabilities are defined. Use Cases (UCs), written in an extended notation PLUC, are the basis for test case derivation. They use the well-known Category Partition method, which is a functional testing approach that focuses on functional units that can be separately tested. In PLUTO, the functional units are UCs, and the approach addresses the generation of test data for each UC. For representing variability, PLUC uses variation tags, so that the product is obtained from the instantiation of given values to these tags. Since they do not use a Feature Model (FM), it is necessary to use constraints among these tags in the PLUC, to avoid the creation of invalid feature combinations. Besides UCs, there is also a Test Specification, derived from a PLUC using the selected test method. A Test Specification may contain several scenarios: one is the main (or basic) scenario and the others correspond to alternative scenarios. In the process of test derivation, they instantiate the tag values, according to the constraints, to obtain the test cases for specific products. At the moment, this process of test case derivation was not yet automated, but a tool was under construction. In this approach, test cases are automatically derived from a specific product, using a state machine model-based testing tool.

Nebut [34] presented how to derive both functional and robustness test objectives from use case contracts. Their notation allows not only to extend use cases with parameters, representing system inputs, but also the sequential dependencies among use cases through the use of contracts (pre- and post-conditions). Similar to the proposal presented here, they also generate transition system as a test model for product-specific testing. The transition system represents valid sequences of use cases, which is obtained from the contracts. They propose an algorithm to extract this transition system from use cases description, but they express UCs as logical predicates, whereas the proposed approach uses a structured natural language.

The PLUS (Product Line UML based Software Engineering) method, proposed by Gomaa [18] as its name indicates, uses various UML notations for modeling a SPL. The method comprises Requirements, Analysis, and Design phases, in which different artifacts are produced, as use cases, feature models, class and sequence diagrams, statecharts and so on. The goal is to guide users to produce test-ready models, that explicitly identify the mappings, or relationship between features, model elements, test specifications, and test dependencies, in order to generate tests for specific products derived from the SPL. Besides UML diagrams, they also use feature conditions in models, which they are set to true when a feature is selected and feature dependency tables to link features to associated classes.

Customizable Activity Diagrams, Decision Tables and Test Specifications (CA-DeT) is a functional testing method that creates test specifications from both the UC and FM models of a SPL [36]. As with the proposed approach in our work, CADeT covers both Domain and Application Engineering of an SPL development process. CaDET uses Activity Diagrams for each use case. Decision Tables are associated with each Activity Diagram, in that a path in the latter corresponds to a column in the table. Each column corresponds to a reusable Test specification. Test cases for specific products are obtained from the Test Specification according to the selected features. The approach proposed in our work, on the other hand, uses State Machine models to represent the whole set of use cases. Besides, we do not represent variability beyond the Use Case model.

Oliveira et al. proposed the FeDRE (Feature-Driven Requirements Engineering) approach [12], which consists of a set of artifacts, activities, and guidelines that provide support to the requirements specification of SPL. There are three main activities: **Scoping** (produces the artifacts: Feature Model, the Feature Specification, and the Product Map), **Requirements Specification for Domain Engineering** (produces the artifacts: Glossary, Functional Requirements and Traceability Matrix), and **Requirements Specification for Application Engineering**. They use the feature model as a basis mainly because features are realized into functional requirements, by considering them as the context of the use case specification. They use feature models and features specifications to model and specify common and variable properties, also defined some artifacts as The Product Map (maps features to products) and the Traceability Matrix (maps use cases to features) help in keeping the feature model, products, and the specification synchronized. FeDRE is not intended for testing, consequently, no test models are generated.

3.2 Model extraction from textual specifications

The extraction of state machines from use cases has been researched in the last years, for single products. The problem of using automatic methods to transform textual specifications into Analysis models has been addressed in different works. There are also approaches that use state machine models to represent the whole product behavior.

There are various works aimed at state-model extraction for single products. For example, Somé [45] proposes a formalization of use cases descriptions and an algorithm that generates a finite state transition model for single products. This approach uses a form of natural language to write use case descriptions defined by precondition, flows (basic and alternatives) and postconditions. An algorithm generates a hierarchical type of finite state transition machines from use cases by looking at a use case as a network of connected scenarios.

AGADUC process (Automated Generation of Activity Diagrams from UCs) generates UML Activity Diagrams to present the workflows in use cases descriptions [14]. They propose the SUCD (Structured Use Case Descriptions), which is a simple structure with a very limited syntax used to guide the description of use cases workflows. SUCD allows describing concurrent flows, looping, and branching and condition evaluation. For every UC, AGADUC process generates a single activity diagram to represent the Basic Flow and Alternative Flows merged together. AGADUC also generates a separate activity diagram for each subflow and Extension Points. All the activity diagrams are generated simultaneously to show how they relate to each other. They present the AREUCD tool, which uses a limited syntax to write the use case descriptions [14].

The works of Yue et al. extract various Analysis models for single systems, such as class [54], activity and sequence models [53] and state machines [51] from a structured notation for use case description called RUCM (Restricted Use Case Modeling), which is a structured notation for use case specification. The use of a restricted notation aimed at reducing ambiguities and facilitates the processing of tools. Besides the structured language and a use case template, their work also proposes a set of transformation rules and algorithms to derive automatically preliminary Analysis models. The authors did an extensive literature review in order to derive their notation, and also, they perform experimental studies to assess the usefulness of the approach for end users [52]. The approach is also supported by a tool, aToucan [54].

The above-mentioned approaches are focused on describing use cases and generating analysis models for single systems. In what concerns the testing of Product Families, several authors also proposed techniques to derive test cases from use case descriptions modified to cope with variability. The survey presented by [40] presents some of these works. As pointed out in the aforementioned reference, there are many challenges for product-family testing, among them, how to deal with variability in the test artifacts. In this respect, our work gets inspiration from the FeDRE (Feature-Driven Requirements Engineering) approach for requirements specification of SPL [12]. Although our approach is only concerned with the requirements phase, and not with testing, it provides guidelines that establish traceability links among features, products, and requirements, which is useful for developers as well as for testers.

The interest of our work is the automatic generation of a product test model from use case specifications that is useful for test case generation. It gets inspiration from the approaches aforementioned, which were focused on single systems, to propose our SPL testing approach. It was also necessary to search for approaches that addressed the problem of SPL Requirements modelling, and there are several in the literature. For our work gets inspiration from [18], with respect to the representation of variability in Use Case modeling. From FeDRE approach [12] our work adopted the Domain Requirements Engineering process and artifacts as the first step to our approach.

Our work uses the RUCM approach as a basis for use case description; RUCM was modified in order to handle variability of product lines and exceptional behavior. The reason for using RUCM is that the authors did an extensive literature review in order to define an adequate template for use case description, and also, they assess the usefulness of their approach through experimental studies [52].

The work of Greghi et al. extracts information from natural language standard documents and semi-automatically generates Extended Finite State Machines (EFSMs) [21], and it is supported by TXT2SMM prototype. Our work gets inspiration from [21], by analyzing the use cases descriptions and using a set of rules to categorize the sequence of words that characterize states, transitions, triggers and so on.

3.3 MBT and software product lines

Software product line testing is a tedious task since the common and the shared variant requirements have to be tested for each derived product. Indeed, the same functionality in a derived product, cannot be reused exactly in other product (the functionality may be different from one product to another, due to the crossing of different variation points). So, for testing a given function common to all products, specific test cases may have to be written for each specific product [35].

The use of a Model-Based Testing approach, proposed for single systems testing, can thus be amenable to SPL testing. MBT is useful for SPL testing because it allows test case generation automation, which improves testing productivity, in that way, automating test case design [10]. Also, since models can be obtained from the requirements, they are useful not only to validate these requirements but also to set up test cases early in the development cycle. Another point is that models are suitable to represent commonalities and variabilities present in SPL requirements.

In the context of software product line testing, different approaches to generate test cases from test model have been proposed. Nearly all of these approaches use the strategies presented in subsection 2.2 for test case generation. For example, Nebut [35] presents an approach to automate the test cases generation for any chosen product, from the functional requirements of a software product line. The requirements are modeled using UML use cases. The use cases describe commonality and variability, and they are enhanced with parameters and with contracts. Use cases and scenarios are combined to generate test objectives that are refined into test scenarios. A Use Case Transition System (UCTS) is derived, which is a behavior model for each specific product. Test cases are generated in two steps: sequences of use cases are deduced from use case contracts and scenarios are substituted to each use case to produce a test scenario that is finally transformed into a set of test cases. The test cases derived from product-specific behaviors can be executed against the chosen end product to check that the expected functionalities have been correctly implemented. A behavior model can be simulated, besides serving as a basis for test case generation. However, they do not use a Feature Model, but a decision model to select the specific features for a product.

Weißleder [49] proposes the use of state machines to model product variants. They use the generalization-specialization relationship in object-oriented paradigm in the following way: in UML, a state machine is associated with a context class, that is, the state machine represents the behavior (a property) of its context class. In this way, instead of specializing the state machine, they use the inheritance relationship between classes and reuse the state machine of the parent context class as a behavioral description of its subclasses. Variability is addressed with the use of guard conditions as well as the use of submachine states. They propose the reuse of state machines for automatic test case generation, indicating that a reasonable amount of test effort can be saved by using this method. Test cases are then automatically generated with a tool developed by the group, called Partition Test Generator (ParTeG), that supports state-based test criteria but also test data criteria, such as boundary value.

The ScenTED technique (Scenario-based TEst case Derivation) was aimed at reducing effort in product family testing [41]. This is a model-based, reuse-oriented technique for test case derivation. The reuse of test cases is ensured by preserving variability during test case derivation. In domain engineering, the reusable artifacts (use case scenarios and activity diagrams including variability) are created and in application engineering, these artifacts are reused to create customer-specific applications. The problem that presents this form of reuse is that it is not performed systematically, there is no method that supports the tester in selecting reusable test cases. Functionalities of the new product might not have been tested completely if the selection of the test cases was carried out falsely, i.e., the test coverage of an application is not guaranteed [42].

The application of MBT approach for SPL driven by UML and CVL (Common Variability Language) models is proposed in [37], the test cases created from the combination of these two models is a simple skeleton that must be refined by a human tester. [38] proposes a toolchain combining combinatorial testing and model-based testing for a SPL. The combinatorial testing is used to cover all pairwise feature configurations, whereas the model-based testing approach is used to generate test cases for each derived feature configuration on. In this approach, they also generate tests for specific products, using available open-source tools.

Test every product from a software product line requires a high effort, due to the test cases generation can be a difficult process taking into account different alternatives to apply the reuse. In SPL is possible to reuse analysis artifacts (as use cases), test artifacts (as test models, test cases), code and so on. Many testing techniques are proposed to create reusable artifacts, which in domain engineering generates reusable test artifact and in application engineering selects and adapts these test. Unfortunately, it often takes longer to adapt the generated test cases to the current product than it would take to create a new suite of test cases to the current product to guarantee that the product is tested completely.

The main concern in SPL Engineering is reuse: reuse of artifacts produced in the Domain Engineering in the Application Engineering, as well as the reuse of test artifacts, as seen in Section 2.2. However, the proposed approach requires the recreation of the test model every time a new product or a new version is created. This occurs because when requirements are modified, test cases must be adapted/updated to ensure that the new functionalities are covered. In principle, the only assets reused are the ones built during Domain Engineering, but nothing prevents Test Engineers to reuse test models and test scenarios well. We believe that this is better, as it is not necessary to maintain test models and test cases, which becomes expensive over time.

3.4 Chapter Summary

Use cases are an easy way to express the functional requirements of a system, as these are well-structured, easy-to-understand documents written in a controlled natural language. In software product lines by enhancing use cases, many approaches have been proposed to describe software product line dealing with variability. On the other hand, manual or automatic test models extraction from these textual descriptions is performed using algorithms, transformation rules and so on. The quality of these models is directly related to the quality of the artifacts domain, which comes from the user specification. This is a classical problem for testing since tests are always generated to validate an implementation with respect to a specification. If the quality of the specification is low, the same happens with the test cases generated from it.

Recently MBT has been used in software product line context because this technique allows introducing variability into test models from requirements. Approaches to generating test cases from test model as class models, sequence models, activity models and state machines are proposed and used. A comparison of different approaches to describe SPL requirements using use cases: extracts test models from use cases descriptions and that generate test cases from test models is presented in Table 3.1.

Table 3.1: Comparison among related works and proposed approach

Approach		Represent SPL requirements with UC	Generate models from UC specification	Generate test cases from state machine models	Strategies for reuse
1	FeDRE [12]	Yes	No	No	No
2	AGADUC [14]	No	Yes Activity diagrams	No	No
3	RUCM [54] [53]	No	Yes Class, activity and sequence diagrams	No	No
4	RUCM [52]	No	Yes State machines	Yes	No
5	SOMÉ [45]	No	Yes State models	No	No
6	PLUTO [5]	Yes	No	No	Instantiate test cases for a specific product
7	ScenTED [42]	Yes	Yes Activity diagrams	No	Design test cases for reuse
8	GREGHI [21]	No	Yes State machines	No	No
Our work		**Yes**	**Yes State Machines**	**Yes**	**Reuse domain artifacts**

Chapter 4

A model-based product testing approach

This Chapter presents the model-based product testing approach (MBPTA). Figure 4.1 shows the schematic view of the approach, with the artifacts necessary for model-based testing of products created from a SPL. Section 4.1 describes the Domain engineering where artifacts that represent SPL requirements are defined. Section 4.2 presents the Application engineering, in which is shown the artifacts needed to the test case generation. An illustrative example is used throughout the Chapter, which is based on the microwave oven product line. The requirements and the product features are the same defined in Gomaa's book [18].

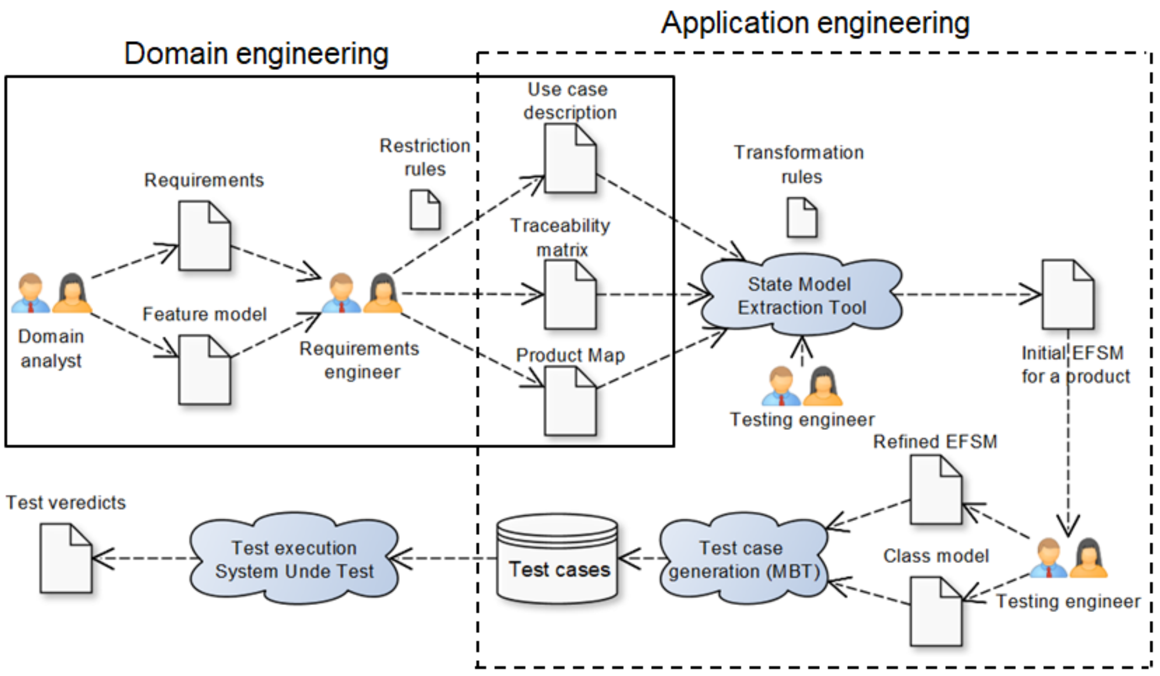


Figure 4.1: Process to generate state machines

4.1 Domain Engineering: Requirements modeling

The requirements modeling in the domain engineering phase is an important activity, in which the artifacts that are used to extract the state machines are defined. The artifacts generated in domain engineering are: feature model of a SPL, use case diagrams to represent user's interaction with the system, use case template to write the use case specifications of the system, and traceability matrix to define links between the features and use cases.

4.1.1 Feature model

The feature model can be used to capture, organize and visualize features of a family of systems. Features are the attributes of a system that directly affect end-users. In this model common, optional and alternative features are used to represent the variability in the SPL. Selection of optional or alternative features is not made arbitrarily. It is usually made based on concerns that the end-user has. In the MBPTA, the feature model is used to determine and visualize the products that can be derived from a SPL by the configuration of available features.

Figure 4.2 shows the feature model of the Microwave oven software product line [18]. The model represents common, alternative and optional features. The microwave oven systems can be configured to choose from many optional and alternative features, and the features dependency constraints must be taken into account. If a feature should be used in a specific product, the feature alternative or optional should be selected. Features can be selected for all products or just some of them. The selection of one feature can prohibit the selection of another one.

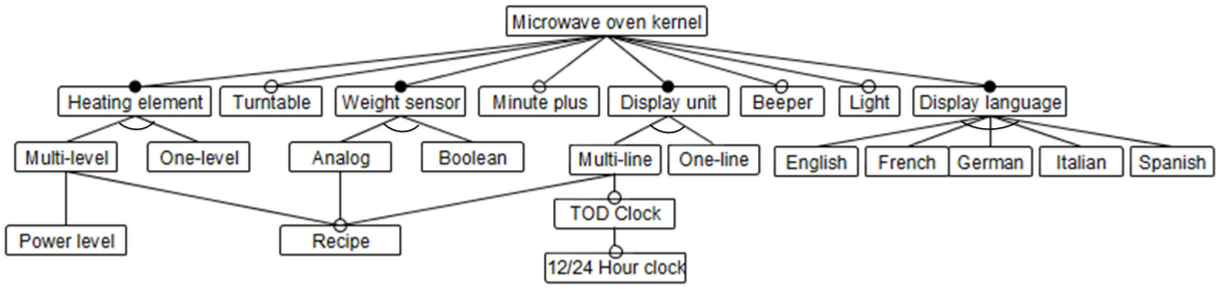


Figure 4.2: Feature model of the Microwave oven software product line (based on [18]

The microwave oven SPL has common functionalities for all products like input buttons for selecting Cooking Time, Start and Cancel, as well as a numeric keypad. Cooking is possible only when the door is closed, when there is something in the oven and when the cooking time is not zero.

For some products, alternative features can be selected, for example, there is a choice of language for displaying messages. The default language is English, but other possible languages are French, Spanish, German and Italian. It also has a display to show the cooking time left. Prompts and warning messages are displayed on the display unit. The basic oven has a one-line display; more-advanced ovens can have multi-line displays. The time-of-day option can be used only when the multi-line display is present. A microwave heating element with two options: multi-level and one-Level (selected for default) is provided. The default sensor is boolean weight, but the analog weight can be selected if desired.

Specific products can have or not have the optional features. The top-of-the-line oven has a recipe optional feature, which needs an analog weight sensor, a multi-line display, and a multi-level heating element. A beeper indicates when cooking is finished. The oven has also a lamp, which is switched on when the door is open and remains on while cooking food. It is switched off when the door is closed and when cooking stops. There is also a turntable, which rotates during cooking. The Power level optional feature can be selected only if the multi-level heating element is selected.

4.1.2 Use case diagrams

Use Case Diagrams are used to represent user's interaction with the system. This diagram describes the functional requirements of the system in terms of actors and use cases, considering the system as a black box and describing the interactions between the actor(s) and the system in a narrative form consisting of user inputs and system responses [18].

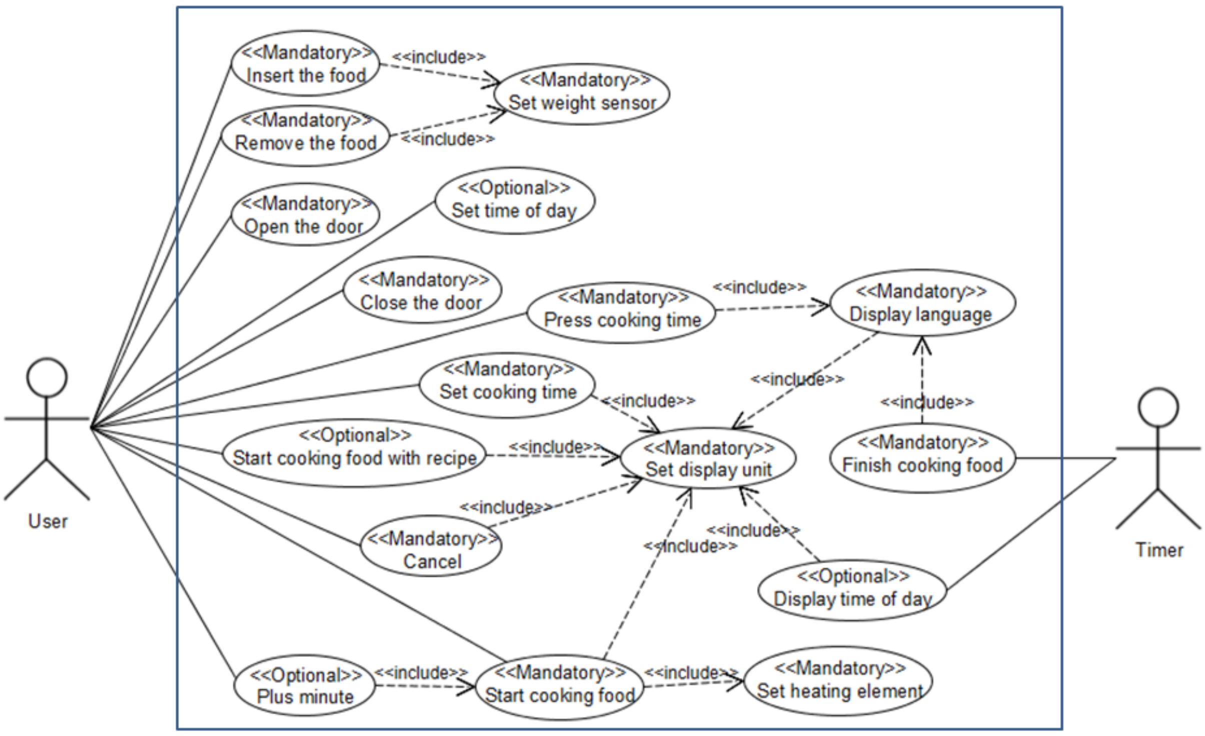
In order to specify the functional requirements of software product line, different kinds of use cases are defined: *kernel use cases*, which are needed by all members of the SPL; *alternative use cases*, where different use cases are needed by different members of the SPL; and *optional use cases*, which are needed by only some members of the SPL [18]. Each use case represents the interactions between the user and the system. In the MBPTA, is proposed that in each flow of use cases exists only one request from the user. Although the system can execute one or more actions in response to a user request. The use case diagram presented in Figure 4.3 has more use cases than the original one presented in [18] because our structured use case description required some original use cases to be split and rewritten.

4.1.3 Use cases description

For the description of use cases, a template was defined. Restriction rules and conventions to be followed were also defined to allow the processing by a state machine extraction tool. These are explained in the following.

Use case template.

Use Case Descriptions (UCD) are an important artifact used in Requirements Engineering. These descriptions can be written using natural language, which therefore increases the risk of ambiguity in UCD. A template well-structured and a set of restrictions rules of natural language can help to write, read and reduce the ambiguity of UCD. In the MBPTA, a template and restrictions rules are used to write UCD. UML provides no specific notation for specifying UCs, so it is common practice to adopt some template to help to read and review use cases. The template adopted in our work is based on Restricted Use Case Modeling (RUCM) approach [52]. There were two main reasons for



User
Timer
<<Mandatory>> Insert the food
<<Mandatory>> Set weight sensor
<<Mandatory>> Remove the food
<<Mandatory>> Open the door
<<Optional>> Set time of day
<<Mandatory>> Close the door
<<Mandatory>> Press cooking time
<<Mandatory>> Display language
<<Mandatory>> Set cooking time
<<Optional>> Start cooking food with recipe
<<Mandatory>> Set display unit
<<Mandatory>> Finish cooking food
<<Mandatory>> Cancel
<<Optional>> Display time of day
<<Optional>> Plus minute
<<Mandatory>> Start cooking food
<<Mandatory>> Set heating element
<<include>>

Figure 4.3: Use case diagram of the Microwave oven software product line

this choice: *i)* it gets inspiration from various templates proposed in the literature and, most important; *ii)* it uses a structured language, proposing a set of restriction rules to write UCs in order to facilitate automated UC analysis necessary to extract the behavior models.

However, there was a need to create some extensions, first, to handle variability for SPL, as RUCM is intended for single systems. Second, to allow the separate specification of exceptional behavior necessary to specify exception handling mechanisms. TABLE 4.1 shows the use case template used in our work.

Table 4.1: Use Case Template

Use Case Id	The identifier of the use case.	
Use Case Name	The name of the use case. It usually starts with a verb.	
Use case type	The type specifies whether the use case is mandatory, optional or alternative.	
Brief Description	Summarizes the use case in a short paragraph.	
Associate feature	The feature(s) associated with the use case.	
Feature cardinality	Cardinality associated with the alternative, mandatory and optional features.	
Primary Actor	The actor which initiates the use case.	
Secondary Actor	Other actors the system relies on the accomplish the services of the use case.	
Pre-condition	What should be true before the use case is executed.	
Dependency	Include and extend relationships to other use cases.	
Generalization	Generalization relationships to other use cases.	
Basic Flow Steps (BFS)	Specifies the main successful path.	
	Step (numbered)	Flow of events

	Postcondition.	What should be true after the basic flow executes.
Specific Alternative Flow (SAF)	Applies to one specific step of the basic flow.	
	RFS	A reference flow step number where flow branches from.
	Step (numbered)	Flow of events.
	Postcondition.	What should be true after the specific alternative flow executes.
Bounded Alternative Flow (BAF)	Applies to more than one step of the basic flow, but not all of them	
	RFS	A list of reference flow steps where flow branches from
	Step (numbered)	Flow of events.
	Postcondition.	What should be true after the bounded alternative flow executes
Global Alternative Flow (GAF)	Applies to all the steps of the basic flow.	
	Step (numbered)	Flow of events.
	Postcondition.	What should be true after the global alternative flow executes
Specific Exceptional Flow (SEF)	Applies to one specific step of the basic, specific alternative or specific variation flow.	
	RFS	A reference flow step number where flow branches from.
	Step (numbered)	Flow of events.
	Postcondition.	What should be true after the specific exceptional flow executes.
Bounded Exceptional Flow (BEF)	Applies to more than one step of the basic, specific alternative or specific variation flow, but not all of them.	
	RFS	A list of reference flow steps where flow branches from.
	Step (numbered)	Flow of events.
	Postcondition.	What should be true after the bounded exceptional flow executes
Global Exceptional Flow (GEF)	Applies to all the steps of the basic flow.	
	Step (numbered)	Flow of events.
	Postcondition.	What should be true after the global exceptional flow executes.
Specific Variation Flow (SVF)	Applies to one specific step of the basic flow. This flow is used to handle the alternative features.	
	RFS	A reference flow step number where flow branches from.
	Step (numbered)	Flow of events.
	Postcondition.	What should be true after the specific variation flow executes.
Bounded Variation Flow (BVF)	Applies to more than one step of the basic flow, but not all of them. This flow is used to handle the alternative features.	
	RFS	A list of reference flow steps where flow branches from.
	Step (numbered)	Flow of events.
	Postcondition.	What should be true after the bounded variation flow executes
Global Variation Flow (GVF)	Applies to all the steps of the basic flow. This flow is used to handle the alternative features..	
	Step (numbered)	Flow of events.
	Postcondition.	What should be true after the global variation flow executes.

The fields *Use Case Id, Use Case Name, Brief Description, Primary Actor, Secondary Actor, Precondition, Dependency*, and *Generalization* are not necessary to explain since they are commonly encountered in many templates. The fields *Use Case Type, Feature*

Cardinality, and *Associated Feature* are used to handle variability in the SPL. *Use Case Type* specifies whether the UC is mandatory, optional, or alternative [18]. *Associated Feature* field specifies the name of the feature(s) associated with the UC [12]. *Feature Cardinality* specifies (for alternative features), whether it can have one or more variation points in the use case. The values for Cardinality field can be [2]: (1..1) only one feature can be selected (OR) and (1..*) one or more feature(s) can be selected (XOR) for alternative features. For optional feature must be (0..1) and for mandatory feature must be (1..1).

A use case descriptions can also contain variation points [18], which identify locations at which variation occurs. In the following is explained the specification of the use case flows. The structure of the template was inspired by [45]. The use case template is defined as a tuple «**Header, Pre, Flows**», in which **Header** was described above, **Pre** is a set of pre-conditions and **Flows** are a set of UC's flows. Flows is also represented by a tuple «**Type, Steps, Post**»: **Type** refers to the type of flow, which can be one of the following:

i) *Basic (or main) flow* represents the "normal flow" of steps of a use case, each UC can only have one basic flow and it is composed of a sequence of steps numbered sequentially;

ii) *Alternative flow* describes conditional steps from the basic flow (designated as reference flow, or RF), after which the use case continues to be executed. Alternative flows describe all the other scenarios (success and failure). These flows also are composed of a sequence of numbered steps. An alternative flow always depends on a condition occurring in a specific step in a flow of reference (basic flow or an alternative flow itself) [51];

ii) *Exceptional flow* describes error conditions (are a response to the error of a step in another flow), this flow can be categorized as recoverable and failure flows. In a recoverable flow the use case can continue to be executed, whereas in a failure flow the use case execution can no longer continue [44];

iv) *Variation flow* which is associated with a variation point. This flow is used to represent variability in SPL.

Non-basic flows can also be categorized as specific (refers to a single step in a RF), bounded (refers to more than a step, consecutive or not of the RF) and global (refers any step in the RF) [51]. The branching condition is specified in the reference flow by restriction rules, which are explained below. **Post** is a set of post-conditions that must be true when each flow terminates. **Steps** represent a set of steps in each flow and can also be represented by a tuple **«SNumber, Cond, Oper»**: **SNumber** is the step number; in each flow, the steps are numbered sequentially (each step is completed before the next one is started). **Cond** is a set of conditions that must hold for the step to be possible; it may be empty. The **IF-THEN-ENDIF** and **VALIDATE THAT** keywords proposed by RUCM are used to express the conditions in specifying steps. **Oper** is an action that describes an interaction between *(i)* the primary or secondary actor send a request to the system, *(ii)* the system with itself validate a request (e.g., modifying its internal state) *(iii)* the system with an actor (e.g., sending a message or result to an actor) [8].

Restriction rules

The RUCM approach proposes a set of restrictions rules on the use of natural language [52] such as actions should be written in the present tense; verbs should be in active voice, modal verbs are not allowed, among others. In the MBPTA, these rules were restricted a bit more, by requiring that actions should have the form "The <actor> does something" or "The system does something". These restrictions are aimed to ease the processing the use case specification by a tool. The restriction rules are classified into two groups: restrictions on the use of natural language (this group is divided into two categories R1-R7 and R8-R16, according to their location of application), and restrictions enforcing the use of specific keywords for specifying control structures [52]. This set of restriction rules were adapted and extended in our work to write use case specifications of SPL because these rules were considered simply and useful to be used.

The first seven restriction rules (R1-R7) are used to restrict the use of natural language aimed to reduce ambiguity (see Table 4.2). These rules are used only to action steps, they do not apply to condition steps, preconditions, and postconditions [52].

Table 4.2: Restriction rules R1-R7

#	**Description**
R1	The subject of a sentence in basic and alternative flows should be the system or an actor
R2	Describe the flow of events sequentially
R3	Actor-to-actor interactions are not allowed.
R4	Describe one action per sentence. (Avoid compound predicates.)
R5	Use present tense only.
R6	Use active voice rather than passive voice.
R7	Clearly describe the interaction between the system and actors without omitting its sender and receiver.

The next nine restrictions rules (R8-R17) are applied to all sentences in a UCSs (action steps, condition steps, preconditions, postconditions, and brief description). The R8-R16 restriction rules were defined by [52], the R17 was added in our work by include "The <actor> does something" sentence to identify the actions of actors. R8-R10 and R16-R17 are used to reduce ambiguity in UCSs, and R9-R15 are used to automatically generate analysis models (See Table 4.3).

In the last eleven restriction rules (see Table 4.4): **INCLUDE USE CASE (R18)** and **EXTENDED BY USE CASE (R19)** keywords are used to represent dependencies of include and extend. **RFS** (Reference flow step) (R20) keyword specifies the step number where flow (specific, alternative, exceptions or variation) branches from. The alternative, Exceptional and Variation flows have a name and a number to differentiate each one of them, and this name is used to go back to the reference flow. The **IF-THEN-ENDIF** keyword (R21) is used to specify simple conditional logic sentences, it must appear in one flow only.

Exceptional flows are triggered when there is a violation of the condition associated with the **VALIDATE THAT** keyword (R22). The **ABORT** keyword (R23), on the

Table 4.3: Restriction rules R8-R17

#	**Description**
R8	Use declarative sentences only. "Is the system idle?" is a non-declarative sentence. Commonly required for written Use Cases.
R9	Use words in a consistent way. Keep one term to describe one thing.
R10	Don't use modal verbs (e.g., might). Modal verbs and adverbs usually indicate uncertainty; Instead, metrics should be used if possible.
R11	Avoid adverbs (e.g., very).
R12	Use simple sentences only. A simple sentence must contain only one subject and one predicate.
R13	Don't use negative adverb and adjective (e.g., hardly, never), but it is allowed to use not or no.
R14	Don't use pronouns (e.g. he, this).
R15	Don't use participle phrases as adverbial modifier.
R16	Use "The system" to refer to the system under design consistently. (The system does something).
R17	Use "The actor" to refer to the actor that interact with the system. (The actor does something).

other hand, indicates that the system execution can no longer continue. **RESUME STEP** keyword (R24) terminates Recoverable flows. This indicates that the exceptional flow goes back to the reference flow, meaning that exception handling succeeded well. **POSTCONDITION** keyword (R25) specifies that each flow (basic, alternative, exceptional and variation) should have its own postcondition. **IGNORE** keyword (R26), is used to represent when the system does not execute any action, the system is waiting for the next user interaction. The curly brackets **"{}"** (R27) are used to represent variability in variation flows. The features are written into these brackets with its different values in each variation flow. Rules R26 and R27 were added in our work to handle variability when specifying a SPL.

Table 4.4: Restriction rules R18-R27

#	**Description**	#	**Description**
R18	INCLUDE USE CASE	R23	ABORT
R19	EXTENDED BY USE CASE	R24	RESUME STEP
R20	RFS	R25	POSTCONDITION
R21	IF-THEN-ENDIF	R26	IGNORE
R22	VALIDATE THAT	R27	{}

An example of a use case description according to the template and restrictions rules just described is presented in Table 4.5.

Table 4.5: Description of use case Finish cooking food

Use Case Id: UC012
Use Case Name: Finish cooking food.
Use Case Type: Mandatory.
Cardinality feature: 1..1
Brief Description: The system finishes cooking the food.
Associate feature: Microwave Oven Kernel, Light, Turntable, Beeper.
Primary Actor: Timer.
Secondary Actor: None.
Pre-condition: The door is closed AND the food is inserted AND the time is not zero AND the start button is pressed.
Dependency: INCLUDE USE CASE Display language.
Generalization: None.
Basic Flow Steps (BFS):
1. The timer sends finished message to the system.
2. The system stops cooking.
3. The system sets the time TO zero.
4. The system sets the start button TO not pressed.
5. **INCLUDE USE CASE** Display language.
6. The system displays end message.
7. The system clears the display.

Postcondition: The door is closed AND the food is inserted AND the time is zero AND the start button is not pressed.
Specific Alternative Flow (SAF):
1.

Postcondition:
Bounded Alternative Flow (BAF):
1.

Postcondition:
Global Alternative Flow (GAF):
1.

Postcondition:
. . .
Specific Variation Flow (SVF-1):
1. the light is selected
2. The system sets the lamp TO off.

Postcondition: The lamp is switched off.
Specific Variation Flow (SVF-2):
1. the turntable is selected
2. The system sets the rotation TO off.

Postcondition: The rotation is off.
Specific Variation Flow (SVF-3):
1. the beeper is selected
2. The system sets beep TO on.

Postcondition: The beep is on.

4.1.4 Traceability matrix

The Traceability matrix (TM) specifies the links among features and use cases. A use case can contain one or more associated features and a feature can be associated with one or more use cases. This matrix is an artifact produced in the Requirements phase, inspired by the FeDRE approach [36], and an example of it is shown in TABLE 4.6. The traceability matrix stores the relationship between features and use cases to facilitate understanding and to keep the feature model, products, and the specification synchronized. This information is used when features for a specific product are selected. The TM shows which use cases are required by these features so that changes in the feature model can be traced to the use cases and vice-versa. The "**X**" in the TM means that a feature is associated with the use case and the "/" indicates that there is no relation between the feature and the use case.

Table 4.6: Traceability Matrix

Features		**Use cases**																
		Mandatory													**Optional**			
Type	**Name**	Open the door	Close the door	Insert the food	Remove the food	Press cooking time	Set cooking time	Start cooking food	Finish cooking food	Cancel	Display language	Set display unit	Set weight sensor	Set heating element	Plus minute	Set time of day	Display time of day	Cook food with recipe
Common	Microwave oven kernel	X	X	X	X	X	X	X	X	X	/	/	/	/	/	/	/	/
Alternative	Display language English	/	/	/	/	/	/	/	/	/	X	/	/	/	/	/	/	/
Alternative	Display language French	/	/	/	/	/	/	/	/	/	X	/	/	/	/	/	/	/
Alternative	Display language Spanish	/	/	/	/	/	/	/	/	/	X	/	/	/	/	/	/	/
Alternative	Display language German	/	/	/	/	/	/	/	/	/	X	/	/	/	/	/	/	/
Alternative	Display language Italian	/	/	/	/	/	/	/	/	/	X	/	/	/	/	/	/	/
Alternative	One-line display unit	/	/	/	/	/	/	/	/	/	/	X	/	/	/	/	/	/
Alternative	Multi-line display unit	/	/	/	/	/	/	/	/	/	/	X	/	/	/	/	/	X
Alternative	Boolean weight sensor	/	/	/	/	/	/	/	/	/	/	/	X	/	/	/	/	/
Alternative	Analog weight sensor	/	/	/	/	/	/	/	/	/	/	/	X	/	/	/	/	X
Alternative	One-level heating element	/	/	/	/	/	/	/	/	/	/	/	/	X	/	/	/	/
Alternative	Multi-level heating element	/	/	/	/	/	/	/	/	/	/	/	/	X	/	/	/	X
Optional	Light	X	X	/	/	/	/	X	X	X	/	/	/	/	/	/	/	/
Optional	Turntable	X	/	/	/	/	/	X	X	X	/	/	/	/	/	/	/	/
Optional	Beeper	/	/	/	/	/	/	/	X	/	/	/	/	/	/	/	/	/
Optional	Power level	/	/	/	/	/	/	/	/	/	/	/	/	X	/	/	/	/
Optional	Minute plus	/	/	/	/	/	/	/	/	/	/	/	/	/	X	/	/	/
Optional	Recipe	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	X
Optional	TOD Clock	/	/	/	/	/	/	/	/	/	/	/	/	/	/	X	X	/
Optional	12 Hour clock	/	/	/	/	/	/	/	/	/	/	/	/	/	/	X	X	/
Optional	24 Hour clock	/	/	/	/	/	/	/	/	/	/	/	/	/	/	X	X	/

4.1.5 Product Map

Product Map relates products to features, indicating which features are used by the foreseen products. A product can contain one or more associated features and a feature can be associated with one or more products. This information is used to facilitate the process of state machine extraction. This matrix is an artifact produced in the Requirements phase, the rows in the matrix show the features and the columns show the products as shown in TABLE 4.7. The "X" in the PM means that a feature is associated with a product and the "/" indicates that there is no relation between the feature and the product.

Table 4.7: Product Map

Type	**Feature**	**Default**	**Prod 1**	**Prod 2**	**Prod 3**	**Prod 4**	**...**
Common	Microwave oven kernel	X	X	X	X	X	...
Alternative	Display language English	X	X	/	/	/	...
Alternative	Display language French	/	/	X	/	/	...
Alternative	Display language Spanish	/	/	/	/	X	...
Alternative	Display language Italian	/	/	/	/	/	...
Alternative	One-line display unit	X	X	/	/	X	...
Alternative	Multi-line display unit	/	/	X	X	/	...
Alternative	Boolean weight sensor	X	X	X	/	/	...
Alternative	Analog weight sensor	/	/	/	X	X	...
Alternative	One-level heating element	X	X	X	/	/	...
Alternative	Multi-level heating element	/	/	/	X	X	...
Optional	Light	/	/	X	/	/	...
Optional	Turntable	/	/	X	X	/	...
Optional	Beeper	/	/	/	X	X	...
Optional	Power level	/	/	/	X	/	...
Optional	Minute plus	/	/	/	/	X	...
Optional	Recipe	/	/	/	/	/	...
Optional	TOD Clock	/	/	/	/	/	...
Optional	12 Hour clock	/	/	/	/	/	...
Optional	24 Hour clock	/	/	/	/	/	...

4.2 Application Engineering: Testing Models

This phase considers the development of products from the reusable artifacts created in the Domain Engineering. In MBPTA, this phase considers the creation of artifacts for test case generation. A static model, consisting of a UML class diagram is created, which provides the context for the dynamic model, that is, the state machine model which is used as a test model. These artifacts are explained in the following.

4.2.1 Conceptual model

The conceptual model represents a static, blackbox view of the problem domain. It shows the relationship between the system and the environment elements in terms of classes and their relationships. For a state machine its context is represented for a class. Transitions (events, guards, and effects) of a state machine can refer to properties and operations of its context class [48]. Although there are tools that derive class models from use case descriptions, e.g. aToucan [52], this step is manually accomplished by tester in MBPTA for the moment.

4.2.2 Behavior model

State machines are commonly used for model-based test case generation. The construction of such state machines is manual, expensive, and error-prone. Besides, automatic extraction of state machines from use cases to UML would help reduce time, costs and errors in software development process.

State machine generation has always been a challenge, especially for large-scale systems. In the context of SPL development, this modeling is more complex by the fact that variability should be considered. The point is: how to represent variant states, transitions, events and actions? There are two main approaches to consider in this case [19]: specialization and parameterization. In the former, inheritance mechanism is used, but this implies the representation of one state machine for each alternative or optional feature, or for each feature combination, which can lead to a combinatorial explosion. In parameterized state machines, on the other hand, there are feature dependent states, events, and transitions guarded by feature conditions. Although this approach avoids the combinatorial explosion problem, it requires the introduction of new transitions and states, which can make the model very complex.

In order to avoid the problem of variability representation, our work proposes to model a product-specific behavior. In this case, there is no risk of combinatorial explosion, as the model only contains the features already selected for the product under test. The extracted model is an extended finite state machine, which can be defined as:

$$M = < S,\ initial,\ start,\ I,\ O,\ V,\ T >$$

Where

- **S** is the finite set of states;
- **Initial** represents the initial pseudostate of a UML state machine;
- **Start** indicates the initial state;
- **I** is the finite set of input events (request from an actor);
- **O** represents a finite set of output actions or the outputs generated from the system to an actor;
- **V** represents a finite set of context variables used in state invariants or in guards;
- **T** represents a finite set of transitions.

MBPTA comprises a state model extraction step, aimed at helping test analysts to obtain a first draft of the behavior model. These steps are explained in the next Section.

4.2.3 State machine Extraction

UML requires that every state diagram has an initial transition. In MBPTA this transition connects the initial pseudo-state to the **start** state, and its trigger is named as **'construct'**. Except for the initial pseudo-state, all states are represented by the pre-

and postconditions of the use cases (UC). A final transition is added as a link to the final state. The other transitions represent possible flows among the use cases.

Since states represent pre- and post- conditions, and there are as many post-conditions as flows in a use case, there are as many transitions as flows for a given a use case (UC). In other words, each transition corresponds to a (basic or other) flow in the UCTS, with the use case name as the trigger. The guards differentiate the transitions from the origin to a destination state for a given UC. Guards correspond to the condition step that precedes each non-basic flow.

The set of rules necessary for the state machine extraction are presented in TABLE 4.8. These establish the sequence of words that characterize states, transitions, triggers and so on. Some of these rules were extended from [53]; two more rules were necessary to handle the exceptional flows and the variability in SPLs.

Table 4.8: Transformation rules

#	**Description**
1	Generate an instance of *StateMachine* for the use case model.
2	Generate the **initial** state of the state machine.
3	Generate an instance of State, named as **start**, representing the start state of the state machine.
4	Generate an instance of **Transition**. Its trigger is named as **construct**. This transition connects the initial state to the start state.
5	Invoke rules 5.1-5.6 to process each use case of the use case model.
5.1	Generate an instance of State for each sentence of the precondition of the use case, as long as such a state has not been generated, which is possible because two use cases might have the same preconditions.
5.2	Generate an instance of State for each sentence of the postcondition of the basic flow of the use case, as long as such a state has not been generated.
5.3	Connect the states corresponding to the precondition to the states representing the postcondition of the basic flow of the use case with the transition whose trigger is the name of use case. The guards are the conjunction of all the conditions of the condition sentences in the basic flow. The effects are the system responses in the basic flow.
5.4	Process the postcondition of each alternative flow of the use case.
5.4.1	Generate an instance of State for each sentence of the postcondition of each alternative flow.
5.4.2	Connect the states corresponding to the precondition of the basic flow to the states corresponding to the postconditions of the alternative flows with the transition whose trigger is the name of use case.
5.5	Process the postcondition of each exceptional flow of the use case.
5.5.1	Generate an instance of State for each sentence of the postcondition of each exceptional flow.

5.5.2	Connect the states corresponding to the precondition of the basic, alternative or variation flow where the exception occurs to the state corresponding to the postcondition of the exceptional flow with a transition. The trigger is the name of use case, the effects are the responses of the system. For this rule is used the **VALIDATE THAT** keyword, to determine if an exception is identified, then the guard is the negative condition of **VALIDATE THAT** keyword. For example, if the condition is "The system **VALIDATE THAT** the door is closed", the guard is "the door is not closed".
5.6	Process the postcondition of each variation flow of the use case.
5.6.1	Combinate the postcondition of the variation flow with of the postcondition of the flow (basic, alternative or exceptional) where the variation flow is referred. For this rule are used the curly brackets **"{}"**, to determine if a feature of the PF is selected.
5.6.2	Connect the state corresponding to the precondition of the flow (basic, alternative or exceptional) where the variation flow is referred to the postcondition corresponding to the combination of the postconditions (variation and basic, alternative or exceptional flow). The transition whose trigger is the name of use case. The effects are the combination of system responses of the variation flow and the flow where the variation flow is referred.
6	**INCLUDE USE CASE** sentence is used to indicate that the behavior of an included UC is inserted into the behavior of the including use case. The source state is the precondition of including use case and this precondition must be present in the included use case precondition. The target state is the combination of including use case postcondition with the included use case postcondition. Source and Target states are connected through a transition where the trigger is the name of including use case. The guards are the conjunction of all the conditions of the condition sentences of the previous steps containing the name of the included use case. The effects are the combination of included use case responses with the including use cases responses.
7	**EXTEND USE CASE** sentence is used to indicate that an use case adds an optional behavior to the extended use case. The source state is the precondition of extending use case and this precondition must be present in the extended use case precondition. The target state is the combination of extending use case postcondition with the extended included use case postcondition. Source and Target states are connected through a transition where the trigger is the name of extending use case. The guards are the conjunction of all the conditions of the condition sentences of the previous steps containing the name of the extended use case. The effects are the combination of extended use case responses with the extending use cases responses.
8	**ABORT** sentence is used to indicate that the system execution can no longer continue. The state system is leading to a final state - as it is not possible to know what is the final state, an abstract final state is added so that it can represent the interruption of the machine.This keyword must be used in exceptional and alternative flows only.
9	**RESUME STEP** sentence is used to terminate Recoverable flows. This indicates that the exceptional or alternative flow goes back to the reference flow, meaning that exception handling succeeded well. The system goes back to the previous state.

10	**IGNORE** sentence is used to indicate that the system does not execute any action when this keyword is present. The system remains in the same state, the pre- and post- condition are the same (source state = target state).

In order to ease the automatic extraction of state machine elements presented above, users should adopt some conventions in use case writing.

- Convention number one is that a use case specification contains scenarios that represent particular paths through a use case from the point of view of a primary actor. **IF** commands are only allowed at the beginning of alternative or exceptional scenarios.
- Convention two states that actions performed by the system must be explicitly stated as *"The system does something"* so that the output actions of a transition are identified.
- Convention three requires that the user should identify the initial state, by indicating the precondition that characterizes this state. In this way, it is possible to create a transition from the start state to the indicated state.
- Convention four is relative to the use of curly brackets "{}" to represent the variability in variation flows, so that features are written into these brackets with their different values in each variation flow. The curly brackets in alternative or mandatory use cases are only allowed in variation flows, only in optional use cases, these curly brackets are allowed in the basic flow.

Figure 4.4 presents an excerpt of a state machine extracted manually from the use case **Use Case Finish cooking food** using the extraction rules.

4.2.4 State model refinement

After obtaining of the first state machine draft, the test engineer refines this state machine, which can be used as an input to automatically generate test cases. The refining process is important because the extraction method has no rules to make a semantic analysis of the pre and postconditions yet. For this reason, redundant states can be generated in the extraction process. For the refinement, the following steps can be followed:

- Add missing transitions, e.g., it is necessary to create a transition between the start state and the initial state of the state model; .
- remove extra transitions and states, e.g., it is necessary to remove duplicate states;
- modify transitions and states, e.g. to modify names of the state;
- create the context class, it must contain all attributes and operations that are mapped to variables and operations (effects) of the state machine;
- map all the triggers and operations in the state machine diagram to the attributes and operations of the context class associated with the state model;
- replace textual guards conditions with corresponding OCL constraints based on input parameters of the triggers associated with the guards, e.g., *The door is closed and The food is inserted* may be mapped for *doorIsOpen=false and foodIsInserted=true.*

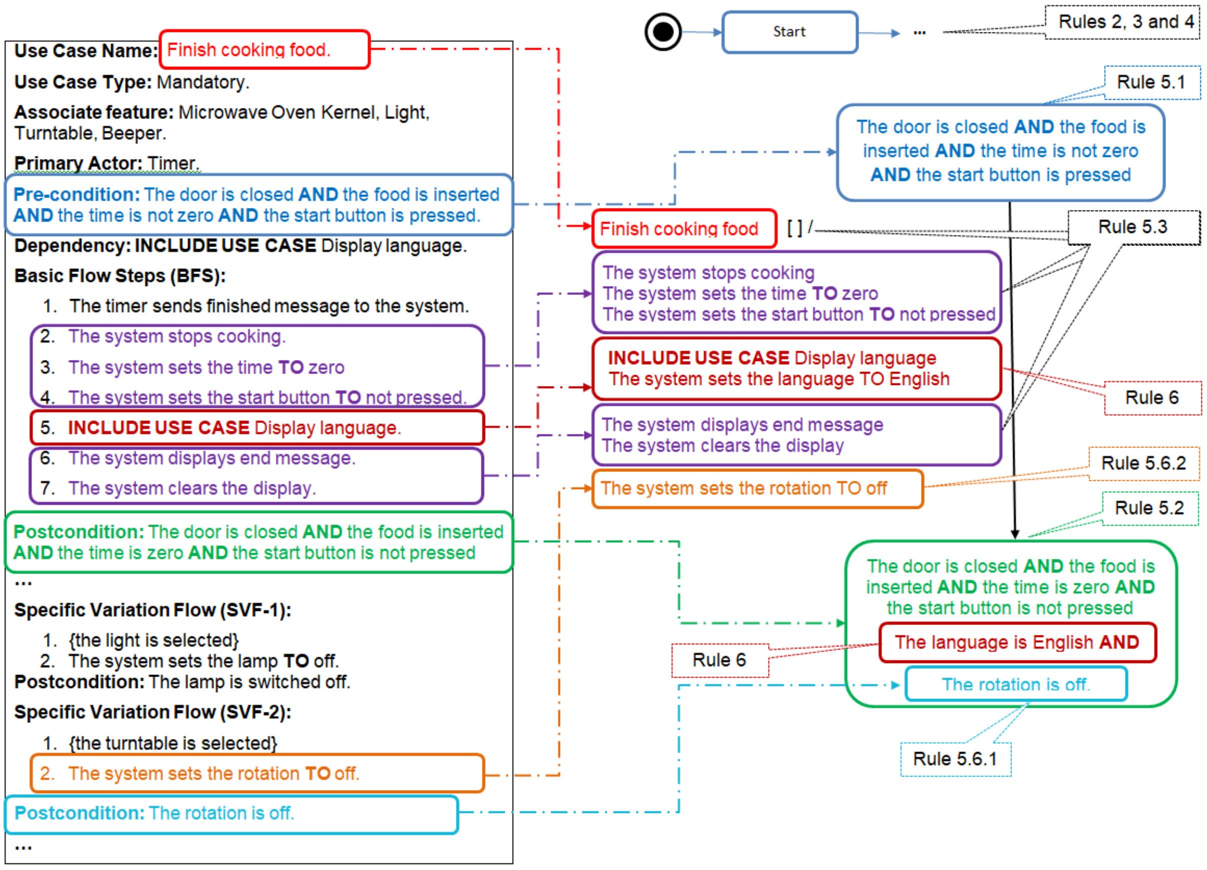

Use Case Name: Finish cooking food.
Use Case Type: Mandatory.
Associate feature: Microwave Oven Kernel, Light, Turntable, Beeper.
Primary Actor: Timer.
Pre-condition: The door is closed AND the food is inserted AND the time is not zero AND the start button is pressed.
Dependency: INCLUDE USE CASE Display language.
Basic Flow Steps (BFS):
1. The timer sends finished message to the system.
2. The system stops cooking.
3. The system sets the time TO zero
4. The system sets the start button TO not pressed.
5. INCLUDE USE CASE Display language.
6. The system displays end message.
7. The system clears the display.
Postcondition: The door is closed AND the food is inserted AND the time is zero AND the start button is not pressed
...
Specific Variation Flow (SVF-1):
1. {the light is selected}
2. The system sets the lamp TO off.
Postcondition: The lamp is switched off.
Specific Variation Flow (SVF-2):
1. {the turntable is selected}
2. The system sets the rotation TO off.
Postcondition: The rotation is off.
...
Start
...
Rules 2, 3 and 4
Rule 5.1
The door is closed AND the food is inserted AND the time is not zero AND the start button is pressed
Finish cooking food
[ ] /
Rule 5.3
The system stops cooking
The system sets the time TO zero
The system sets the start button TO not pressed
INCLUDE USE CASE Display language
The system sets the language TO English
Rule 6
The system displays end message
The system clears the display
Rule 5.6.2
The system sets the rotation TO off
Rule 5.2
The door is closed AND the food is inserted AND the time is zero AND the start button is not pressed
The language is English AND
The rotation is off.
Rule 6
Rule 5.6.1

Figure 4.4: Excerpt of the state machine for use case Finish cooking food

Figure 4.5 presents a context class created for Product 2.

Operations and properties of the context class are mapped to effects and parameters of the state machine, respectively. To illustrate this process Figure 4.6 shows an excerpt of the Initial state machine for Product 2.

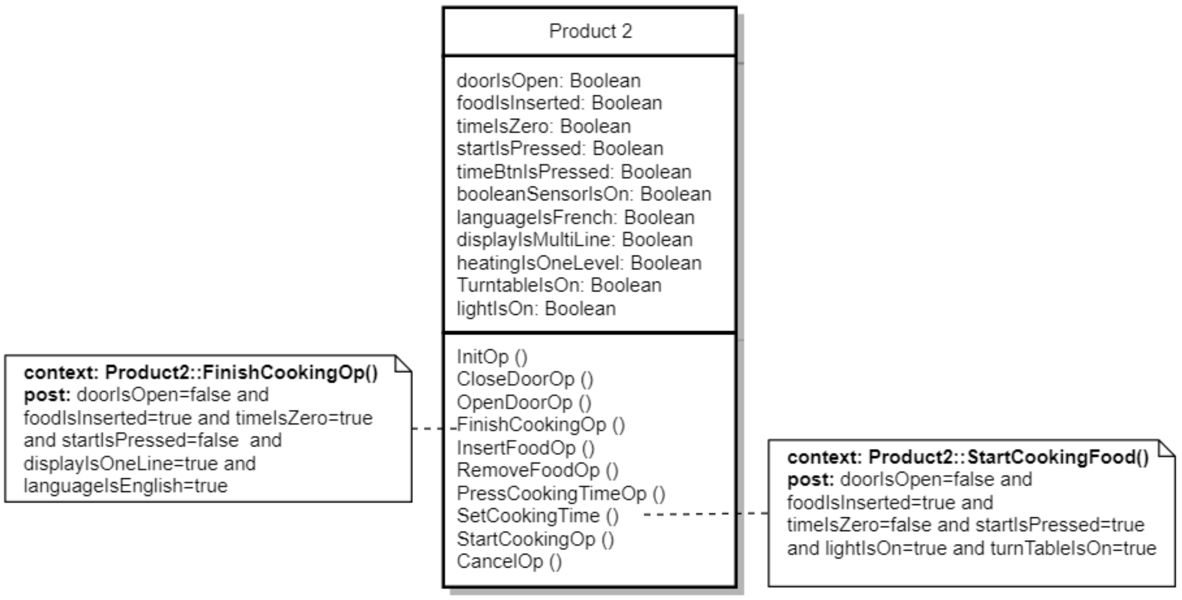

Product 2
doorIsOpen: Boolean
foodIsInserted: Boolean
timeIsZero: Boolean
startIsPressed: Boolean
timeBtnIsPressed: Boolean
booleanSensorIsOn: Boolean
languageIsFrench: Boolean
displayIsMultiLine: Boolean
heatingIsOneLevel: Boolean
TurntableIsOn: Boolean
lightIsOn: Boolean
InitOp ()
CloseDoorOp ()
OpenDoorOp ()
FinishCookingOp ()
InsertFoodOp ()
RemoveFoodOp ()
PressCookingTimeOp ()
SetCookingTime ()
StartCookingOp ()
CancelOp ()
context: Product2::FinishCookingOp()
post: doorIsOpen=false and foodIsInserted=true and timeIsZero=true and startIsPressed=false and displayIsOneLine=true and languageIsEnglish=true
context: Product2::StartCookingFood()
post: doorIsOpen=false and foodIsInserted=true and timeIsZero=false and startIsPressed=true and lightIsOn=true and turnTableIsOn=true

Figure 4.5: Context class for Product 2

The states with dashed lines were considered the same during the refinement since the difference is about the postcondition when optional features were selected (bold letters). These states can be merged as is shown in Figure 4.7 (this figure still does not show the mapping of guards, effects, events and state names to OCL constraints).

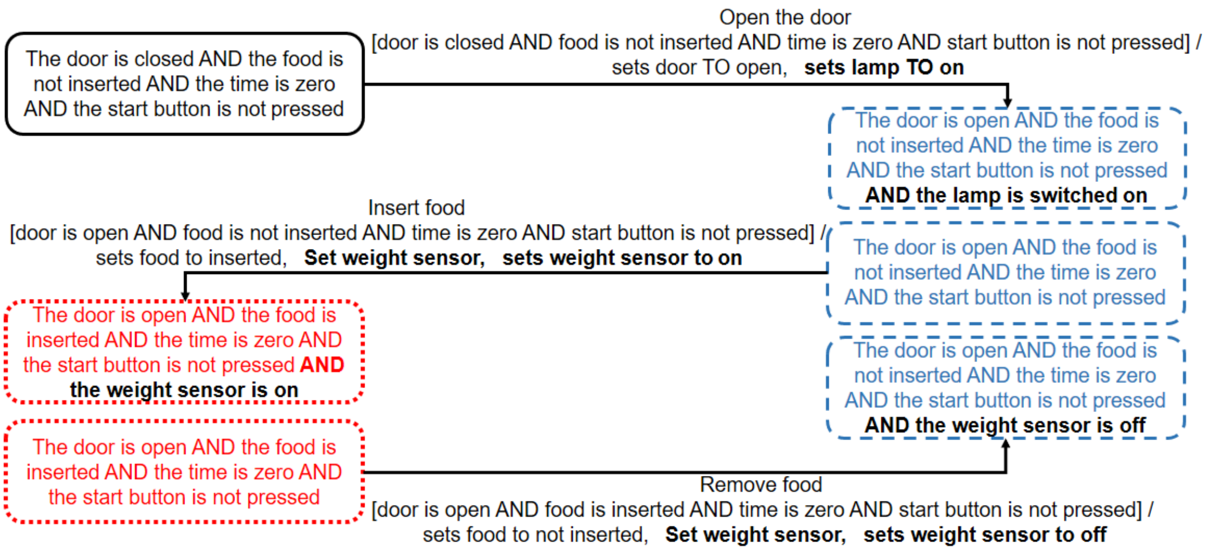

Open the door
[door is closed AND food is not inserted AND time is zero AND start button is not pressed] /
sets door TO open, **sets lamp TO on**
The door is closed AND the food is not inserted AND the time is zero AND the start button is not pressed
The door is open AND the food is not inserted AND the time is zero AND the start button is not pressed **AND the lamp is switched on**
Insert food
[door is open AND food is not inserted AND time is zero AND start button is not pressed] /
sets food to inserted, **Set weight sensor, sets weight sensor to on**
The door is open AND the food is not inserted AND the time is zero AND the start button is not pressed
The door is open AND the food is inserted AND the time is zero AND the start button is not pressed **AND the weight sensor is on**
The door is open AND the food is not inserted AND the time is zero AND the start button is not pressed **AND the weight sensor is off**
The door is open AND the food is inserted AND the time is zero AND the start button is not pressed
Remove food
[door is open AND food is inserted AND time is zero AND start button is not pressed] /
sets food to not inserted, **Set weight sensor, sets weight sensor to off**

Figure 4.6: Excerpt of Initial state machine

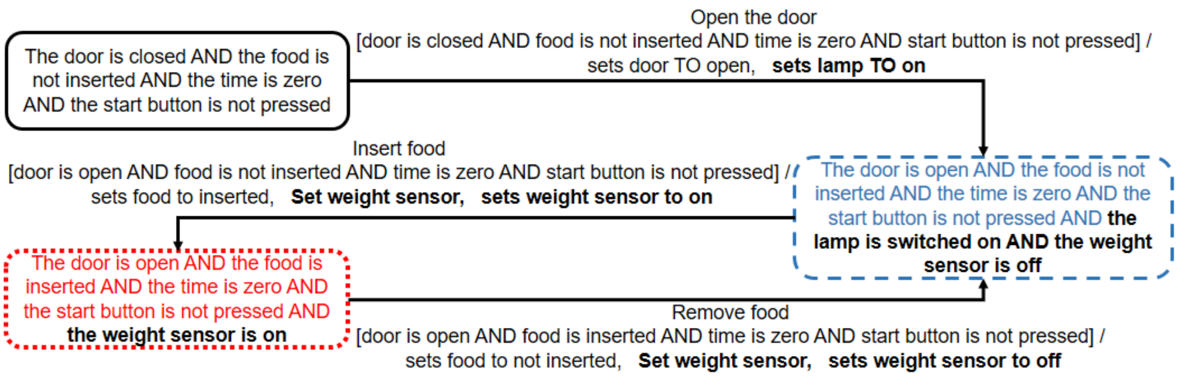

Open the door
[door is closed AND food is not inserted AND time is zero AND start button is not pressed] /
sets door TO open, **sets lamp TO on**
The door is closed AND the food is not inserted AND the time is zero AND the start button is not pressed
Insert food
[door is open AND food is not inserted AND time is zero AND start button is not pressed] /
sets food to inserted, **Set weight sensor, sets weight sensor to on**
The door is open AND the food is not inserted AND the time is zero AND the start button is not pressed AND **the lamp is switched on AND the weight sensor is off**
The door is open AND the food is inserted AND the time is zero AND the start button is not pressed AND **the weight sensor is on**
Remove food
[door is open AND food is inserted AND time is zero AND start button is not pressed] /
sets food to not inserted, **Set weight sensor, sets weight sensor to off**

Figure 4.7: Merging redundant states in refinement process

Duplicate states were removed, states and events were renamed, e.g., *Open the door* for `OpenDoorEv` event. Operations and attributes of the context class were mapped in variables and effects of state model. For example, the sentence *door is open* was mapped to the context class attribute `doorIsOpen`, the effect *sets food to inserted* was mapped to the context class operation `FoodInsertedOp` with the postcondition `foodIsInserted=true`.

The guard condition *[door is open AND food is inserted AND time is zero AND start button is no pressed]* was mapped for `[doorIsOpen=true and foodIsInserted=false and timeIsZero=true and startIsPressed=false and booleanSensorIsOn=false]`. A refined state model is shown in Figure 4.8.

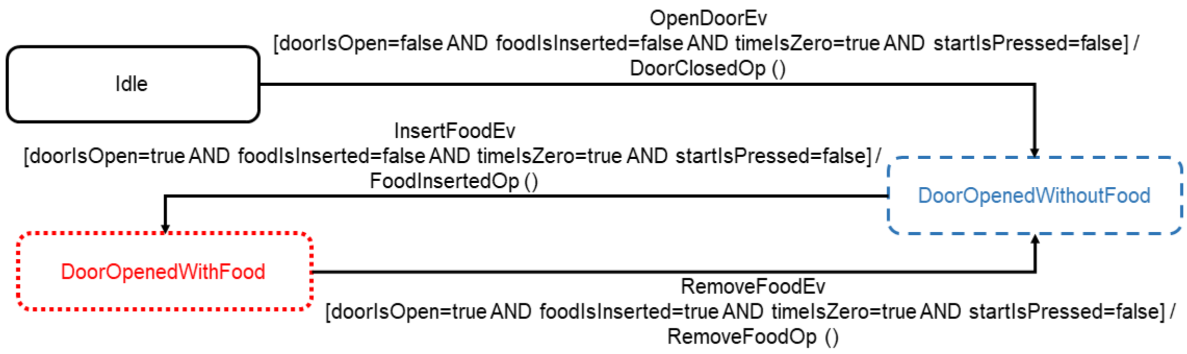

Idle
OpenDoorEv
[doorIsOpen=false AND foodIsInserted=false AND timeIsZero=true AND startIsPressed=false] /
DoorClosedOp ()
InsertFoodEv
[doorIsOpen=true AND foodIsInserted=false AND timeIsZero=true AND startIsPressed=false] /
FoodInsertedOp ()
DoorOpenedWithoutFood
DoorOpenedWithFood
RemoveFoodEv
[doorIsOpen=true AND foodIsInserted=true AND timeIsZero=true AND startIsPressed=false] /
RemoveFoodOp ()

Figure 4.8: Excerpt refined state machine

4.3 Chapter Summary

This chapter introduced MBPTA, a model based product testing approach for SPL, that is aimed to guide test engineers during system testing of specific products. MBPTA supports test engineers to solve two problems: i) how to cope with variability for the derivation of product-specific test cases? and ii) how to create test model for a specific product?

To cope with problem i), variability modelled during Domain Engineering phase guides test model creation during Application Engineering phase. Variability is represented in the use case models. The other domain artifact also helps to determine which features are used in a product under test. In this way, the same artefacts used to develop the product are also useful for test case generation. Another benefit of the approach is that traceability between development and test activities can be achieved.

MBPTA is use-case centered, that is, the use case model is the main reusable artefact in what concerns test activities. Use case descriptions are written in a structured natural language for reducing ambiguity and facilitate automated analysis. Non-UML artifacts such as Product Map and Traceability Matrix are used in this approach to guide test model derivation.

To cope with problem ii), a set of transformation rules was provided to automatically transform textual use cases into state machine models. This approach provides a solution that is capable of dealing with the complexity involved in SPLs to extract test models with a large number of requirements.

Using MBPTA, test engineers can not only determine whether a given product satisfies its functional requirements but also if it does not omit to implement a required feature. For the approach to be useful in practice, to have a tool to support the automatic derivation of the test model from the Domain Engineering artefacts was developed. The tool is presented in next Chapter.

Chapter 5

The prototype tool

The model-based product testing approach presented in the previous Chapter guides the creation of a state machine as a test model in the Application Engineering phase. The creation of state-machine is not straightforward and requires a certain training, as most practitioners are more used with textual specifications, like use case descriptions. Therefore, a tool was developed to support test engineers in the construction of the state machine representing the behavior of the product under test. To show the feasibility of this support, we built MARITACA (state MAchine geneRatIon from TextuAl use CAses) tool, which uses the use case descriptions, Traceability Matrix, Product Map, and Extraction rules to support the test engineers in the extraction of an initial state machine. Section 5.1 describes the extraction and generation process of state machines from use cases descriptions. Section 5.2 presents the MARITACA interface.

5.1 The extraction process

MARITACA tool was implemented as a proof of concept of the proposed approach described in Chapter 4. This tool applies techniques from Natural Language Processing (NLP), more specifically, a Part-of-Speech (PoS) Tagger [28]. By analyzing the tagged text, the tool uses the set of rules presented in TABLE 4.8 to categorize the sequence of words that characterize states, transitions, triggers and so on. Figure 5.1 below shows a schematic view of the steps necessary for the extraction of a state machine model from the artifacts created in the Domain Engineering phase.

Use cases descriptions must be written using the template and the restriction rules by Requirements engineer as explained in Section 4.1.3. The use case description must be in .txt extension to be processed by Part-of-Speech Tagger. The PoS tagger processes the words in the text and attaches to each one a tag according to its grammatical category producing as output a text like shown in TABLE 5.1. Some used categories are NN (Noun, singular), NNP (Proper noun, singular), JJ (Adjectives), VB (Verb, base form), VBG (Verb, gerund), VBZ (Verb, 3rd person singular), DT (Determiner), VBN (Verb, past participle), RB (Adverb), CC (Conjunction), TO (to), RP (Participle), CD (Cardinal number) and so on.

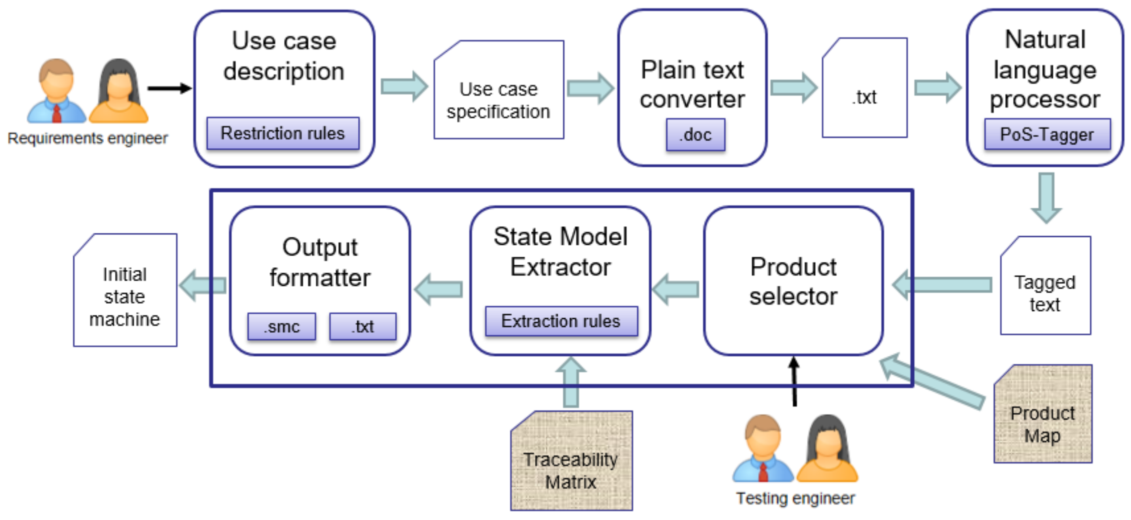


Figure 5.1: A schematic view of the extraction process

Table 5.1: UCD tagged of Finish cooking food

```
Use_NN Case_NN Id_NN UC012_NNP ._.
Use_NNP Case_NNP Name_NN Finish_VB cooking_JJ food_NN ._.
. . .
Precondition_NN :_:  The_DT door_NN is_VBZ closed_VBN AND_CC the_DT
food_NN is_VBZ inserted_VBN AND_CC the_DT time_NN is_VBZ not_RB zero_CD
AND_CC the_DT start_NN button_NN is_VBZ pressed_VBN ._.
Basic_JJ Flow_NN Steps_VBZ LRB_LRB BFS_NNP RRB_RRB :_:
1_CD ._.
The_DT timer_NN sends_VBZ finished_JJ message_NN to_TO the_DT system_NN
._.
2_CD ._.
The_DT system_NN stops_VBZ cooking_NN ._.
3_CD ._.
The_DT system_NN sets_VBZ the_DT time_NN TO_TO zero_CD ._.
4_CD ._.
The_DT system_NN sets_VBZ the_DT start_NN button_NN TO_TO not_RB
pressed_VBN ._.
5_CD ._.
INCLUDE_NNP USE_NNP CASE_NN Display_NN language_NN ._.
6_CD ._.
The_DT system_NN displays_VBZ end_NN message_NN ._.
7_CD ._.
The_DT system_NN clears_VBZ the_DT display_NN ._.
Postcondition_NNP :_:  The_DT door_NN is_VBZ closed_VBN AND_CC the_DT
food_NN is_VBZ inserted_VBN AND_CC the_DT time_NN is_VBZ zero_CD AND_CC
the_DT start_NN button_NN is_VBZ not_RB pressed_VBN ._.
.  .  .
```

The Traceability Matrix (TM) and the Product Map (PM) must be written in .csv format. When the input documents are ready, they must be loaded in the MARITACA tool, which has incorporated the transformation rules to generate state models.

First, the testing engineer loads the PM, the TM, and the tagged use case descriptions. Then, MARITACA shows the available products with its features (from the PM). Second, the testing engineer selects a product. Finally, the user selects the option to extract the state machine, then MARITACA starts processing the documents, as follow:

i) **Information extraction:** all use cases are processed and their extracted information is stored, which establishes the relationship between the information and its roles in the state machine. This information is obtained by using the transformation rules presented in TABLE 4.8, some of these rules are explained in following:

   – Name of use cases: are identified the triggers and they represent the events of the state machine.

   – Use case type: allows to identify when a use case is always used in all state machines (for all products) and when a use case is used for some state machines (for some products).

   – Basic flow: there is a sequence such as *The_DT <any_noun>_NN <any_verb>_ VBZ <any_subsentence>* or *The_DT <any_noun>_NNP <any_verb>_ VBZ <any_subsentence>*. It is transformed in effects of the state machine. These events are denoted by verb subsentence. Only verbs with the _VBZ tag are used to identify effects. When exist a sequence *INCLUDE_NNP USE_NNP CASE_NN* or *EXTEND_NNP USE_NNP CASE_NN*, the included/extended use case name is stored as an effect of the state machine (rules 6 and 7).

   – Alternative flow: when a sequence with *IF_IN* and *THEN_NNP* tags exist, the information between these two tags must be present in the use case precondition and this information is identified as a guard. Similar to the basic flow the rules to identify effects, include and extend are applied in alternative flows.

   – Exceptional flow: when the sequence with *VALIDATE_NNP THAT_WDT* tag exists, two guards are created. First, the positive conditions of VALIDATE THAT keyword and second with the negative condition of VALIDATE THAT keyword. Then, possible actions *IGNORE_ VB* (representing a self-loop), *ABORT_ VB* (representing the interruption of the state machine) and *RESUME_ VB* (representing to go back to the reference flow) are identified (see rules 8, 9 and 10).

   – Variation flow: when existing "LCB_LRB" and "RCB_RRB" tags, they correspond to the opening and closing of keys of the optional features ({<feature>}), precondition, effects, guards, and postcondition are identified and stored.

ii) **Selection of the use cases:** the use cases are selected using the Product Map and the Traceability Matrix for the chosen product. As defined in Section 4.1, the Product Map relates a specific product to the features it contains, whereas the Traceability Matrix relates each feature to the Use Cases.

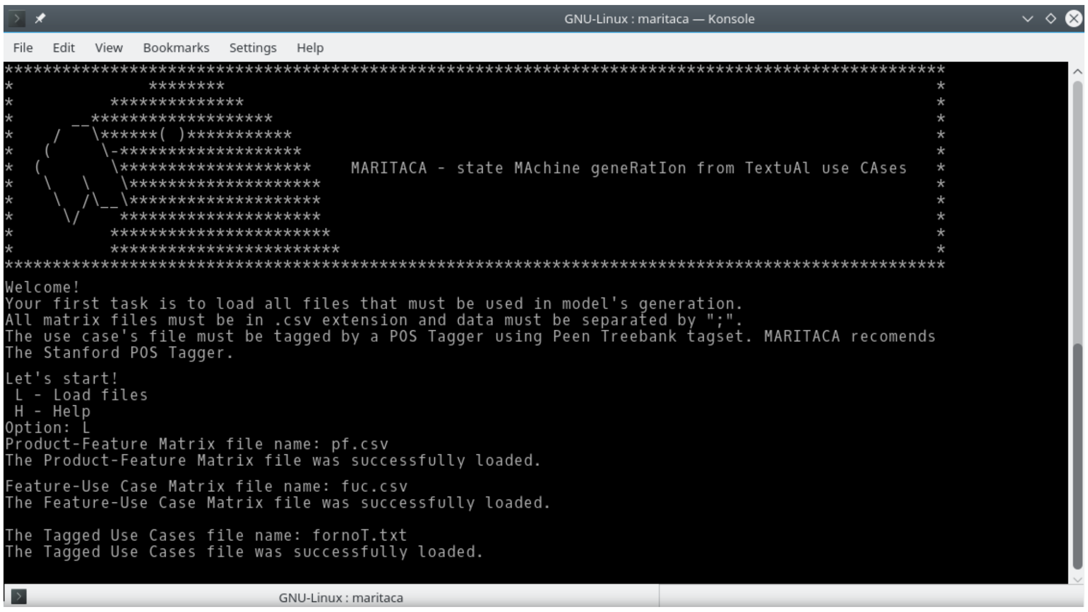
iii) **Selection of information from selected use cases and state machine composition:** the information of each use case selected is retrieved from the data structure created during the extraction step and it is copied to other data structure that stores exclusive information of the state machine that is being created.

iv) **State machine output:** the extracted state machine is available in two formats: pure textual (.txt file) or in a format amenable to be processed by another tool for test case generation.

5.2 Tool interface

MARITACA tool was developed using the C++ programming language. This tool extracts information from the Product Map (to know which products can be selected and its features), the Traceability Matrix (to know which use cases are required for a chosen product) and the use case descriptions tagged.

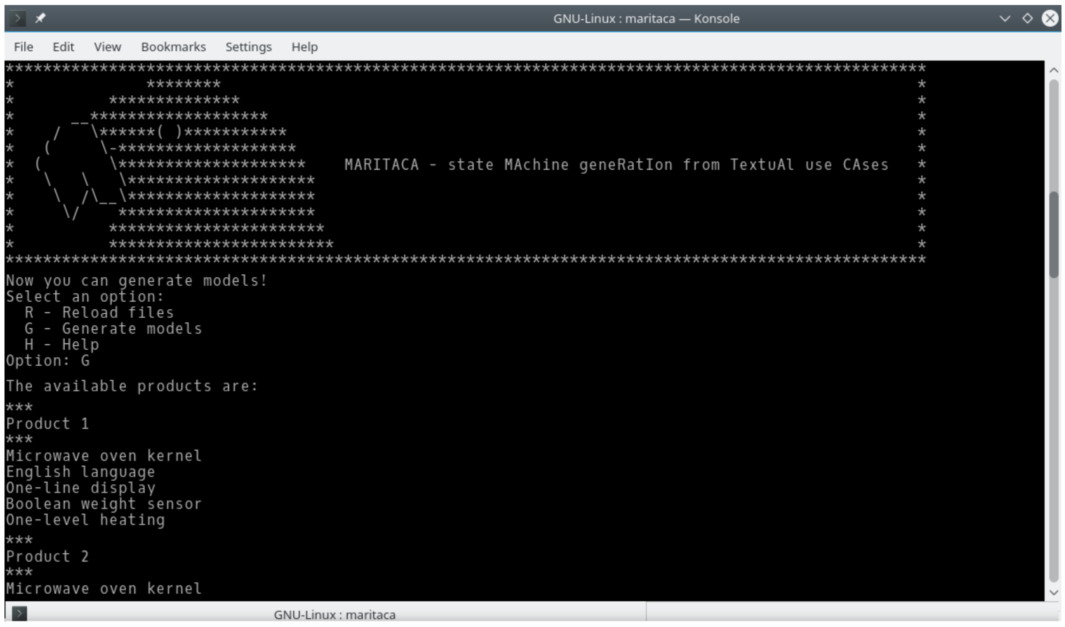
Since this is a prototype tool, developed as a proof of concept, the interaction with it is through a simple, command line interface. First, the user is presented with the initial interface shown in Figure 5.2, which asks the user to provide the files containing the Traceability Matrix, Product Map, and tagged use case description. The user can also see a Help, providing information about MARITACA.


GNU-Linux : maritaca — Konsole
File Edit View Bookmarks Settings Help
GNU-Linux : maritaca

```
************************************************************************************************
*              ********                                                                        *
*          **************                                                                      *
*       __*******************                                                                  *
*    /  ‾‾\******( )***********                                                                *
*   (      \-*******************                                                               *
*  (        \********************    MARITACA - state MAchine geneRatIon from TextuAl use CAses   *
*   \   \    \********************                                                             *
*    \  /\__\********************                                                              *
*     \/     ********************                                                              *
*          ***********************                                                             *
*          ************************                                                            *
************************************************************************************************
Welcome!
Your first task is to load all files that must be used in model's generation.
All matrix files must be in .csv extension and data must be separated by ";".
The use case's file must be tagged by a POS Tagger using Peen Treebank tagset. MARITACA recomends
The Stanford POS Tagger.

Let's start!
 L - Load files
 H - Help
Option: L
Product-Feature Matrix file name: pf.csv
The Product-Feature Matrix file was successfully loaded.

Feature-Use Case Matrix file name: fuc.csv
The Feature-Use Case Matrix file was successfully loaded.

The Tagged Use Cases file name: fornoT.txt
The Tagged Use Cases file was successfully loaded.
```

Figure 5.2: Initial interface of MARITACA tool

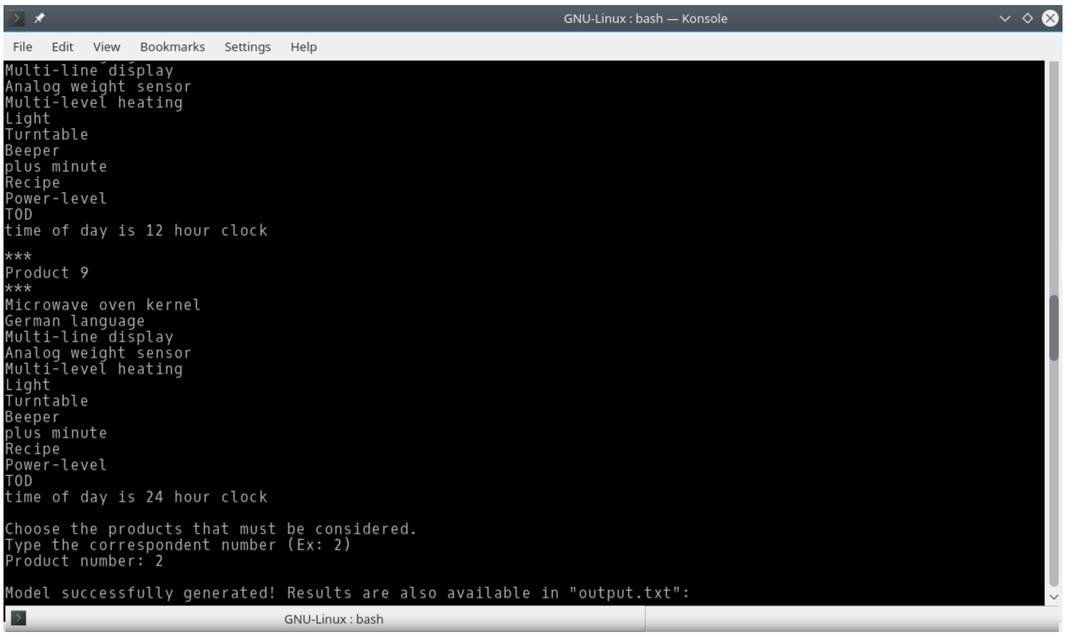
Once the files are loaded, three options are presented: *i)* Reload files: used to load input files once more; *ii)* Generate models: used to generate the state machines (available products and its features are presented); and *iii)* Help: used to provide information about MARITACA. Figure 5.3 shows the available products for the Microwave oven product line.



```
***************************************************************************************************
*              ********                                                                           *
*          **************                                                                         *
*      __*******************                                                                      *
*    /   \******( )***********                                                                    *
*   (     \-*******************                                                                   *
*  (       \********************    MARITACA - state MAchine geneRatIon from TextuAl use CAses    *
*   \   \   \********************                                                                 *
*    \  /\__\********************                                                                 *
*     \/    *********************                                                                 *
*          ***********************                                                                *
*          ************************                                                               *
***************************************************************************************************
Now you can generate models!
Select an option:
  R - Reload files
  G - Generate models
  H - Help
Option: G

The available products are:
***
Product 1
***
Microwave oven kernel
English language
One-line display
Boolean weight sensor
One-level heating
***
Product 2
***
Microwave oven kernel
```

Figure 5.3: Interface to select Generate models option

When the product is selected, the state machine is extracted and saved in the folder where MARITACA is stored. Figure 5.4 shows the final interface to state machine process.



```
Multi-line display
Analog weight sensor
Multi-level heating
Light
Turntable
Beeper
plus minute
Recipe
Power-level
TOD
time of day is 12 hour clock

***
Product 9
***
Microwave oven kernel
German language
Multi-line display
Analog weight sensor
Multi-level heating
Light
Turntable
Beeper
plus minute
Recipe
Power-level
TOD
time of day is 24 hour clock

Choose the products that must be considered.
Type the correspondent number (Ex: 2)
Product number: 2

Model successfully generated! Results are also available in "output.txt":
```

Figure 5.4: Interface to select a product

Finally, when the state machine is extracted and saved in .txt extension, this can be seen as shown TABLE 5.2.

Table 5.2: Excerpt of a pure textual description of a generated state machine

```
Pre:  The door is open AND the food is inserted AND the time is not zero
AND the start button is not pressed
Trigger:  Close the door
Guard:  the door is open AND the food is inserted AND the time is not
zero AND the start button is not pressed
Effect:  sets the door TO closed
Post:  The door is closed AND the food is inserted AND the time is not
zero AND the start button is not pressed
Pre:  The door is closed AND the food is inserted AND the time is not
zero AND the start button is not pressed
Trigger:  Start cooking food
Guard:  The door is closed AND the food is inserted AND the time is not
zero AND the start button is not pressed
Effect:  Set heating element, sets heating TO one-level, sets the start
button TO pressed, starts cooking, Set display unit, sets the display TO
one-line, displays the time
Post:  The door is closed AND the food is inserted AND the time is not
zero AND the start button is pressed AND The heating is one-level AND The
one-line is on
```

5.3 Chapter Summary

This chapter presented the MARITACA tool, developed to support the partitioners in the state machine extraction process. This tool answered one of our research questions: do the rules and conventions defined in Chapter 4 allow the generation of a state machine that represent the scenarios presented in the use case descriptions? Although there are a number of similar works [54] [45] [14], the main difference of our work is that it allows to generate state machines for all products of a SPL, and not for single products

This first version of MARITACA tool generates semi-automatically initial textual state machines from use cases descriptions. Besides the use case descriptions, the tool also accepts as inputs the product map and the traceability matrix. The latter allows the user to select a product for which to generate the model with the required features. Using natural language processing techniques, MARITACA extracts the state model for the selected product. It can produce not only a pure textual description of the model but also a format that is compatible with one tool used in previous works of the group. Naturally, the extracted state machine is partially complete, and not amenable to be processed by a tool, requiring further refinements.

Chapter 6 presents the use of the approach and the MARITACA tool in a family of CHDP-based fault-tolerant applications.

Chapter 6

Application of the approach

This Chapter presents the application of the approach to the Connection Handler Design Pattern (CHDP), a reusable design solution for the development of distributed applications tolerant to connection crashes [23]. Although CHDP was not developed as an SPL, it is based on commonalities present on different distributed applications to recover from connection crashes. Moreover, the design pattern was adapted to different types of applications, each one with its specificities. Therefore, MBPTA is applicable to the family of fault-tolerant, distributed applications that are built form the design pattern.

The CHDP is part of the Devasses research project [1], in which this dissertation is also inserted. Section 6.1 describes the Connection Handler Design Pattern (CHDP). Section 6.2 shows the application of the proposed approach for the family of CHDP-based fault-tolerant applications. Finally, section 6.3 presents the state machine extracted using the MARITACA tool.

6.1 The Connection Handler Design Pattern (CHDP)

The CHDP is built on a basic pattern and proposes a design solution for the development of connection-oriented applications. This pattern is independent of the application's logic, the programming language, or the platform where the application is running. Once a connection is established, client and server exchange data to accomplish a given service. At any moment, any of the two peers can close the connection. When a connection crash occurs, the server must distinguish a reconnection attempt from a new connection request, to replace the failed connection with a fresh one, if necessary. Once the successful reconnection phase finishes, the server and the client retransmit lost data. Figure 6.1 shows a sequence diagram of a failure-free scenario [23], where the client and server establish a new connection and can start writing and reading data.

To show that CHDP can be reused and specialized for different types of distributed applications tolerant to connection crashes, three different types of applications were considered [23]:

- **Stream-based applications:** that transmit flow of bytes instead of structured messages. Examples of such applications are file systems and multimedia applications;

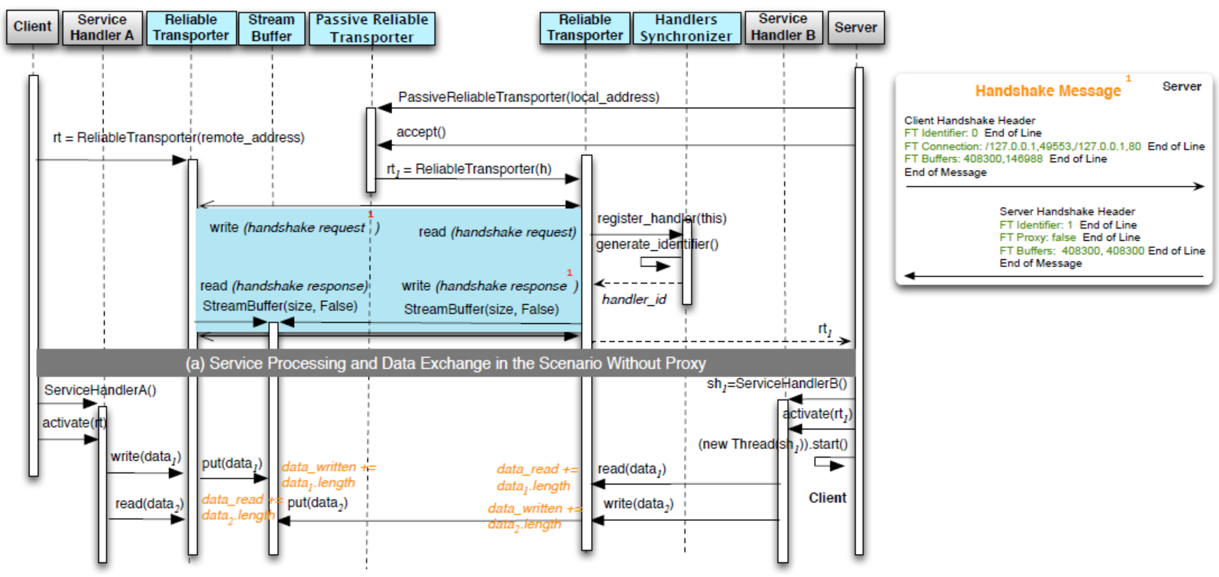

Client
Service Handler A
Reliable Transporter
Stream Buffer
Passive Reliable Transporter
Reliable Transporter
Handlers Synchronizer
Service Handler B
Server
PassiveReliableTransporter(local_address)
accept()
rt = ReliableTransporter(remote_address)
rt1 = ReliableTransporter(h)
write (handshake request)
read (handshake request)
register_handler(this)
generate_identifier()
read (handshake response)
write (handshake response)
StreamBuffer(size, False)
handler_id
rt1
(a) Service Processing and Data Exchange in the Scenario Without Proxy
ServiceHandlerA()
sh1=ServiceHandlerB()
activate(rt)
activate(rt1)
(new Thread(sh1)).start()
write(data1)
put(data1)
data_written += data1.length
data_read += data1.length
read(data1)
read(data2)
data_read += data2.length
put(data2)
data_written += data2.length
write(data2)
Client
Handshake Message
Server
Client Handshake Header
FT Identifier: 0 End of Line
FT Connection: /127.0.0.1,49553,/127.0.0.1,80 End of Line
FT Buffers: 408300,146988 End of Line
End of Message
Server Handshake Header
FT Identifier: 1 End of Line
FT Proxy: false End of Line
FT Buffers: 408300, 408300 End of Line
End of Message

Figure 6.1: Sequence diagram of a failure free scenario [23]

- **HTTP-based applications:** web applications which usually are based on HTTP messaging protocol, but here they rely on the stream-based solution, for the sake of efficiency;
- **Message-based applications:** which communicate by exchanging messages on top, for example, of a TCP connection.

In the sequel, it is presented how these applications are considered as an SPL and how the proposed approach could be applied for model-based test case generation.

6.2 Domain Engineering

In order to apply our approach, the distributed applications tolerant to connection crashes were considered as an SPL, in which the CHDP contains the common features of these applications. Therefore, the domain artifacts consisted of the feature model, traceability matrix, product map and use cases descriptions, as explained in Subsection 4.1.

Figure 6.2 presents the feature model. Features as Transport handle, Service Handler, Passive transport handle, and Event are mandatory for all applications. Read and write features are optional for the products. Buffer and Reliable Endpoint are alternative features, where only one feature can be selected. When the Stream buffer feature is selected, cannot be selected the HTTP Buffer or the Message Buffer. Also, the selection of Stream buffer implies the Reliable transport feature selection.

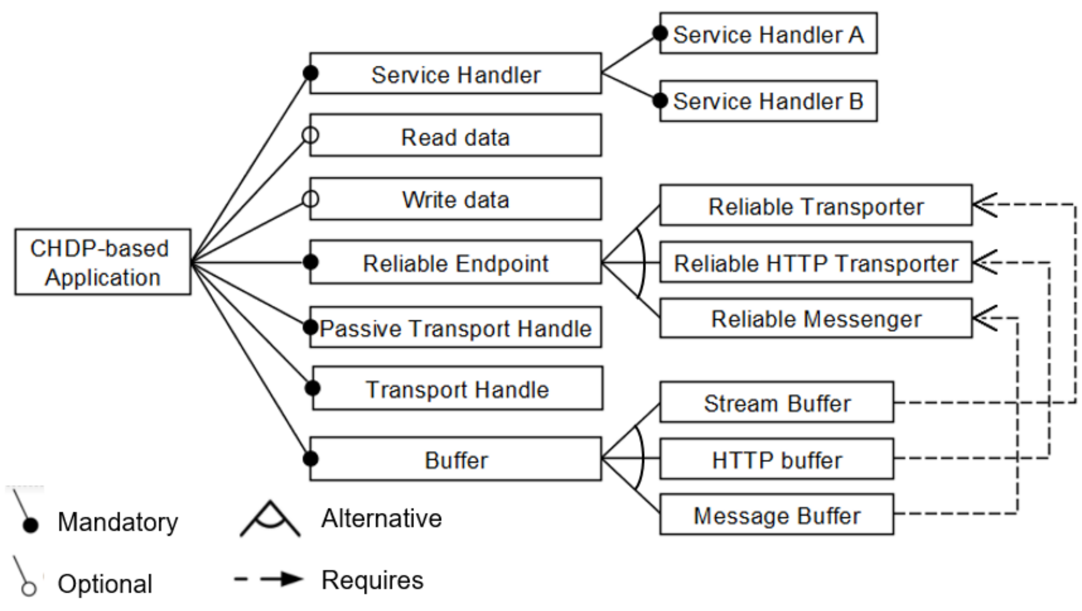


Figure 6.2: Feature model of CHDP-based application

The use cases diagram is shown in Figure 6.3. The stereotypes in the diagram are used to distinguish the use case types, which can be: mandatory, alternative and optional, as explained in Subsection 4.1.3. Include and/or Extend relationships among use cases are represented too. An excerpt of the description of Create Passive Transport Handler use case is shown in TABLE 6.1. This use case was described using the restriction rules presented in Subsection 4.1.3.

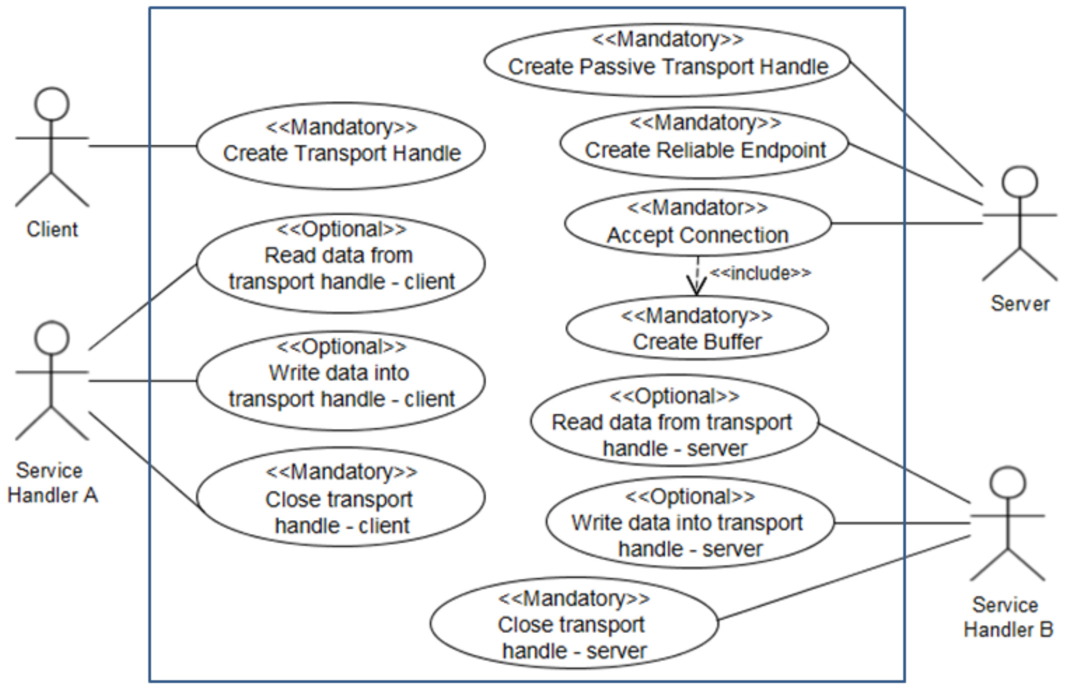

<<Mandatory>> Create Passive Transport Handle
<<Mandatory>> Create Transport Handle
<<Mandatory>> Create Reliable Endpoint
Client
<<Optional>> Read data from transport handle - client
<<Mandator>> Accept Connection
<<include>>
Server
<<Mandatory>> Create Buffer
<<Optional>> Write data into transport handle - client
<<Optional>> Read data from transport handle - server
Service Handler A
<<Mandatory>> Close transport handle - client
<<Optional>> Write data into transport handle - server
<<Mandatory>> Close transport handle - server
Service Handler B

Figure 6.3: Use case diagram for CHDP

Table 6.1: Create Passive Transport Handle use case description

Case Id: 005
Use Case Name: Create Passive Transport Handle
Use Case Type: Mandatory
Cardinality feature: 1..1
Brief Description: The Passive Transport Handle is created
Associate feature: Passive Transport Handle
Primary Actor: Server
Secondary Actor: None
Pre-condition: An IP address is provided AND A Port number is provided
Dependency: None
Generalization: None
Basic Flow Steps (BFS):
1. The servers request to create new Passive Transport Handle by providing an IP address and a Port Number.
2. The system VALIDATE THAT the IP address and the Port Number are ready to use.
3. The system creates a new Passive Transport Handle.

Postcondition: A Passive Transport Handle is created.
...
Specific Exceptional Flow (SEF-1): BFS 2
1. The system throws a BindException
2. ABORT

Postcondition: A BindException is thrown.
...

TABLES 6.2 and 6.3 show the Product Map and the Traceability Matrix created to represent the relation between features and use cases, and Features and products.

Table 6.2: Product Map of CHDP-based application

Type	**Feature**	**Default**	**Stream-based application**	**HTTP-based application**	**Message-based application**
Common	Service Handler A	X	X	X	X
Common	Service Handler B	X	X	X	X
Common	Transport Handle	X	X	X	X
Common	Passive Transport Handle	X	X	X	X
Common	Reliable Transporter	X	X	/	/
Alternative	Reliable HTTP Transporter	/	/	X	/
Alternative	Reliable Messenger	/	/	/	X
Alternative	Stream Buffer	X	X	/	/
Alternative	HTTP Buffer	/	/	X	/
Alternative	Message Buffer	/	/	/	X
Optional	Write data	/	X	X	/
Optional	Read Data	/	X	X	/

Table 6.3: Traceability Matrix of CHDP-based application

Feature		**Use cases**										
		Mandatory							**Optional**			
Type	**Name**	Create Transport Handle	Close transport handle - client	Create Passive Transport Handle	Create Reliable Endpoint	Accept Connection	Create a Buffer	Close Transport Handle - server	Read data from transporthandle - client	Write data into transport handle - client	Read data from transport handle - server	Write data into transport handle - server
Common	Service Handler A	X	/	/	/	/	/	/	/	/	/	/
Common	Service Handler B	/	/	/	/	/	/	/	/	/	/	/
Common	Transport Handle	X	X	/	/	X	/	X	/	/	/	/
Common	Passive Transport Handle	/	/	X	/	/	/	/	/	/	/	/
Common	Passive Transport Handle	/	/	/	X	/	/	/	/	/	/	/
Alternative	Reliable Transporter	/	/	/	X	/	/	/	/	/	/	/
Alternative	Reliable Messenger	/	/	/	X	/	/	/	/	/	/	/
Alternative	Stream Buffer	/	/	/	/	/	X	/	/	/	/	/
Alternative	HTTP Buffer	/	/	/	/	/	X	/	/	/	/	/
Alternative	Message Buffer	/	/	/	/	/	X	/	/	/	/	/
Optional	Write data	/	/	/	/	/	/	/	/	X	/	X
Optional	Read Data	/	/	/	/	/	/	/	X	/	X	/

6.3 Application Engineering

As was explained in Chapter 4 in this phase, two artifacts are generated: the context class and the state model, which is extracted from the domain artifacts. These models are presented in the sequel.

Using the proposed approach and the MARITACA tool, initial state machines for CHDP-based application are generated. TABLE 6.4 presents an excerpt of the textual state machine for Stream-based application generated by MARITACA.

The context class associated with the generated state machine was manually created. This context class represents the attributes and operations used in the state machine and is shown in Figure 6.4. In our work is not presented a way to generate the context class, but when initial version of a state machine is extracted, this state machine can help to complete the context class with the operations and attributes that belong to the state machine.

Table 6.4: Excerpt of textual state machine for Stream-based application

```
Pre:  An IP address is provided AND A Port number is provided
Trigger:  Create Transport Handle
Guard:  the connection is reachable AND the Passive Transport Handle is
created
Effect:  creates a new Transport Handle
Post:  A Transport Handle is created AND A new Service Handler A is
activated AND The Transport Handle is not closed
Pre:  A Passive Transport Handle is created
Trigger:  Create Reliable Endpoint
Guard:
Effect:  creates a new Reliable Transporter
Post:  A Passive Transport Handle is created AND The Reliable Transporter
is created
Pre:  A Passive Transport Handle is created AND A new Service Handler B
is activated AND The Transport Handle is not closed
Trigger:  Read data from transport handle - server
Guard:  the channel is not empty during T
Effect:  puts the data into the data buffer
Post:  The Service Handler B is run AND The Transport Handle is not
closed
```

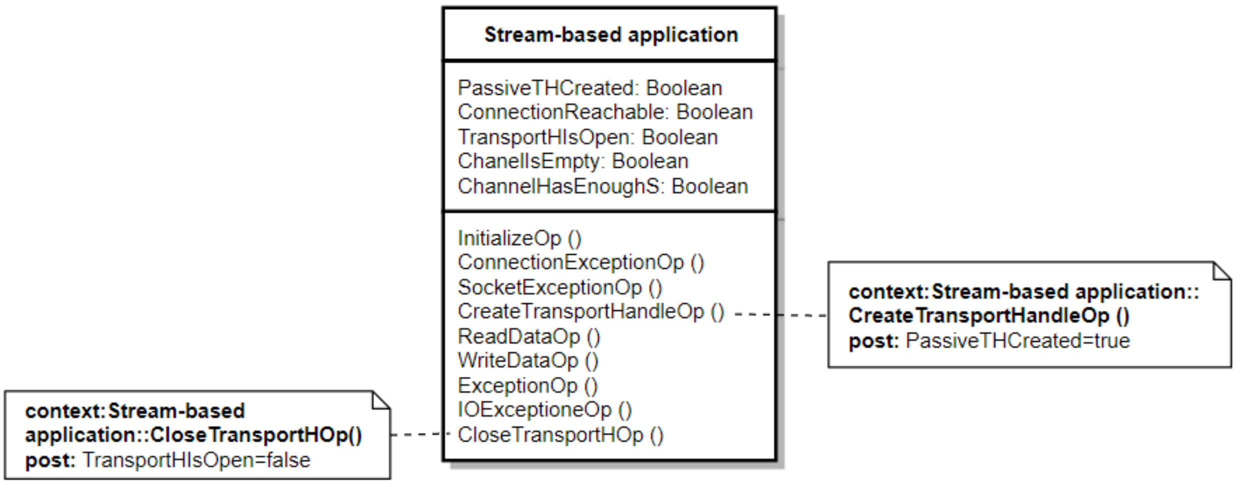

Stream-based application
PassiveTHCreated: Boolean
ConnectionReachable: Boolean
TransportHIsOpen: Boolean
ChanelIsEmpty: Boolean
ChannelHasEnoughS: Boolean
InitializeOp ()
ConnectionExceptionOp ()
SocketExceptionOp ()
CreateTransportHandleOp ()
ReadDataOp ()
WriteDataOp ()
ExceptionOp ()
IOExceptioneOp ()
CloseTransportHOp ()
context: Stream-based application:: CreateTransportHandleOp ()
post: PassiveTHCreated=true
context: Stream-based application::CloseTransportHOp()
post: TransportHIsOpen=false

Figure 6.4: Context class for CHDP-based Application

Figure 6.5 represents graphically the initial version of the state machine for Stream-based application extracted using the MARITACA tool.

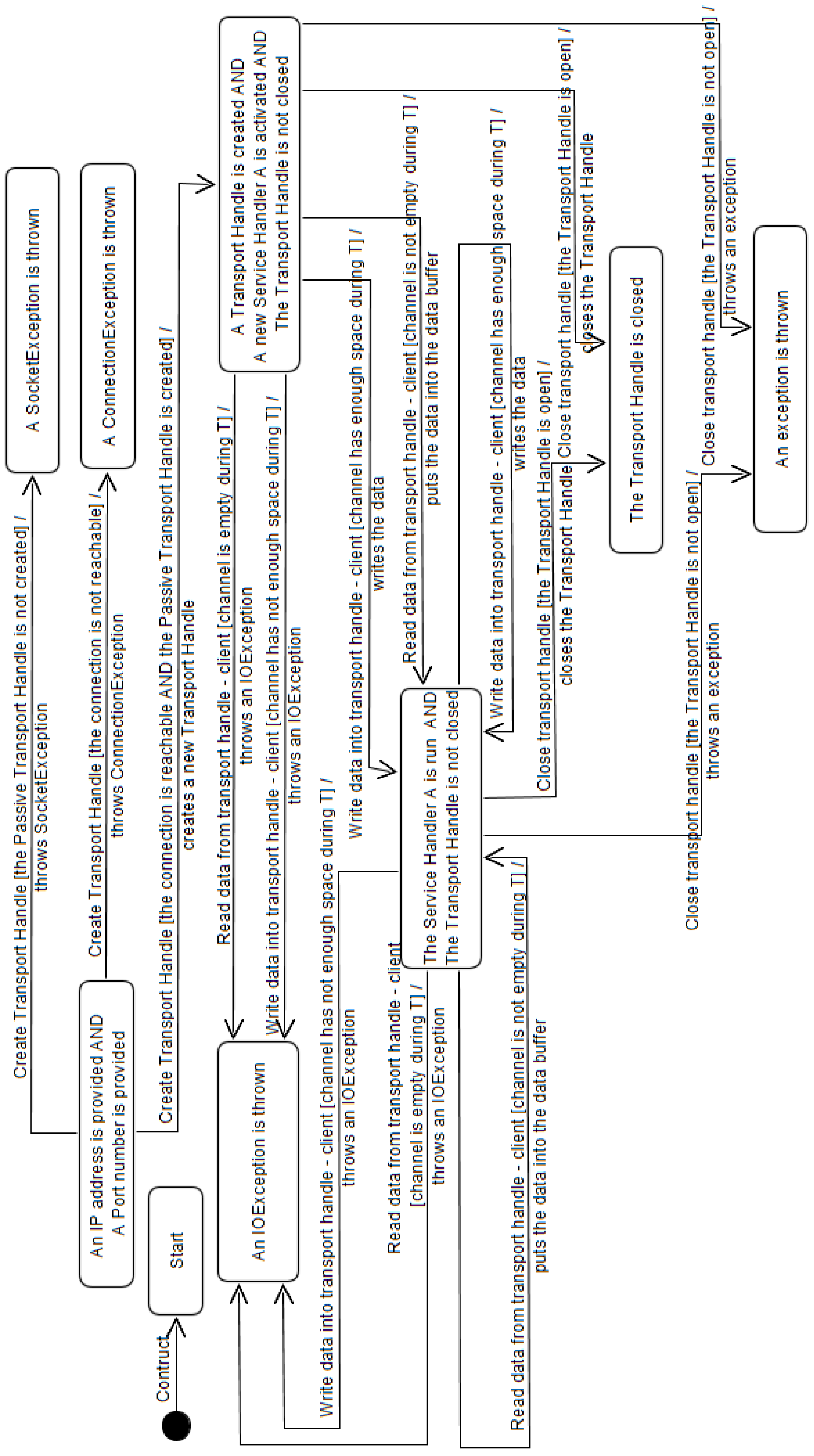

Contruct
Start
An IP address is provided AND A Port number is provided
Create Transport Handle [the Passive Transport Handle is not created] / throws SocketException
A SocketException is thrown
Create Transport Handle [the connection is not reachable] / throws ConnectionException
A ConnectionException is thrown
Create Transport Handle [the connection is reachable AND the Passive Transport Handle is created] / creates a new Transport Handle
A Transport Handle is created AND A new Service Handler A is activated AND The Transport Handle is not closed
Read data from transport handle - client [channel is empty during T] / throws an IOException
Write data into transport handle - client [channel has not enough space during T] / throws an IOException
An IOException is thrown
Write data into transport handle - client [channel has enough space during T] / writes the data
Read data from transport handle - client [channel is not empty during T] / puts the data into the data buffer
The Service Handler A is run AND The Transport Handle is not closed
Write data into transport handle - client [channel has enough space during T] / writes the data
Write data into transport handle - client [channel has not enough space during T] / throws an IOException
Read data from transport handle - client [channel is empty during T] / throws an IOException
Read data from transport handle - client [channel is not empty during T] / puts the data into the data buffer
Close transport handle [the Transport Handle is open] / closes the Transport Handle
Close transport handle [the Transport Handle is open] / closes the Transport Handle
The Transport Handle is closed
Close transport handle [the Transport Handle is not open] / throws an exception
Close transport handle [the Transport Handle is not open] / throws an exception
An exception is thrown

Figure 6.5: Initial state model for Stream-based application

By using the steps presented in subsection 4.3, the initial state machine was refined. States were renamed for short and representative names, for example, the state *"An IP address is provided AND A Port number is provided"* was renamed by *"Initial"*

New transitions were added to link: the *"Start"* state with the *"Initial"* state, and the *ConnectionException, SocketEXception, IOException, Exception"* states with final pseudostates. Taking in count the attributes and operations of the context class, Parameters, Events, and Operations were added or modified in the state machine. The transitions were modified by replacing textual guards with corresponding OCL constraints. For example, TABLE 6.5 presents the transformation made between two states and the elements of its transition.

Table 6.5: Example of refinement

Element	**Textual state machine**	**Refined state machine**	**Stream-based application context class**
Source state's name	An IP address is provided AND A Port number is provided	Initial	
Target state's name	A Transport Handle is created AND A new Service Handler A is activated AND The Transport Handle is not closed	Ready	
Event	Create Transport Handle	CreateTransportHandleEv	
Parameters	the connection is reachable	connectionReachable = True	connectionReachable: Boolean
	the Passive Transport Handle is created	PassiveTransportHandleCreated = True	PassiveTransportHandleCreated : Boolean
Operation	creates a new Transport Handle	CreateTransportHandleOp	CreateTransportHandleOp

Figure 6.6 shows the refined state machine using the parameters and operations of the context class (Figure 6.4).

For test case generation, the ParTeG tool [48] was used because it is a model-based testing tool that automatically generates test cases from state machines and class diagrams. The class diagrams are annotated with OCL expressions. For the test case generation, Parteg supports some coverage criteria [48].

This tool allows the code generation of test suites for JUnit 3.8, JUnit 4.3, Java Mutation Analysis, and CppUnit 1.12. test suites. An excerpt of test cases for Stream-based application is presented in TABLE 6.6.

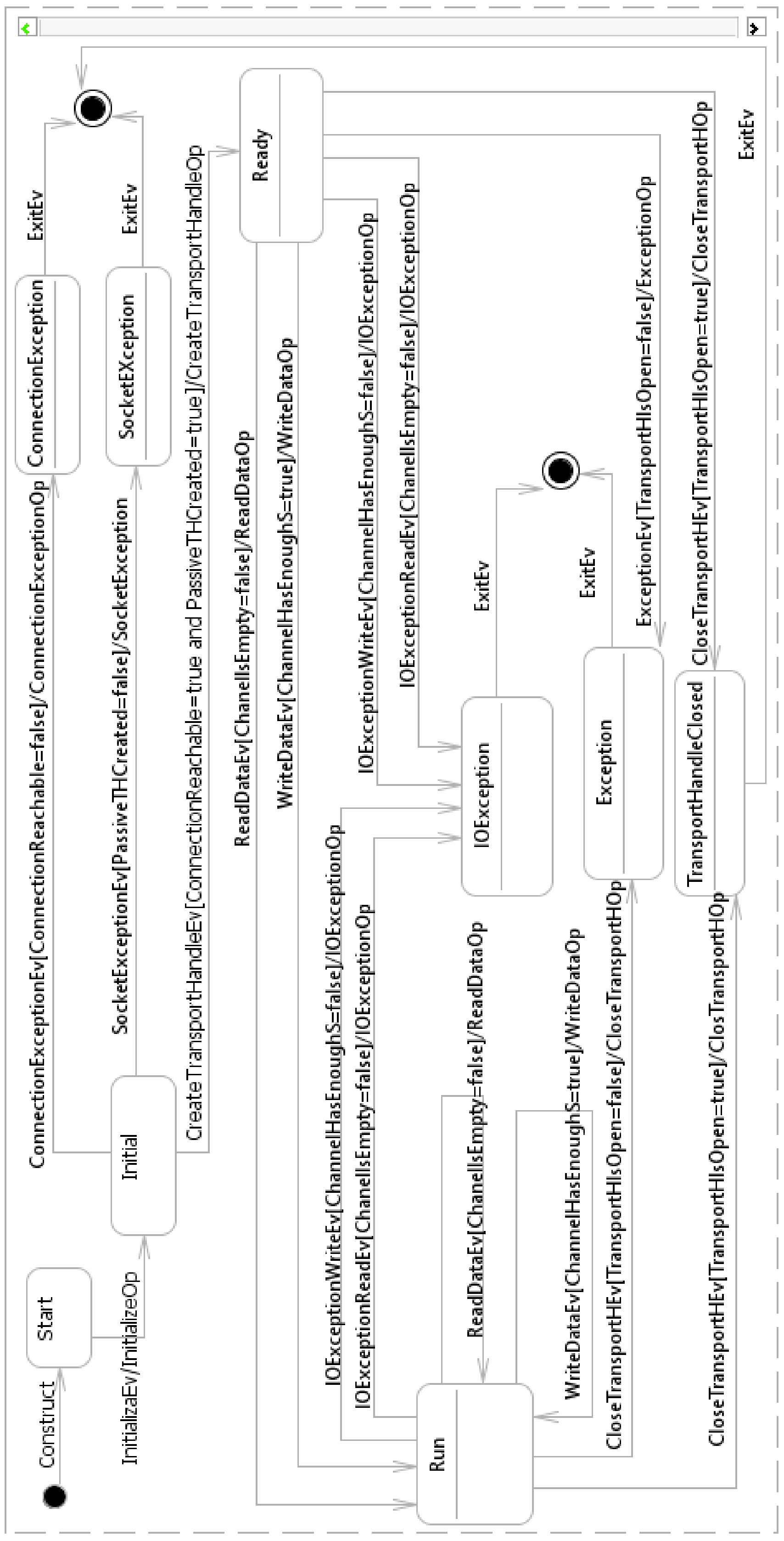

Construct
Start
InitializaEv/InitializeOp
Initial
ConnectionExceptionEv[ConnectionReachable=false]/ConnectionExceptionOp
ConnectionException
ExitEv
SocketExceptionEv[PassiveTHCreated=false]/SocketException
SocketEXception
ExitEv
CreateTransportHandleEv[ConnectionReachable=true and PassiveTHCreated=true]/CreateTransportHandleOp
Ready
ReadDataEv[ChanelIsEmpty=false]/ReadDataOp
WriteDataEv[ChannelHasEnoughS=true]/WriteDataOp
IOExceptionWriteEv[ChannelHasEnoughS=false]/IOExceptionOp
IOExceptionReadEv[ChanelIsEmpty=false]/IOExceptionOp
ExitEv
ExitEv
ExceptionEv[TransportHIsOpen=false]/ExceptionOp
CloseTransportHEv[TransportHIsOpen=true]/CloseTransportHOp
ExitEv
IOException
Exception
TransportHandleClosed
IOExceptionWriteEv[ChannelHasEnoughS=false]/IOExceptionOp
IOExceptionReadEv[ChanelIsEmpty=false]/IOExceptionOp
ReadDataEv[ChanelIsEmpty=false]/ReadDataOp
WriteDataEv[ChannelHasEnoughS=true]/WriteDataOp
CloseTransportHEv[TransportHIsOpen=false]/CloseTransportHOp
CloseTransportHEv[TransportHIsOpen=true]/ClosTransportHOp
Run

Figure 6.6: Refined state machine for Stream-based application

Table 6.6: Excerpt of test cases for Stream-based application generated by ParTeG

Test case	Description

public void test1() {
TestClient oTestObject = new TestClient();
	Initialization of test case object.

Boolean ChanelIsEmptyVal1 = false;
Boolean ConnectionRVal1 = true;
Boolean PassiveTHCreatedVal1 = true;
Boolean TransportHIsOpenVal1 = true;
Boolean ChannelHasEnoughSVal1 = true;
	Definition of internal variables used for Parteg. Parteg assign to each variable a value. For Boolean: true or false, boundary values for arbitrary integer values are created.

assertEquals(true, (oTestObject.state==State.Start));	First state of the state machine
oTestObject.handleEvent("InitializeEv", ConnectionRVal1, TransportHIsOpenVal1, PassiveTHCreatedVal1, ChanelIsEmptyVal1, ChannelHasEnoughSVal1);	Event used to initialize parameters of the state machine.

assertEquals(true, (oTestObject.ConnectionReachable.booleanValue()==ConnectionRVal1.booleanValue()));
assertEquals(true, (oTestObject.PassiveTHCreated.booleanValue()==PassiveTHCreatedVal1.booleanValue()));
assertEquals(true, (oTestObject.ChanelIsEmpty.booleanValue()==ChanelIsEmptyVal1.booleanValue()));
assertEquals(true, (oTestObject.ChannelHasEnoughS.booleanValue()==ChannelHasEnoughSVal1.booleanValue()));
assertEquals(true, (oTestObject.TransportHIsOpen.booleanValue()==TransportHIsOpenVal1.booleanValue()));
	Parameters of the state machine are initialized with the variables create by Parteg. It is made in the InitializeOp obtained from the context class.

assertEquals(true, (oTestObject.state==State.Initial));	The initial state is activated when the InitializeEv is executed.

assertEquals(true, (oTestObject.ConnectionReachable.booleanValue() == true));
assertEquals(true, (oTestObject.PassiveTHCreated.booleanValue()==true));
oTestObject.handleEvent("CreateTransportHandleEv");
assertEquals(true, (oTestObject.state==State.Ready));
	Guard between Initial and Ready states, these parameters are evaluated when CreateTransportHandleEv event is triggered.

assertEquals(true, (oTestObject.ChanelIsEmpty.booleanValue()==false));
oTestObject.handleEvent("ReadDataEv");
assertEquals(true, (oTestObject.state==State.Run));
	Guard between Ready and Run states when ReadDataEv is triggered.

assertEquals(true, (oTestObject.ChanelIsEmpty.booleanValue()==true));
oTestObject.handleEvent("IOExceptionReadEv");
assertEquals(true, (oTestObject.state==State.IOException));
}
	Guard between Run and IOException states, when IOExceptionReadEv is triggered.

6.4 Chapter Summary

This chapter is focused on the use of proposed approach in a distributed applications tolerant to connection crashes. The Domain Engineering and the Application Engineering phases of the approach described in Chapter 4 were employed, and the artifacts produced in each of these phases were presented.

To use this approach, the Use cases were described, Product Map and Traceability Matrix were defined. Using MARITACA tool an initial state machine for Stream-based application was extracted. Finally, the initial state machine was refined and using ParTeG tool a set of test cases were generated.

Next Chapter presents the discussion about the obtained results for Microwave oven SPL and Connection Handle Design Pattern SPL.

Chapter 7

Results and discussion

This chapter presents a preliminary validation of the proposed approach and tool. Section 7.1 presents the questions that are used to evaluate the approach and the procedure followed for this evaluation. Section 7.2 shows a comparison between manually and automatically state models extracted from Microwave oven SPL. This section also presents the analysis of the test cases generated from the refined state models. Section 7.3 presents the obtained result after to use the proposed approach in the Connection Handler Design Pattern (CHDP), a comparison between automatically and manually state models extracted are presented, and an analysis of the test cases generated from these state models is shown.

7.1 Validation procedure and tools

The validation of MBPTA and MARITACA presented here is based on SPL applications: in the microwave oven, used along the chapters to illustrate the approach, and the CHDP-based applications presented in Chapter 6. With this validation, the goal was to answer the following questions:

Q1 Do the transformation rules allow the generation of partial, but syntactically correct state models?

Q2 Does MARITACA generate state machines that are syntactically correct?

Q3 Applying MBPTA is it possible to generate relevant test cases for a new product?

Regarding Q1, the transformation rules were applied manually to the two families. The "syntactically correct" term means that the state machines, although partially generated from these rules, are in accordance with the state diagram notation presented in Section 4.2.3. Once the transformation rules were validated, in order to answer Q2, it was performed a comparison between the textual representation of the manually and automatically extracted models. In order to compare these two versions of the state models the MELD tool [50] is used. Meld is a tool that helps to compare files and visualizes global and local differences. When MELD detects a difference between two documents, it highlights words or set of words where the differences were found.

In order to answer Q3, it is necessary, first, to generate test cases for the different products and then, analyze the similarity among these test cases. For test case generation a tool, Parteg, was used. This tool allows different test criteria for test case generation from state models. For this evaluation, the All-Transitions and Decision Coverage criteria are applied. Satisfy the coverage criterion All-Transitions requires traversing all transition sequences up to length one. In [48], the term "up to length one" includes length one and length zero (paths with length equal to zero are interpreted as states). Decision coverage is a simple control-flow-based coverage criterion. It is satisfied iff each guard condition is evaluated to true and false, respectively.

To determinate whether test cases generated for different products are similar or not, cluster analysis was used. This is a statistical multivariate method that divides data into groups (clusters) in such a way that data in the same group are more similar to each other than to those data in other groups [43]. For the grouping, some measure of similarity is necessary to determine the proximity (or distance) between pairs of data. Also, there are different types of clustering. Therefore, the analysis is composed of three steps: *i)* determine how to represent the test cases for similarity comparison; *ii)* generate a similarity (or distance) matrix, in which each cell represents a distance between a pair of test cases; *iii)* use the matrix for test case clustering and analyze the obtained clusters. The following paragraphs present a brief description of these steps.

Test case representation

Although the Parteg tool produces concrete test cases in Java, that is, test cases that are ready to execute, an abstract representation of the test cases, that is platform independent, was used, for two reasons: *i)* it makes the analysis independent of the test case generation tool used, and *ii)* in the context of MBT, abstract test cases are generally considered, as the model represents an abstract representation of the system under test. In state-based testing, an executable test case is an instance of a path starting at the initial state and ending in some state. These test paths may vary according to the criteria used for test case generation. There can be various ways to represent such paths, depending on what elements (states, transitions, events, guards) from the state machine are used to characterize the path. In this study, test paths are represented as a sequence of transition ids. Figure 7.1 shows an illustrative example of a UML state machine, whereas Figure 7.2 represents the transition tree that produced for the all-transitions criteria applied to this model. The bold lines represent an example transition path, identified as T0-T2-T4-T6.

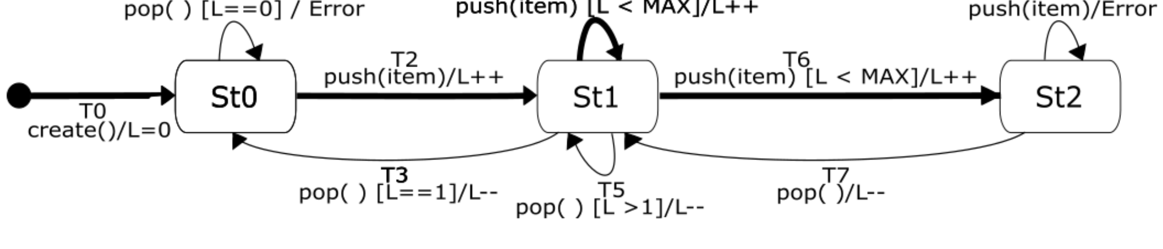


Figure 7.1: Part of the stack state machine

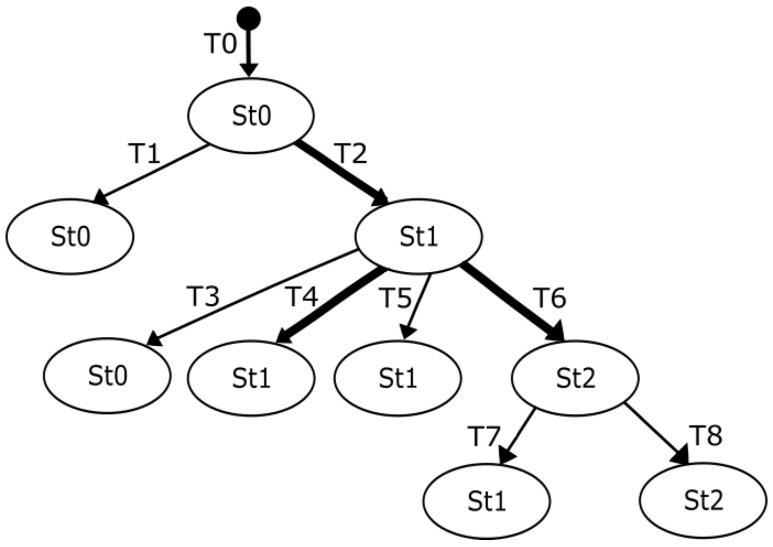


Figure 7.2: Transition tree of the stack state machine

Similarity matrix generation

The test path represented as above can be seen as transition sequence, so, to obtain the similarity matrix between pairs of test cases it is necessary to use similarity measures between two sequences [20]. The advantage of using sequence-based similarity measures is that the order in which the transitions appear in the sequence will be taken into account. A number of measures have been proposed in the literature, supported by a number of different tools. In this study, we used Harry, a tool that implements various similarity measures for strings [43].

Cluster analysis

With the similarity matrix obtained in the previous step, it is possible to use different techniques for cluster analysis (see, for example [30]). In this study a hierarchical clustering technique was used, which allowed the graphical representation of the results in the form of a dendrogram, that displays a hierarchical relationship among the test cases. A dendrogram is a tree-like diagram that shows the groups that are formed to create clusters of data and their levels of similarity. Similar data are connected to the links such that the position in the diagram is determined by the level of similarity/dissimilarity between data. The level of similarity is measured on the vertical bars (alternatively it can show the distance level) and the different variables are specified on the horizontal bars. The dendrograms were obtained with the use of the R tool [33].

7.2 Obtained results using the Microwave oven SPL

The microwave oven SPL was introduced to serve as an example of the approach in Chapter 4. For this SPL, manually and automatically state models were extracted for four products (See TABLE 4.7).

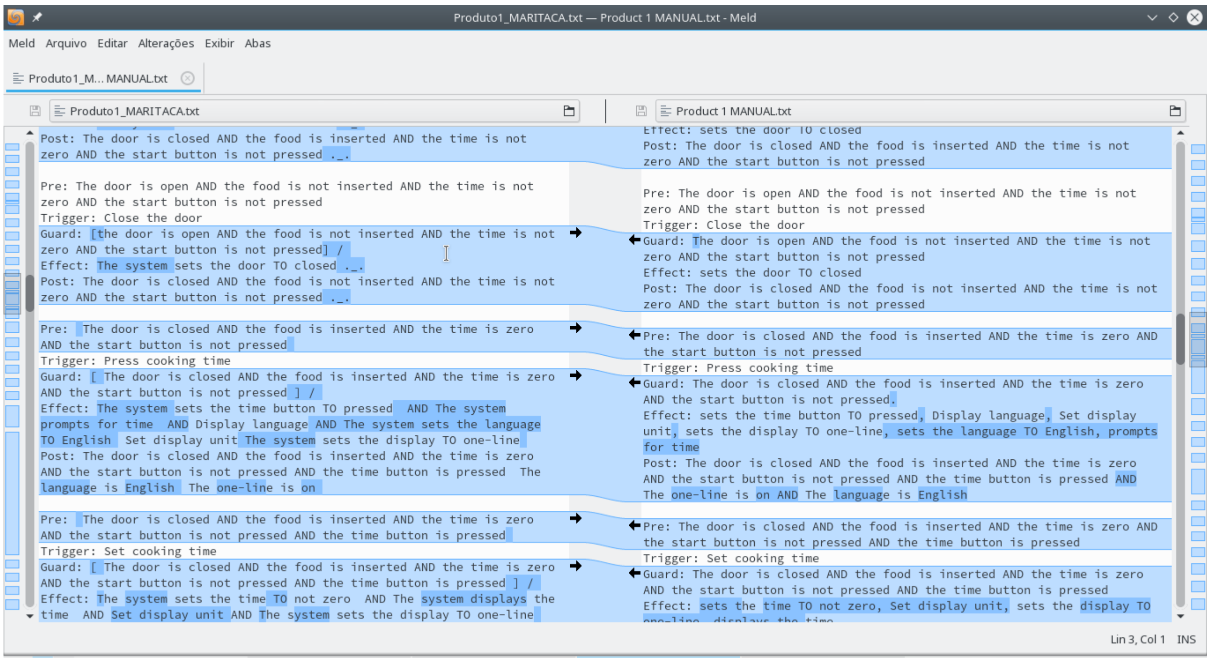
To answer Q1, the automatically extracted state models were compared with the manually created to evaluate if MARITACA can create syntactically correct state models by following the transformation rules. This evaluation showed that the state models were extracted in accordance with the structure of state model shown in Section 4.2.3.

The initial state model has some duplicate states because as was explained in Section 4.2.4, MARITACA does not make difference between two postconditions semantically similar when optional features are selected. For each product, 20 initial states were created and when the state models were refined 9 duplicate states were removed from each stat model.

7.2.1 Comparison between manually and automatically generated state models

Following the procedure mentioned in the Section 7.1, manually generated and automatically extracted state models were compared using MELD tool. Figures 7.3 and 7.4 show the comparison for Products 1 and 2, respectively. Meld shows the manually generated and automatically extracted state model side-by-side. The differences between the state models are highlighted to indicate where they are different.

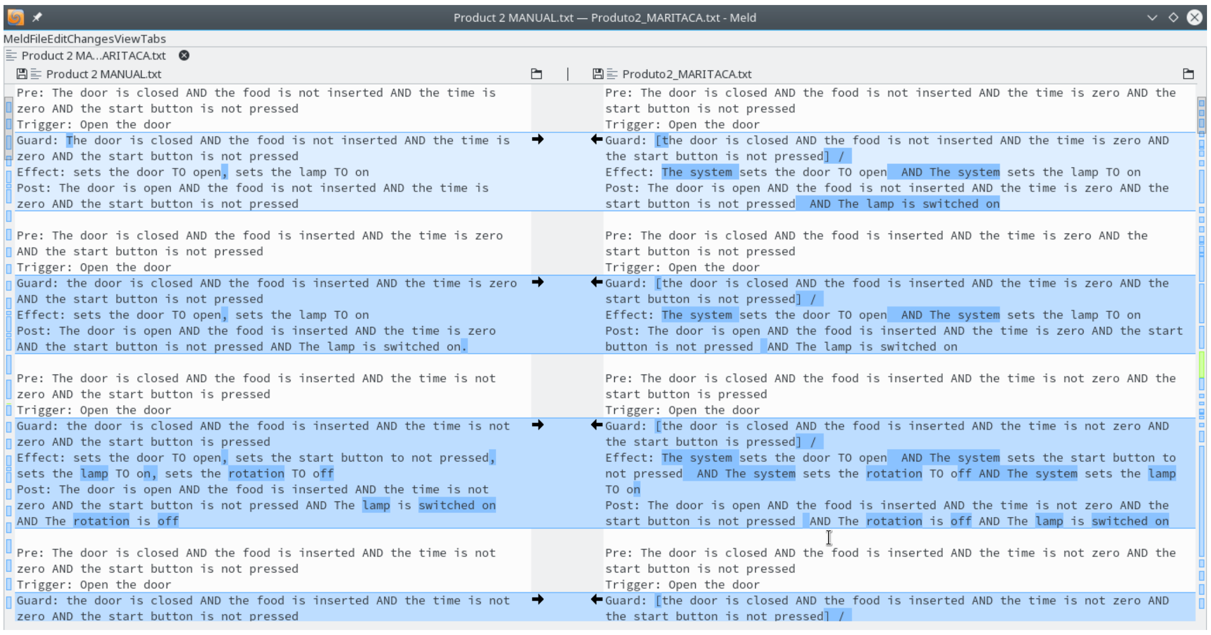
To answer Q2, the two figures show that the differences between manually and automatically extracted textual state models are not significant, these differences are about punctuation marks, spaces, lowercase, and uppercase. Although some differences are related to the order as MARITACA and the user read the use case descriptions, for this example all states and transitions agree. This comparison allows confirming that MARITACA tool generates the state models in accordance with the rules established in Chapter 4.


Produto1_MARITACA.txt — Product 1 MANUAL.txt - Meld
Meld Arquivo Editar Alterações Exibir Abas
Produto1_M... MANUAL.txt
Produto1_MARITACA.txt
Product 1 MANUAL.txt
Lin 3, Col 1 INS

```
Post: The door is closed AND the food is inserted AND the time is not
zero AND the start button is not pressed ._.

Pre: The door is open AND the food is not inserted AND the time is not
zero AND the start button is not pressed
Trigger: Close the door
Guard: [the door is open AND the food is not inserted AND the time is not
zero AND the start button is not pressed] /
Effect: The system sets the door TO closed ._.
Post: The door is closed AND the food is not inserted AND the time is not
zero AND the start button is not pressed ._.

Pre:  The door is closed AND the food is inserted AND the time is zero
AND the start button is not pressed
Trigger: Press cooking time
Guard: [ The door is closed AND the food is inserted AND the time is zero
AND the start button is not pressed ] /
Effect: The system sets the time button TO pressed  AND The system
prompts for time  AND Display language AND The system sets the language
TO English  Set display unit The system sets the display TO one-line
Post: The door is closed AND the food is inserted AND the time is zero
AND the start button is not pressed AND the time button is pressed  The
language is English  The one-line is on

Pre:  The door is closed AND the food is inserted AND the time is zero
AND the start button is not pressed AND the time button is pressed
Trigger: Set cooking time
Guard: [ The door is closed AND the food is inserted AND the time is zero
AND the start button is not pressed AND the time button is pressed ] /
Effect: The system sets the time TO not zero  AND The system displays the
time  AND Set display unit AND The system sets the display TO one-line
```

```
Effect: sets the door TO closed
Post: The door is closed AND the food is inserted AND the time is not
zero AND the start button is not pressed

Pre: The door is open AND the food is not inserted AND the time is not
zero AND the start button is not pressed
Trigger: Close the door
Guard: The door is open AND the food is not inserted AND the time is not
zero AND the start button is not pressed
Effect: sets the door TO closed
Post: The door is closed AND the food is not inserted AND the time is not
zero AND the start button is not pressed

Pre: The door is closed AND the food is inserted AND the time is zero AND
the start button is not pressed
Trigger: Press cooking time
Guard: The door is closed AND the food is inserted AND the time is zero
AND the start button is not pressed.
Effect: sets the time button TO pressed, Display language, Set display
unit, sets the display TO one-line, sets the language TO English, prompts
for time
Post: The door is closed AND the food is inserted AND the time is zero
AND the start button is not pressed AND the time button is pressed AND
The one-line is on AND The language is English

Pre: The door is closed AND the food is inserted AND the time is zero AND
the start button is not pressed AND the time button is pressed
Trigger: Set cooking time
Guard: The door is closed AND the food is inserted AND the time is zero
AND the start button is not pressed AND the time button is pressed
Effect: sets the time TO not zero, Set display unit, sets the display TO
one-line, displays the time
```

Figure 7.3: Differences between manually generated and automatically extracted textual state models for product 1


Product 2 MANUAL.txt — Produto2_MARITACA.txt - Meld
MeldFileEditChangesViewTabs
Product 2 MA...ARITACA.txt
Product 2 MANUAL.txt
Produto2_MARITACA.txt

Product 2 MANUAL.txt:

```
Pre: The door is closed AND the food is not inserted AND the time is
zero AND the start button is not pressed
Trigger: Open the door
Guard: The door is closed AND the food is not inserted AND the time is
zero AND the start button is not pressed
Effect: sets the door TO open, sets the lamp TO on
Post: The door is open AND the food is not inserted AND the time is
zero AND the start button is not pressed

Pre: The door is closed AND the food is inserted AND the time is zero
AND the start button is not pressed
Trigger: Open the door
Guard: the door is closed AND the food is inserted AND the time is zero
AND the start button is not pressed
Effect: sets the door TO open, sets the lamp TO on
Post: The door is open AND the food is inserted AND the time is zero
AND the start button is not pressed AND The lamp is switched on.

Pre: The door is closed AND the food is inserted AND the time is not
zero AND the start button is pressed
Trigger: Open the door
Guard: the door is closed AND the food is inserted AND the time is not
zero AND the start button is pressed
Effect: sets the door TO open, sets the start button to not pressed,
sets the lamp TO on, sets the rotation TO off
Post: The door is open AND the food is inserted AND the time is not
zero AND the start button is not pressed AND The lamp is switched on
AND The rotation is off

Pre: The door is closed AND the food is inserted AND the time is not
zero AND the start button is not pressed
Trigger: Open the door
Guard: the door is closed AND the food is inserted AND the time is not
zero AND the start button is not pressed
```

Produto2_MARITACA.txt:

```
Pre: The door is closed AND the food is not inserted AND the time is zero AND the
start button is not pressed
Trigger: Open the door
Guard: [the door is closed AND the food is not inserted AND the time is zero AND
the start button is not pressed] /
Effect: The system sets the door TO open  AND The system sets the lamp TO on
Post: The door is open AND the food is not inserted AND the time is zero AND the
start button is not pressed  AND The lamp is switched on

Pre: The door is closed AND the food is inserted AND the time is zero AND the
start button is not pressed
Trigger: Open the door
Guard: [the door is closed AND the food is inserted AND the time is zero AND the
start button is not pressed] /
Effect: The system sets the door TO open  AND The system sets the lamp TO on
Post: The door is open AND the food is inserted AND the time is zero AND the start
button is not pressed  AND The lamp is switched on

Pre: The door is closed AND the food is inserted AND the time is not zero AND the
start button is pressed
Trigger: Open the door
Guard: [the door is closed AND the food is inserted AND the time is not zero AND
the start button is pressed] /
Effect: The system sets the door TO open  AND The system sets the start button to
not pressed  AND The system sets the rotation TO off AND The system sets the lamp
TO on
Post: The door is open AND the food is inserted AND the time is not zero AND the
start button is not pressed  AND The rotation is off AND The lamp is switched on

Pre: The door is closed AND the food is inserted AND the time is not zero AND the
start button is not pressed
Trigger: Open the door
Guard: [the door is closed AND the food is inserted AND the time is not zero AND
the start button is not pressed] /
```

Figure 7.4: Differences between manually generated and automatically extracted textual state models for product 2

The state machines are refined as shown in Section 4.2.4. The TABLE 7.1 presents a summary of states, transitions, events, and parameters obtained for the four state machines, addresing the product described in Product Map (See TABLE 4.7)

Table 7.1: Summary of obtained results for the four state machines

Product	States	Transitions	Events	Parameters
Product 1	11	29	11	9
Product 2	11	29	11	11
Product 3	11	29	11	12
Product 4	11	32	12	11

The state machines for the four products were refined and although they are similar in relation to the number of states, transitions, and events, it is important to highlight that there are important differences among the parameters that are evaluated in each guard conditions. Some parameters represent optional (lightIsOn = light) and alternative (languageIsEnglish = English) features of SPL. For this example the Q2 is answered, the four extracted state machines are syntactically correct and all features selected for each product are presented in the parameters of the state machines.

A comparison with the state model proposed by Gomma in [18] is not possible to carry out because the use cases descriptions provided in the textbook and created in our work are different. In the MBPTA, the use cases descriptions have a direct impact on the generated state machines and the proposed transformation rules used to generate state model are different to the way used by Gomma to generate his state model.

7.2.2 Evaluation of generated test cases

The TABLE 7.2 presents the number of generated test cases for the four products by using Parteg tool.

Table 7.2: Test cases for the four products

Product	**Test Cases All-Transitions criteria**	**Test cases Decision Coverage criteria**
Product 1	19	27
Product 2	19	27
Product 3	18	26
Product 4	20	29

Figure 7.5 shows the dendrogram created to represent the similarity among the generated test cases for the four products by using coverage criterion All-Transitions, following the steps described in Section 7.1. The tree shows the pairwise dissimilarity among the test cases generated for the four products. These test cases are represented as leaf nodes. The vertical axis represents the distance between test cases or groups of test cases (a non-leaf node). Each non-leaf node has a right and a left branch. The height of a node represents the distance between the right and the left branch groups. Highly similar groups are nearer to the bottom of the tree. As we move up, the groups become bigger and the the distance among them also increases. Therefore, according to this informal interpretation of the dendrogram, the test cases connected by a horizontal line indicate high similar test cases, that is, the distance between them is zero. Given the way test cases are represented, that is, as sequences of transition ids, this means that they all have the same transitions, in the same order, which could indicate that these test cases are redundant. However, by further analyzing these test cases, it was possible to verify that, in some cases, although they have the same transitions in the same order, they differ in terms of the features they exercise. This is the case, for example, of test cases V6 (Product 1), V25 (Product 2) and V62 (Product 4), highlighted in Figure 7.5. As shown in TABLE 7.3, in which the test cases are presented in detail, they exercise different features.

Table 7.3: Excerpt of sequence of transitions for products 1, 2, and 4

Product 1	**Product 2**	**Product 4**
test6() InitEv OpenEv InsertFoodEv CloseDoorEv PressCookingTimeEv SetCookingTimeEv StartCookingFoodEv OpenEv CloseDoorEv	**test25()** InitEv OpenEv InsertFoodEv CloseDoorEv PressCookingTimeEv SetCookingTimeEv StartCookingFoodEv OpenEv CloseDoorEv	**test62()** InitEv OpenEv InsertFoodEv CloseDoorEv PressCookingTimeEv SetCookingTimeEv StartCookingFoodEv OpenEv CloseDoorEv
StartCookingFoodEv doorIsOpen == false foodIsInserted == true timeIsZero == false startIsPressed == false timeBtnIsPressed == true *heatingIsOneLevel == false*	**StartCookingFoodEv** doorIsOpen == false foodIsInserted == true timeIsZero == false startIsPressed == false timeBtnIsPressed == true *heatingIsOneLevel == false* *lightIsOn == false* *turnTableIsOn == false*	**StartCookingFoodEv** doorIsOpen == false foodIsInserted == true timeIsZero == false startIsPressed == false timeBtnIsPressed == true *heatingIsMultiLevel == false* *beeperIsOn == false*
CloseDoorEv doorIsOpen == true foodIsInserted == true timeIsZero == true startIsPressed == false	**CloseDoorEv** doorIsOpen == true foodIsInserted == true timeIsZero == true startIsPressed == false *lightIsOn == true*	**CloseDoorEv** doorIsOpen == true foodIsInserted == true timeIsZero == true startIsPressed == false

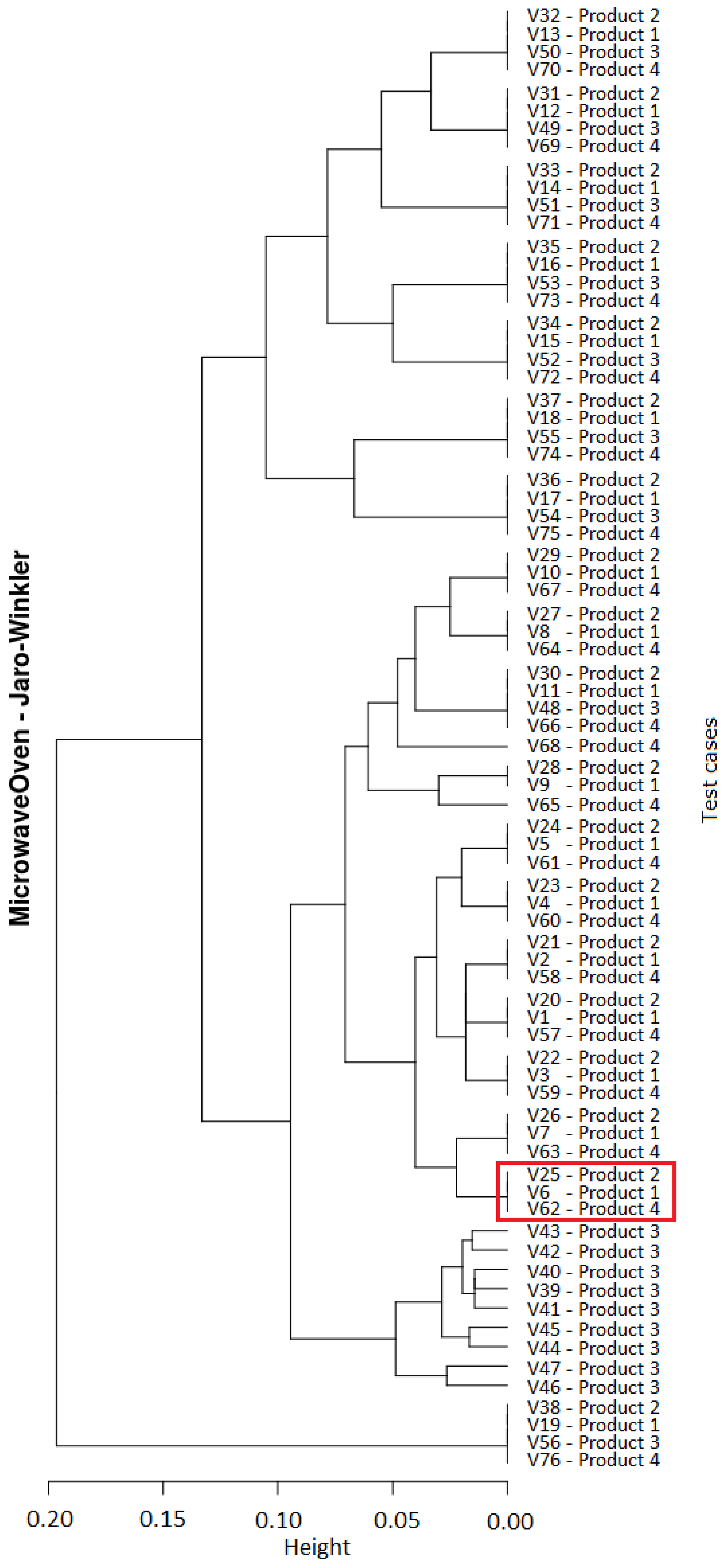


Figure 7.5: Dendrogram of generated test cases

The **StartCookingFoodEv** event of test case 25 has two parameters more than the **StartCookingFoodEv** event of test cases 6 and 62, *lightIsOn* and *turnTableIsOn* that represent optional features of SPL. The parameters that represent heating element alternative feature are different *heatingIsOneLevel* and *heatingIsMultiLevel*). The **Start-CookingFoodEv** event for test case V62 has the *beeperIsOn* parameter that represents an optional feature selected for this product. For the **CloseDoorEv** event, the test case V25 has one parameter more than test cases V6 and V62, that represents an optional feature selected only for this product.

In order to answer Q3, the proportion of redundant test cases were calculated, that is, among the 76 test cases obtained for the four products, what is the proportion of those that are completely similar, i.e., that cover the same transitions, in the same order, and with the same parameters. It is obtained that only 5,26% of the test cases are redundant and therefore, can be reused for all products; 23,57% of the test cases can be used to test products 1, 2 and 3; 29,03% of test cases would be used for products 3 and 4. Analysing manually the generated test cases for the four products shows that even when the sequence transition id is similar, to reuse test cases among the derived products is necessary to modify manually the parameters evaluated in the guard conditions.

More information about the comparison done between test cases taking into count the sequence is shown in Annex A.

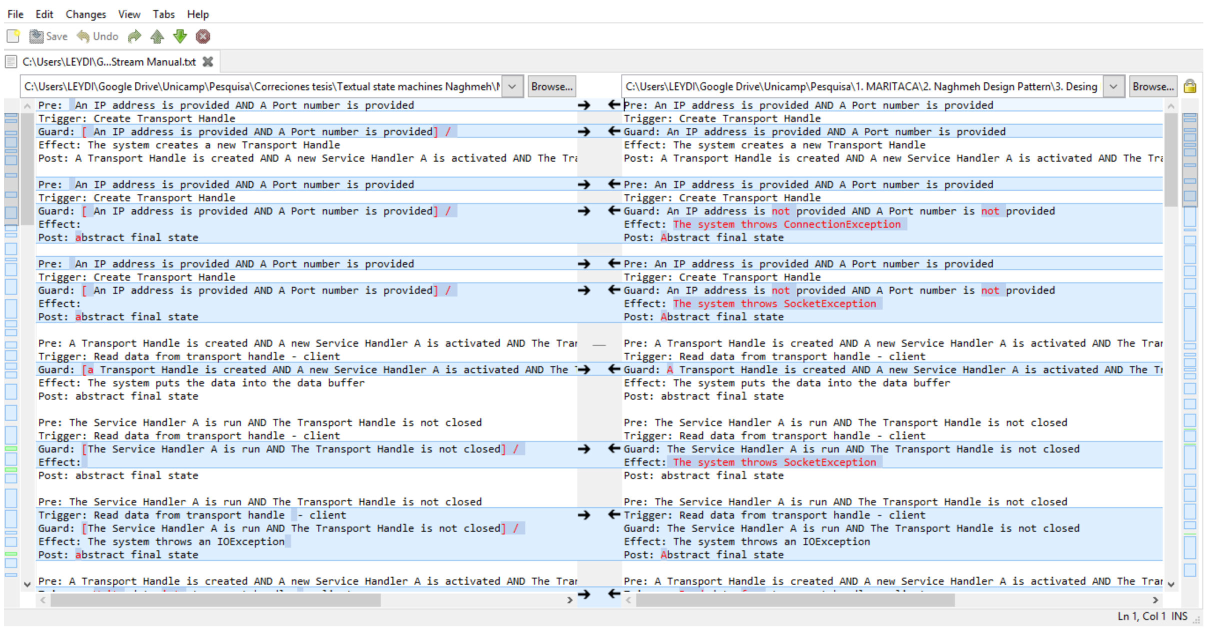
7.3 Obtained results using the Connection Handle Design Pattern SPL

The Connection Handle Design Pattern SPL was used to apply the MBPTA in Chapter 6. For three different distributed applications manually and automatically state models were extracted (See TABLE 6.2) from the CHDP SPL.

To answer Q1, similar to the evaluation presented in Section 7.2, the automatically extracted state models using MARITACA were compared with manually generated state models to evaluate if this tool can be to extract state models by using the transformation rules proposed in Section 4.2.3. For the Stream-based application, 25 states were created and when the state model was refined 6 duplicate states were removed. For the HTTP-based application, 20 states were created and in the refinement process 3 duplicate states were removed. And for the Message-based application, 22 states were created and when the state model was refined 4 duplicate states were removed.

7.3.1 Comparison between manually and automatically generated state model

Using the procedure presented in Section 7.1 the manually generated and automatically extracted state machines were compared. Figures 7.6 and 7.7 show the obtained results for Stream-based and HTTP-based applications, using MELD tool.



C:\Users\LEYDI\G...Stream Manual.txt

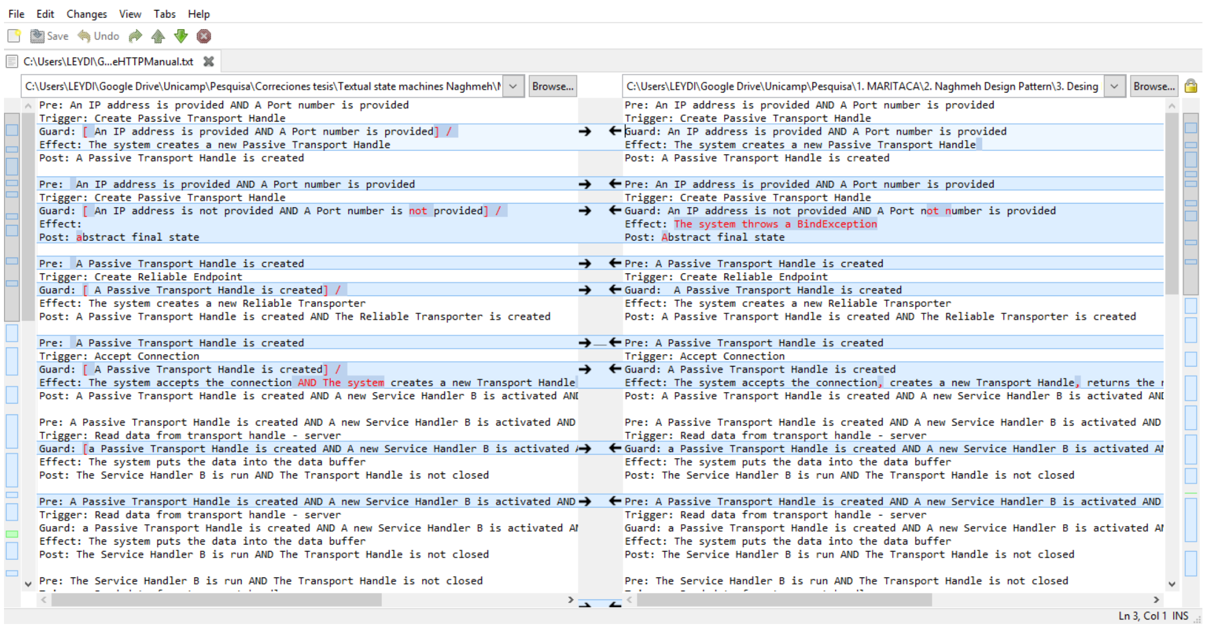
C:\Users\LEYDI\Google Drive\Unicamp\Pesquisa\Correciones tesis\Textual state machines Naghmeh\

```
Pre:  An IP address is provided AND A Port number is provided
Trigger: Create Transport Handle
Guard: [ An IP address is provided AND A Port number is provided] /
Effect: The system creates a new Transport Handle
Post: A Transport Handle is created AND A new Service Handler A is activated AND The Tra

Pre:  An IP address is provided AND A Port number is provided
Trigger: Create Transport Handle
Guard: [ An IP address is provided AND A Port number is provided] /
Effect:
Post: abstract final state

Pre:  An IP address is provided AND A Port number is provided
Trigger: Create Transport Handle
Guard: [ An IP address is provided AND A Port number is provided] /
Effect:
Post: abstract final state

Pre: A Transport Handle is created AND A new Service Handler A is activated AND The Trar
Trigger: Read data from transport handle - client
Guard: [a Transport Handle is created AND A new Service Handler A is activated AND The
Effect: The system puts the data into the data buffer
Post: abstract final state

Pre: The Service Handler A is run AND The Transport Handle is not closed
Trigger: Read data from transport handle - client
Guard: [The Service Handler A is run AND The Transport Handle is not closed] /
Effect:
Post: abstract final state

Pre: The Service Handler A is run AND The Transport Handle is not closed
Trigger: Read data from transport handle  - client
Guard: [The Service Handler A is run AND The Transport Handle is not closed] /
Effect: The system throws an IOException
Post: abstract final state

Pre: A Transport Handle is created AND A new Service Handler A is activated AND The Trar
```

C:\Users\LEYDI\Google Drive\Unicamp\Pesquisa\1. MARITACA\2. Naghmeh Design Pattern\3. Desing

```
Pre: An IP address is provided AND A Port number is provided
Trigger: Create Transport Handle
Guard: An IP address is provided AND A Port number is provided
Effect: The system creates a new Transport Handle
Post: A Transport Handle is created AND A new Service Handler A is activated AND The Tra

Pre: An IP address is provided AND A Port number is provided
Trigger: Create Transport Handle
Guard: An IP address is not provided AND A Port number is not provided
Effect: The system throws ConnectionException
Post: Abstract final state

Pre: An IP address is provided AND A Port number is provided
Trigger: Create Transport Handle
Guard: An IP address is not provided AND A Port number is not provided
Effect: The system throws SocketException
Post: Abstract final state

Pre: A Transport Handle is created AND A new Service Handler A is activated AND The Trar
Trigger: Read data from transport handle - client
Guard: A Transport Handle is created AND A new Service Handler A is activated AND The Tr
Effect: The system puts the data into the data buffer
Post: abstract final state

Pre: The Service Handler A is run AND The Transport Handle is not closed
Trigger: Read data from transport handle - client
Guard: The Service Handler A is run AND The Transport Handle is not closed
Effect: The system throws SocketException
Post: abstract final state

Pre: The Service Handler A is run AND The Transport Handle is not closed
Trigger: Read data from transport handle - client
Guard: The Service Handler A is run AND The Transport Handle is not closed
Effect: The system throws an IOException
Post: Abstract final state

Pre: A Transport Handle is created AND A new Service Handler A is activated AND The Trar
```

Figure 7.6: Differences between manually generated and automatically extracted textual state models for Stream-based application



C:\Users\LEYDI\G...eHTTPManual.txt

C:\Users\LEYDI\Google Drive\Unicamp\Pesquisa\Correciones tesis\Textual state machines Naghmeh\

```
Pre: An IP address is provided AND A Port number is provided
Trigger: Create Passive Transport Handle
Guard: [ An IP address is provided AND A Port number is provided] /
Effect: The system creates a new Passive Transport Handle
Post: A Passive Transport Handle is created

Pre:  An IP address is provided AND A Port number is provided
Trigger: Create Passive Transport Handle
Guard: [ An IP address is not provided AND A Port number is not provided] /
Effect:
Post: abstract final state

Pre:  A Passive Transport Handle is created
Trigger: Create Reliable Endpoint
Guard: [ A Passive Transport Handle is created] /
Effect: The system creates a new Reliable Transporter
Post: A Passive Transport Handle is created AND The Reliable Transporter is created

Pre:  A Passive Transport Handle is created
Trigger: Accept Connection
Guard: [ A Passive Transport Handle is created] /
Effect: The system accepts the connection AND The system creates a new Transport Handle
Post: A Passive Transport Handle is created AND A new Service Handler B is activated ANI

Pre: A Passive Transport Handle is created AND A new Service Handler B is activated AND
Trigger: Read data from transport handle - server
Guard: [a Passive Transport Handle is created AND A new Service Handler B is activated A
Effect: The system puts the data into the data buffer
Post: The Service Handler B is run AND The Transport Handle is not closed

Pre: A Passive Transport Handle is created AND A new Service Handler B is activated AND
Trigger: Read data from transport handle - server
Guard: a Passive Transport Handle is created AND A new Service Handler B is activated AI
Effect: The system puts the data into the data buffer
Post: The Service Handler B is run AND The Transport Handle is not closed

Pre: The Service Handler B is run AND The Transport Handle is not closed
```

C:\Users\LEYDI\Google Drive\Unicamp\Pesquisa\1. MARITACA\2. Naghmeh Design Pattern\3. Desing

```
Pre: An IP address is provided AND A Port number is provided
Trigger: Create Passive Transport Handle
Guard: An IP address is provided AND A Port number is provided
Effect: The system creates a new Passive Transport Handle
Post: A Passive Transport Handle is created

Pre: An IP address is provided AND A Port number is provided
Trigger: Create Passive Transport Handle
Guard: An IP address is not provided AND A Port not number is provided
Effect: The system throws a BindException
Post: Abstract final state

Pre: A Passive Transport Handle is created
Trigger: Create Reliable Endpoint
Guard:  A Passive Transport Handle is created
Effect: The system creates a new Reliable Transporter
Post: A Passive Transport Handle is created AND The Reliable Transporter is created

Pre: A Passive Transport Handle is created
Trigger: Accept Connection
Guard: A Passive Transport Handle is created
Effect: The system accepts the connection, creates a new Transport Handle, returns the r
Post: A Passive Transport Handle is created AND A new Service Handler B is activated ANI

Pre: A Passive Transport Handle is created AND A new Service Handler B is activated AND
Trigger: Read data from transport handle - server
Guard: a Passive Transport Handle is created AND A new Service Handler B is activated AI
Effect: The system puts the data into the data buffer
Post: The Service Handler B is run AND The Transport Handle is not closed

Pre: A Passive Transport Handle is created AND A new Service Handler B is activated AND
Trigger: Read data from transport handle - server
Guard: a Passive Transport Handle is created AND A new Service Handler B is activated AI
Effect: The system puts the data into the data buffer
Post: The Service Handler B is run AND The Transport Handle is not closed

Pre: The Service Handler B is run AND The Transport Handle is not closed
```

Figure 7.7: Differences between manually generated and automatically extracted textual state models for HTTP-based application

This comparison can help to answer the Q2 because, using the informal interpretation presented in 7.2.1. The family of distributed applications presents few differences (punctuation marks, spaces, lowercase, and uppercase) between the manually generated and automatically extracted state models. MARITACA can generate syntactically correct state models according to the format defined in Section 4.2.3.

When state models for this SPL were created, some issues were found and corrected. One of these issues were related to incompleteness in rule 8 for the treatment of exceptions when the ABORT keyword is used (See section 4.2.3). This rule was modified and adapted to MARITACA tool.

Another issue was related to words written using uppercase letters in the use case descriptions. The PoS tagger tagged these works as a Proper noun (NNP) and not as a Singular noun (NN), and when MARITACA reads the tagged description it applies a different rule for each tag. The words in the use cases description were corrected, use cases description was tagged and state models were generated again.

Table 7.4 presents a summary of states, transitions, events and parameters obtained for the three refined state machines.

Table 7.4: Summary of obtained results for the three state machines

Application	**States**	**Transitions**	**Events**	**Parameters**
Stream-based application	19	43	25	13
HTTP-based application	17	35	21	8
Message-based application	18	39	23	11

7.3.2 Evaluation of generated test cases

As soon as the state machines were refined taking in count their context classes, the test cases were generated using Parteg tool. The results are presented in Table 7.5.

Table 7.5: Generated test cases for the four products

Application	**Test Cases All-Transitions criteria**	**Test cases Decision Coverage criteria**
Stream-based application	25	49
HTTP-based application	19	38
Message-based application	23	43

The Figure 7.8 shows the similarity analysis done for generated test cases to determine whether there is or there is no redundancy among them.

Test cases from V1 to V19 from the HTTP-based application, test cases are related to V20 to V44 from the Message-based application and test cases are related to V45 to V69 are related to the Stream-based application.

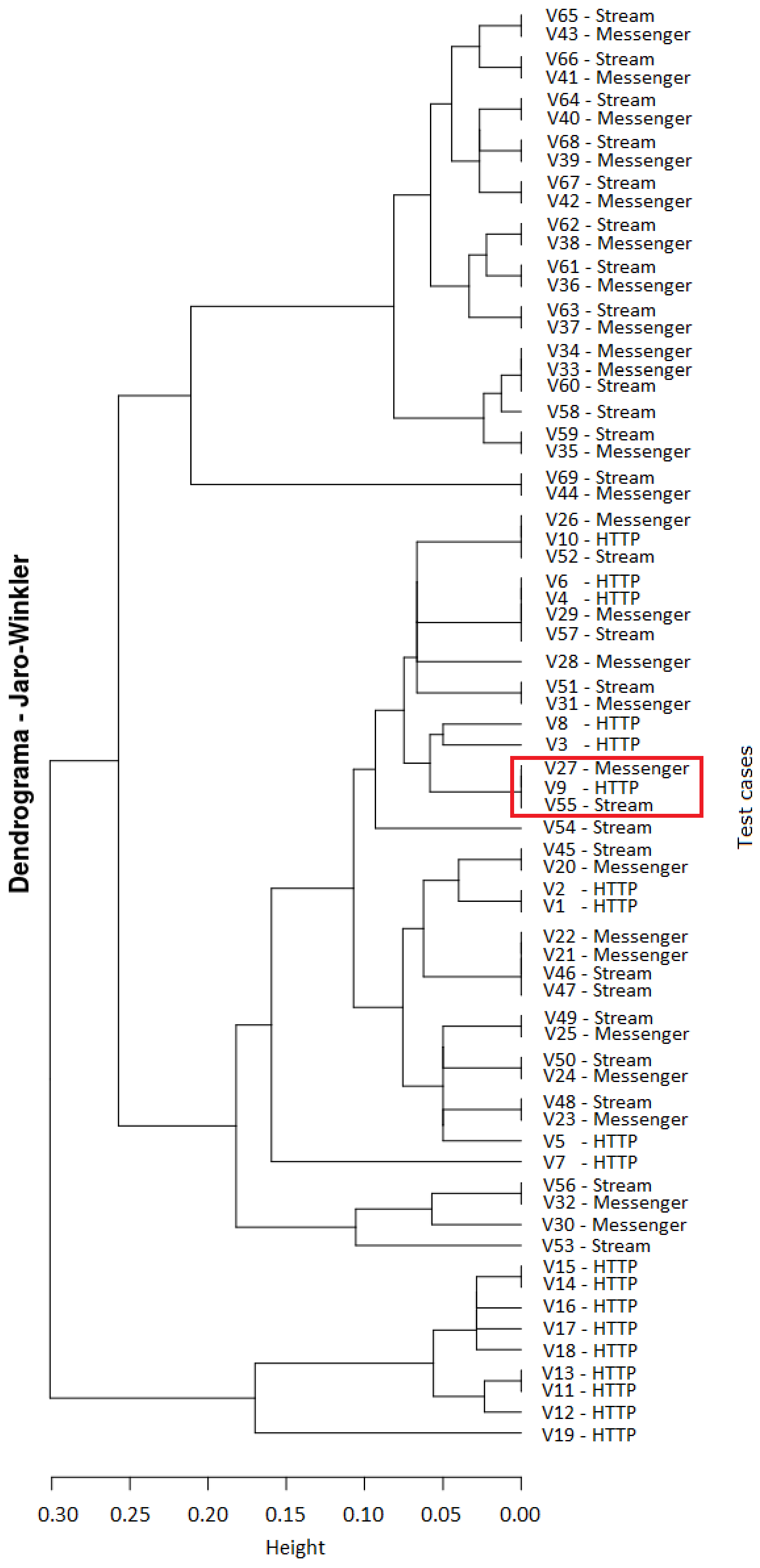


Figure 7.8: Dendrogram of generated test cases

TABLE 7.6 shows an example of test cases 9, 27 and 55 with the same sequence of transitions, but 9 and 55 are different of 27 in terms of the evaluated parameter in the guard.

Table 7.6: Excerpt of sequence of transitions for the three applications

HTTP-bases	**Message-based**	**Stream-based**
test9() InitializaCEv CreateTransportHandleCEv IOExceptionReadCEv	**test27()** InitializaCEv CreateTransportHandleCEv IOExceptionReadCEv	**test55()** InitializaCEv CreateTransportHandleCEv IOExceptionReadCEv
IOExceptionReadCEv chanelIsEmpty == false	**IOExceptionReadCEv**	**IOExceptionReadCEv** chanelIsEmpty == false

The proportion of redundant test cases to answer Q3 is: 69 test cases were generated for the three applications. Only 10,76% of the test cases can be reused for the three applications (these test cases are the same in the three applications: sequence and parameters); 25,25% of the test cases can be used to test Stream-based and HTTP-based applications. Analysing manually the generated test cases for the three applications shows that even when the sequence transitions id is similar, to reuse test cases among the derived products is necessary to manually modify the parameters evaluated in the guard conditions. But this attempt to reuse similar test cases could involve a manual analysis of all generated test cases that exercise a given transition and to modify the guard condition.

Therefore, in this case, most test cases were relevant as they exercised the specific features for each product.

Whether test cases of Stream-based application are used to test the Message-based application, some specifically created test cases for the Stream-based application could not test important features of Message-based application because they could be incomplete or test cases created to test the Stream-based application could be not executed because features could no exist in the Message-based application.

More information about the comparison done between test cases taking into count the sequence is shown in Annex B.

7.4 Final considerations

This Chapter presented the obtained results by using the Model-based product testing approach (MBPTA) with Microwave oven SPL and Connection Handle Design Pattern SPL.

After use the MBPTA the research questions defined in Section 1.2 can be answered: *"i) how to describe use cases in order to cope with the variability present in SPL requirements?"* The SPL's variability is captured in the use case descriptions in sections as the Associate feature, Cardinality feature, Use case type, variation flows (using {} tags). Moreover, the Product Map and Traceability Matrix are used to link the features with products and use cases respectively. Since use cases are written in natural language, they are prone to ambiguity and inconsistency, the restrictions rules used in MBPTA to write the use cases can help to reduce ambiguities and eases the state model extraction.

To answer the research question *"ii) how to take into account the constraints among features, as well as the features selected for a specific product?"*, two artifacts are used: the Feature Model (FM), which show constraints such as "requires", "excludes", "xor", and the Product Map (PM). The latter allows associating features with products. When a feature is required by another feature this association must be represented in the PM.

The set of transformation rules and MARITACA tool can be used to generate state models from requirements written in form of use case description, then the question *" iii) How to automatically generate the state model from the textual requirements?"* is resolved.

In research question *"iv) How to refine the skeleton test model automatically extracted from the use cases in order to obtain a model amenable to a test case generation tool?"* To the refinement process, a set of steps were suggested. Using these steps and with the context class, the state machines can be refined to be used in a MBT tool to generate test cases.

This approach can be used to help in the generations of models from textual requirements and to generate test cases from these generated state models. An important aspect is that to test new products, new test models, and test cases must be generated from domain artifacts. The reuse is not about application artifacts (test models and the test cases) but about domain artifacts (Use cases descriptions, PM and TM). There is a certain degree of effort in the refinement process of new state models, but this can be rewarded over time as engineers acquire expertise in the refining process of state machines. In software product line testing the reuse of test cases is not straightforward because the existing test cases must be maintained and evolved as each new product is created. Manually maintain and create the test cases for new products requires a significant work, and over time, the number of test cases increases so much (some of them obsolete) that is easier to simply ignore them and create new test cases. It is not possible to generalize the obtained results for all SPLs because sufficient experiments were not made. It is necessary to apply the MBPTA in a real context to find validation threats and improvement opportunities. Next Chapter presents the conclusions and future works.

Chapter 8

Conclusion and Future works

The software product lines have been used recently both in industry and academia, due to commonalities among the products that can be derived from them. Considerable benefits as reducing costs, improving software quality and productivity are provided by this paradigm. As in the development process for single systems, testing plays an important role. Generation of an effective set of test cases to the derived products is a challenge in software product lines context because large number of required test must be created.

Model-based testing (MBT) is an useful technique because it allows creating models from requirements which can derive test case. Moreover, when the test engineers are creating the test models, ambiguities and errors may be detected. Thus, it is cheaper to correct them during the development of the test model than to correct them in later development phases.

To describe the behavior of a system, state machines are the most commonly used notation. From these state machines, test suites are often automatically generated by using model-based testing tools. However, to create the state machines from requirements are manually created, expensive, and error-prone task. Therefore, transitions from requirements to state machines would help to reduce the effort of the test engineers.

This dissertation presented an approach to support the automatic generation of state models from use cases descriptions. The proposed approac hMBPTA aims to help in the generation of state models from use cases requirements. The created state models can be used to automatically generate test cases using a MBT tool. The use cases descriptions are written in a controlled natural language, in which restrictions rules are defined in order to reduce ambiguity and facilitate automated analysis. By using the domain artifacts and applying a set of transformation rules, an initial version of a state model for a specific product can be generated.

The MBTPA can support the traceability between the requirements and test cases because the test models are generated from the requirements. Thus, when the requirements change, the test models will change accordingly. The generation of a initial version of a state machine could help the use of MBT in software product line context.

To support the extraction process the MARITACA tool was also prototyped. This tool uses the domain artifacts, the transformation rules and automatically generates a textual version of state machine. These state machines must be refined by test engineers and in order to make them amenable to be treated by a test case generation tool. It is also worth noting that the extraction procedure is highly dependent on the description of use cases. So, to reduce effort in the refinement phase, test engineers must be careful with the way they write the use cases, to avoid ambiguities and improve state machine connectivity.

8.1 Contributions

The main contributions of our work is the MBPTA, an approach for model-based system testing of products from software product lines. Moreover, two papers were produced and accepted in international conferences.

Another important contribution was the work on the Connection Handle Design Pattern performed in collaboration with the researches of Department of Informatics Engineering of the University of Coimbra.

8.2 Future works

A feasible future work is to perform case studies with real SPL to obtain more information about the advantages of the proposed MBPTA.

Another future work is to improve the extraction process to deal with the exceptions and self-loop in the extracted state machine. As well as, to implement aspects reducing the use of IF-THEN-ENDIF keywords, because the nesting of these keywords can make a use case description very confusing.

Next, it is under consideration the use of semantic analysis to improve the natural language processing of the use case descriptions. In this way, it is expected to reduce redundant states generation, decreasing the refinement effort.

It is also envisaged to develop a use case editor to guide users in writing their use cases. This editor could help the test engineers to reduce errors, incompleteness and inconsistencies in the use case descriptions. The Product Map can be implemented as extension for the tool to guide user in the selection of features for a specific product.

Bibliography

[1] Devasses - design, verification and validation of large-scale, dynamic service systems. Available on http://www.devasses.eu/, 2017. Accessed: 03-11-2017.

[2] Sandra António, João Araújo, and Carla Silva. Adapting the i* framework for software product lines. In *Proceedings of Advances in Conceptual Modeling - Challenging Perspectives*, pages 286–295, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[4] Antonia Bertolino and Stefania Gnesi. Use case-based testing of product lines. *ACM SIGSOFT Software Engineering Notes*, 28:355–358, 2003.

[5] Antonia Bertolino and Stefania Gnesi. Pluto: A test methodology for product families. In *International Workshop on Software Product-Family Engineering: 5th International Workshop*, volume 28, pages 181–197. Springer Berlin Heidelberg, 2004.

[6] Robert Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[7] Paul Clements and Linda Northrop. Salion, inc.: A software product line case study. Technical report cmu/sei-2002-tr-038, SEI - Software Engineering Institute, 2002.

[8] Alistair Cockburn. Writing effective use cases, the crystal collection for software professionals. *Addison-Wesley Professional Reading*, 2001.

[9] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John McGregor, Eduardo Santana De Almeida, and Silvio Romero de Lemos Meira. A systematic mapping study of software product lines testing. *Information and Software Technology*, 53(5):407–423, 2011.

[10] Siddhartha Dalal, Ashish Jain, Nachimuthu Karunanithi, JM Leaton, Christopher Lott, Gardner Patton, and Bruce Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, pages 285–294, New York, NY, USA, 1999. ACM.

[11] Stanley Davis. *Future perfect.* Addison-Wesley, 1987.

[12] Raphael Pereira de Oliveira, Emilio Insfran, Silvia Abrahão, Javier Gonzalez-Huerta, David Blanes, Sholom Cohen, and Eduardo Santana de Almeida. A feature-driven requirements engineering approach for software product lines. In *Software Components, Architectures and Reuse (SBCARS), 2013 VII Brazilian Symposium on*, pages 1–10. IEEE, 2013.

[13] Ivan do Carmo Machado, John McGregor, and Eduardo Santana de Almeida. Strategies for testing products in software product lines. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–8, 2012.

[14] Mohamed El-Attar and James Miller. Agaduc: Towards a more precise presentation of functional requirement in use case mod. In *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 346–353, Washington, DC, USA, 2006. IEEE Computer Society.

[15] Emelie Engström. Regression test selection and product line system testing. In *Proceedings of Third International Conference on Software Testing, Verification and Validation. Frace, Paris.*, pages 512–515. IEEE, 2010.

[16] Leydi Erazo, Eliane Martins, and Juliana Galvani Greghi. Modeling dependable product-families: From use cases to state machine models. In *Proceedings of Dependable Computing (LADC), 2016 Seventh Latin-American Symposium on*, pages 131–134, Cali, Colombia, 2016. IEEE.

[17] Leydi Erazo, Eliane Martins, and Juliana Galvani Greghi. Maritaca - from textual use case descriptions to behavior models. In *Dependable Systems and Networks Workshop (DSN-W), 2017 47th Annual IEEE/IFIP International Conference on*, pages 83–90, Denver, CO, USA, 2017. IEEE.

[18] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures.* Addison-Wesley Professional, USA, 2004.

[19] Hassan Gomaa and Michael Shin. A multiple-view meta-modeling approach for variability management in software product lines. In *International Conference on Software Reuse*, pages 274–285. Springer, 2004.

[20] Wael H Gomaa and Aly A Fahmy. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13), 2013.

[21] Juliana Galvani Greghi, Eliane Martins, and Ariadne Maria Brito Rizzoni Carvalho. Semi-automatic generation of extended finite state machines from natural language standard documents. In *Dependable Systems and Networks Workshops (DSN-W), 2015 IEEE International Conference on*, pages 45–50. IEEE, 2015.

[22] Günter Halmans and Klaus Pohl. Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2(1):15–36, 2003.

[23] Naghmeh Ivaki, Nuno Laranjeiro, and Filipe Araujo. A design pattern for reliable http-based applications. In *Services Computing (SCC), 2015 IEEE International Conference on*, pages 656–663. IEEE, 2015.

[24] Michel Jaring and Jan Bosch. Representing variability in software product lines: A case study. In *Proceedings of the Second International Conference on Software Product Lines*, pages 15–36, London, UK, 2002. Springer-Verlag.

[25] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report cmu/sei-90-tr-21, Software Engineering Institute, Carnegie-Mellon University. Pittsburgh Pennsylvania, 1990.

[26] Peter Knauber and Giancarlo Succi. Perspectives on software product lines. In *Proceedings of first international workshop on software product lines: economics, architectures, and implications. New York, USA*, pages 29–33. ACM SIGSOFT Software Engineering Notes, 2001.

[27] Jihyun Lee, Sungwon Kang, and Danhyung Lee. A survey on software product line testing. In *Proceedings of the 16th International Software Product Line Conference*, pages 31–40, New York, NY, USA, 2012. ACM.

[28] Geoffrey Leech. Adding linguistic annotation. In *Developing linguistic corpora: a guide to good practice*, pages 17–29. Oxbow Books, 2005.

[29] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[30] Ricardo Linden. Técnicas de agrupamento. *Revista de Sistemas de Informação da FSMA*, pages 18–36, 2009.

[31] Andreas Metzger and Klaus Pohl. Software product line engineering and variability management: Achievements and challenges. In *Proceedings of the on Future of Software Engineering*, pages 70–84, New York, NY, USA, 2014. ACM.

[32] Marc H Meyer and Alvin P Lehnerd. *The power of product platforms*. Simon and Schuster, 1997.

[33] Fionn Murtagh. R: Hierarchical clustering. Available on https://stat.ethz.ch/R-manual/R-devel/library/stats/html/hclust.html. Accessed: 05-11-2017.

[34] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. A requirement-based approach to test product families. In *International Workshop on Software Product-Family Engineering*, pages 198–210. Springer, 2004.

[35] Clémentine Nebut, Yves Le Traon, and Jean-Marc Jézéquel. System testing of product lines: From requirements to test cases. In *Software Product Lines*, pages 447–477. Springer, 2006.

[36] Erika Mir Olimpiew. *Model-based Testing for Software Product Lines.* PhD thesis, George Mason University, Fairfax, VA, USA, 2008.

[37] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. *Software Product Lines: Going Beyond*, pages 196–210, 2010.

[38] Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. Moso-polite: Tool support for pairwise and model-based software product line testing. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 79–82, New York, NY, USA, 2011. ACM.

[39] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., 2005.

[40] Beatriz Pérez Lamancha, Macario Polo Usaola, and Mario Piattini. Software product line testing - a systematic review. In *Proceedings of 4th International Conference on Software and Data Technologies. Sofia, Bulgaria*, pages 23–30. Springer, 2009.

[41] Andreas Reuys, Erik Kamsties, Klaus Pohl, and Sacha Reis. Model-based system testing of software product families. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, pages 519–534, Berlin, Heidelberg, 2005. Springer-Verlag.

[42] Andreas Reuys, Sacha Reis, Erik Kamsties, and Klaus Pohl. The scented method for testing software product lines. In *Software Product Lines*, pages 479–520. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[43] Konrad Rieck and Christian Wressnegger. Harry: a tool for measuring string similarity. *Journal of Machine Learning Research*, 17:1–5, 2016.

[44] Gisele Rodrigues Mesquita Ferreira, Cecília MF Rubira, and Rogério de Lemos. Explicit representation of exception handling in the development of dependable component-based systems. In *High Assurance Systems Engineering, 2001. Sixth IEEE International Symposium on*, IEEE Conference Proceedings, pages 182–193, USA, 2001. IEEE.

[45] Stephane S. Some. Beyond scenarios: Generating state models from use cases. In *ICSE 2002 Workshop Scenarios and state machines: models, algorithms, and tools*, 2002.

[46] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[47] David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-based Software Development Process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[48] Stephan Weißleder. *Test models and coverage criteria for automatic model-based test generation with UML state machines.* PhD thesis, Humboldt University of Berlin, 2010.

[49] Stephan Weißleder, Dehla Sokenou, and Holger Schlingloff. Reusing state machines for automatic test generation in product lines. *Model-Based Testing in Practice (MoTiP), Fraunhofer IRB Verlag*, pages 19–28, 2008.

[50] Kai Willadse. Meld. Available on http://meldmerge.org/, 2011. Accessed: 11-10-2017.

[51] Tao Yue, Shaukat Ali, and Lionel Briand. Automated transition from use cases to uml state machines to support state-based testing. In *Proceedings of the 7th European Conference on Modelling Foundations and Applications*, pages 115–131, Berlin, Heidelberg, 2011. Springer-Verlag.

[52] Tao Yue, Lionel C. Briand, and Yvan Labiche. A use case modeling approach to facilitate the transition towards analysis models: Concepts and empirical evaluation. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 484–498, Berlin, Heidelberg, 2009. Springer-Verlag.

[53] Tao Yue, Lionel C. Briand, and Yvan Labiche. An automated approach to transform use cases into activity diagrams. In *Proceedings of the 6th European Conference on Modelling Foundations and Applications*, ECMFA'10, pages 337–353, Berlin, Heidelberg, 2010. Springer-Verlag.

[54] Tao Yue, Lionel C. Briand, and Yvan Labiche. atoucan: An automated framework to derive uml analysis models from use case models. *ACM Trans. Softw. Eng. Methodol.*, pages 13:1–13:52, 2015.

Appendix A

Test cases similarity - Microwave oven system

Similar test cases in terms of the sequence of transition ids are clustered and shown in the following tables. Unique test cases for each product are not presented here.

Table A.1: Similar test cases in terms of the sequence of transition ids for the Products 1, 2, 3 and 4

Product 1	**Product 2**	**Product 3**	**Product 4**
test11()	**test30()**	**test48()**	**test66()**
InitEv	InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv	CloseDoorEv
PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv
SetCookingTimeEv	SetCookingTimeEv	SetCookingTimeEv	SetCookingTimeEv
OpenEv	OpenEv	OpenEv	OpenEv
test12()	**test31()**	**test49()**	**test69()**
InitEv	InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv	CloseDoorEv
PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
test13()	**test32()**	**test50()**	**test70()**
InitEv	InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv	CloseDoorEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv

Product 1	Product 2	Product 3	Product 4
test14()	**test33()**	**test51()**	**test71()**
InitEv	InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv	CloseDoorEv
OpenEv	OpenEv	OpenEv	OpenEv
test15()	**test34()**	**test52()**	**test72()**
InitEv	InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv	InsertFoodEv
RemoveEv	RemoveEv	RemoveEv	RemoveEv
test16()	**test35()**	**test53()**	**test73()**
InitEv	InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv	InsertFoodEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
test17()	**test36()**	**test54()**	**test75()**
InitEv	InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv	OpenEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
test18()	**test37()**	**test55()**	**test74()**
InitEv	InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv	OpenEv
CloseDoorEv	CloseDoorEv	CloseDoorEv	CloseDoorEv
test19()	**test38()**	**test56()**	**test76()**
InitEv	InitEv	InitEv	InitEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv

Table A.2: Similar test cases in terms of the sequence of transition ids for the Products 1, 2 and 4

Product 1	Product 2	Product 4
test1()	**test20()**	**test57()**
InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv
PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv
SetCookingTimeEv	SetCookingTimeEv	SetCookingTimeEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
OpenEv	OpenEv	OpenEv
RemoveEv	RemoveEv	RemoveEv
CloseDoorEv	CloseDoorEv	CloseDoorEv
OpenEv	OpenEv	OpenEv

Product 1	**Product 2**	**Product 4**
test2()	**test21()**	**test58()**
InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv
PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv
SetCookingTimeEv	SetCookingTimeEv	SetCookingTimeEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
OpenEv	OpenEv	OpenEv
RemoveEv	RemoveEv	RemoveEv
CloseDoorEv	CloseDoorEv	CloseDoorEv
CancelEv	CancelEv	CancelEv
test3()	**test22()**	**test59()**
InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv
PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv
SetCookingTimeEv	SetCookingTimeEv	SetCookingTimeEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
OpenEv	OpenEv	OpenEv
RemoveEv	RemoveEv	RemoveEv
CloseDoorEv	CloseDoorEv	CloseDoorEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
test4()	**test23()**	**test60()**
InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv
PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv
SetCookingTimeEv	SetCookingTimeEv	SetCookingTimeEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
OpenEv	OpenEv	OpenEv
RemoveEv	RemoveEv	RemoveEv
CancelEv	CancelEv	CancelEv
test5()	**test24()**	**test61()**
InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv
PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv
SetCookingTimeEv	SetCookingTimeEv	SetCookingTimeEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
OpenEv	OpenEv	OpenEv
RemoveEv	RemoveEv	RemoveEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv

Product 1	**Product 2**	**Product 4**
test7()	**test26()**	**test63()**
InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv
PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv
SetCookingTimeEv	SetCookingTimeEv	SetCookingTimeEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
OpenEv	OpenEv	OpenEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
test8()	**test27()**	**test64()**
InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv
PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv
SetCookingTimeEv	SetCookingTimeEv	SetCookingTimeEv
StartCookingFoodEv	StartCookingFoodEv	StartCookingFoodEv
CancelEv	CancelEv	CancelEv
test10()	**test29()**	**test29()**
InitEv	InitEv	InitEv
OpenEv	OpenEv	OpenEv
InsertFoodEv	InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv	CloseDoorEv
PressCookingTimeEv	PressCookingTimeEv	PressCookingTimeEv
SetCookingTimeEv	SetCookingTimeEv	SetCookingTimeEv
CancelEv	CancelEv	CancelEv

Table A.3: Similar test cases in terms of the sequence of transition ids for the Products 1 and 2

Product 1	**Product 2**
test9()	**test28()**
InitEv	InitEv
OpenEv	OpenEv
InsertFoodEv	InsertFoodEv
CloseDoorEv	CloseDoorEv
PressCookingTimeEv	PressCookingTimeEv
SetCookingTimeEv	SetCookingTimeEv
StartCookingFoodEv	StartCookingFoodEv
FinishCookingEv	FinishCookingEv

Appendix B

Test cases similarity -CHDP

Similar test cases are clustered taking into count the sequence of transition ids. Unique test cases for each product are not presented here.

Table B.1: Similar test cases in terms of the sequence of transition ids for the three applications

HTTP-based	**Message-based**	**Stream-based**
test4()	**test29()**	**test57()**
InitializaCEv	InitializaCEv	InitializaCEv
CreateTransportHandleCEv	CreateTransportHandleCEv	CreateTransportHandleCEv
CloseTransportHCEv	CloseTransportHCEv	CloseTransportHCEv
test10()	**test26()**	**test52()**
InitializaCEv	InitializaCEv	InitializaCEv
CreateTransportHandleCEv	CreateTransportHandleCEv	CreateTransportHandleCEv
WriteDataCEv	WriteDataCEv	WriteDataCEv

Table B.2: Similar test cases in terms of the sequence of transition ids for two applications

Message-based	**Stream-based**
test20()	**test45()**
InitializaSEv	InitializaSEv
CreateTransportHandleCEv	CreateTransportHandleCEv
ReadDataCEv	ReadDataCEv
IOExceptionReadCEv	IOExceptionReadCEv
ExitCEv	ExitCEv
test22()	**test46()**
InitializaSEv	InitializaSEv
CreateTransportHandleCEv	CreateTransportHandleCEv
ReadDataCEv	ReadDataCEv
CloseTransportHCEv	CloseTransportHCEv
ExitCEv	ExitCEv

Message-based	**Stream-based**
test23() InitializaSEv CreateTransportHandleCEv ReadDataCEv IOExceptionWriteCEv	**test48()** InitializaSEv CreateTransportHandleCEv ReadDataCEv IOExceptionWriteCEv
test24() InitializaSEv CreateTransportHandleCEv ReadDataCEv ReadDataCEv	**test50()** InitializaSEv CreateTransportHandleCEv ReadDataCEv ReadDataCEv
test25() InitializaSEv CreateTransportHandleCEv ReadDataCEv WriteDataCEv	**test49()** InitializaSEv CreateTransportHandleCEv ReadDataCEv WriteDataCEv
test32() InitializaSEv SocketExceptionCEv ExitSEv	**test51()** InitializaSEv SocketExceptionCEv ExitSEv
test33() InitializaSEv ConnectionExceptionCEv ExitSEv	**test560()** InitializaSEv ConnectionExceptionCEv ExitSEv
test34() InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ReadDataSEv CloseTransportHSEv ExitSEv	**test60()** InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ReadDataSEv CloseTransportHSEv ExitSEv
test35() InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ReadDataSEv IOExceptionReadSEv ExitSEv	**test59()** InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ReadDataSEv IOExceptionReadSEv ExitSEv
test36() InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ReadDataSEv IOExceptionReadSEv	**test61()** InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ReadDataSEv IOExceptionReadSEv

Message-based	Stream-based
test37() InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ReadDataSEv ReadDataSEv	**test63()** InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ReadDataSEv ReadDataSEv
test38() InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ReadDataSEv WriteDataSEv	**test62()** InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ReadDataSEv WriteDataSEv
test39() InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv IOExceptionReadSEv	**test68()** InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv IOExceptionReadSEv
test40() InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv WriteDataSEv	**test64()** InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv WriteDataSEv
test41() InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ExceptionSEv	**test66()** InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv ExceptionSEv
test42() InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv CloseTransportHSEv	**test67()** InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv CloseTransportHSEv
test43() InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv IOExceptionWriteSEv	**test65()** InitializaSEv CreatePassiveTransportHandleEv CreateReliableEndPointEv AcceptConnectionEv IOExceptionWriteSEv