Integrating Performance Analysis in Model Driven Software Product Line Engineering

By

Rasha Tawhid

A thesis submitted to
The Faculty of Graduate Studies and Research
in partial fulfilment of
the degree requirements of
Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University
Ottawa, Ontario, Canada
May 2012

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-89310-4

Our file Notre référence
ISBN: 978-0-494-89310-4

Abstract

A *Software Product Line* (SPL) is a set of similar software systems that share a common set of features satisfying a particular domain, and are built from a shared set of software assets using a common means of production. This research proposes to integrate performance analysis in the early phases of the model-driven development process of SPL. We start by adding generic performance annotations to the UML model representing the set of core reusable SPL assets using the MARTE Profile adopted by OMG. A model transformation realized in Atlas Transformation Language (ATL), derives the UML model of a specific product with concrete performance annotations from the SPL model, which is further transformed into a performance model by using a previously developed transformation called PUMA.

The automatic derivation of a specific product model based on a given feature configuration is enabled through the mapping between features from the feature model and their realizations in the design model. An efficient mapping technique is proposed that aims to minimize the amount of explicit feature annotations in the UML model of SPL. Implicit feature mapping is inferred during product derivation from the relationships between annotated and non-annotated model elements as defined in the UML metamodel. The mapping technique is used to derive automatically a given product model.

Performance is a run-time property of the deployed system and depends on other factors that are external to the design model, characterizing the underlying platforms and

run-time environment. Performance completions provide a means to extend the modeling constructs of a system by including the influence of these factors. The variability space of the performance completions is covered and represented through Performance Completion-feature model (PC-feature model).

Dealing manually with a large number of performance parameters annotating a UML+MARTE product model is an error-prone process. A model-driven user-friendly technique is proposed to automatically collect all generic performance parameters that need binding from the generated product model and present them to developers in a spreadsheet format, together with context and guiding information where each PC-feature is mapped to certain MARTE annotations corresponding to UML model elements in the product model.

To my sons Aly and Seif

Acknowledgements

Words alone cannot express the thanks I owe to my supervisor, Dr. Dorina Petriu, not only for her support, guidance, contributions of time and ideas to the research, but also for her understanding and consideration.

My most sincere and warm thanks go to my husband, Mohamed, and to my mom for their continuous support and encouragement.

To my sons, Aly and Seif, I dedicate this research to you both because without your patience, help and support in different ways over these past few years, this dissertation would not have been completed. Special thanks should be given to you!

Table of Contents

List of Tables

List of Figures

List of Acronyms

AOM	Aspect-Oriented Modeling
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
ATC	Atomic Transformation Code
ATL	Atlas Transformation Language
CBFM	Cardinality-Based Feature Modeling
CSM	Core Scenario Model
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
GQAM	Generic Quantitative Analysis Modeling
GRM	General Resource domain model
FODA	Feature-Oriented Domain Analysis
FORM	Feature-Oriented Reuse Method
LQN	Layered Queueing Model
MARTE	UML Profile for Modeling and Analysis of Real-Time and Embedded systems
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDSD	Model-Driven Software Development
MOF	Meta-Object Facility
MTTL	Model Template Transformation Language
NFP	Non-Functional Property
OCL	Object Constraint Language
OMG	Object Management Group

OVM	Orthogonal Variability Model
QVT	Query/View/Transformations
PAM	Performance Analysis Modeling
PC	Performance Completions
PIM	Platform Independent Model
PLUS	Product Line UML-Based Software Engineering
PSM	Platform Specific Model
PUMA	Performance by Unified Model Analysis
SAM	Schedulability Analysis Modeling
SPE	Software Performance Engineering
SPL	Software Product Line
SPT	UML Profile for Schedulability, Performance and Time
UML	Unified Modeling Language

Chapter 1: Introduction

Software Product Line (SPL) engineering is a software development approach that takes advantage of the commonality and variability between products from a family. Commonality includes the characteristics that are common to all SPL members, while variability distinguishes the members of a family from each other and needs to be explicitly modeled and separated from the common parts [CLA01]. The main challenge in the context of SPL approach is to model and manage the variability and to support the generation of specific products by reusing a set of core family assets. The SPL paradigm has received a lot of attention in recent years since it aims to improve productivity and decrease realization times by gathering the analysis, design and implementation activities of a family of systems. It is based on the reuse of core assets instead of starting from scratch for every family member. A software product line (or a family of systems – we use both terms as synonyms in this research) can be defined as "a set of software intensive systems sharing a common, managed set of features that satisfy specific needs of a particular market or mission, and that are developed from a common set of core assets in a prescribed way" Clements and Northrop [CLE01].

There are two major development processes in SPL engineering: a) *domain engineering* for analyzing the commonality and variability between members of the product line and establishing reusable SPL assets, and b) *application engineering* for deriving manually or automatically individual products that are SPL members from the

reusable assets defined in the domain engineering processes. A very important concept in SPL is that of *feature*, used to represent reusable characteristics of a product line. Features are used to differentiate among members of the product line and to define the commonality and variability in the functionality offered by different SPL products. Feature modeling is essential for both variability management and product derivation where a specific product member is defined by a unique combination of features. Mapping features from the feature model to model elements realizing them in design model is necessary for automatic product derivation.

Model-driven development (MDD) improves software development by capturing the key features of a system in models which are developed and refined as the system is created [VOE07]. MDD promotes the use of models to cope with the complexity of current software systems. Models aim to capture every important aspect of software systems through their entire lifecycle. MDD's objective goes beyond considering models as just auxiliary documentation artifacts, models can be gradually refined all the way to implementation by means of model transformations. Models can be considered as primary building blocks for building software systems [PER08]. Several model-based software development paradigms are developed such as OMG's Model-Driven Architecture (MDA) [OMG04] and Microsoft's Software Factories [GRE04].

The Unified Modeling Language (UML) is a well-known wide-spread notation for modeling software systems during the development process [OMG07]. It includes a graphical notation to create visual models of software-intensive systems by providing several kinds of diagrams supporting multiple views of a system (such as structural and behavioral, logical and physical, interactions amongst different system components and

implementation details). Therefore, it would be useful to use UML not only for modeling SPL artifacts, but also for describing variability in order to get all the advantages of UML, including tool support and standardization. Furthermore, UML model transformations can be realized by model transformation languages for which there are implementations and transformation engines, such as Atlas Transformation Language (ATL) [ATL], Query/View/Transformations (QVT) [OMG08] and Atomic Transformation Code (ATC) [EST].

However, since UML does not support directly variability modeling as needed for SPL, several UML extension mechanisms to specify product line variability were introduced by different authors [CLA01], [ROB02], [PRE02], [ZIA03], [CHO05], [GOM05], [KOR07]. Each one of these works proposes a set of stereotypes, tagged values and constraints for SPL, but so far there is no standard UML profile for SPL. These works will be explained in detail in chapter 2.

The evaluation of software and system designs for non-functional properties (NFP) such as performance, reliability, and security can be enabled by attaching to the UML software model suitable additional information specific to the property to be evaluated [WOO05]. Performance properties can be annotated on UML models by using the *UML Profile for Schedulability, Performance and Time (SPT)* or its recent replacement, the *UML Performance Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE)* [MARTE11]. The SPT and MARTE profiles define stereotypes and tagged values that can be attached to design model elements, particularly in the architecture, behaviour and deployment specifications. An annotated UML model

can be transformed into a performance model and analyzed with known analysis techniques and tools [BAL04], [WOO08].

1.1 Motivation

Software Performance Engineering (SPE) is a methodology introduced in [SMI90] that aims to insure that software products are built to meet their performance requirements and promotes the integration of performance analysis into the software development process from the early stages and continuing throughout the whole software life cycle. SPE uses predictive performance models to evaluate the responsiveness of the system (such as response times, throughputs, queueing delays and utilizations). In order to integrate performance analysis into the SPL development process, two major model transformations are required: a) from an annotated UML SPL model to a product model with concrete performance annotations, and b) from the outcome of the first step to a performance model. This research focuses on the first transformation as illustrated by the shaded area in Figure 1.1, whereas the second transformation for deriving automatically a LQN performance model for a specific product applies the PUMA transformation approach previously developed in our research group [WOO05].

One of the main concepts of software product line development process is to take advantage of the reusability of a set of core assets shared among the members of a family of products, instead of building each product from scratch. In this work, we apply the same concept to the reuse of performance annotations, by integrating software performance engineering techniques in the early phases of SPL development. Instead of annotating from scratch each UML model of each product, we propose to annotate the

SPL model once with generic annotations, and to provide binding information when deriving the annotated model of a desired product from the generic SPL model.

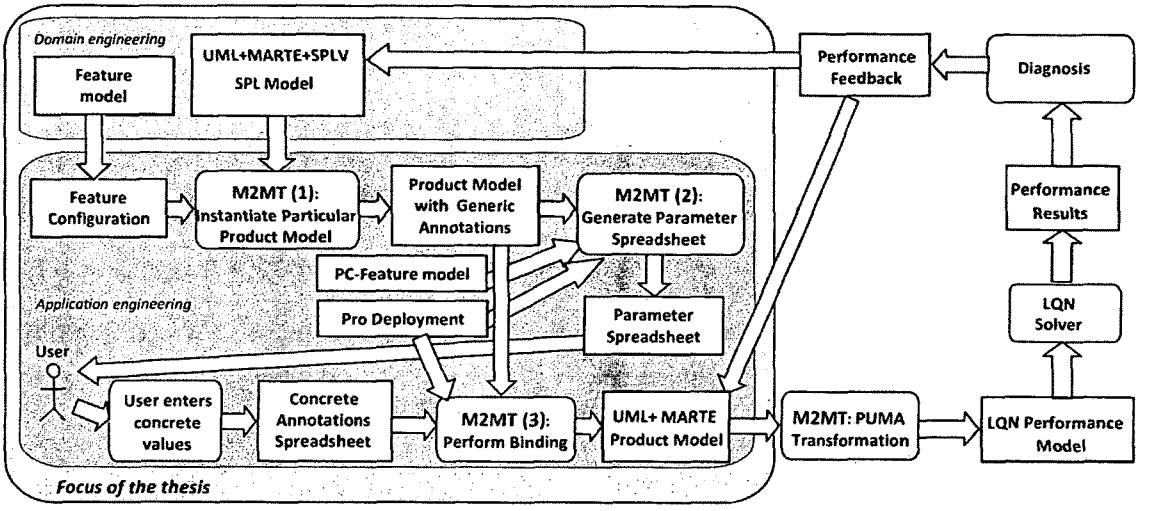


Figure 1.1: Approach for deriving a product performance model.

The automatic derivation of a concrete product model based on a given feature configuration is enabled through the mapping between features from the feature model and their realizations in the design model. In this research, we propose an efficient mapping technique that aims to minimize the amount of explicit feature annotations in the UML design model of SPL.

Due to the fact that models are more abstract and less detailed than code, variability at model level is less scattered and therefore simpler to manage [GRO07]. Previous research has been done in the area of integrating model-driven development (MDD) into SPL to achieve the complementary benefits of the two paradigms. Most of such papers address variability management and product derivation in SPL by adopting MDD at the structural level, but only a few are concerned with the behavioural level. In

this work, we apply MDD to SPL development for modeling variability and deriving a product in both structural and behavioural views with a double goal: to derive a UML model for a given product from the SPL model, which can be further used both for developing a product implementation and for generating a performance model for the product for early performance analysis.

Aspect-Oriented Modeling (AOM) techniques improve software development by providing means for modularizing crosscutting concerns, and then waiving them with other software artifacts. There are some previous works applying AOM to SPL development with the intention to manage variability and to derive a product. In this work we do not apply AOM, but meet the same objectives by building a SPL model that superimposes the realization of all features, as described in chapter 4.

Performance is a run-time property of the deployed system and depends on two types of factors: some are contained in the design model while others are external to the design model, characterizing the underlying platforms and run-time environment. Performance models need to reflect both types of factors. Woodside et al. proposed the concept of performance completions to close the gap between abstract design models and external factors [WOO02]. Performance completions provide a means to extend the modeling constructs of a system by including the influence of the underlying platforms and execution environments in performance evaluation models. Since our goal is to automate the derivation of a performance model for a specific product from the SPL model, we propose to deal with performance completions in the early phases of the SPL development process.

Dealing manually with a large number of performance parameters annotating a UML+MARTE product model, by asking the developer to inspect every diagram in the model in order to extract these annotations, is an error-prone process. This research deals with the problem of collecting all the generic parameters that need to be bound to concrete variables from the annotated product platform independent model and presents them to the user in a user-friendly format.

1.2 Objectives

This research aims to take advantage of the SPL model-driven engineering approach to integrate performance analysis in the early phases of the development process for SPL. We start by introducing generic performance annotations to the UML model representing the set of core reusable SPL assets. The annotations are generic and use the MARTE Profile adopted by OMG [MARTE11]. A chain of model transformations realized in the Atlas Transformation Language (ATL) derives the platform specific model of a given product with concrete MARTE performance annotations from the SPL model. Therefore, the derivation process of a member of the SPL not only considers binding the variability expressed in the SPL to a specific product, but also binding the generic performance annotations to concrete values for this product.

Mapping features from the feature model to model elements realizing them in different SPL design models is essential for the automatic derivation of concrete product models from the SPL model. The mapping allows developers to analyze and reason over variability in SPL. One of the outcomes of this research is to propose a new technique for explicit and implicit mapping of features to model elements, and to show how such a

mapping is used in a model transformation for product derivation. The proposed mapping technique aims to reduce the clutter of variability specifications in the SPL design model, and therefore to ensure that the models are more concise, less error-prone and easier to visualize. It is based on the idea of explicitly annotating the minimum amount of model elements with the feature(s) requiring them while, the mapping between features and non-annotated model elements is implicit and can be inferred based on the explicit annotations of related model elements. The evaluation of implicit mapping is done in the transformation rules by navigating the model according to the UML metamodel and well-formedness rules. The proposed mapping technique ensures that the derived model is well-formed by enforcing the well-formedness constraints during model transformation.

Model-driven development (MDD) is a new paradigm for the software development process which changes the focus of software development from code to models and promotes the automatic generation of code from models. We believe that MDD has an important role to play in SPL engineering for managing variability and supporting product derivation. Many existing works have investigated ways of applying MDD to SPL development, with the goal of generating code for given products from the SPL model. In this work, we propose to add another dimension to the model-driven development of SPL, by generating a performance model for a given product from the SPL model, in the early development phases.

MDD enables developers to evaluate the non-functional properties of the software under construction by transforming UML design specifications annotated with extra information corresponding to the property to be assessed into appropriate analysis models. Traditionally, performance analysis models were built "by hand" by specialists

in the field, who "abstracted" from the software only the properties of interests. However, in the context of model-driven development, a new approach for constructing analysis models has emerged, where software models with performance annotations are automatically transformed into performance analysis models. In this direction, we propose to generate from SPL the UML model of a specific product member annotated with concrete performance specifications (which can be further transformed automatically into a performance model by previously developed transformations, such as PUMA).

The extensible nature of UML allows for the definition of new *profiles*, such as MARTE, without extending the UML metamodel (known as "heavyweight extension"). A UML profile, known as a "lightweight extension" enables us to use existing standard UML editors and transformation tools without any adaptations. An advantage of the MARTE annotations is that they can be parametric, raising the level of abstraction and reusability. Therefore, in this research we benefit from the MARTE profile and its generic reusable performance annotations by applying them to a UML model of SPL. During product derivation, a set of binging directives is provided to specify the mapping between the generic and concrete performance annotations. A given product with actual values of performance specifications is generated through model-to-model transformation.

Different SPL members may vary from each other in terms of their functional requirements, quality attributes, platform choices, network connections, physical configurations, and middleware. Many details of the system that are not part of its design model may affect the run-time performance (such as the underlying platforms); such

details need to be included in the performance model. Performance completions are a manner to close the gap between the high-level design model and its different implementations [WOO02]. They provide a means to extend the modeling constructs of a system by including the influence of the underlying platforms and execution environments in performance evaluation models. In this research, the variability space of the performance completions is covered and represented through so-called Performance Completion-feature model (PC-feature model). The PC-feature model explicitly captures the variability in platform choices, execution environments, different types of communication realizations, and other external factors, such as protocols for secure communication channels and represents the dependencies and relationships between them. Therefore, our approach uses two feature models for SPL: 1) a regular feature model for expressing the variability between member products (as described in section 4.3.2), and 2) a PC-feature model introduced for performance analysis reasons to capture platform-specific variability (as described in chapter 6).

In order to analyze the performance of a specific product running on given platforms, we propose to include the performance impact of underlying platforms in the UML+MARTE model of a product as aggregated platform overheads, expressed in MARTE annotations attached to existing processing and communication resources in the generated product model. This will keep the model simple and still allow us to generate a performance model containing the performance effects of both the product and the platforms. Every possible PC-feature choice is mapped to certain MARTE annotations corresponding to UML model elements in the product model. This mapping is realized by a transformation generating parameter spreadsheets, which presents the user not only

with the annotation parameters needing to be bound to concrete values, but also with their context information, as described in chapter 7.

A very simple approach for binding performance annotation parameters would require a developer to carefully inspect the UML model to extract all the generic parameters and then to provide concrete values for each of them. The binding information is fed to the binding procedure as a manually-built list of couples {<generic_parameter>, <concrete_value>}, which contains no context information and no guidelines. This approach requires a lot of work from the developer and is error prone. In general, the number of generic parameters is quite large, so the need for a user-friendly approach is stringent. In this research, we propose a model-driven user-friendly approach for collecting automatically all generic performance parameters that need binding from the generated UML+MARTE product model, which are presented to developers in a spreadsheet format, together with context and guiding information, as described in chapter 7.

To the best of our knowledge, our research is the first to propose an approach for integrating quantitative performance analysis in the early phases of UML-based model-driven development of SPL and generating automatically a performance model of a product from the software model of the family by a chain of model transformations. The main research challenge stems from the semantic mismatch between the SPL model and performance model. A SPL model is a collection of core "generic" asset models, which are building blocks for many different products with all kind of options and alternatives. Since an SPL model does not represent a clearly defined system that could be implemented, run and measured as a whole, we cannot talk about analyzing its

performance. On the other hand, a performance model is an instance-based representation of a run-time system, focusing on how the system uses its resources and how the competition for resources impacts the overall performance (response time, throughput, utilization, etc.). Hence, we need to derive first a concrete product with a well-defined structure and behaviour from the SPL model, and then add to it platform details, transforming into a platform specific model whose performance can be analyzed.

The challenges of the proposed research are both SPL-related and performance-related. The SPL-related challenge is to automate the derivation of a UML product model that contains all the views necessary for performance analysis (i.e. software architecture, key-performance scenarios and deployment) from a SPL model. The performance-related challenge is due to the performance annotations. The SPL model should have reusable generic parametric performance annotations, which will be bound to concrete values when generating a platform specific model of a given product.

In order to understand what kind of performance-related annotations need to be added to the UML-based SPL model to enable performance analysis, one needs to look at the basic concepts contained in the performance domain model which describes three main types of concepts: resources, scenarios, and workloads. Generic performance annotations mean placeholders for information regarding these three concepts. Concrete values will be given for every generated product member to be run on a specific platform/environment.

1.3 Thesis Contributions

The research of this thesis is an integral part of a larger project aiming to generate a performance model for a given product from an annotated UML SPL model (see Figure 1.1). In order to analyze the performance of a given product running on a given platform, three major model transformations are required: a) the transformation of a SPL model to a product platform independent model (PIM) with generic performance annotations; b) the binding of the generic parameters to concrete values provided by the user to generate a platform specific model (PSM) of a given product and c) the transformation of the outcome of the previous step (a given product PSM with concrete performance annotations) into a performance model. This research focuses on the first two transformations as shown in Figure 1.1, whereas the third transformation to derive automatically a Layered Queueing Network (LQN) performance model for a specific product will apply the PUMA transformation previously developed in our research group [WOO05][WOO08]. After the performance model for a given product is generated, it can be analyzed with existing LQN solvers and feedback regarding its performance properties will be given to the software development team. Although domain engineering is more complex, Figure 1.1 shows that the only domain engineering process covered in this thesis is the construction of the SPL model (that represents the source model of our transformation approach) and the creation of the regular feature model (that represents variability in the SPL). The focus of this thesis is mainly on the application engineering phase, where the proposed model transformation approach is applied to generate a given product model.

The contributions of this research are as follows:

- Automatic model transformation to derive a given product PIM from a SPL model. (as shown in Figure 1.1 - M2MT (1)), which includes:
    - Defining a UML-based SPL profile to represent variability in both structure and behaviour SPL artifacts (as described in section 4.1).
    - Efficient mapping technique to establish traceability links between features from the feature model to the model elements from the SPL design model realizing them (as described in section 4.2).
    - Model transformation for deriving a product PIM (structural and behavioural views) with generic performance annotations from the multi-view SPL model (as described in section 4.4.2).
- Transforming the generated product PIM with generic performance annotations into a product PSM with concrete annotations (as shown in Figure 1.1 - M2MT (2) and M2MT (3)), which includes:
    - Handling performance completions and presenting them as a second feature model called PC-feature model (as described in chapter 6).
    - Mapping each PC-feature represented in the PC-feature model to the product model element(s) affected by it through MARTE performance specifications (as described in section 7.2).
    - Model-driven user-friendly approach for automatic extraction of all generic performance parameters that need binding from the generated product PIM and presenting them to developers in a spreadsheet format (as described in section 7.2).

- Performing the actual binding while transforming a product PIM with generic performance annotations into a product PSM with concrete performance annotations (as described in section 7.4).

- Automatic approaches for passing actual parameters to the ATL transformation.
  - Passing a given feature configuration to the transformation that derives a concrete product corresponding to this feature configuration from the SPL model (as described in section 5.1).
  - Passing the concrete performance annotations contained in the spreadsheets to the transformation that generates a product PSM from the PIM (as described in section 7.4).

The following papers are the outcome of this research work:

1. R. Tawhid, D.C. Petriu, "Towards Automatic Derivation of a Product Performance Model from a UML Software Product Line model", Proceedings of the 2008 International Workshop on Software Performance (WOSP08), pp. 91-102, 2008.
2. R. Tawhid, D.C. Petriu, "Integrating Performance Analysis in the Model Driven Development of Software Product Lines", Proc. of the 11th international conference on Model Driven Engineering Languages and Systems (MODELS 2008), Springer, LNCS Vol. 5301 (K. Czarnecki, Ed), pp. 490-504, 2008.
3. R. Tawhid, D.C. Petriu, "Product Model Derivation by Model Transformation in Software Product Lines", Proc. of the 2nd IEEE Workshop on Model-based Engineering for Real-Time Embedded Systems (MoBE-RTES 2011), Newport Beach, CA, USA, 2011.

4. R. Tawhid, D.C. Petriu, "Automatic Derivation of a Product Performance Model from a Software Product Line Model", Proc. of the 15th International Conference on Software Product Lines (SPLC'11), Munich, Germany, 2011.
5. R. Tawhid, D.C. Petriu, "Integrating Performance Analysis in Software Product Line Development Process", book chapter in *Software Product Lines – Advanced Topic*, InTech - Open Access Publisher, 2011.
6. R. Tawhid, D.C. Petriu, "User-Friendly Approach for Handling Performance Parameters during Predictive Software Performance Engineering", Proc. of the 3rd ACM/SPEC international conference on Performance Engineering (ICPE 2012), Boston, USA, 2012.
7. R. Tawhid, D.C. Petriu, "Survey on the Use of Models in Software Product Line Development", submitted to *Advances in Software Engineering* Journal, 2012.
8. D.C. Petriu, M. Alhaj, R. Tawhid, "Software Performance Engineering", chapter in *Formal Methods for Model-Driven Engineering*, (M. Bernardo, V. Cortellessa, A. Pierantonio, Eds.), Springer, Lecture Notes in Computer Science (in press).

1.4 Contents of the Thesis

This section presents the overall organization of the thesis and the content of each chapter.

Chapter 2. SPL - State of the Art: presents an overview of the SPL literature. The Software Product Line (SPL) development methods and UML variability modeling are discussed first, followed by product derivation. The advantages of introducing Model-Driven Development approaches to SPL are explained next. Aspect oriented modeling

and its integration in SPL techniques is briefly discussed. Approaches for non-functional properties in SPL are presented as well. The chapter is concluded by a discussion and comparison between these approaches.

Chapter 3. Background Review: presents the background material needed for this research. Different approaches addressing the concept of platform independent and specific are discussed. UML and MARTE profile, performance analysis methodologies and different model transformation languages are briefly presented.

Chapter 4. Proposed Product Derivation Approach: describes first the UML product line profile for modeling variability (SPLV) and the proposed mapping technique for relating features to SPL design model elements. The derivation approach is illustrated by presenting the SPL model that represents the input of our algorithm, followed by the steps of our model transformation algorithm. The output model of this transformation (i.e., a specific product) is described. An e-commerce case study with MARTE annotations is used to illustrate the approach.

Chapter 5. ATL Transformation for Generating a Product PIM: A detailed analysis of the ATL implementation of UML SPL model to concrete product model is given. The interpretation of explicit and implicit feature mapping is explained when describing the transformation rules.

Chapter 6. Performance Completions: The variability space of the performance completions is covered in this chapter and represented through a Performance Completion-feature model (PC-feature model).

Chapter 7. Handling Performance Parameters: describes the model transformation for generating a user-friendly representation of the binding information for the generic

performance annotations. The actual binding into the generated UML+MARTE product model is also presented.

Chapter 8. Performance Analysis: presents different performance analysis experiments conducted with the LQN models obtained for different generated product models.

Chapter 9. Verification and Validation: evaluates the proposed model transformation approach for product model derivation. The verification of the proposed approach is ensured through the evaluation of the well-formedness of the participating models: feature model, feature configuration, and SPL model. Several test cases were developed to verify completeness, correctness and containment conformance of the derived product model.

Chapter 10. Conclusions: presents the contributions and the limitations of this research. The lessons learned from using multiple open source tools are stated. Possible future directions to expand this work are also addressed.

Chapter 2: SPL - State of the Art

This chapter reviews the state of the art of software product lines (SPL). Different SPL methods are reviewed in section 2.1. In order to take advantages of UML tools and standardization, section 2.2 shows how UML is extended to model variability. Product derivation approaches are discussed in section 2.3. The integration of two important procedures into SPL is addressed in section 2.4 and 2.5, while section 2.6 addresses the SPL requirements level methods. Section 2.7 presents non-functional properties in SPL. It is important to note that the SPL methods discussed in sections 2.1 to 2.7 are compared according to different criteria in section 2.8, and a summarizing comparison organized around the context and content of the methods is presented in the form of tables at the end of the chapter.

2.1 Software Product Line Methods

Modeling communality and variability is an important activity in developing software product lines. Variability in a SPL can be modeled at the software requirements level and at the design level. Feature modeling is the most widely used approach for capturing and managing commonalities and variabilities in SPL throughout all the stages of its development process.

FODA & FORM. Feature Oriented Domain Analysis (FODA) method has been developed at the Software Engineering Institute, Carnegie Mellon University [KANG90]. The feature models represent reusable software elements and consist of the feature diagram, feature definitions, composition rules and rationale for features. FODA introduces the notion of *feature* as an important SPL concept that represents reusable requirements or characteristics of a family of products. Features are analyzed and classified as *mandatory* (common to all product line members), *optional* (required in some product line members), and *alternative* (select exactly one feature out of many). Features are organized into a hierarchical tree in the feature diagram with the root of the tree representing the concept described and the remaining nodes indicating features and sub-features. Compound features are located at the top of the tree and more refined features below. *Mandatory* features are shown by nodes with a black circle, and *optional* features with an empty circle. A set of alternative features is depicted by an arc spanning two or more edges. Feature definitions describe the time at which a feature will be bound (compile time, activation time, run-time). The FODA notations specify dependency relationships and constrains between features called "composition rules". The composition rules indicate which combination of features will be a valid set. There are two types of composition rules: *requires* indicates that a feature depends on some other feature to make sense in a system and *mutually exclusive* indicates the "excludes" relationship between two features (i.e., both features cannot be included in the same system). The reasons for choosing a feature are indicated by annotating it with the rationale.

FODA method is extended by Feature-Oriented Reuse Method (FORM) in [KANG98] to cover the entire activities of domain and application engineering. FORM is a systematic method that captures commonalities and differences of applications and uses a feature model to develop the domain architecture and components for reuse. FORM consists of two main processes: domain engineering and application engineering. During domain engineering, a reference architecture as well as reusable components are created. The FORM domain engineering process consists of three phases: context analysis, domain modeling, and architecture modeling. During the domain modeling phase, commonalties and differences between products are defined. A feature model is created, then used in the architecture modeling phase to develop reusable architecture and components. During the application engineering process, the feature model is used for selecting a consistent set of features for a specific application and the software artifacts developed in the domain engineering process are used to develop applications.

Several extensions and modifications of the original FODA notation have been proposed. For instance, Czarneski in [CZA00] introduces FODA adaptations, such as: a) mandatory features whose ancestors are also mandatory are considered common to all family members; b) or-features; c) priority for variable features; d) feature cardinality and attributes. As FODA has no defined mechanism to specify the relation "at-least-one-out-of-many", Czarnecki defines a category called *Or-features*, depicted by a filled arc spanning two or more edges of feature nodes. Optional and alternative features can be combined to form another feature type called *optional alternative* features. Czarnecki describes weak constraints as default values which can be overridden. Variable features may be annotated with priority in some situations. Features can have cardinalities and

attributes as shown in [CZA02]. Mandatory features are specified with cardinality [1..1], while optional features with cardinality [0..1]. Or relationship can have n:m cardinality, a minimum of n and at most m features can be selected. Cardinality "*" means zero or more features.

Feature models are integrated into Generative Programming in [CZA00]; a Domain Specific Language (DSL) used to specify family members is derived. Generative Programming is a software engineering paradigm based on the notion of *generator* for system families and designed to accomplish reusability. Variability in system families can be managed by implementing components (i.e., code) and generators as generic artifacts. A specific instantiation can be used to generate the implementation of a family member. The mapping between problem and solution spaces is done through configuration knowledge. The problem space is the set of all the application concepts and features, while the solution space consists of generic implementation software components with all their possible combinations. Feature modeling provides the configuration knowledge required to automate the production of the family members corresponding to valid feature combinations.

CBFM. Further extending the original FODA notation, in [CZA04] [CZA05a] the authors introduce cardinality-based feature modeling (CBFM) and the notion of staged configuration. Three different kinds of features are defined: *root feature*, *grouped feature* and *solitary feature*. A maximum of one attribute can be given to any feature. A feature model defines configuration choices within a system (each configuration corresponding to a concrete system). A family member is specified by selecting the desired features from the feature model, respecting the variability constraints. This process can be done in

stages; each stage eliminates some configuration choices (hence the name "staged configuration"). Staged configuration is obtained by successive specialization of a feature model, or by multi-level configuration as in [CZA05c]. In each stage, a set of specialized systems (which is a subset of the original system) is produced. The specialization process is a transformation process that takes a feature model and creates another feature model. In multi-level configuration, the choices available in each stage are presented by separate feature models [CZA05c]. A translation algorithm is proposed to translate cardinality-based feature models into context-free grammars. Specialization of a feature model can also be done at the grammar level. Cardinality-based feature modeling and its configuration and specialization are implemented in the FeaturePlugin tool, which is an Eclipse plug-in.

OVM. Another technique for modeling and documenting variability is introduced by Pohl et al. in [POH05], [BUH05]. They propose a separate model called Orthogonal Variability Model (OVM) to capture the commonality and variability in a SPL. This separate variability model can be used during the different life cycles of software product lines to describe variability in feature models, use case models, design models, component models, and test models where a choice has to be made. There are two principal concepts in OVM models: "variation point" and "variant". *Variation point* (VP) represents a variable item or a variable property of an item whose realization or occurrence may vary across product line members. It is depicted as a triangle in the OVM

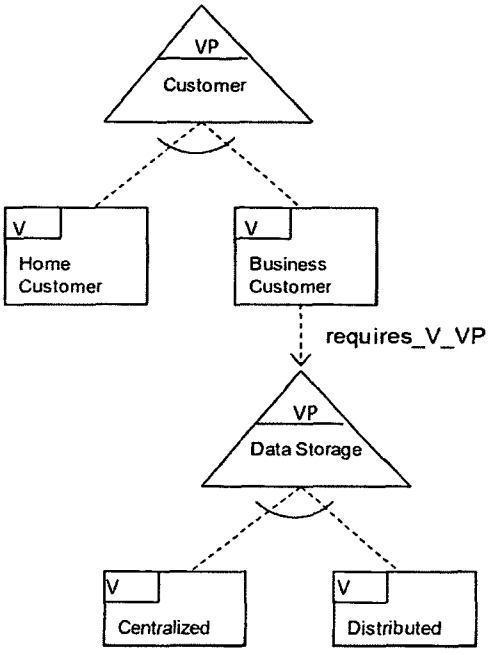


Figure 2.1: Orthogonal Variability Model (OVM).

model and must have a variability dependency relationship with at least one variant. *Variant* (V) defines a specific realization of a VP and can constrain it with a constraint dependency. Furthermore it has at least one variability dependency to a VP. Variant is depicted as a rectangle in the OVM model as shown in Figure 2.1. A set of variants is connected to a variation point through a *variability dependency* which is depicted as an edge between a triangle and a rectangle. OVM presents several types of dependencies: *mandatory* dependencies (depicted as a solid edge), *optional* dependencies (dashed edge) and *alternative choice* dependencies (solid arc spanning a set of optional variability dependence edges). An *alternative* choice has upper and lower bounds denoted by [i, j], which means at least i and at most j variants in a particular product line member. OVM defines different types of constraints between variants, variation points and variants to variation points (such as *requires* and *excludes*). OVM creates hierarchical relationships

between variation points, where binding a specific variant to a variation point cause the inclusion or exclusion of child variation points. Modeling variability in the OVM model is the first part; the second part relates the variability defined in the OVM model to the software artifacts specified in other models by defining traceability links. The OVM documents variability in a similar way across different models without modifying the existing notations. For instance, variability is documenting in the requirements artifacts by defining traceability links between the OVM model and the feature models, use case models, and traditional requirements models. Pohl et al. claim that defining the variability of a feature tree through the OVM enhances the expressive capability compared to the feature tree itself.

Pohl et al. apply OVM in their SPL engineering framework, consisting of domain engineering and application engineering, to define and manage variability throughout the entire development process. The domain engineering process contains five sub-processes: product management, domain requirements engineering, domain design, domain realization, and domain testing. The application engineering process, where applications are built by reusing domain artifacts and binding the variability of the PL to the specific needs of the application, contains four sub-processes: application requirements engineering, application design, application realization, and application testing. The OVM is created during the domain requirements engineering sub-process and refined during the domain design sub-process to include internal variability. The OVM is used through the application engineering sub-processes to support the consistent reuse of the domain assets. For instance, the application architecture is derived from the reference architecture by binding the variability according to OVM. In [BUH05] a variability

metamodel is proposed, extended with the concept of "requirements artifact", which in turn is related to a variant in the variability model through a "specified by" dependency, indicating that this requirements artifact is a variable. A product line PL is represented in the proposed metamodel through the class *single product line,* whose instantiations represent concrete products. The extended metamodel is implemented in the requirement tool DOORS to enhance the presentation of variability information in SPL.

COVAMOF. In [SIN04], the authors propose a framework for variability modeling called COVAMOF, which is able to model the relations between simple and complex dependencies. The concept of variation points is introduced in three abstraction layers of a product family: features, architecture, and component implementations. Variability in product families is realized by *variation points* and is implemented in the product family artifacts by a *realization technique*. Variability dependencies are related to each others. Solving one dependency may affect the validity of others (referred to as dependency interaction). These interactions are represented explicitly in the proposed framework. The COVAMOF Variability View (CVV) derived from the product family artifacts, provides two views on variability: *variation point view* and *dependency view.* Dependency interactions in the CVV identify how dependencies mutually interact and how to deal with them during product derivation. Product derivation problems related to dependencies are introduced in [SIN06] which shows how COVAMOF handles these problems. COVAMOF provides a repository containing all product derivation knowledge to reduce the gap between implicit, documented and formalized knowledge. Since COVAMOF specifies dependencies between multiple variants instead of two variants, this provides an abstract view on relations between choices, which reduces the

complexity of the variability model. COVAMOF framework was applied to several industrial case studies. A variability assessment method called COSVAM is introduced in [DEE09] for the COVAMOF framework. Variability assessment is a technique that answers questions related to variability evolution by analyzing the mismatch between provided and required variability. Provided variability is the variability in the product family artifacts, while required variability is the required functionality and quality in a set of product scenarios. COSVAM provides five deliverables: provided variability model, required variability model, mismatches, solution scenarios, and required modifications. The solution scenarios identify possible solutions to solve the variability mismatches. In order to provide these deliverables, COSVAM defines an iterative process that consists of five steps: identify assessment goal, specify provided and required variability, variability evaluation, and interpret evaluation results.

2.2 Modeling Variability with UML

The Unified Modeling Language (UML) is a well-known accepted notation for modeling software systems. When UML is used for the development of SPL designs, it would be also useful to use it for specifying and modeling SPL variability. However, since UML was intended to model single software systems and does not support modeling variability, several UML extensions to specify product line variability were introduced [CLA01], [ROB02], [CHO05], [PRE02], [KOR07], [ZIA03], [GOM05]. Each one of these methods (labeled below with "*M_<author_name>{letter}>*") proposes a set of stereotypes, tagged values and constraints, which extend UML to represent generic

models needed in domain engineering for specifying product line models and for developing SPL core assets, but so far there is no standard UML profile for SPL.

M_Clauß. A UML extension to support feature models and to describe variability in UML diagrams is presented in [CLA01], [CLA01a]. The feature tree is represented as a UML class diagram with stereotypes to distinguish between feature types. The stereotypes «mandatory», «optional», «alternative» and «external» extend classes to represent the respective type of features. The relations between features are modeled through composition and generalization relationships. Composition rules are modeled as stereotype constraints on the associations between classes. The required and mutual exclusion constraints are modeled though the stereotypes «requires» and «mutex», respectively. Additional properties such as binding time are represented as tagged values. The notion of *variation point* is introduced to explicitly mark the location of a variation and its different variants that can be applied to all UML model elements. Variation point and variant are described by classes with the stereotype «variationPoint» and «variant», respectively. The relationship between a variation point and its variants is modeled by the generalization relationship. Several variation points may be specified for a modeling element. Optional element notation is also introduced.

M_Robak. Another lightweight extension for mapping features to UML diagrams is presented in [ROB02]. A stereotype «variable» with a tagged value named "feature" (whose values indicate the actual feature) is used to model an element that specifies a variable feature. This approach guarantees forward and backward traceability from the feature diagram to a system model; the consistency between the model and the feature diagram is maintained as well. The information required to choose a variant (i.e. binding

time, binding mode) is captured through tagged values associated with the stereotypes. This approach shows how to express variability in the UML activity and component diagrams, but does not introduce a development process for SPL.

M_Choi. An approach focused on representing variability in the SPL architecture is proposed by Choi et al. in [CHO05]. UML 2.0 is extended to represent both commonality and variability of the product line architecture. The approach enhances the architecture visibility and makes it easy for instantiating product architectures. Variability is classified into optional and alternative characteristics. Component and connector are classified in the same manner. Alternative components which are implemented without modifying their interfaces are distinguished from those whose implementation and interfaces can be changed. Both static and dynamic views of optional and alternative components, as well as connectors, are addressed.

[PRE02] introduces the UML-F profile that defines product line extensions. The author provides notational elements to specify well known design patterns. The profile is is concerned only with structural aspects. Several stereotypes and tags annotate classes and interfaces belonging to the product line, while others annotate application-specific classes which present new classes to the product line. A tag «utility» is used to specify the base classes for libraries or run-time system. The class diagram is considered incomplete by default, so its elements are annotated with the tag "..."; otherwise the tag "©"explicitly defines an element as complete. Incomplete class may be missing attributes or methods. Other UML-F tags indicate both the static structure of the framework construction principles, as well as design patterns.

M_Korherr. In [KOR07] the OVM model is described in UML notation. A UML2 profile for variability models is introduced, which extends activity diagrams to show the impact of variability on business processes. Another UML extension is proposed in [CLA01] for representing feature diagrams. The work doesn't introduce a development process for SPL; it demonstrates only how the proposed UML profile for variability modeling is mapped to an activity diagram.

M_Jézéquel-a. A number of papers from a group led by M. Jézéquel propose a UML profile for modeling variability at structural and behavioural levels [MON02], [ZIA02], [ZIA03], [ZIA04]. In [MON02], variability is modeled in the class diagram using abstraction, parameterization and the defined stereotype «optional» to present optional information. Two types of constraints for product line are proposed, expressed as OCL constraints at the UML metamodel level. Generic constraints, such as inheritance and dependency constraints, are applied to any SPL architecture. However, specific constraints such as presence and mutual exclusion constraint are applied to specific product architecture. In [ZIA02], a model for behavioural requirements in SPL is introduced. These requirements are expressed by high level message sequence charts (HMSC) extended with constructs for handling variability. HMSC are used to build dynamic assets for product line and OCL is used for expressing the coherence between static and dynamic aspects of the product line. MSC is extended to express variability by introducing constructs such as optional instances, optional parts, inheritance, and variation points. A UML 2.0 profile for SPL including stereotypes, tagged values, and structural constraints is proposed in [ZIA03]. Optional features that can be omitted in some products are presented through the stereotype «optional» in classes, packages,

attributes or operations in UML diagrams. Stereotypes «variation» and «variant» are used to specify abstract class and each concrete subclass, respectively. The alternative variability is modeled as a mutual exclusion constraint between variants through OCL constraints. Variability in sequence diagrams is introduced in terms of three constructs: *Optionality*, *Variation* and *Virtuality*. Stereotypes «optionalLifeline» and «optionalInteraction» are used to present Optionality for an object and optional interaction, respectively. The variation of the behavior is modeled as a combined SD stereotyped by «variation», which refers to a set of subinteractions stereotyped by «variant». Each subinteraction specifies a variant behavior. The stereotype «variant» has a tagged value {variation=VariationName} to specify its variation point. The virtuality of a SD indicates that its behavior may be redefined by another SD or refinement according to a specific product. Virtuality is introduced by the stereotype «virtual». In [ZIA04], an approach to create detailed behaviour for each product member of the PL is proposed. First, the author uses an algebraic construct to specify variability in the sequence diagrams. Then, the algebraic expressions are interpreted to resolve variability and to derive product expressions which are subsequently transformed into a set of statecharts. PL behavioural requirements are specified using scenarios represented as UML2.0 SD. An algebraic approach to synthesize product statecharts from PL scenarios is presented. First, PL scenarios are specified as algebraic expressions extended by three algebraic constructs for variability to produce so called reference expression for SD (RESD). Specific product expressions in RESD are derived from the product line RESD by resolving the variability using a decision model. Then, product scenarios given as a

RESD is transformed into a composition of statecharts. The proposed approach was implemented in Java.

M_Gomaa-a. Another group addressing variability at both structural and behavioural levels is led by H. Gomaa. An extension to UML for capturing the variability of a SPL at the requirements and analysis level is presented in [GOM02], [GOM04a]. Multiple SPL views (use case model, static model, collaboration model, statechart model, and feature model) are represented in UML. Variation points are defined explicitly by annotating features and classes as optional or variant. The alternative decision concept is used to model SPL variability in collaboration and statechart models. Dependencies are modeled by dependency meta-classes and restrict the selection of two variants. This work is extended in [GOM07a], [GOM08]. In [GOM07a], the multiple view model of SPL is developed and stored in a repository. Feature model is used to derive a product member by selecting a consistent set of the desired features, and then the corresponding artifacts that recognize these features are extracted from the repository to create the product. They provide an automated tool support for presenting the multiple view model of SPL and for checking consistence between the multiple views, as well as for product derivation from the product line repository at the metamodel level. In [GOM08], a multiple-view meta-model is introduced to represent the relationship between different SPL views. The relationships are defined in each phase and between different phases. For instance, in the SPL analysis modeling phase, the relationships between the class, statechart, and collaboration models are defined at the meta-model level. The relationships between different phases such as the SPL requirements modeling and SPL analysis modeling phases are defined as well. Variability is addressed in the multiple-views by explicitly

modeling variation points in each view. The traceability between the features and the artifacts realizing them is done through feature/class dependencies and feature/use case dependencies. In order to maintain the consistence between the multiple views of SPL, consistency checking is handled at the metamodel level. The rules of consistency checking defined at the metalevel must be maintained at the multiple-view model level. These rules are specified through OCL. A process called product line UML-based software engineering environment (PLUSEE) is developed to support the multiple-view model of SPL where several tools such as knowledge-based requirement elicitation (KBRET) is integrated to assist users selecting features and to check feature dependencies.

In [WEB03], [GOM04], four different approaches to model variability by using parameterization, information hiding, inheritance, and variation points are described. Variability is modeled through parameterization where the variation is bound to the parameter's value defined in the core assets. Variability can be modeled through information hiding where different versions of the same component are created with the same interface. The variability is presented in each version of the component. The inheritance mechanism can be used to model variability, where subclasses represent different variants of a superclass. Each subclass has a different interface by extending the interface provided by its superclass by adding new operations or replacing them. The fourth approach to model variability is through variation points. Variation points are specific locations in the core asset components where variability takes place. Particular system components are built from these core asset components by binding variation

points to specific components. The paper shows how the variation point model (VPM) can be used to model other three different approaches.

PLUS. A UML profile to explicitly model commonality and variability in a SPL model is introduced by Gomaa in [GOM05]. The Product Line UML-Based Software Engineering (PLUS) method is a UML-based object-oriented analysis and design method for software product lines. PLUS method extends the UML-based modeling methods that are used for single systems to deal with software product lines. It provides several concepts and additional modeling processes to handle SPL. For modeling commonality and variability in the requirements modeling phase, an approach to model kernel, optional, and alternative use cases, as well as modeling variation points in use cases is proposed. PLUS also provides an approach for modeling common, optional, and alternative features. In the analysis modeling phase, a static and dynamic modeling approach is developed. The static model addresses the commonality and variability among the members of the SPL by categorizing classes as kernel, optional, or variant. In the Feature/Class dependency modeling, the dependencies between features and classes are determined, while in the dynamic modeling, interaction diagrams to realize kernel, optional and alternative use cases are developed. Also, kernel, optional, and alternative statecharts are created during the dynamic modeling phase. Variability in state machine is handled through inheritance and parameterization. During SPL design modeling phase, a component-based software design is develop to model kernel, optional, and variant components. The link between a feature and model elements realizing it is done through Feature/Use case and Feature/Class dependency modeling. A specific application model

can be derived from the SPL model according to a set of chosen features and subject to the feature dependencies.

An approach to make requirements and analysis SPL models test ready in the PLUS method, in order to create reusable test cases for applications derived from those product lines, is described in [OLI05]. Additional information has to be explicitly added to the SPL model to make it test-ready. The approach explicitly identifies the relationship between features, model elements, test specifications, and test dependencies, in order to generate functional test cases for the applications.

2.3 Product Derivation Methods

The process of deriving a product from SPL core assets is called product derivation. The derivation resolves the variability in the SPL model correctly in terms of the dependencies and constraints between features.

M_White. A tool named *Scatter* is presented in [WHI07] to provide automatic product variant selection for a mobile device taking into consideration the configuration and resource constraints such as available memory and network bandwidth. The tool is based on a Constraint Logic Programming Finite Domain. The product line architectures composition rules and the local non-functional requirements are used to reduce the solution space so only variants running on the target infrastructure are considered. After reducing the solution space, the resource requirements are considered and the resource constraints are solved. A branch and bound algorithm is used to select a valid solution. Scatter presents a visual modeling tool for capturing the structure, commonality, and variability in product line architecture to facilitate specifying which components are

available, what products can be created, and how each product is composed. Scatter is implemented using the open-source Generic Eclipse Modeling System (GEMS). This approach applies the same concept as in [CZA05c] for feature diagram references and staged configuration through successive specialization of a feature model to eliminate some configuration choices in the solution space.

M_Rabiser. In the context of variability management in SPL, Deelstra identifies one of the problems faced by organizations in product derivation as the complexity caused by a large number of variation points and variants in the variability model, which makes the selection of variants unmanageable [DEE05]. In this direction, an approach for deriving a product from a prepared and adapted variability models is introduced in [RAB07]. The main objective is to effectively employ variability models during product derivation. Unrelated aspects need to be pruned and additional product-specific information needs to be added to variability models before the derivation process. A pre-defined product, such as an existing product configuration, is added to the variability model to create an initial configuration and reduce the decision space. A tool suite called DOPLER for modeling variability and supporting product derivation was created. The proposed approach is illustrated through an example of an industrial system Siemens VA. Filtering large variability models is similar to the idea of specializing them though staged configuration as proposed in [CZA05c].

In [DEE05] a generic two-phased product derivation process is presented. In the initial phase, a first configuration is created from the product family assets and modified in a number of subsequent iterations until the product satisfies all its requirements. Three alternative approaches are used to derive the initial product configuration; assembly,

configuration selection, and a hybrid of the two approaches. The assembly approach puts together a subset of the shared core assets to the initial product configuration, while the configuration selection approach chooses the closest matching existing configuration. An existing configuration is a legal combination of components. The hybrid approach selects some of the chosen configurations and integrates them to a larger configuration. The approach is applied to an industrial case study to drive a product configuration. The notions of variation point and variant are used to express variability in the code, compilation, linking, or run-time phases of the lifecycle.

2.4 Model-Driven Development Approaches for Product Line

Model-driven development (MDD) is a promising approach since models are treated as first class software artifacts, similar to what traditional development does with code.

Model-driven development improves software development by capturing the key features of a system in models which are developed and refined as the system is created [PRE02]. In literature there are many publications investigating the application of MDD to SPL. A lot of work has been done in the area of integrating MDD into SPL to achieve the benefits of the two paradigms. The ultimate MDD objective in most of the cases is to generate code for a product directly from the SPL model; in some cases, however, a product model is also obtained.

M_Jézéquel-b. Jézéquel et al. propose a body of work in product derivation. A framework for the derivation of a product from a single general UML model based on a creational design pattern is introduced in [ZIA03a]. A model for the entire variability is created. Then, a systematic way of product derivation is proposed. The UML metamodel

is extended to represent the product line constraints at the OCL metamodel level. The design pattern *abstract factory* is used as a model decision during product derivation. This approach especially focuses on static models represented by UML class diagrams. This work is extended in [ZIA06] to address a UML model derivation technique for static and behaviour views. The static derivation is started from a product line class diagram with a decision model and generates the product class diagram which is implemented in the INRIA Model Transformation Language MTL. Additionally, an algebraic approach is proposed to derive statecharts for a specific product from the sequence diagrams of the product line, by transforming product scenarios given as a reference expression for SD into a composition of statecharts as in [ZIA04]. Before the product derivation process, the SPL model has to satisfy the generic constraints presented in [MON02]. Specific constraints have to be satisfied by the derived product model. The generic constraints represent the preconditions of the derivation approach, while specific constraints represent the post-conditions. The UML 2.0 profile presented in [ZIA03] for introducing variability in sequence diagrams is extended here by introducing several tagged values associated to the defined stereotypes. Tagged values *optionalPart, virtualpart* and *variationpoint* are attached to the stereotypes «optionalInteraction», «virtual», and «variation» to specify the occurrence name of the optional SD, the virtual interaction, and the name of the variation point, respectively. The approach for modeling and deriving behaviour views of SPL is implemented by a tool called Product Line Behaviour Synthesis (PLiBS) based on Eclipse in [ZIA07]. The approach is divided into two steps; in the first step, SPL behaviour is modeled through a sequence diagram which is specialized through model transformation while in the second step, state machines are

used to specify the specific product behaviours. However, the tool doesn't support specifying or checking PL constraints as indicated in [ZIA06]. Another dimension for product derivation process is addressed in [PER08], which proposes a flexible and automated process that takes into consideration SPL's customers specific and unanticipated requirements that are partially addressed by the SPL's core assets. The authors distinguish between three kinds of products: 1) products explicitly identified in the domain engineering process and automatically derived from SPL core assets, 2) products satisfying customer-specific requirements that are close to the SPL requirements so reusing the SPL core assets is useful, 3) products which are out of the SPL's scope. A product pre-configuration model is performed by selecting the chosen features and composing their related product line core assets. Then, the pre-configured product is customized to its customer-specific requirements through a model transformation which is validated against OCL instantiation constraints defined on core assets. This product derivation approach is applied for both structural and behavioral views of SPL and implemented on top of Kermeta, a general purpose metamodeling platform. However, it seems that the approach imposes consistency constraints to the model, but the authors don't explain how the model consistency is validated.

M_Czarnecki. Based on the extended feature model annotations, an approach for mapping feature models to artifacts models to present variability in these models is introduced in [CZA05b] [CZA05]. A model represented as a superimposition of all variants whose elements are linked to the corresponding features through annotations is proposed. The advantage of this mapping is that it directly shows the consequence of a selected feature on the resulting model. A *feature-based model template* which consists

of a feature model and model template is created. The model template contains all model elements in the family. These model elements are annotated with presence conditions to indicate whether the element is presented in a template instance (i.e., a product). Product derivation starts by creating a feature configuration based on the feature model. Then, the model template is automatically instantiated by evaluating its presence conditions based on this feature configuration. The approach is implemented on their feature modeling tool FeaturePlugin. This approach applies the same concept of expressing negative variability in structural models as proposed by Voelter et al. in [GRO07a]. However, the linking between a feature model and UML models is through stereotypes rather than a separate dependency model. Another difference is that Voelter provides a generic EMF-based solution through the tool XVar. To verify that all correct configurations of a feature model will generate well-formed template instances, an automated verification procedure is proposed in [CZA06] which only considers presence conditions. The desired well-formedness constraints are expressed in OCL, which map to an alternative semantics of OCL called *template interpretation* which allows evaluating an OCL expression over a template. The new semantics maps OCL constraints to propositional formulas, which are fed into a SAT solver to verify that no ill-formed template instances can be produced. The result of evaluating a standard OCL expression over all instances of a template can be obtained by evaluating this OCL expression over the template using the *template interpretation*. A template verifier according to this procedure is implemented as part of the modeling tool FeaturePlugin.

M_Heidenreich. In the direction of supporting the automatic derivation of a product member of a SPL based on a specific feature's configuration, Heidenreich et al.

propose in [HEI07], [HEI08] an Eclipse-based tool called FeatureMapper that defines the mapping of features in the problem space to model elements realizing these features in the solution space. Based on a set of selected features from the feature model, a feature configuration model is created. This is combined with the mapping model and interpreted by the FeatureMapper transformation component to derive a product model by eliminating model elements that are mapped to non-selected features. This approach is similar to the one proposed in [CZA05b] to map features to other models. However, the mapping in [CZA05b] is done through annotating models with presence conditions that refer to features from the problem space instead of using a separate mapping model. This approach also has some similarities with XVar proposed by Voelter in the way how it tailors models according to a specific feature configuration. An important advantage of XVar is that it models dependencies between features and model elements explicitly in a dependency model. The authors argue in [HEI08a] for the need for different visualizations of mappings between features and their realization artifacts in SPL. A new visualization technique called MappingViews is proposed that provides four different visualization views: realization, variant, context, and property-changes view. Each view represents a different vision of the SPL and exposes properties of the system that were hidden before. The proposed MappingViews technique is implemented in the FeatureMapper tool. Different approaches are proposed in [HEI09] to ensure the well-formedness of all participating SPL models (i.e., feature models, mapping models, and solution-space models). FeatureMapper enforces constraints on feature models and feature configuration models for ensuring valid models. Since FeatureMapper uses a generic mapping model, two well-formedness rules have to be enforced during the

creation of the mapping model: referenced features and referenced modeling artifacts of a mapping must exist. For solution-space models, three constraint classes are proposed to ensure that all output models conform to the rules in the metamodel of the language used for the solution-space models. These different verification approaches are generic and can be applied to any modeling language. On the other hand, the verification approach proposed in [CZA06] only addresses well-formedness rules for UML models. A comparison between the mapping tool FeatureMapper and a customized model transformation tool, VML* is discussed in [HEI10]. VML* is chosen as a representative of operational approaches. A set of comparison criteria is defined such as the expressiveness of the approach, useability and analysis support. The similarities and differences between the two approaches are illustrated through a common case study. They concluded that since FeatureMapper is based on a generic approach, no initial overhead is required to apply it to any modeling language and it is better for visualizing the variability in a product line. On the other hand, VML* is based on customizing a language for each target modeling language, thus it requires some setup cost at the beginning of a project, but it allows for much richer semantics in the mapping specification.

M_Gomaa-c. In [GOM05a], the author claims that developing and evolving SPL considering the software architecture model rather than starting from the code is much clearer. The evolution approach starts with the kernel software architecture representing the commonality of the SPL and considers the effect of optional and/or alternative features by adding optional or variant components to the product line architecture or modifying existing components. A feature dependent approach for adapting component-

based software architectures at design time to derive an application member from the SPL architecture is described in [GOM07b]. Optional and alternative features required for the adaptation of the original kernel software architecture which represents the commonality of the product line are added by designing optional and variant components to realize these features. These feature dependent components are used to manage the execution of kernel, optional and variant components in component-based software architecture. A different approach for run-time adaptation of the SPL architecture is proposed in [GOM07]. Variability is modeled in the component-based software architectures and each version of the architecture as it evolves is considered as a member of a SPL. Architectural patterns are used to build the architecture. Each architecture pattern has a corresponding software reconfiguration pattern to describe how the architecture can be dynamically modified. Software architecture configuration is the process of customizing the SPL architecture to derive a specific product line member in terms of components and their interconnections. On the other hand, dynamic architecture reconfiguration is the process of adjusting the application configuration at run-time after it has been deployed. Another approach for customizing dynamically at run time product line architecture and implementation to derive an executable application is described in [GOM06]. This process is called Dynamic Client Application Customization (DCAC) approach which is driven by the feature model developed during product line modeling. During application engineering, features are selected and a customization file is created which contains the selected feature names and values for parameterized variables. This file is used to customize a target application at run time.

M_Avila. A domain-specific transformation language (DSTL) called Model Template Transformation Language (MTTL) for specifying the transformations of model templates based on feature models is described in [AVI07]. This approach addresses the work of from [CZA05b] with a DSTL rather than UML. First, the product line model is developed by creating a feature model and a model template. The model template contains all model variants of the family model, while the feature model (which is an instance of the Cardinality-Based Feature Model (CBFM)) represents all possible features in the system. Second, according to the selected features, a feature configuration model is created from the feature model and a MTTL model is created which is transformed to an Atomic Transformation Code (ATC) model. Then, the ATC model is executed on the already existing transformation engine of the ATC for specializing the model template. In order to simplify the template specialization process in the generic transformation language ATC, the proposed domain-specific transformation language MTTL is used to provide specific constructs for the problem domain. By transforming the MTTL model to ATC model, the authors avoid creating a compiler to execute the ATTL model. An example of a family of StateMachine models is used to explain the approach.

M_Trujillo-a. [TRU09] deals with variability in model as well as metamodels and model transformations by applying the concept of step-wise refinement to them. A realization of a feature consists of refinements of models, metamodels and model transformations. Refinement of models means extending a base model by adding elements for customizing it for different requirements. In the context of SPL, the base model is the one with the common model elements in the SPL and the refinement models present variant/optional features or a new feature. For each new feature added to the base

model, a separate refinement model is defined. In this paper, the authors focus on variability in model transformations that define the mappings between models and models as well as between models and code. Since model transformations are defined at the metamodel level, variability on metamodels is discussed as well. A language called XAK to refine and compose XML documents is used. XML documents need additional annotations to be a base model. Some attributes are added to the document element to specify for example the name of the feature being supported. Since not all elements in the model are refinable, the attribute xak:module is added to the refinable elements. The composition of the base model and the refinement is achieved using XAK composer tool. In the case of a standard metamodel (e. g. UML) and/or the model transformation shares common library (e. g. ATLAS Transformation Language), there is no need to express variability in them. Otherwise, the same approach of model refinement and composition is applied to them. The authors present an example of refining and composing an Ecore metamodel and a MOFScript transformation. The concept of model composition in this approach is similar to the one in [JAY07], but rather than composing UML models with MATA, it composes XML models with XAK.

M_Trujillo-b. The authors propose an approach to handle variability in the System Modeling Language (SysML) which is a general-purpose modeling language for systems engineering applications in [TRU10]. SysML is an extension of a subset of the UML. Variability is handled by using the notion of variation point and expressed in different SysML models using stereotypes. The mapping between variation points and features as well as the derivation of a specific model are specified by the IBM Rational Rhapsody/Gears Bridge tool. The derivation process starts by selecting the desired

features for a product, and then all the common model elements will be included in the product model. The conditions associated to the variation point of each element are evaluated to check if the element is part of the product model or not.

M_Barreiros. [BAR09] proposes a model to present configuration knowledge that addresses some issues such as dependencies and interactions between features. A generic configuration module which consists of a list of parameters and configuration description is created to describe feature implementations. The detail of this configuration description depends on the type of the selected artifacts. The parameters are tailored for each specific configuration. The proposed model is a generic one since there is no restriction on the type of artifacts and composition method that are used. Their model can be added to any existing tools to display and edit the configuration knowledge of a SPL. A case study of multimedia application for mobile devices is presented to show how the configuration knowledge model is used to capture the necessary details to configure a specific product from a SPL based on the selected features and the interactions between them. The configuration module is specialized to represent a specific product configuration by binding the parameters to specific values.

M_Haugen .Another approach in [HAU04] presents a conceptual model for SPL engineering related to MDA standards. The product line is represented by a model called "product line model" which consists of a set of functional and quality features. The functional features are modeled as UML use cases, while the quality features as UML classes stereotyped with Qos. A system family model which represents the structure and behaviour of the systems at the architectural level is modeled in terms of UML 2.0 composite structures. The system family class view includes variability definition

through stereotypes. Stereotype «vp» represents a variation point that requires to be resolved at configuration time. Variation points can be an association, attribute or operation. Product derivation proceeds by defining the product line model. Then, the model transformation takes as inputs both the product line model and the system family model as parameters and produces as output the "Product/System Model" corresponding to the platform-independent model of the product. The authors try to minimize the use of stereotypes for modeling variability, making use instead of UML built-in mechanisms, such as association multiplicity 0..1 instead of «optional» stereotype proposed by Clauß [CLA01].

M_Oldevik. The same authors of [HAU04] propose the use of a high-order transformation to represent variability in SPL and generate a product code from a product line model in [OLD07]. The high-order transformation applies two text transformations: base transformation and hi-transform, which is an aspect-based extension of the model-to-text transformation language MOFScript. Each variant in SPL is presented in terms of hi-transform, which is applied to the base transformation using the higher-order transformation. The result is a modification of the base transformation by adding or removing features according to the hi-transform's instructions in order to generate a specific product code. This approach applies an idea similar to Voelter [VOE07], where AOP is used to implement positive variability by using the transformation language Xtend [VOE07].

M_Kim. An extension of the metamodel of Reusable Asset Specification (RAS) recently adopted by the Object Management Group (OMG) is presented in [KIM07]. They integrate the main ideas and concepts of AOM and component-based SPL

architecture into the RAS metamodel to represent variability in the PL architectures. Variability is modeled using aspects and variability mechanisms such as *variability point* and *variant*. This approach has the same idea of integrating variability mechanisms into aspects as in [MOR08] which uses the keywords *Alternative* and *Variant* rather than *variability point* and *variant*.

M_Botterweck-a. [BOT07] presents an approach for deriving the architecture of a product by selectively copying elements from the SPL architecture based on a product-specific feature configuration. The SPL architecture model contains variability to cover all products' aspects. During the derivation process, the variability is resolved since decisions whether components should be included or not are determined from the application feature model. The traceability between features and their realized architecture models are described by two *realizedBy* references in their metamodel. The feature model property *architecture* references the related architecture model, while the feature property *architectureElement* references the related element in this architecture model. The approach is realized by the model transformation language ATL. This approach is extended in [BOT09] to automate the derivation of an assembled executable product by integrating model transformation and Aspect Oriented Programming (AOP). The domain engineering process is divided into two phases: feature analysis and feature implementation. Feature model is created during the feature analysis phase using FODA notations with extra annotations to capture dependencies between features, while the SPL is implemented with AspectJ and Java during the feature implementation phase. The mapping between features and their realizing implementation components is done through an implementation model. During application engineering process, a feature

configuration model is created, and then ATL is used to generate an implementation configuration model from the feature configuration model and the implementation model. The implementation configuration model is further transformed to an equivalent textual model, which contains all components required to implement a specific feature configuration. Finally, AspectJ is used to assemble a specific product.

M_Botterweck-b. Each product member of a product line may need an individual, unique, and tailored user interface of its own to achieve the required usability. Usually, this is performed manually. [PLE10] proposes to integrate concepts from SPL product derivation and Model-based User Interface Development (MBUID) to obtain a systematic and semi-automated creation of user interfaces during product derivation. The user interface derivation is performed at a higher level of abstraction using the models from MBUID by relating both the Abstract Interaction Objects and Task Model from MBUID to the feature model during the domain engineering process. In the application engineering, the product-specific Abstract User Interface Model can be generated automatically from the product's feature configuration, and then the final user interface is derived using semi-automatic approaches from MBUID.

M_Filho. [FIL10] proposes a technique for deriving a component-based architecture for a given product from a software product line repository where several SPLs, feature model, reference architectures with the interface specification of the components as well as components that compose the reference architecture are registered. A configuration model for each SPL that defines the mapping between features, architecture components and component implementation is registered in the repository as well. The reference

architecture is instantiated with real components based on the selected features for a specific product.

The advantages of using Model Driven Architecture (MDA) in the process of developing SPL core assets that capture the common and different characteristics of the product family are presented in [DEE03]. The benefits of postponing the selection of a specific binding time and the choice of a realization mechanism for a variation point to the compilation of the application model during the application engineering process are discussed. The management of the platform variation points and the variation of functional characteristics between products can be achieved automatically in MDA approaches.

A literature survey on model-driven transformation approaches that address variability on the domain level is presented in [TAJ11]. The comparison between the surveyed approaches is based on several evaluation criteria such as the type, language and tool of modeling; supporting variability, traceability, and execution. The survey reveals some open issues that need to be taken into account in this area of variability modeling and model-driven transformation.

2.5 Aspect-Oriented Modeling Techniques for Product Line

Aspect-Oriented Software Development (AOSD) aims at separating the concerns addressed by "aspects" that are generally spread throughout the system and scrambled with core features. AOSD improves the way software is developed by increasing modularity, reusability, maintainability and ease of evolution.

A number of recent works have demonstrated that applying Aspect-Oriented Software Development (AOSD) to SPL provides an improved mechanism to encapsulate and model variabilities and commonalities throughout the entire software lifecycle [VOE07], [GRO08], [LAH 07].

M_Jézéquel-c. Another technique for product derivation through the integration of the Aspect-Oriented Modeling (AOM) approaches is introduced by Jézéquel et al. First, variability is integrated into AOM approaches in [LAH07], Then, a generic AOM framework extended with variability is introduced in [MOR07], [MOR08]. In [LAH07], the authors claim that creating complex systems with reusable aspects helps managing software complexity. In general, reusability and flexibility of aspects are limited due to the fact that current AOM approaches describe only one possible variant of an aspect and offer one way to weave it into the target model. In order to make aspects reusable in various contexts, there is a need to introduce variability in the AOM approaches. The authors apply the SPL mechanisms to address variability into the composition mechanisms of AOM approaches. Their method is explained though the SmartAdapters approach and the reuse of the *Observer* design pattern. Each reusable concern in the SmartAdapters approach has an adapter to specify its composition protocol, which is a set of adaptations and adaptation targets describing how the concern can be composed with other concerns when it is reused. Variability is added to this composition protocol in two contexts, *matching* and *adaptation*, to make it and the concern more reusable. The keyword *derivable* is introduced to the adapter meaning that the adapter can represent different alternatives to implement the composition. A choice is described by a clause *Alternative* which specifies two or more variants. The two clauses *optional* and

constraint are also added to extend the composition protocol. The composition metamodel of the proposed AOM approach was extended and used to build a modeling tool which was implemented on the Eclipse Modeling Framework (EMF) with the Kermeta language. The approach deals with packages, classifiers features, and associations. However, it doesn't concern the behaviour view of the systems. Therefore, the authors introduce a generic aspect-oriented modeling framework that can be applied to any domain metamodel in [MOR07]. This generic aspect-oriented modeling approach is extended with variability in [MOR08]. Introducing variability mechanisms into aspects turns standard aspects into configurable ones and makes them reusable in different contexts. Aspects are used to design optional and variant parts of a model as they do at code level. By weaving aspects into a base model, a specific product can be constructed. First, a domain metamodel is defined; then, several variability mechanisms are integrated into aspects. For instance, the keywords *Alternative* and *Variant* are used to identify several ways to compose aspects or different places to place them. Variability is introduced for both composition protocol and targeting. Composition protocol is described by adaptations, which are weaving operations as explained in [LAH07]. An *aspect configuration* which is constructed from a specific selection of variants and options is woven into the base model to derive the desired product. An approach is presented in [MOR08b] to verify that aspect models with variability are successfully integrated into an existing model. This verification approach concentrates only on the parts of the model that are affected by the aspect, analysing and testing all the possible variants of each variable aspect model. The checking process consists of two steps: 1) static analysis which checks the consistence of each part of the aspect as well as the

consistence of the composition protocol, 2) testing aspect composition, where each variable aspect model is considered as a test case.

M_Jézéquel-d. Another generic aspect-oriented modeling approach called GeKo is addressed by the same group in [MOR08a], which can be applied to any domain specific modeling language without modifying its metamodel. It is applied to a SPL model to derive a new product with new feature requiring the modification of the basic behaviour of the core assets. The GeKo composer is based on the definition of mappings between the pointcut and the base model, and the pointcut and the advice. (The pointcut is a subset of the base model, which indentifies the place in the base model where an aspect should be composed). These mappings are defined by linking model elements that are compatible. The approach is implemented in Kermeta, which is an extension of the Essential Meta-Object Facilities (EMOF). The composition of the base model and an advice model is based on the use of a third model called pointcut and two morphisms allowing the identification of the objects of the base which have to be kept, to be removed and to be replaced with those of advices. A different technique to design models containing variability is introduced in [MOR09], where a reusable variability aspect metamodel describing the variability concepts and their relationships independently from any domain metamodel is introduced. This aspect can be woven into any domain metamodel conforming to Ecore/EMOF to extend it with variability mechanisms. A meta-class called *PointOfVariability* is introduced in their generic variability metamodel which has a concrete sub-class called *VariabilityOfElement* to provide a specific domain with variability. Variability is applied to elements through boolean operators: *And*, *Xor*, and *Or* to denote mandatory, alternative, and at least one elements, respectively. Two

distinguished operators are defined; *HomogeneousOperators* linked to *VariabilityOfElement* and applied to elements of the same type, *HeterogeneousOperators* linked to *PointOfVariability* and applied to elements of different types. Constraints are also defined between *PointOfVariability.* Their aspect model weaver SmartAdapters is used to weave variability into its metamodel. The process for deriving a product model from a PL model conforming to the extended metamodel where the variability aspect has been woven is also described.

M_Voelter. An approach that integrates aspect-oriented (AOSD) and model-driven software development (MDSD) techniques to support variability management and the derivation process for SPL is introduced by Voelter et al [VOE07]. The authors claim that since models are more abstract and less detailed than code; variability at model level is less spread and therefore easier to manage than at code level. The authors use models to express as many PL artifacts as possible, which are processed using model transformations. Mapping from problem to solution domain is done through Model-to-Model transformation. Code is generated from the solution domain model by aspect-oriented code generation at the level of code generation templates. Variability management is generally handled at model level, except for unpredicted variability, where aspect-oriented programming is used to define it. [VOE07] describes the aspect oriented model-driven product line engineering (AO-MD-PLE) approach in general that defines PL and variants at model level and generates running applications through transformations. AO techniques are used to identify variability in the models as well as in the transformers and generators. Based on product requirements in the problem domain, SPL is modeled using a Domain Specific Language (DSL), where each variation needs to

be configured with various options. In the solution domain, a component-based architecture is built by model transformation from which code is generated using AO techniques. The authors distinguish between two types of variability: structural and non-structural, which are described through creative construction DSLs and configuration languages, respectively. In [GRO07a] two different ways are used to define variability in structural models built with a creative construction DSL. 1) Using negative variability by selectively removing parts of a creative construction model based on the presence or absence of features in the configuration models. An overall SPL model is created manually and its elements are connected to specific features in a configuration model. The model is then tailored to contain only model elements that realize the selected features. Deleting means that the element itself and all references to this element are removed from the model. The linking between features and model elements is specified in a separate dependency model. The XVar tool is used to implement the negative variability. 2) Using positive variability by starting with a minimal core which presents common artifacts in SPL and then selectively adding additional product-specific parts. Positive variability is implemented into models through model weaving, which composes different separated models into a consistent one. The places where the new parts will be added are defined using aspect modeling. A pointcut expression is used to specify the connection points. A model weaver tool XWeave described in [GRO07b] is used to implement the positive variability in structural models. XWeave takes a base model and one or more aspect models as input and weaves the content of the aspect model into the base model according to the product configuration. Weaving is done by matching names of elements in the aspect and the base model or by explicit pointcut expressions defined

by an openArchitectureWare (oAW) expression language. [VOE07a] explains how variability is handled in model-to-model transformations and at code generators level. The approach is implemented on top of an open source tool called *openArchitectureWare (oAW).* An *oAW* workflow is created, which consists of loading several models, checking constraints on them, transforming them into another model and then generating code from them. Variants of Model-to-Model transformation are built in *oAW* with a language called *Xtend,* while variants of code generators are created through a language called *Xpand.* A variant is created according to the selected feature in the configuration feature model built by the *pure::variants* tool. Therefore, in [GRO07] the variant management tool *pure::variants* and the open source MDD tool *openArchitectureWare (oAW)* are integrated to enable expressing variability at model level. *pure::variants* checks the validity of selected features from the variability model and automatically resolves dependency conflicts. It provides valid configuration information to the MDD tool which instantiates models and checks constraint violations as well as transforms models into other models and generates code. The execution of the model transformation and code generation in the MDD tool depends on the presence or absence of the feature in the configuration model provided by *pure::variants.* [GRO08] discusses the integration of aspect-oriented and model-driven software development to handle variability in models, model transformation, and code generation. The model weaver XWeave which allows describing product line variability as aspects is presented in [GRO08a]. During domain engineering are created the core model representing the commonality of the product line as well as the aspects representing features required by one or more members. During application engineering, the aspect models are composed with the core model based on

the selection of features in the configuration model to create a complete model for a member of this product line. Aspect models are linked to features defined in configuration models created by *pure::variants*. These links are specified in the oAW workflows, thus the weaving is based on the feature selection. The whole processes of the AO-MD-PLE approach is thoroughly described in [GRO09] with the tools that support it. The approach is illustrated with a home automation system case study. In some cases, variability has to be considered at code level. The authors explain in [GRO09] how AOP is used to realize positive variability; also, the XVar language is extended to support negative variability at code level. The concept of handling negative variability in this approach is similar to [ZIA03a] for deriving a UML class diagram of a product, where a design pattern *abstract factory* is used as model decision during product derivation instead of a separate dependency model. However, the approach in [GRO09] uses AOSD technique to compose variants at the model level, which is different from [MOR09] where AOM is used to integrate variability mechanisms at the metamodel level.

M_Gomaa-b. A new technique to model features separately from other features is proposed in [JAY07]. UML analysis and design models are used for each feature separately. The models can be combined at any point during the modeling lifecycle, and interaction between features is detected automatically. A model composition language, Modeling Aspects using a Transformation Approach (MATA) based on the graph transformation formalism, is used to relate features to each other. MATA describes where and how aspects are woven into base models using UML stereotypes «create» and «delete» for creating or deleting model elements, respectively. A stereotype «context» is applied to model elements to specify that they should not be affected when

creating/deleting their container. Critical pair analysis is applied to detect any conflict or dependency relationships between graph rules which are marked in the feature dependency diagram developed during requirements engineering. In the context of SPL derivation process, a user can select a subset of the available variant feature model and automatically compose them with the kernel feature model to generate the product model which consists of class, sequence and statechart diagrams. First, kernel features are modeled using UML and each variant feature model is converted into MATA language model which is translated to their equivalent graph transformation rules. Then, the graph rules are executed to generate a composed model which is transformed back to UML model. The idea of this approach is similar to the one in [LAH07]. The stereotypes proposed for composing features in MATA are similar to the Create/Set/Unset adaptations proposed by Jézéquel.

M_Stoiber. In [STO07], [STO09], it is claimed that UML has several difficulties with handling and maintaining variability. Therefore, a new variability modeling approach is proposed by combining aspect-orientation with a table-based decision model; the approach uses the ADORA modeling language. The decision table contains information about the variability constraints and decision support. The commonality of the system is modeled in an ADORA model called core model, while every variant is modeled as an ADORA aspect container. The behavior and the user-interaction of a component are described with statecharts and scenario-charts, respectively. These variability aspects are waived to the core model based on the evaluation of decision variables managed through the decision model. Product derivation is supported by creating products from this model based on the aspect weaving mechanism of ADORA.

This approach is concerned with SPL domain requirements by using use cases to connect variability with commonality. A concept similar to the positive variability proposed in [VOE07] is applied. Several visualization techniques are developed in [STO08] to enhance the ones from the ADORA language for supporting variability analysis and traceability as well as product derivation in SPLE. These techniques include visualization of constraints, visual tracing between the graphical model and the decision table as well as the consequences of adding or omitting variants in the derived product model. In [STO10], a semi-automated approach called feature unweaving is proposed. The approach starts by identifying manually the elements that vary in the ADORA model, and then automatically extracts them with the feature unweaving function that refactors the requirements model and specifies the selected elements with aspect-oriented modeling.

2.6 Methods covering the SPL Requirement Phase

The requirement phase is the first and most abstract SPL view where commonality and variability is addressed. It is to be expected that a number of SPL methods - some already mentioned in the previous sections - cover this phase. For instance, the FODA notation is the most used method for modeling variability at the requirement level. Gomaa's methods capture commonality and variability in the requirements modeling phase by defining kernel, optional, and alternative use cases, as well as modeling variation points in use cases. PLUS also provides an approach for modeling common, optional, and alternative features. Pohl et al. in [POH05] document variability in requirement artifacts by defining traceability links between the OVM model and the feature models, use case models, textual requirements, and traditional requirements

models. Ziadi et al. [ZIA04] specify PL requirements as algebraic expressions extended with constructs to specify variability on a UML sequence diagrams. Other methods that focus on the SPL requirement phase will be discussed in this section.

M_Anthon. One of the works interested in modeling variability and communality in the requirement stage of SPL is [ANTH08]. In [SOM08] an aspect-oriented use case modeling approach is proposed, where a relation called *aspect* for crosscutting requirements is defined. A composition mechanism that integrates use cases with crosscutting concerns to produce a global behaviour model is introduced. A global behaviour model is obtained by integrating all crosscutting concerns and base use cases. Crosscutting concerns defined as advice use cases are linked to the base use cases using the *aspect* relationship. The linking of advice use cases with base use cases takes place according to syntactical matching of joinpoints and pointcut expressions. A pointcut defines how advice use cases are weaved at the joinpoints. In [ANTH08], this approach is applied to software product line to generate a specific product. A relation called *variability* which is a specialization of the relation *aspect* is introduced for variability and crosscutting commonality in SPL. Variabilities are modeled as advice use cases and weaved to the base use cases based on conditions and pointcut expressions. A condition specifies whether a given member of a product line provides the functionality described by the advice use case, while a pointcut expression specifies where the advice is weaved. Instead of generating a global behaviour model of a system, a behaviour model of a specific product is generated. This approach is implemented in a tool called *Use Case Editor* (UCEd) that takes a set of related use cases written in a natural language and automatically generates executable State Charts. The approach starts by selecting

variabilities for a specific product through an integration tool included in the UCEd during the composition level. Then, A Petri nets model for this product is created from use cases as an intermediate model by mapping use cases to Petri nets. Finally, a UML State Chart is generated from Petri nets model by using the UCEd tool.

M_Braganca. Braganca et al. propose a method for formalizing UML 2.0 use cases to support variability in SPL systems. A method for integrating requirement models into model-driven methods (called *4 Step Rule Set* (4SRS)) is applied to handle SPL variability in [BRA05]. 4SRS is a model-driven method where the system requirements are specified through use cases and activity diagrams, then transformed to component diagram to derive the SPL architecture from its requirements. In [BRA06], the formalization of UML 2.0 use cases by extending its metamodel is addressed. The authors address variability in use cases through the *extend* relationship. In their proposed metamodel, the *extend* relationship can have a condition, make several extensions to a base use case, and require rejoin points different from the original extension points. In [BRA07a], they show how to transform functional requirements represented as use cases to component based requirements for SPL by extending their method 4SRS to handle variability. This extension is based on the UML-F profile [PRE02] with additional stereotypes to support variability modeling in requirements and analysis diagrams. The 4SRS method supports the feature diagrams and relates features to the SPL requirements. Since use case behavior is modeled as activity diagrams, relationships between elements of the activity diagrams and features are established through use case elements. Use case realizations are created automatically through the 4SRS method. Component diagrams are adapted to model use case realizations and a link between a feature and its realization

element is presented. A global SPL architecture model based on use case realizations is created manually by integrating these realizations as a first step for creating an executable code for SPL. Another approach for transforming UML use cases to feature models is presented in [BRA07] with the goal to obtain a use case model for a specific product. A metamodel for feature diagrams is proposed and the dependencies between features are modeled using *OCL* constraint language. The UML use case metamodel is extended to model variability. First, each use case in the SPL use case model is mapped to a feature in the feature model. Then, the feature model is transformed to a configuration feature model which is finally transformed with the SPL use case model to a product use case model. The approach is implemented with the QVT transformation language. In [AZE10], the authors represented three types of variability: alternatives, specializations and options that can be modeled in use case diagrams. Specialization is a special kind of alternative. They concluded that alternatives and specializations can be modeled through the «extend» relationship, while options through stereotyped use cases.

PLUSS. [ERI05] defines a domain modeling approach called Product Line Use Case modeling for Systems and Software engineering (PLUSS) that uses features, use cases and use case realizations for modeling variants in product line use case models. PLUSS uses a feature modeling notations based on FODA. A complete use case model is created for the whole product line and the feature model is used to instantiate a concrete product use case model. PLUSS defines four different variants in the use case models. The first variant differentiates between the use cases by relating one or more use cases to a feature in the feature model. The second variant relates one or more included use cases to a feature, while the third variant relates included scenario steps with features. The

fourth variant models cross-cutting aspects that affect several use cases as use case parameters. PLUSS distinguishes between two types of use cases: change cases that define future requirements and use cases that define accepted functionality in a domain. The integration between features, use cases and use case realization is presented in the PLUSS metamodel. The approach was applied in an industrial vehicle information system case study. A product use case model is created from a PL model by selecting features from a feature model. PLUSS approach is extended in [ERI09] for managing natural-language requirements specifications in the context of SPL. A PLUSS parameter to present variability in requirement statements is defined, which consists of a parameter nametag and a parametric feature in the feature model to specify that a selected value for the parameters results in a new child feature of the parametric feature. The parameter nametag specifies a position in requirement statements where parameter values are inserted during product model instantiation. The product model is instantiated by defining views/filters showing only those requirements that are related to this product. These filters are defined by selecting the desired features from a feature model. Then, they are exported as product requirements specifications using the custom tool support where every parameter nametag is given its concrete value. The commercial requirement management tool Telelogic DOORS is extended to support this approach and the approach is applied to an industrial case study to evaluate its performance.

M_John. One of the first works for extending use cases with a variability mechanism to employ them for modeling SPL functional requirements is [JOH02]. The authors propose the stereotype *variant* to model alternative or optional elements in use case diagrams, while in use case description, XML tags <*variant*> and </*variant*> are

used to mark optional or alternative steps. During the application engineering process, a decision is made for each variant use case whether to be included or not in a particular product based on a separate decision model, which captures the relationships and dependencies between variation points.

M_Halmans. Rather than using a feature model to document variability at the SPL requirement level, Halmans et al. in [HAL03] propose using UML use cases as a communication medium for SPL variability. An example to illustrate the documentation of functional SPL variability through use cases and scenarios is presented. Use cases and scenarios are represented through use case diagrams and use case templates, respectively. Different variants can be represented in use case diagrams using the standard relationships: *generalization*, *extend*, and *include*. However, there are some limitations in this presentation such as variation points are invisible, cardinality of variability is not modeled, and the different behaviors of variability and constraints are not presented. The example points out some considerable limitation of the UML notation and shows the need for extensions to explicitly represent variation points and to capture important constraints. Therefore, the authors propose a graphical notation which facilitate the representation of functional variability in use case diagrams to support the explicit representation of variation points and the cardinalities of the relationships between the variation points and its variants, as well as to indicate if a variation point is mandatory or not. The stereotype «variant» is used to model variant use case while a triangle is used to define a variation point which can be associated to one or more use cases using the <<include>> relationship. A "tree-like" relationship is used to express a relation between variation point and its variants. The cardinality of this relationship is modeled with a

[min..max] notation, where *min* and *max* define the minimum and maximum number of variants which have to be selected for this variation point, respectively.

PLUC. A notation for product line use cases PLUCs is proposed in [FAN04] to address SPL requirements. It divides the structure of use cases into two levels: product line level and product level in order to instantiate product use cases. Variability is expressed through tags to specify the parts of the SPL requirements that need to be instantiated. Three different types of tags are presented in the use case scenarios: alternative, parametric, and optional. Constraints are also added to this notation at the product level through Boolean conditions associated to variability tags. This notation allows expressing variants which distinguish products of SPL as well as valid combinations of variants. The authors describe how to instantiate these tags for a specific product of a mobile phone system.

M_Loughran. An approach proposed in [LOU05] integrates natural language processing (NLP) and AOSD to handle requirements analysis and concern identification to support the derivation of feature oriented models for implementation. This approach is implemented in a tool called NAPLES, which takes as input the requirements documents and produces structured models such as feature model and AORE model which shows viewpoints, composition rules, and early aspects. These structured models are exploited to derive framed aspects and classes which are used to generate code. The feature model is delineated by the framed aspect process to common and variable parts in order to identify which aspects and classes will be used in the architecture of a specific product.

2.7 Approaches addressing Non-functional Properties in SPL

This section presents related research on performance analysis of software systems, addressing quality attributes in SPL approaches. In the context of SPL, to the best of our knowledge, no work has been done to evaluate and predict the performance of a given product by generating a formal performance model. Most of the work aims to model non-functional requirements (NFRs) in the same way as functional requirements. Some of the works are concerned with the interactions between selected features and the NFRs and propose different techniques to represent these interactions and dependencies.

M_Belategi. Model-driven development and software product line paradigms are integrated together to model and validate embedded software systems in [BEL10]. An analysis method that takes into account scenarios, platform, and variability for embedded software product lines has been proposed. Although the authors consider the SPL architecture as a critical asset for representing quality attributes and their compliance to quality goals, they have not addressed how quality attributes are modeled in the architecture.

The MARTE profile is analyzed in [BEL10a], to identify the variability mechanisms of the profile in order to model variability in embedded SPL models. Although MARTE was not defined for product lines, the paper proposes to combine it with existing mechanisms for representing variability modeling and management for SPL. The paper does not explain how this can be achieved.

A model analysis process for embedded SPL is presented in [BEL11] to validate and verify quality attributes variability. The concept of multilevel and staged feature model is applied by introducing more than one feature models that represent different

information at different abstraction levels; however, the traceability links between the multilevel models and the design model are not explained.

M_Bartholdt. In [BART09] is stated that each functional feature influences all system quality attributes to some degree, making the manual tracking of quality attributes difficult. The system quality attributes are a cross-cutting concern that changes with each (de)selection of a feature. The authors propose an integrated tool-supported approach that considers both qualitative and quantitative quality attributes without imposing hierarchical structural constraints. This approach addresses the integration of quality attributes in SPL by assigning quality attributes to software elements in the solution domain and linking these elements to features. An aggregation function is used to collect the quality attributes depending on the selected features for a given product.

M_Billig. A framework for improving model reuse is provided in [BIL04]. A customized mapping between models and source code is generated according to specific requirements. Each mapping is conducted to a specific feature to specify when it can be applied. A feature model is used to define mapping alternatives and to allow the customization of the model transformation.

A literature survey on approaches that analyze and design non-functional requirements in a systematic way for SPL is presented in [NGU09]. The main concepts of the surveyed approaches are based on the interactions between the functional and non-functional features.

Svamp. An approach called Svamp is proposed to model functional and quality variability at the architectural level of the SPL [RAA08]. The approach integrates several models: a Kumbang model to represent the functional and structural variability in the

architecture and to define components that are used by other models; a quality attribute model to specify the quality properties and a quality variability model for expressing variability within these quality attributes.

M_Benavides. Reference [BEN05] extends the feature model with so-called extra-functional features representing non-functional features. Constraint programming is used to reason on this extended feature model to answer some questions such as how many potential products the feature model contains.

The Product Line UML-Based Software Engineering (PLUS) method is extended in [STR06] to specify performance requirements by introducing several stereotypes specific to model performance requirements such as «optional» and «alternative performance feature».

2.8 Discussion and Comparative Analysis

In the previous sections, a broad range of SPL development methods were discussed, where a method can cover one publication or more. In this section, we propose an evaluation framework that considers all the discussed SPL methods in which models play a significant role throughout all phases of the SPL lifecycle, starting from requirement and analysis phase (domain engineering) until design and product derivation (application engineering). The characteristics of the proposed evaluation framework cover all the roles played by models in SPL development, such as modeling variability, specifying SPL artifacts, and expressing variability in SPL artifacts.

Our evaluation framework for assessing different SPL development methods consists of two categories of criteria defining the *context* and *contents* of the method.

Each category is further divided into several elements describing different characteristics. The criteria for the method's *context* cover the following:

1. The specific goal or purpose of the SPL development method, which is further classified into four sub-elements as follows:
   a. Defining any notation for explicitly specifying commonalities and differences in order to manage variability.
   b. Generating specific product architecture or implemented components from the SPL reference architecture or from reusable components.
   c. Generating a product design model from a SPL model. The model can cover structural or/and behavioural views of SPL.
   d. Generating code for a particular product.
2. The modeling language used, which can be classified in two sub-categories: the language for specifying variability and the language for modeling SPL artifacts. It can be graphical (e.g. UML, DSML) or textual (e.g. XML). Some methods claim that they can be applied to any modeling language, but are described for a specific language. In this case, we specify the language used for illustration and indicate that the method can be adapted to other languages.
3. The tool/language providing tool support to the users of the method (if it exists).
4. Other methodologies/software development paradigms such as MDD, AOM, or generative programming (GP) integrated with the method into the SPL development in order to achieve the benefit of the combination.
5. The case studies used to verify or evaluate the method, which may be a simple example or a practical industrial case study.

The second category of criteria in our evaluation framework characterizes the method's *contents* and covers the following elements:

1. The SPL artifacts created and managed by the method. This element is further classified into three sub-elements as follows:
   a. Managing the higher level of SPL architecture view and reusable components.
   b. Covering a more detailed structural view of SPL and creating parts of the design model such as class or component diagrams.
   c. Covering the behavioural view of SPL and creating parts of the design model such as sequence or activity diagrams.
2. The notation used to express variability in SPL artifacts (e.g., UML extensions or the Orthogonal Variability Model (OVM) [POH05])
3. When using UML for variability modeling, does the method extend the UML metamodel through standard extensions (i.e., profile mechanism) or non-standard extensions (by adding metaclasses/associations to the UML metamodel)?
4. When using the *feature* concept, what kind of feature model the method is using, the original FODA [KANG90] or an extended one? We state the specific extension to FODA whenever the extension has a name; otherwise we refer to it as "extended FODA".
5. In the case of using the *feature* concept, does the method define traceability links or mappings between a feature from the feature diagram and the SPL artifacts realizing it?

In this evaluation, we call "method" a SPL development technique proposed in one or more research papers by the same authors. If the method has an accepted name, we refer

to it by its name; otherwise we use a method identifier that starts with the letter "M" followed by the surname of the first author or of the group leader.

We provide a comparative analysis below. The summary of the comparison between the SPL methods according to the context criteria is illustrated in Table 2.1.a, 2.1.b and to the content criteria in Table 2.2.a, 2.2.b. Here is a list of general remarks which represent the conclusions of the comparative analysis.

Method's goal. Some authors focused only on the higher level of SPL architecture view either to manage variability in this level or/and generate a product architecture model and implementation components, while others went a step ahead and covered the structure and behaviour views of SPL. On the other hand, a number of authors dealt only with the SPL requirement phase. We found that out of 32 reviewed papers concerned with variability management, 20 methods addressed product derivation as well; 15 methods presented the derivation of the product architecture, while 19 addressed the product derivation at the design model; 7 methods were focused exclusively on the requirements phase.

Generated Code. It seems that no method succeeds yet in generating a complete implementation code for a specific product from a SPL model by using model-driven development (MDD) techniques. Moreover, only four methods of the reviewed papers were able to generate partial of code for a product by integrating other techniques into MDD such as aspect-oriented programming [GOM05a], [GRO09], [OLD07], [TRU10].

Supported Modeling Language. A body of works makes use of the UML and its standard extension mechanism (profiles) to model variability and SPL artifacts as well as map features from the feature model to the model elements realizing them in the design

model by utilizing stereotypes and tagged values as a way to annotate the elements. It is the first step towards introducing a UML profile for SPL, which has the advantage that it is a standard extension mechanism, and therefore enables the use of UML tools without any modifications. 27 methods out of 41 surveyed make use of UML either to model variability or to model SPL artifacts, while 19 methods out of those 27 use UML for both modeling variability and SPL artifacts. The rest of the methods use DSL, XML or text. 13 methods of those that use of UML to model SPL artifacts employ the stereotype mechanism to link features to model elements realizing them. Only two methods out of those using UML extended the UML metamodel through non-standard extension mechanism [HAL03], [BRA06].

Integrated Methodology. Different methodologies and software development paradigms are integrated into SPL methods. 8 methods integrate aspect-oriented modeling or aspect-oriented software development into the SPL paradigm, while 18 methods integrate model transformation and model-driven development. 6 methods integrate aspect-oriented programming in order to generate product code [GRO09], [OLD07].

Tool support. Most of the methods are supported by a software tool. The tool is used either to implement the whole proposed approach or part of it. There is a high percentage of research providing a tool support for their proposed approaches.

Managed SPL Artifacts. Different SPL artifacts are managed by the reviewed methods such as architecture, structure, or behavior views. Most of the reviewed methods are concerned with managing the structure views of the SPL, both at a high level of abstraction of the structure view or a detailed view.

Expressing Variability. Expressing variability in the SPL artifacts is one of the most important issues allowing to manage variability between product line members. However, less than half of the reviewed approaches explicitly express variability in the SPL and only 30% of the methods make use of UML stereotypes and tagged values as a way to annotate elements in the design model.

Decision Model Used. Most of the methods use FODA notations to differentiate between SPL members and model variabilities. Other methods use different graphical notations [SIN04], variation points and variants [POH05], XML [TRU09], or tables [STO10].

Kind of Feature Model Used. When a method applies the concept of feature and feature model, FODA notation or its extended alternatives, such as CBFM, are the most widely used techniques for modeling variability among the surveyed methods. We found that 17 methods out of 25 use a FODA notation to model variability, while other methods model variability through a stereotyped UML class or use case diagram.

Mapping. The automatic derivation of a concrete product model from a SPL model is enabled through the mapping between features from the feature model and their realizations in the design model. We found that more than 70% of the reviewed methods do link features to different artifacts through different techniques. Although the techniques for setting up the mapping are different, the concept of mapping is nonetheless applied.

Addressing Quality Attributes. It is noticeable that the SPL paradigm does not give considerable attention to quality attributes or non-functional properties. We could not find any method that tackles the problem of instantiating a specific product model

from a SPL model according to quality requirements that would make use of special analysis models, such as performance, security, or schedulability models. Moreover, no work has been done to evaluate or predict the quality properties of a given product model based on a specific configuration. Few approaches are proposed to model non-functional requirements (NFRs) in the same way as functional requirements are modeled. Two of the reviewed methods are concerned with the interactions between the selected features and the NFRs [BART09], [RAA08]. They proposed different techniques to represent these interactions and dependencies, but not based on quantitative analysis models.

Table 2.1.a: Comparison between Methods' context.

Methods	Method's goal				Modeling Language used to		Tools implementing the method	Methodology integrated	Case study validiting the method
	Manage variability	Derive prod. architecture	Derive prod. design model	Generate prod. code	represent variability	describe SPL artifacts			
FODA & FORM	Yes	Yes	No	No	DSL (Tree)	DSL	NONE	NONE	Electronic system
M-Clauß	Yes	No	No	No	UML	UML	NONE	NONE	Order processing system
M-Robak	Yes	N/A	N/A	N/A	UML	UML	NONE	NONE	Mail subscription system
M-Cloi	Yes	N/A	N/A	N/A	UML	UML	NONE	NONE	Restaurant reservation system
OVM	Yes	Yes	No	No	DSL (Graphical Notation)	UML	DOORS	NONE	Home Security System
M-Korherr	Yes	N/A	N/A	N/A	UML	UML	NONE	NONE	Business process
M-Jézéquel-a	Yes	No	Yes	No	UML	UML	Java	NONE	Different case studies
M-Jézéquel-b	Yes	No	Yes	No	UML	UML	MTL for class, PLBS for SD	MTrans.	Mercure PL and Banking
M-Jézéquel-c	Yes	No	Yes	No	UML (can be adapted to DSML)	UML	NONE	AOM	Simple phone
M-Jézéquel-d	No	No	Yes	No	UML (can be adapted to DSML)	UML (can be adapted to DSML)	Kermeta	AOM	Arcade Game Maker
M-Voelter	No	Yes	partial	partial	DSL (Tree)	DSL	oAW, Xpand, Xtext	AOSD & MDSD	Home automation system
M-Czarnecki	Yes	No	Yes	No	DSL (Tree)	UML (can be adapted to DSML)	fmp2rsm	MDSD	E-commerce
M-Heidenreich	No	No	Yes	No	DSL (Tree)	UML	Feature Mapper	MDSD	Contact Management application
M-Gomaa-a	Yes	Yes	Yes	No	UML	UML	PLUSEE	NONE	Factory automation & e-commerce
PLUS	Yes	Yes	Yes	No	UML	UML	NONE	NONE	Several case studies
M-Gomaa-b	No	No	Yes	No	UML	UML	MATA	MTrans. & AOM	Microwave oven
M-Gomaa-c	No	Yes	Yes	partial	UML	UML	C++ & C#	MDA & MDD	Microwave oven & Hotel reservation
COVAMOF	Yes	Yes	No	No	DSL (Graphical Notation), XML	N/A	COVAMOF-VS	MDA	Intelligent traffic systems

Table 2.1.b: Comparison between Methods' context (Continued).

Methods	Method's goal				Modeling Language used to		Tools implementing the method	Methodolygy integrated	Case study validiting the method
	Manage variability	Derive pro. architecture	Derive pro. design model	Generate pro. code	represent variability	describe SPL artifacts			
M-Trujillo	No	No	Yes	No	XML	XML	XAK	MDD	An embedded system
M-Barreiros	Yes	No	No	No	UML	N/A	NONE	NONE	Mobile multimedia application
M-White	No	Yes	No	No	DSL	DSL	GEMS	NONE	Mobile Devices
M_Rabiser	Yes	No	No	No	DSL (Tree-based model)	N/A	DOPLER	NONE	Siemens VA system
M-Avila	No	No	partial	No	DSL (Tree)	DSL	ATC	MDE	StateMachine models
M-Haugen	Yes	Yes	No	No	UML	UML	Product derivation tool	MDA	Watch family
M-Oldevik	No	No	No	partial	DSL (Tree)	UML	Java	AOP, GP, MDD	Book store system
M-Kim	Yes	No	No	No	UML	UML	NONE	AOM	Electronic travel service
M-Stolber	Yes	Yes	No	No	DSL (ADORA & table)	DSL (ADORA)	ADORA	AOSD	Electronic security system
M-Botterweck	Yes	Yes	No	No	DSL (Tree)	UML	ATL, Java & AspectJ	M Trans. & AOP	Scientific calculators
M-Petriu	Yes	No	Yes	No	UML	UML	ATL	M Trans.	E-commerce
M-Anthon	Yes	No	Yes	No	UML	UML	UCEd	AOSD	Microwave oven system
M-Braganca	Yes	Yes	No	No	UML	UML	4SRS	M Trans.	mobile phone & library system
PLUSS	Yes	No	Yes	No	DSL (Graphical Notation)	UML	DOORS	NONE	VIS
M-John	Yes	No	Yes	No	UML & XML	UML	NONE	NONE	Cruise control system
M-Halmans	Yes	No	No	No	DSL (Graphical Notation)	UML	NONE	NONE	Flight booking systems
PLUC	Yes	No	Yes	No	Text	Text	NONE	NONE	Mobile phone system
M-Loughran	Yes	Yes	No	No	DSL (Tree)	NLP	NAPLES	NLP & AOSD	Mobile phone system
M_Belategi	Yes	Yes	No	No	N/A	N/A	NONE	MDD	NONE
M_Bartholdt	Yes	No	No	No	DSL (Tree)	NONE	pure::variants	NONE	eHealth
M_Billing	Yes	No	No	No	DSL (Tree)	UML	Triple	MDD	federated information systems
Svarrp	Yes	No	No	No	N/A	N/A	Kumbang & QPE	NONE	NONE
M_Benavides	Yes	No	No	No	DSL (Tree)	NONE	OPL Studio	NONE	NONE

Table 2.2.a: Comparison between Methods' content.

Methods	SPL artefacts managed by the method			Notations used to express variability in SPL artefacts	kind of feature model used	Non-standard extension to UML metamodel	Linking artifacts to features
	Architecture	Stucture	Behaviour				
FODA & FORM	Yes	No	No	N/A	FODA	N/A	No
M-Clauß	No	UML Class diag.	No	stereotypes	FODA expressed in UML class Diag.	N/A	Yes
M-Robak	No	UML component diag.	UML activity diag.	stereotypes	Extended FODA	No	Yes
M-Choi	Yes	No	No	stereotypes	N/A	No	No
OVM	Yes	No	No	OVM	N/A	N/A	Yes
M-Korherr	No	No	UML activity diag.	OVM	N/A	No	N/A
M-Jézéquel-a	No	UML Class diag.	UML SD	stereotypes	N/A	No	No
M-Jézéquel-b	No	UML Class diag.	UML SD	stereotypes	Extended FODA	No	No
M-Jézéquel-c	No	UML Class diag.	UML StateChart diag.	N/A	N/A	N/A	N/A
M-Jézéquel-d	No	No	UML Activity diag.	N/A	N/A	N/A	N/A
M-Voelter	Yes	Class Dia.	State machine	N/A	Extended FODA	N/A	Yes
M-Czarnecki	No	UML Class diag	UML activity diag.	stereotypes	Cardinality-based feature model	No	Yes
M-Heidenreich	No	UML Class diag	UML StateChart diag.	N/A	Cardinality-based feature model	N/A	Yes
M-Gomaa-a	Yes	UC, feature, class diags.	Collaboration, Statechart diags.	stereotypes	N/A	No	Yes
PLUS	Yes	UC, feature, class, component diags.	Communication, statechart diags.	stereotypes	Feature dependency class diag.	No	Yes
M-Gomaa-b	No	UML Class diag.	UML StateChart, SD	N/A	Feature dependency class diag.	No	Yes
M-Gomaa-c	Yes	UML component diag.	UML StateChart	N/A	Feature dependency class diag.	No	N/A
COVAMOF	Yes	No	No	COVAMOF Variability View	N/A	N/A	Yes
M-Trujillo	No	XML model	No	Attributes	N/A	N/A	Yes
M-Barreiros	No	No	No	N/A	Extended FODA	N/A	Yes

Table 2.2.b: Comparison between Methods' content (Continued).

Methods	SPL artefacts managed by the method			Notations used to express variability in SPL artefacts	kind of feature model used	Non-standard extension to UML metamodel	Linking artifacts to features
	Architecture	Studure	Behaviour				
M-White	Yes	No	No	N/A	FODA	N/A	No
M_Rabiser	No	No	No	N/A	N/A	N/A	No
M-Avila	No	No	No	N/A	Cardinality-based feature model	N/A	Yes
M-Haugen	Yes	UML Class diag.	No	stereotypes	UML use cases	No	N/A
M-Oldevik	No	UML Class diag	No	N/A	FODA	N/A	No
M-Kim	Yes	Studural model	SD	N/A	N/A	No	N/A
M-Stoiber	Yes	No	StateChart & scenario	N/A	N/A	N/A	Yes
M-Botterweck	Yes	No	No	N/A	Extended FODA	N/A	Yes
M-Petriu	No	UC, feature, class, deployment diags.	SD	stereotypes	Feature dependency class diag.	No	Yes
M-Anthon	No	Use case	StateChart	N/A	Extended FODA	No	No
M-Braganca	Yes	Use case	Activity Dia.	stereotypes	Feature dependency class diag.	Yes	Yes
PLUSS	No	Use case	No	PLUSS Notation	Extended FODA	No	Yes
M-John	No	Use case	No	stereotypes and XML tags	N/A	No	N/A
M-Halmans	No	Use case	No	Graphical Notations	N/A	Yes	N/A
PLUC	No	Use case scenario	No	tags	N/A	N/A	N/A
M-Loughran	No	Requirements documents	No	N/A	Extended FODA	N/A	Yes
M_Belategi	Yes	Yes	No	No	N/A	N/A	N/A
M_Bartholdt	No	No	No	N/A	FODA	N/A	N/A
M_Elling	Yes	No	No	N/A	FODA	No	Yes
Svamp	Yes	No	No	N/A	N/A	N/A	N/A
M_Benavides	N/A	N/A	N/A	N/A	FODA	N/A	N/A

Chapter 3: Background Review

This chapter presents the background material needed for this research. The concept of platform independent and specific is addressed in section 3.1. UML and MARTE profile as well as performance analysis techniques are presented in section 3.2. Different model transformation languages are described in section 3.3.

3.1 Methods handling Platform Independent/Specific Concept

This section surveys briefly work from literature related to software performance engineering in the context of Model-Driven Architecture (MDA), an initiative introduced by the Object Management Group as an approach to system-specification and interoperability based on the use of models [OMG04]. In MDA, platform-independent models (PIMs) are composed with platform models in order to produce platform-specific models (PSM).

The Model-Driven Architecture approach is extended in [COR07] with non-functional modeling and analysis concepts by adding new models and transformations for validation activities.

A model transformation framework is proposed in [VER05] for automatically including the impact of middleware on the architecture and the performance of distributed systems. Different middleware descriptions (i.e., platform models) are stored in a library, so the designers can compose the PIM of their applications with different

types of middleware and derive a platform-specific model. A Layered Queueing Network (LQN) model build by hand from PSM is used for performance evaluation.

An approach for performance prediction of component-based software systems proposed in [BAL06] is based on operational analysis of QN models where performance bounds are computed without deriving a QN model from the software specification. Performance bounds for system throughput and response time are used to answer several performance-related what-if questions, such as identifying the bottleneck resource if the platform configuration is changed.

A method for designing parametric performance completions is proposed in [HAP10]. The variability in the platforms is described by using a feature model. The completions can be instantiated for different environments by explicitly coupling the transformations to performance models and implementations to add the necessary details to both.

A method called "coupled transformations" (CT), whose goal is to produce more accurate predictive performance models from a software model is proposed in [BEC08]. CT refers to two parallel transformation chains: one is generating code from a source software model, while the other is producing a predictive analysis model for a given non-functional property, such as performance. The CT method keeps the two transformation chains in sync, by using knowledge about the transformations in the first chains to improve the outcome of the transformations from the second chain.

A queueing model for the performance of Web servers is presented in [MEI01]. The model includes the impacts of workloads, hardware/software configurations,

communication protocols, and interconnect topologies. It is implemented in a simulation tool and the results are validated with results from a test lab environment.

A literature survey on approaches that address non-functional requirements (NFRs) is presented in [CHU09]. The classification is based on software variability, requirements analysis, elicitation, reusability, and traceability as well as aspect-oriented development. Variability related to SPL is also discussed.

3.2 Analysis of Non-Functional Properties

Model-Driven Development (MDD) is a well-known paradigm that aims at capturing every important aspects of developing a software system through models. Being focused on models, MDD facilitates the analysis of non-functional properties (NFP) (such as performance, reliability, scalability, and security) of software and system designs. Software models annotated with extra information corresponding to the property to be evaluated are automatically transformed into appropriate analysis models. Over the last decade, several modeling formalisms and tools (such as queueing networks, Petri nets, process algebras, Markov chains, fault trees, probabilistic time automata, formal logic, simulation) have been developed for the analysis of different NFP.

In order to bridge the gap between software analysis and non-functional properties analysis, OMG introduced the *UML profile for Schedulability, Performance, and Time Specification* (SPT) and its recent replacement, the *UML profile for Modeling and Analysis of Real-Time and Embedded Systems* (MARTE) [MARTE11]. These profiles represent a lightweight extension of UML with timing and resource information required for schedulability and performance analysis. Adding quantitative performance

annotations to UML models enables their transformation into performance models, which can be analyzed with existing performance analysis tools and techniques. The following section gives a brief overview of using UML and MARTE profile for performance analysis modeling. Section 3.2.2 discusses various works in the area of performance analysis and modeling.

3.2.1 UML and MARTE Profile

The Unified Modeling Language (UML) is a standardized general purpose modeling language which provides a set of graphical notation for analysis, design and implementation of software systems. This standardized modeling language is used for the specification, visualization, architecture design, construction, simulation and testing as well as documentation of software systems.

UML provides a set of modeling concepts and notations to meet the requirements of various software modeling systems. However, designers may need to create new model elements or attach some extra information to the model elements. They may also desire to have additional features and/or notations beyond those defined in the standards. This can be achieved in two ways: either by changing the UML metamodel (known as heavyweight extension) or by defining a profile - a set of stereotypes, tagged values and constraints (known as lightweight extensibility mechanisms). Although modifying the UML metamodel by adding or changing its metaclasses offers the highest degree of flexibility, this has the big disadvantage of losing the existing tool support. On the other hand, the lightweight UML extension mechanisms does not change the UML metamodel, which means that the existing standard UML modelling tools can be used without any

extensions or adaptations. Hence, it is recommended to use lightweight extensions supported by three UML built-in extension mechanisms:

1) *Stereotypes*. A stereotype is a model element that extends an existing UML metaclass or inherits from another stereotype. Booch et al. [BOO99] define a stereotype as "an extension of the vocabulary of UML models, which allows you to create new kinds of buildings blocks that are derived from existing ones but are specific to your problem".
2) *Tagged Values*. Stereotypes can have properties referred to them as tags (similar to class attributes). The values of these properties are the tagged values that can add extra information to stereotyped model elements. Tagged values are defined as "an extension of the properties of a UML element, which allows you to create new information in that element's specification" in [BOO99].
3) *Constraints*. These are conditions or restrictions that refine the semantics of a model element. A constraint can be written in OCL, or a natural or programming language.

The above extensibility mechanisms help us customize and extend UML by adding new concepts, introducing new properties, and specifying new semantics in order to make the language suitable for any specific problem domain. Stereotypes, tagged values and constraints are used to define UML profiles which are specializing UML for different application domains. One of these is the *UML Performance Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE)* [MARTE11] which adds concepts relating to timing, resources, and quantitative performance and schedulability information to the UML models.

MARTE defines the foundations for model based description and analysis of both software and hardware aspects of real time and embedded systems. The aim of the profile

is not to define new techniques for analyzing real-time and embedded systems, but to support existing techniques by adding facilities to annotate models with information required to perform a specific analysis. MARTE focuses on performance and schedulability analysis. At the same time, it defines a general analysis framework to refine or specialize other kinds of quantitative analysis based on scenarios and resources (such as reliability, availability, safety, etc.). Using MARTE has the advantage of providing a common technique to model hardware and software aspects and allowing interoperability between development tools [MARTE11]. Quantitative analysis domains have different terminology, concepts, and semantics; however they also share some common quantitative analysis concepts which are defined in the *Generic Quantitative Analysis Model (GQAM)*. GQAM is further specialized by subprofiles such as performance and schedulability, for analysis based on how the software behaviour uses the system resources, [MARTE11]. The sub-profile *Schedulability Analysis Model* (SAM) supports schedulability analysis to predict whether a set of software tasks meet their hard deadlines, while the *Performance Analysis Model* (PAM) subprofile supports performance analysis of a system with non-deterministic behaviour and stochastic characteristics.

The PAM domain extends the GQAM domain, which contain three main types of concepts: workload, resources and behaviour. *WorkloadEvent* describes a stream of arriving events, focusing on workload types with open and closed arrivals, workload generators and traces. PAM uses the behaviour-causality model of *Scenarios* and *Steps* and extends the properties of GQAM *Steps* to include more kinds of operation demands during a step such as *PassResource* which identifies the passing of a shared resource

from one process to another. It also adds the possibility of an asynchronous (non-synchronized) parallel operation, which is forked but never joins [MARTE11]. The PAM domain model includes the GQAM resource package. Important types of resources are hardware processors (*ExecutionHost*), concurrent process threads (*Concurrent Resource*), and *LogicalResources* defined in the software (such as semaphore, lock, buffer pool, or a block of memory). A process resource, or pool of process threads, is also a kind of logical resource which is modeled separately, by the concept of *SchedulableResource* imported from the General Resource domain model (GRM). A special concept of *RunTimeObjectInstance* is introduced in PAM to represent an alias for a process or thread pool resource identified in behaviour specifications by lifelines and swimlanes. The *AnalysisContext* imported from GQAM indicate which behaviours are included in the analysis scope. It has a set of annotation parameters that can be used in MARTE expressions. A context may have any number of workloads, representing different sources of requests or initiations of operations. In a performance analysis, a workload corresponds to a class of traffic, which can be open or closed. An open workload is a *RequestEventStream* in which the events arrive at a given rate in some predetermined pattern. On the other hand, a closed workload defines a stream generated by a fixed number of active or users which cycle between requesting to execute a *BehaviorScenario*, and spending an external delay period called a *Think Time* outside the system.

PAM supports communications modeling through mechanisms for conveying messages called *CommChannel* and a kind of *Step* called *CommunicationStep* which represents the sending/receiving of a message. The *CommunicationStep* annotations determine the level of modeling the cost and delay of communications.

The MARTE annotations for performance properties used in this work are explained in more detail in section 4.3.4.

3.2.2 Performance Analysis and Modeling Techniques

Performance is an important quality attribute of every software systems, especially for real-time systems. Many performance failures are due to architecture or design reasons, meaning that performance problems are introduced in the early phases of the development process. However, most of the time performance problems are ignored until later in the life cycle, when the system is built and can be measured; at this stage, fixing the problems is much more difficult and expensive. It has been recognized that performance analysis needs to be addressed from the early design phases throughout the software life-cycle. Performance models are able to predict performance problems during the architectural and early design phases of the project, allowing evaluating design alternative under different workload conditions [SMI90]. The goal is to help developers to choose better design alternatives as early as possible, so that the systems being built will meet their performance requirements.

Software Performance Engineering (SPE) is a methodology introduced in [SMI90] that promotes the integration of performance analysis into the software development process from the early stages and continuing throughout the whole software life cycle. Software performance models are created from frequently executed scenarios annotated with performance parameters. There are various ways of describing these scenarios in UML, two of which are Interaction Diagrams and Activity Diagrams.

Since the introduction of SPE, there has been a significant effort to integrate performance analysis into the software development process by using different performance modeling paradigms: queueing networks, Petri nets, stochastic process algebras, simulation, etc. [BAL04]. An annotated UML model can be transformed into a performance model and analyzed with known analysis techniques and tools [BAL04][WOO08]. Traditionally, performance analysis models were built "by hand" by specialists in the field, who "abstracted" from the software only the properties of interests. However, in the context of model-driven development, a new approach for constructing analysis models has emerged, where software models with performance annotations are automatically transformed into performance analysis models.

A good survey of transformations of software models into different performance models is given in [BAL04]. Examples of such transformations are from UML to Layered Queueing Networks in [PET02], to Stochastic Petri Nets in [BER02], and to Stochastic Process Algebra in [CAV04]. [COR00] introduces a similar technique following the SPE methodology for generating the same kind of models as in [SMI90]. [KAH01] presents a UML-based notation and framework for describing performance models. [LOP04] transforms a UML models into Petri Nets, but without taking into consideration the contention for hardware resources.

Usually, the interpretation of the performance model results is done by a performance analyst, who understands the formal performance model. Current research is being done to diagnose the performance problems automatically (by following a set of rules similar to the experts) and to suggest advice for improvement in terms that the software developers can easily understand.

In our research group, we are using the transformation framework PUMA described in [WOO05], [WOO08]. PUMA is a tool architecture that provides a unified interface between different kinds of design specifications and different kinds of performance models in the form of an intermediate model called Core Scenario Model (CSM). CSM captures the essence of software performance specification and estimation, and strips away the design detail which is irrelevant to performance analysis. It converts annotated UML models into different kinds of performance models (Layered Queueing Networks, Queueing networks, stochastic Petri Nets). After the performance model is generated, it can be analyzed with existing solvers and feedback regarding its performance properties will be given to the software development team.

The performance modeling formalism used in our research group is the Layered Queueing Model (LQN) [WOO95]. LQN is an extension of the well-known Queueing Network model developed for modelling software systems, which contain nested services. A software server often requires services from other servers in order to fulfill the requests of its own clients [WOO95].

3.3 Model Transformation Language and Tools

Model transformation plays an essential role in MDD since it changes the use of model from a documentation artifact into a more extensive model-driven usage where implementations can be obtained. A model transformation is the process of converting one model to another model of the same system [OMG04]. Ultimately, this means converting a source model into a target model, both of which conform to their respective metamodels. A transformation is based on a set of rules, each expressing a relationship

between one or more source elements and one or more target elements. Depending on the target, there are *model-to-model* and *model-to-text* transformations.

There are several tools and languages for model transformations such as ATL [ATL], RubyTL [SAN06], ATC [EST], QVT [OMG08], and MOFScript. The general purpose model transformation languages such as ATL, QVT, and ATC have already an existing implementation and execution transformation engines to execute them. However, other domain-specific languages require creating a compiler or virtual machine in order to be executed [AVI07].

3.3.1 QVT

In response to a demand for a transformation framework of MDA-compliant models, OMG released the MOF 2.0 Query/View/Transformations (QVT) standard [OMG08]. QVT has three domain-specific languages: Relations, Core, and Operation Mappings. The QVT Relations metamodel and language is a declarative specification that supports complex object pattern matching between MOF models, as well as the creation of trace classes to record the details of a transformation execution [OMG08]. QVT Relations transformations define a set of conditions of the candidate models that must be satisfied before a transformation from one to the other can take place. The QVT Core metamodel and language uses minimal extensions to the Essential MOF and OCL languages. It only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models. As a result, transformation descriptions are defined in a far more verbose technical manner. In addition, unlike the Relations language, trace models have to be explicitly defined. The core model may be directly implemented, or

derived from a Relations model. This is similar to how Java code is transformed in to byte code by the Java compiler [OMG08]. The Operation Mappings language is a specification for providing imperative transformations. It supports several extensions to OCL which allow a more procedural syntax for describing complex relations between models. Operations Mappings use the same trace models as Relations language, meaning they can be used interchangeably with the latter.

3.3.2 ATL

The Atlas Transformation Language, developed by the ATLAS INRIA & LINA Group, is a model transformation tool that is based on MOF 2.0 QVT Core. It is built on top of a model transformation virtual machine. ATL is a hybrid between declarative and imperative programming [ATL]. Declarative programming enables expressing the mapping between the source and target model elements. However, imperative constructs are used to specify mapping that is difficult to express declaratively.

An ATL transformation is composed of a set of rules and helpers. The rules are written in OCL syntax to define the mapping between the source and target model. More specifically, a rule selects a certain type of model element from the source model and navigates the source model around it to find other model elements necessary for creating target model elements and for initializing their attributes. The helpers, which are analogous to Java methods, allow the developer to specify factorized ATL code that can be invoked from different points in the ATL transformation module [ATL]. All helpers have a name, context type, return value type, code expression, and an optional set of parameters.

ATL has two modes of execution, default and refine. Default mode requires the developer to specify the rules that will generate all modified and unmodified elements from the source to target models. Refine mode eases programming in cases where the source and target models are similar.

The model transformations in this research are realized in ATL. A brief description of the ATL syntax to help understanding the transformation rules in chapter 5 is given here: a) the navigation from an element to its attribute is expressed by the element name followed by a dot and the attribute name; b) literals are expressed between quotes; c) a value assignment is done via the left arrow operator "←"; d) filter conditions on the source model are expressed in OCL; and e) a comment statement begins with double dashes "--".

Chapter 4: Proposed Product Derivation Approach

This chapter presents the UML model transformation approach proposed in this thesis for deriving a UML model with performance annotations for a specific product from an annotated SPL model. The product derivation process is defined as a complete process of constructing products from a SPL model. Before explaining our approach in section 4.4, a UML profile for modelling the variability and commonality in SPL is presented in section 4.1. An efficient mapping technique is proposed in section 4.2 to set up traceability links between features from the feature model and model elements realizing them in the design model. The proposed approach is illustrated through a case study of an e-commerce application; section 4.3.1 describes the case study and shows how its variability is modeled. The feature model for representing variability in SPL is explained in section 4.3.2. The SPL model, which is the source model of the proposed model transformation, is presented in section 4.3.3, while section 4.3.4 explains the MARTE profile annotations for performance.

4.1 UML Profile for SPL Variability (SPLV)

Generally a UML model represents a single software system; however the model of a software product line represents multiple products in the same domain. In order to model it, we need to differentiate between elements shared by all the products, and elements that

may vary from one product to another. The notions of *commonality* and *variability* are used to designate common and variable elements in a SPL [ZIA06].

Modeling variability and commonality between products is an important activity for developing software product lines. An important notion in SPL is that of *feature,* used to denote a system property such as any functional or non-functional characteristic at the requirement, design, architectural, platform, or any other level. Features are used to differentiate among members of the product line. Gomaa defines in [GOM05] a feature as "a requirement or characteristic that is offered by one or more members of the product line". The commonality and variability of a SPL are captured by means of features and their dependencies in the feature model. The feature model is an essential technique for managing the common and variable features of a SPL throughout all its development stages. Feature model also plays a central role in the automatic derivation of product members of SPL. A specific product is configured by selecting a valid feature combination – called *feature configuration* – from the feature model based on the product's requirements.

A graphical notation FODA for feature models was introduced in [KANG90]. However, in our work we needed to represent the feature model in the same language as the SPL model, because both are input to the proposed model transformation. Since UML is the language of choice for modeling software designs, we chose UML for the SPL model and consequently for the feature model. We propose here a UML profile for SPL variability, called SPLV, which is mostly based on Gomaa's work PLUS [GOM05] for the following reasons:

1) PLUS is a well-developed method with precise expression of common and variable characteristics.
2) It is based on the standard UML profiling mechanism, which allows extending UML without changing its metamodel or tool support. This enables us to create annotated UML models with any UML editor that conforms to the UML standard.
3) It allows us to use the annotation approach for expressing variability in different UML diagrams (as described in section 4.2), which is simpler than modeling variability through other mechanisms and mapping it to UML diagrams (for instance, using the orthogonal variability model (OVM) introduced in [POH05]).
4) All the phases of SPL development process are covered with a well-organized methodology to proceed from one phase to the other.
5) It is applied to real-time systems and pays a lot of attention to the representation of behaviour, which is very important for performance analysis.
6) The feature model is represented in PLUS as a class diagram, which is more intuitive than other approaches (for instance, simpler than modeling features as OCL constraints).
7) The feature dependency class diagram is able to cover the entire notations of the well-known feature modeling FODA and its extensions.
8) Since the PLUS methodology is designed specifically for SPL, it satisfies both variability representation and product derivation.

However, our UML SPL profile has several significant differences from PLUS.

1) The SPL behaviour view is modeled in our work by sequence diagrams instead of collaboration (communication) diagrams as in PLUS, due to the fact that the

modeling power of sequence diagrams has been considerably enhanced in UML 2 with respect to communication diagrams.

2) Handling variability within a sequence diagram is done in our approach by extending the *alt* and *opt* operators that represent *alterative* and *optional* features, respectively.
3) Different stereotypes and tagged values are proposed to map variability during the construction of a SPL model to the run-time variability of a specific product.
4) A variant element in a sequence diagram is mapped to the feature(s) requiring it through a tagged value of the stereotype associated to the model element.
5) We pay attention to the deployment of software components to hardware devices, which is also important for performance analysis, while PLUS does not.
6) We deal with MARTE performance annotations, both in the structural and behavioral views of SPL.
7) An extension point in a base use case can be extended by several optional extending use cases, each one representing an *optional* feature.
8) Each *optional* and *alternative* use case is linked to the feature(s) requiring it from the feature model through a tagged value associated with the use case stereotype. We express the traceability links between features and model elements realizing them through the annotation approach (as explained in section 4.2).
9) We introduced the concept of *quality feature* in the feature model, which characterizes design decisions that have an impact on the non-functional requirements or properties.

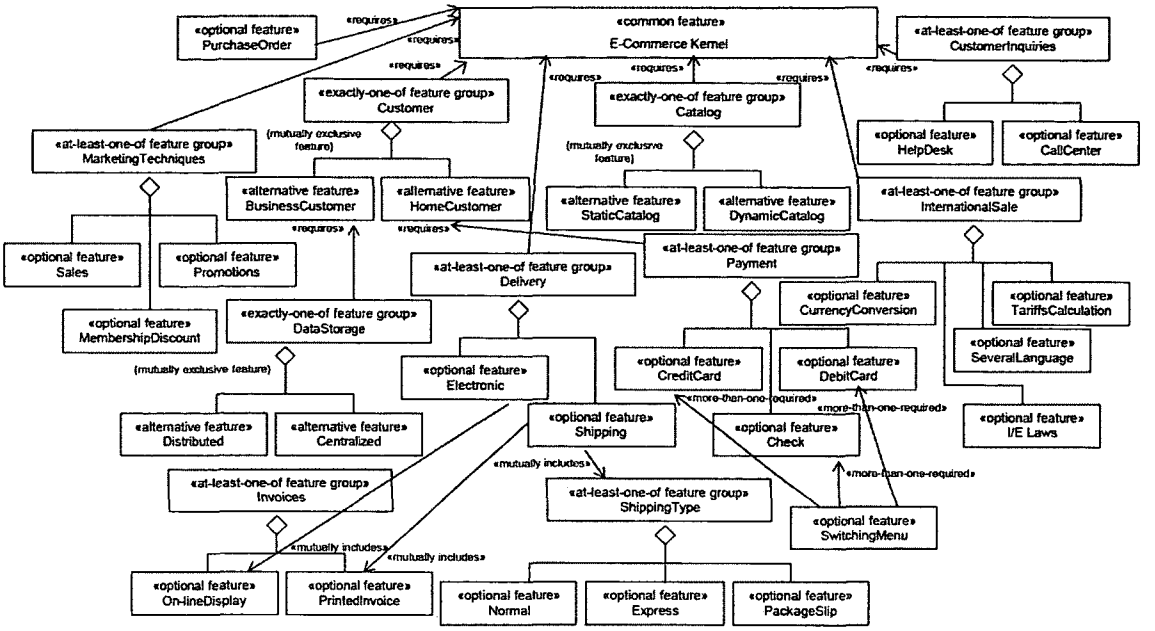


Figure 4.1: Feature model of the e-commerce SPL.

10) We introduce another kind of constraint between features in the feature model to represent additional restrictions on the feature selection. The *more-than-one-required* constraint is used to restrict a selection of a feature that requires at least the selection of other two features.

We believe that the SPLV profile contains all the necessary concepts because we surveyed all the works in this area found in the literature and integrated them [CLA01], [ROB02], [CHO05], [PRE02], [KOR07], [ZIA03], [GOM05]. The reason we have not made the definition of SPVL a central part of this thesis research is due to the fact that in December 2009 OMG has issued a Request for Proposal titled "Common Variability Language (CVL), whose objective is to enable the specification of the variability in product line models in order to support seamless product line modeling across the whole product line engineering process [OMG09]. Since we could not wait for the CVL

standard to be defined and adopted, we proceeded with our own SPVL definition, but we are aware that an OMG standard will make SPVL superfluous.

4.1.1 UML-based Profile for Feature Model

In our research, instead of using a hierarchical tree representation of the feature model as in FODA [KANG90], we use a dependency class diagram, where stereotyped classes represent either features or feature groups (see Figure 4.1). Different stereotypes are used to distinguish between features such as *«Common Feature»*, *«Optional Feature»*, and *«Alternative Feature»* as shown in Appendix A. Related features are grouped into feature groups, which define constraints on how the features are used by a product. Other stereotypes are used for feature groups such as *«At-Least-One-of-Feature Group»*, *«Exactly-One-of-Feature Group»*, and *«Zero-or-one-of Feature Group»* (see Appendix A). Different association stereotypes are used to model dependencies between features such as *«requires»*, *«mutually-includes»*, and *«mutually-exclusive»*, describing the way in which features can be combined within the same SPL. We introduce another kind of dependency *«more-than-one-required»* to represent a choice that requires at least two optional features. In our approach, we also distinguish between two types of features: functional and non-functional features. Functional feature represents a requirement as described in [GOM05]. On the other hand, a non-functional feature represents a quality feature characterizing design decisions that have impact on the non-functional requirements or concerns.

A feature configuration is a valid selection of features that corresponds to a product of the family [CZA05c]. In order to enable the selection of features defining a

configuration, each feature in the feature model has a tagged value associated to its stereotype indicating whether it is selected or not.

Another kind of feature model is introduced in our research to model variability in platform/environment. The stereotype *«PC- feature»* indicates a PC-feature, where the tagged value *{attList}* is a list of MARTE attributes affected by this PC-feature. Another tagged value *{guidance}* contains a list of guideline information.

4.1.2 UML-based SPL Profile for Structure Views

When modeling the structure of a SPL, variability is introduced through abstract classes and subclasses and through parameterization. A generalization/specialization relationship is used to model different variants of a class. The inheritance mechanism is used to specialize the superclass into subclasses, each one representing different features. For parameterization, a class can have configuration parameters which have to be assigned different values for different features. The classes are categorized into *Kernel*, *Optional* and *Variant* class, which are depicted with stereotypes (see Figure 4.2.a to 4.2.e and Appendix A). A *kernel* class is required for all members of the SPL, while an *Optional* class is required for some members only. *A Variant* class represents alternative choices.

In our approach, each *Optional* and *Variant* class in the class diagram is mapped to the feature(s) from the feature model that require the respective class through tagged value(s) associated to its stereotype, as shown in Figure 4.2.a to 4.2.e.

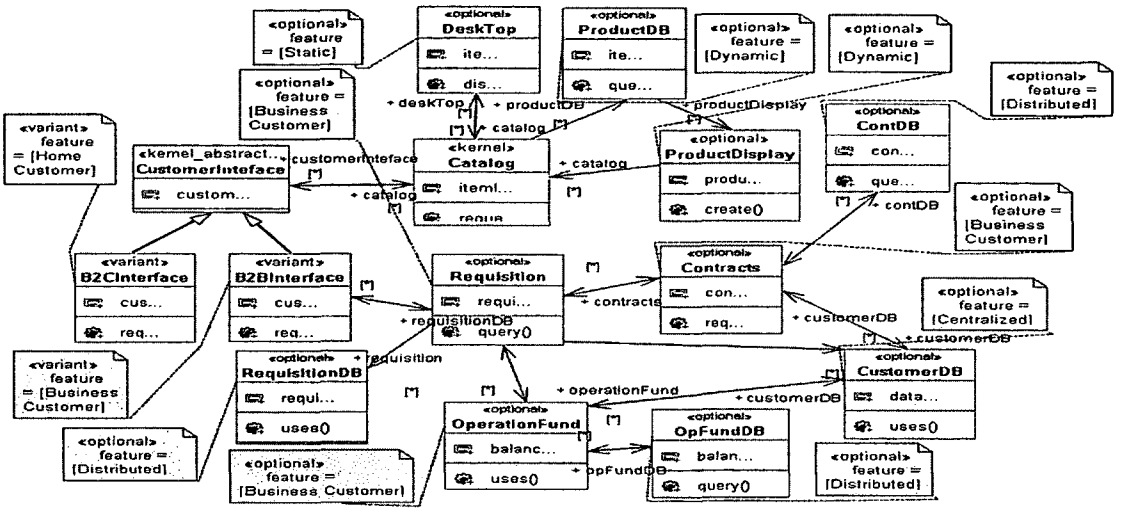


Figure 4.2.a: Part of the class diagram of the e-commerce SPL.

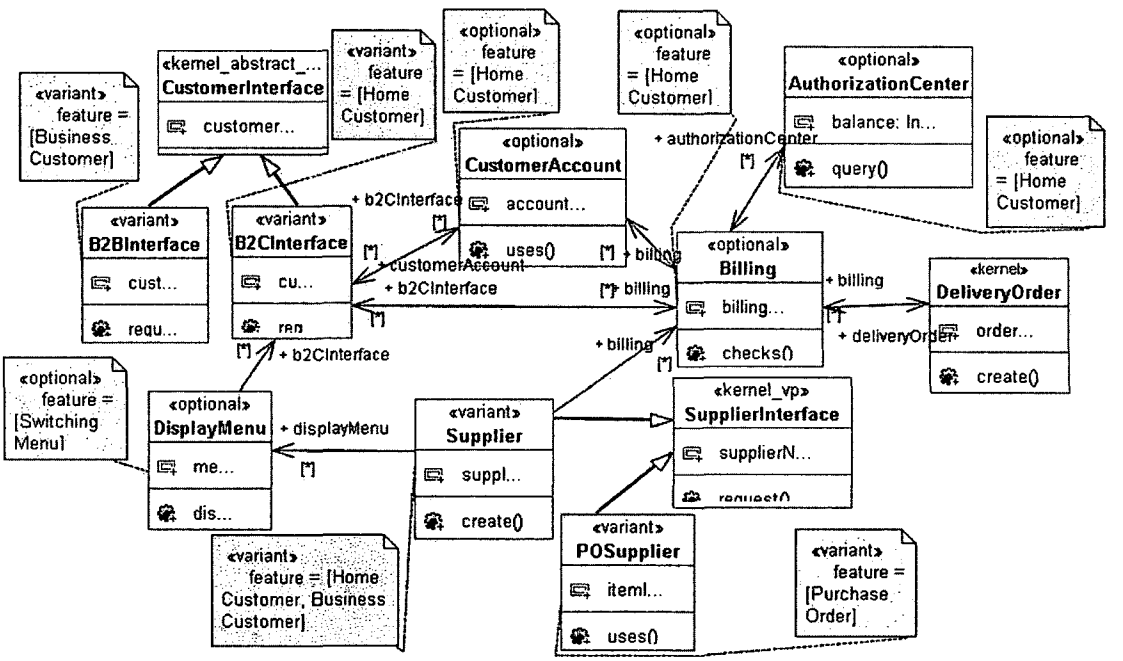


Figure 4.2.b: Part of the class diagram of the e-commerce SPL.

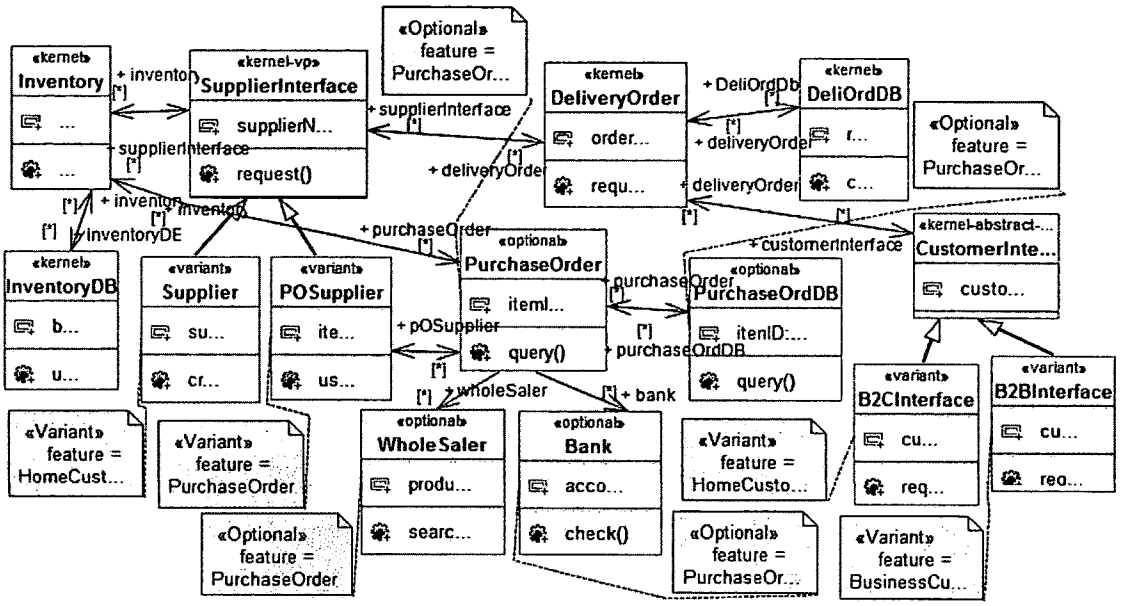


Figure 4.2.c: Part of the class diagram of the e-commerce SPL.

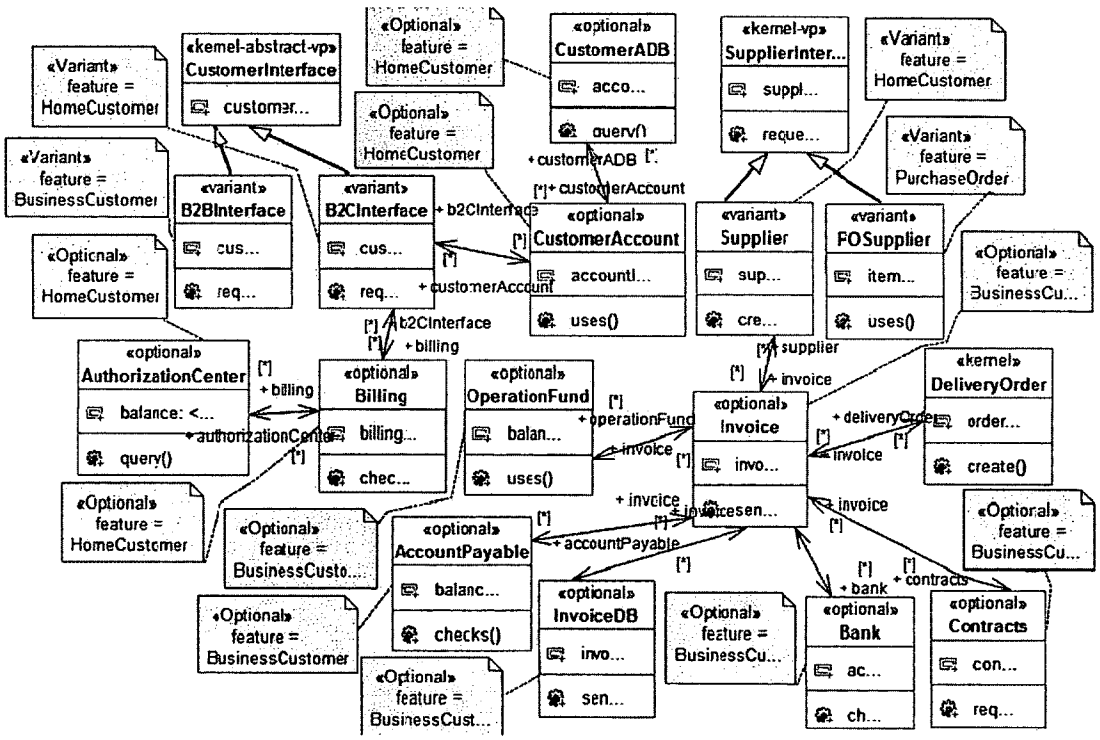


Figure 4.2.d: Part of the class diagram of the e-commerce SPL.

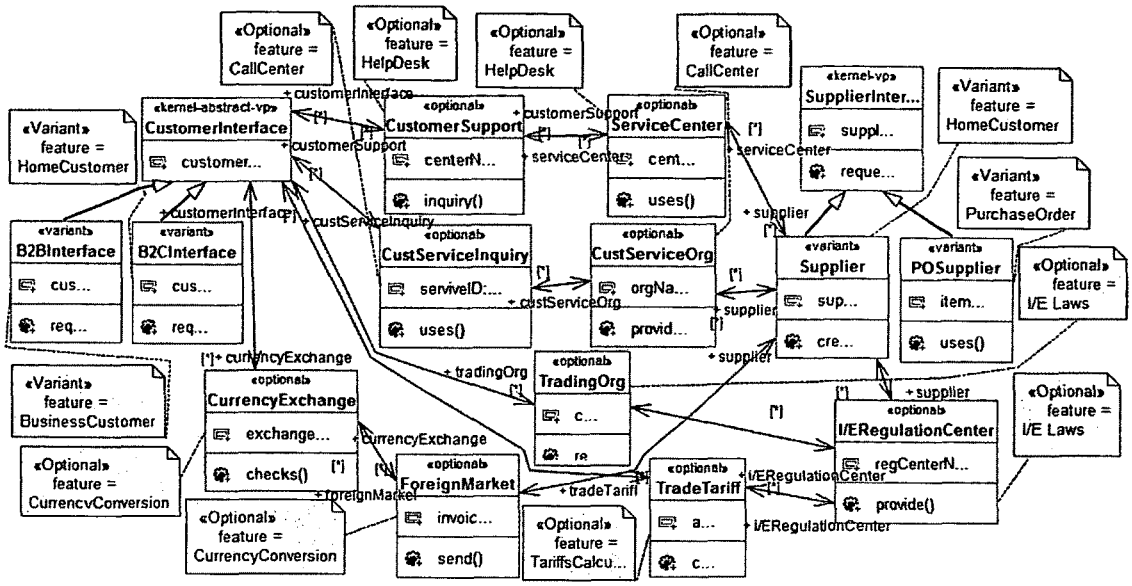


Figure 4.2.e: Part of the class diagram of the e-commerce SPL.

4.1.3 UML-based SPL Profile for Behavioural Views

In order to model the functional requirements of a SPL, Gomaa extends the use case model to differentiate between types of use cases [GOM05]. The stereotypes *«Kernel»*, *«Optional»*, and *«Alternative»* are used to differentiate between use cases that are: a) always required by all members of the SPL, b) required by some but not all members, and c) use cases in which a choice must be made, respectively. Furthermore, the use case variability can be handled through variation points. *Variation points* are one or more locations in a use case scenario where variation could occur. For a small variation, a variation point is identified in the use case scenario to specify the location where change can take place. *Variation points* in a use case scenario are used to define *optional* or *alternative* functionality in SPL. However, for complex variations, the *extend* and *include* relationships between use cases may be used to model variations. Extension

points are used in the base use case scenario to define specific locations where extensions will be added. Alternative variability is modeled by an extension point with several extending use cases, each one defining an alternative. Optional variability is modeled in [GOM05] as an extension point with only one extending use case corresponding to the optional functionality. In our approach, optional variability can be expressed as an extension point with several extending use cases (see Appendix A for the profile).

Regarding behaviour modeling, variability can be modeled in statecharts by using inherited or parameterized statecharts [GOM05]. It is also possible to model variability in sequence or communication diagrams representing each scenario of each use case categorized as *«Kernel»*, *«Optional»*, or *«Alternative»*. In our approach, the SPL behaviour view is modeled as sequence diagrams rather than communication diagrams as used in [GOM05].

A SPL sequence diagram that represents a scenario is generic, describing possible interactions between objects corresponding to more than one feature. In order to be able to model variability within a sequence diagram, we utilize the capabilities and mechanisms provided by UML 2.0. A sequence diagram (SD) in UML 2.0 can be composed from a number of *Combined Fragments* using a set of operators called *Interaction Operators* and a set of operands called *Interaction Operands* (see [OMG07a], page 464). Both *alt* and *opt* operators are used to model variability within a so-called *Variability Combined Fragment* (VCF). In UML, the *alt* operator defines a choice of behaviour where at most one of its operands will be chosen at run-time. To model variability at design time, we stereotype (i.e., extend) the *alt* operator with *«AltDesignTime»* to indicate that the choice happens during product derivation. Similarly,

an extended *opt* operator with the stereotype *«OptDesignTime»* can be used to model optional features. More complex *optional* and *alternative* features can be modeled by extending the *ref* operator associated to an *Interaction Use*.

A small-grained alternative feature presented as a variation point in a use case scenario can be modeled as an extended alternative *Combined Fragment* using the *alt* operator (see [OMG07a] page 468). The alternative *Combined Fragment* is stereotyped as *«AltDesignTime»* with a tagged value referring to the name of the feature group. The guard of each *Interaction Operand* of the alternative *Combined Fragment* is mapped to the feature requiring it. For example, the SPL scenario *ConfirmDelivery* shown in Figure 4.3, contains an alternative *Combined Fragments* corresponding to the features *Distributed* and *Centralized* of the feature group *DataStorage*; the fragment is stereotyped *«AltDesignTime»* with tagged value *{VP=DataStorage}* and each one of its *Interaction Operands* has a guard denoting the corresponding feature. A more complex *alternative* feature represented as an extending use case can be also modeled as an extended *alt* operator which contains an *Interaction Use* referring to the *Interaction* representing the extending use case.

A small-grained *optional* feature can be represented as an extended *opt* operator, while a complex *optional* feature can be represented as an *Interaction Use* referring to an *Interaction* representing this feature. For example, in the SPL scenario *BillCustomer* shown in Figure 4.4, behaviour corresponding to each feature from the group *Payment* is modeled as an optional *Combined Fragment* stereotyped *«OptDesignTime»* with a tagged value *{VP=Payment}*. For instance, the behaviour corresponding to the *optional* feature *CreditCard* is modeled by an optional *Combined Fragment* containing an *Interaction Use*

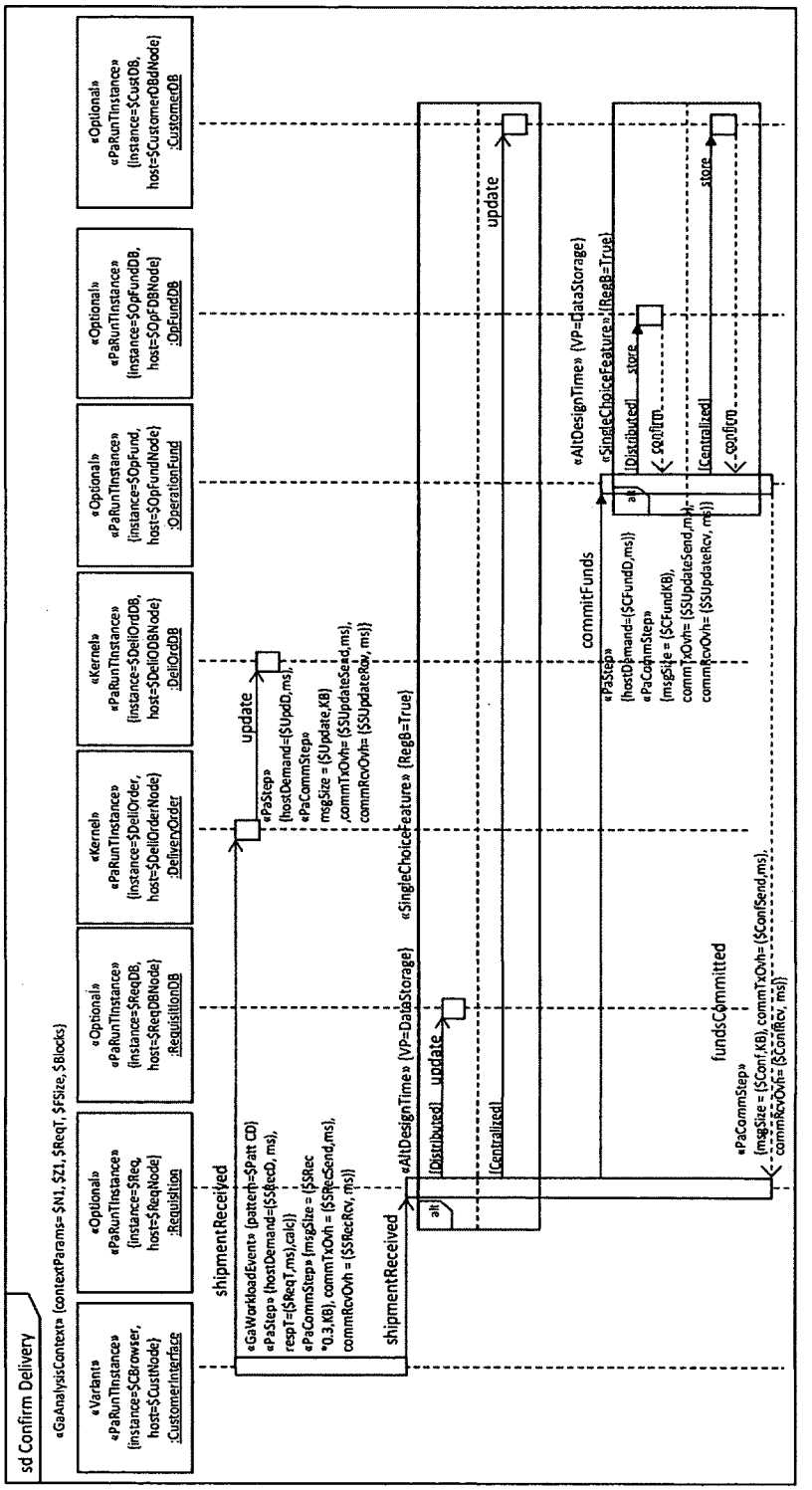


Figure 4.3: SPL Scenario Confirm Delivery.

that refers to the *Interaction CreditCard* which is stereotyped *«Optional»* with the tagged value *{Feature=CreditCard}*. Each one of the other *optional* features (*Check* and *DebitCard*) is modeled in the same manner.

Please note that an operand of an *«AltDesignTime» Combined Fragment* is chosen during product derivation, when the actual SD describing a given scenario for the respective SPL member is actually generated. The derived SD contains only the chosen operand. This operand may represent a regular run-time behaviour in the derived SD, as shown in Figure 4.3, or an optional run-time behaviour. To specify the type of run-time behaviour represented by this operand, we introduce the stereotype *«SingleChoiceFeature»* with two attributes indicating regular and optional run-time behaviours: *RegB* and *OptB*, respectively.

However, during the product derivation process, we may need to indicate that more than one optional *Combined Fragment* corresponding to different features from the same feature group may be chosen. For this reason we introduce the stereotype *«MultiChoiceFeature»* that has three attributes indicating regular, optional, and alternative behaviours: *RegB*, *OptB*, and *AltB*, respectively. Thus, at run-time, the actual SD may contain several optional *Combined Fragments*. Whenever, the tagged value *{AltB=True}*, we propose to model each one of these optional *Combined Fragments* as an operand of an alternative *Combined Fragment*. Then, a user can choose one of these operands at run-time. For example, assume that the scenario *BillCustomer* shown in Figure 4.4 for a given product will contain two choices *CreditCard* and *DebitCard* and the attribute *AltB* is true. Then, these two options are modeled as an alternative *Combined*

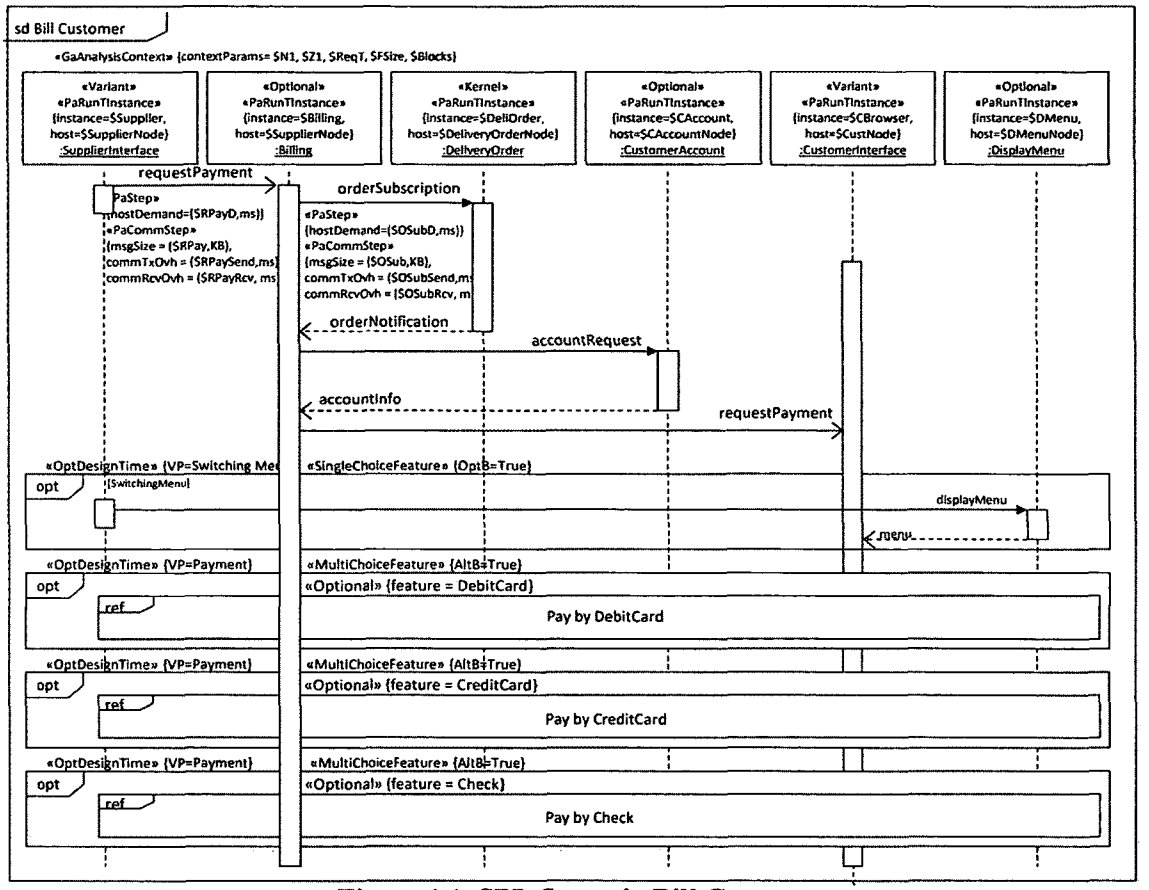


Figure 4.4: SPL Scenario Bill Customer.

Fragment with two operands. At run-time, the user can choose the preferred method of payment between *CreditCard* and *DebitCard* (more details will be given later in section 5.4 and the profile is shown in Appendix A).

4.2 Efficient Mapping of Features to the SPL Design Model

The automatic derivation of a concrete product model based on a given feature configuration is enabled through the mapping between features from the feature model and their realizations in the design model of the SPL. The product model corresponding to the desired feature configuration is instantiated automatically through a model-to-

model transformation, where the transformation process evaluates the SPL model elements' annotations for the selected feature configuration. In this research we propose an efficient mapping technique that aims to minimize the amount of explicit feature annotations in the UML design model of SPL.

Modeling variability in SPL models can be achieved in different ways:

1) Annotating different diagrams of the reusable SPL model with variability specifications, which are mapping features from the feature model to model elements realizing them.
2) Using a separate model to represent variability which can be linked to different model elements of the reusable SPL model.

The idea of mapping features to the model elements realizing them is not a new concept; several papers have been published in this area. For instance, Clauss [CLA01a] proposed a UML extension approach to describe variability in UML diagrams through stereotypes *«variationPoint»* and *«variant»*, where a tagged value indicates the name of the variation point. An optional class stereotyped *«optional»* is also connected to the feature model by a tagged value. However, [CLA01a] does not address the product derivation process, being concerned only with variability modeling. In [ROB02], a stereotype *«variable»* and a tagged value *feature* are introduced to model an element specifying a variable feature. The tagged value *feature* corresponding to a feature name from the feature model is used to set up a traceability link between the feature model and a system model. Variability in the separate model OVM [POH05] is related to software artifacts specified in other design models; at the same time, a traceability link between OVM and a feature model is introduced. Thus, features in a feature model are related to

elements realizing them in different design models through the OVM model. However, the authors address only the derivation of the high level product architecture from the reference architecture, but do not discuss mapping techniques at the design level. Heidenreich et al. [HEI08] propose a tool called FeatureMapper that defines the mapping of features in the problem space to model elements realizing these features in the solution space. A feature configuration is combined with a separate mapping model and interpreted by the FeatureMapper transformation component to derive a product model. Voelter et al. [GRO09] also propose a separate dependency model to specify the linking between features and model elements in their approach. In [STO09], the authors introduced a table-based decision modeling that defines the relationship between variation points and their constraints, as well as their realized artifacts. The table is used by ADORA modeling language to derive a product. The traceability between features and their realizing architecture models are described in [BOT09] by two *realizedBy* references in the metamodel. The feature model property *architecture* refers to the related architecture model, while the feature property *architectureElement* to the related element in this architecture model. The mapping between features and the components realizing their implementation is done through an implementation model.

Another approach proposed by Czarnecki et al. [CZA05b] for expressing variability in a family model by mapping features to model elements realizing them is based on a *feature-based model template,* which consists of superimposed product variants containing all model elements in a SPL. Each model element is annotated with a presence condition (PC), which is a logical expression of features in disjunctive normal form, indicating whether the element should be present in a template instance or not. The

model template is automatically instantiated by evaluating the PCs for a given feature configuration. A drawback of this approach is that the model template is cluttered with variability specifications for each element in the model. The authors propose an implicit presence conditions (IPC) for a given type of elements, where an IPC is specified based on the PC of other elements and the syntax and semantic of the target model. The IPC are given as a table of different choices; the IPC of a specific model element is chosen by searching for the closest matching supertype in the table. In [AVI07] a Model Template Transformation Language (MTTL) is used to specialize model templates based on feature configurations where the mapping between features and their realizing elements in the model template is defined in the MTTL metamodel through the property *feature* of the metaclass *Mapping*. The property *feature* refers to the name of the feature that triggers the operation. In Gomaa's methodology PLUS [GOM05], the mapping between features and model elements realizing them is introduced through a separate tabular representation of feature/use case and feature/class relationships as explained in the previous section.

In our work, we apply the annotation approach by using the SPLV profile described in the previous section. We are aiming to annotate the UML model of SPL with a minimum amount of variability specifications. The reason we chose the UML profiling mechanism for annotations is that a profile extends UML without changing its metamodel or tool support. This allows us to create annotated UML models with any UML editor that conforms to the UML standard [OMG07] by simply importing the profile packages in the tool.

The annotation approach has a number of advantages over the separate variability modeling:

a) Model elements subject to variability are clearly noticeable.

b) The consequence of selecting a feature is directly shown on the design model.

c) The mapping is easier to retrace and understand.

d) The expressive capability is enhanced.

However, a significant drawback of the annotation approach that makes it error-prone is the fact that the SPL models become cluttered with variability specifications, which becomes worse as models grow in size and complexity. This limitation diminishes the advantages of the annotation approach.

The annotation approach proposed in this research mitigates this drawback by reducing the type and number of explicitly annotated model elements as much as possible. The objective is to reduce the clutter of variability specifications in the SPL design model, and therefore to ensure that the models are more concise, less error-prone and easier to visualize. The decision what types of elements to annotate explicitly depends on the application domain and should be taken early in the domain engineering process. The mapping of features to non-annotated model elements is implicit, and can be inferred from their relationships with annotated model elements; such relationships are defined in the UML metamodel and are explored in the transformation rules during product derivation by navigating the model according to the UML metamodel and well-formedness rules. For instance, in a class diagram of the SPL reusable model, annotate explicitly the variability of classes with the names of the features requiring each class, but leave the associations without variability annotations. The unspecified mapping of features to each association can be inferred from the annotations of the two classes

connected to the association ends. Thus, we can say that the mapping of features to classes is *explicit* and that of features to associations is *implicit*.

Whenever a model element is not explicitly annotated with corresponding feature(s) through a stereotype or its attributes in the SPL model, the automatic transformation process needs to decide whether to copy this element to the target model or not. This decision is based on several factors:

a) The type of this non-annotated element.

b) The specifications and well-formedness constraints of the modeling language.

c) The presence or absence of other annotated elements related to it.

d) The containment hierarchies defined in the metamodel.

e) The cardinality of this element.

For example, according to the UML metamodel, a binary association has to be attached to a classifier at each end. Therefore, the decision whether a binary association has to be copied or not to the target is based on the selection of both of its classifiers. If at least one of its classifier is not selected, the association will not be created in the target model. In other words, the binary association is created in the target model if and only if both of its *memberEnd* properties have their classifiers already selected and created. At the same time, if only one of its classifier is selected and created in the target model, the property attached to this unselected association and owned by the selected classifier should not be created in the target model. The interpretations of the implicit mapping will be explained in details in chapter 5, where the transformation rules are described.

The proposed mapping technique ensures that the derived product model is a well-formed model by enforcing the well-formedness constraints during the

transformation process. Each time a new model element is selected and added to the target model, the verification of its well-formedness rules is guaranteed by construction, according to the transformation rules that are based on the UML metamodel. Chapter 9 gives more details of the verification of the derived product model.

4.3 Domain Engineering Phase

The software product line development process is separated in two major phases:

1) *Domain engineering* for creating and maintaining a set of reusable artifacts and introducing variability in these software artifacts, so that the next phase can make a specific decision according to the requirements of the product to be created.
2) *Application engineering* for building products that are family members from reusable artifacts created in the first phase instead of starting from scratch.

The domain engineering process is a development cycle *for* reuse and includes, but is not limited to, creating the requirement specifications, domain models , architecture, reusable software components, documentation and specifications, schedules, budgets, test plans, test cases, work plans, and process descriptions [CLE01]. An important outcome of this process is the SPL assets created by the domain engineering process which are of interest for our research is a multi-view UML design model of the family, called the *SPL model*, which represents a superimposition of all variant products. The creation of the SPL model employs two separate profiles: a *Software Product Line Variability* profile (SPLV) for specifying the commonality and variability between products (presented in section 4.1), and the MARTE profile standardized by OMG for performance annotations. Another important outcome of the domain engineering process is the feature model used to

represent commonalities and variabilities between family members in a concise taxonomic form. Additionally, we create a PC-feature model to represent the variability space of performance completions (will be explained in chapter 6).

The SPL model should contain, among other assets, structural and behavioural views which are essential for the derivation of performance models. It consists of:

1) Structural description of the software showing the high-level classes or components, especially if they are distributed and/or concurrent.
2) A set of key performance scenarios defining the main system functions frequently executed.

Please note that this research does not cover the entire domain engineering phase neither all of its sub-processes. However, in order to explain our derivation approach, the construction of the UML model for the SPL that represents the source model of our model transformation approach is illustrated through an e-commerce case study in the next section. As already mentioned, two profiles are applied to the source model: MARTE for performance annotations and SPLV for modeling variability and mapping it to SPL artifacts. The creation of the e-commerce case study used to illustrate the proposed approach is done during the domain engineering phase.

4.3.1 E-commerce Case Study

Electronic commerce (e-commerce) is a highly distributed application which uses the Internet and the World Wide Web (WWW) for purchasing and trading products and services. The Internet is an efficient mechanism for promoting and allocating product information and customer services where suppliers can be linked to large manufacturers

or service organizations. Furthermore, e-commerce makes use of computing and communication technologies in several industries such as financial industries, insurance companies, online airline reservation systems, order processing, etc [TRE03].

Fong et al. [FON00] classify the commercial web sites into three main categories: 1) Business-to-Consumer commerce that directly advertises their physical products and/or services to end consumers. 2) Business-to-Business commerce having an online catalogs advertising goods to other businesses. 3) Information commerce which distribute digital goods such as information products and services online. The three categories have some common features to present products, process orders, accomplish transactions, and deliver customer service.

Another way to categorize an e-commerce system, based on the definition that an e-commerce carries out transactions of any type through an electronic media such as the Internet, takes into account the roles played by suppliers and customers in these transactions such as consumers (C), administrators (A), businesses (B) or employees (E). An e-commerce can be classified into B2E, B2A, C2C, B2B, and B2C [GRU 02]. Business-to-employee (B2E) allows companies to provide products and/or services to their employees through a business network. Business-to-administration (B2A) involves a large number of online transactions between companies and public administration such as social security, employment, registries, etc. Consumer-to-consumer (C2C) includes transactions between consumers through some third party such as the online auction.

Our case study used throughout this research to demonstrate that our proposed approach can handle variability categorizes the e-commerce system according to the roles

played by customers. It can support both business-to-business (B2B) and business-to-customer (B2C) products in the following manner.

In B2B, a business customer browses various WWW catalogs, views various items from a given supplier's catalog and selects items to purchase. Each customer has a contract with a supplier for purchases as well as bank accounts through which payments can be done. An operation fund is associated with each contract for fund availability. A delivery order is created and sent to the supplier if the contract and fund are ready. The supplier confirms the order and goes into a planned shipping date if the order is a physical good. If the system supports several shipping methods, the customer has to choose one of them. A printed invoice is issued and the customer is notified when the order is shipped. When the shipment is received, the customer acknowledges, the delivery order is updated, and payment of the invoices is authorized. The invoice is sent to account payable which authorizes the payment of fund.

In B2C system, a customer can browse through several catalogs provided by the suppliers to select items to purchase. The customer requests to purchase items from the supplier and provides personal details, such as address and credit card information which are stored in a customer account. If the credit card is valid, a delivery order is created and sent to the supplier. When the order is set up, the delivery is scheduled and the customer is notified and requested to pay. If the system supports several type of payment, the customer has to choose one of them. If the customer purchases digital goods, the delivery will be online after payment approval and the invoice will be displayed on-line too.

A supplier may create a purchase order requesting new inventory supplies from a wholesaler. When the purchase order is delivered to the supplier, payment is made to the wholesaler [GOM05]. Our SPL case study supports this feature as an optional one.

A catalog consists of a set of web pages describing items for sale. The web pages might be created statically or dynamically from database items. A supplier may be a small shop with a static catalog, and simple requirements for record keeping and order access, or it can be a large dealer which requires a dynamic web site to handle changing in products and prices [FON00]. Therefore, in our case study there are two types of supplier catalog: 1) Static catalog which means static content that contains ready pages which have to be prepared whenever the information changes. 2) Dynamic catalog or dynamic content which is created whenever requested and exploits some information sources to create the required page. It needs to be up to date with accurate product information, availability, and prices. A static catalog may be created through a desktop publishing application, while an online catalog (Dynamic catalog) needs a database with product display to create catalog pages on the fly or a real-time inventory. Our case study offers both types of catalog.

Furthermore, our case study provides two optional features to handle customer inquiries: a help desk and a call center. For international sales, it offers four optional features: currency conversion, presenting several languages, handling import/export laws and tariffs calculation [FON00]. Different merchandising techniques can be provided to attract customers such as promotions, occasionally sales, promotion events, membership discounts, or frequent buyer programs. The case study supports three marketing techniques only: sales, promotions, and membership discounts. Digital goods and

services can be electronically delivered with on-line display invoices, while physical products require shipping delivery with printed invoices. The case study offers both kind of invoices (printed invoice and on-line display) as well as both delivery types (electronic and shipping) in addition to different shipping forms such as normal, express, and package-Slip. An essential requirement of an e-commerce is to support the exchange of payments for goods and services. Payment has different methods such as credit card, charge card, checks, money orders, electronic funds transfer, payment card, smart cards, electronic cash, micropayment, or peer to peer payment systems. The case study supports three payment methods for B2C applications: credit card, credit card, and check. In B2B applications, Business often buys from each other using purchase orders. It is a charge card that doesn't extend credit and its balance must be paid in full each month. The case study supports only one payment method for B2B through charging his bank account.

4.3.2 Feature Model

The feature models are used in our approach to represent two different variability spaces. This subsection describes the regular feature model representing functional variabilities between products. An example of feature model of an e-commerce SPL is represented in Figure 4.1. The feature model is taken as input by our ATL transformation as an extended UML class diagram, where the features and feature groups are modeled as stereotyped classes and the dependencies and constraints between features as stereotyped associations. For instance, the two alternative features *BusinessCustomer* and *HomeCustomer* are mutually exclusive, so they are grouped into an exactly-one-of feature group called *Customer*. Another example of mutually exclusive features is the

two alternative features *StaticCatalog* and *DynamicCatalog* that are grouped into an exactly-one-of feature group called *Catalog*, while the three optional features *Normal*, *Express*, and *PackageSlip* are grouped into an at-least-one-of feature group called *ShippingType*. Thus, an individual product can provide at least one of these features or any number of them. In the case of a product providing all of these features, the user can choose one of them during the run-time execution. The feature group *Payment* with the three optional features *CreditCard*, *DebitCard*, and *Check* are handled in the same manner.

In addition to functional features, we add to the diagram another type of features characterizing design decisions that have an impact on the non-functional requirements or properties. For example, the architectural decision related to the location of the data storage (centralized or distributed) affects performance, reliability and security, and is represented in the diagram by two mutually exclusive quality features. This type of feature related to a design decision is part of the design model, not just an additional PC-feature required only for performance analysis.

One of the limitations of this work is that we consider only string features that can be bound at design time, not at run-time.

This feature model represents the set of all possible combinations of features for the products of the family. It describes the way features can be combined within this SPL. A specific product is configured by selecting a valid feature combination from the feature model, producing a so-called feature configuration based on the product's requirements.

To enable the automatic derivation of a given product model from the SPL model, the mapping between the features contained in the feature model and their realizations in

a reusable SPL model needs to be specified, as shown in section 4.2. Also, each stereotyped class in the feature model has a tagged value indicating whether it is selected in a given feature configuration or not.

4.3.3 SPL Model

The SPL model represents a multi-view UML design model of a family of products which represents a superimposition of all variant products.

The first step for creating the SPL model is to build a use case model for the e-commerce SPL as shown in Figure 4.5, representing the functional requirements of the SPL. The kernel use cases required by all the family members are shown in white, the optional use cases that may be used by any member are drawn in light grey, and the alternative use cases used only by some members are shown in dark grey.

In order to avoid polluting our model with extra annotations and to ensure the well-formedness of the derived product model as in [CZA06], we propose to annotate explicitly the minimum amount of model elements within each diagram of our SPL model. For instance, in the use case diagram, only the *optional* and *alternative* use cases are annotated with the name of the features requiring them (given as stereotype attributes); since a *kernel* use case represents commonality, it is sufficient to just stereotype it as «*Kernel*». Other model elements, such as actors, associations, generalizations, properties, are mapped implicitly to the features through their relationship with the use cases, so there is no need to clutter the model with their annotations. The evaluation of implicit mapping during product derivation is explained in chapter 5. Please note that the annotations proposed here indicate only the feature

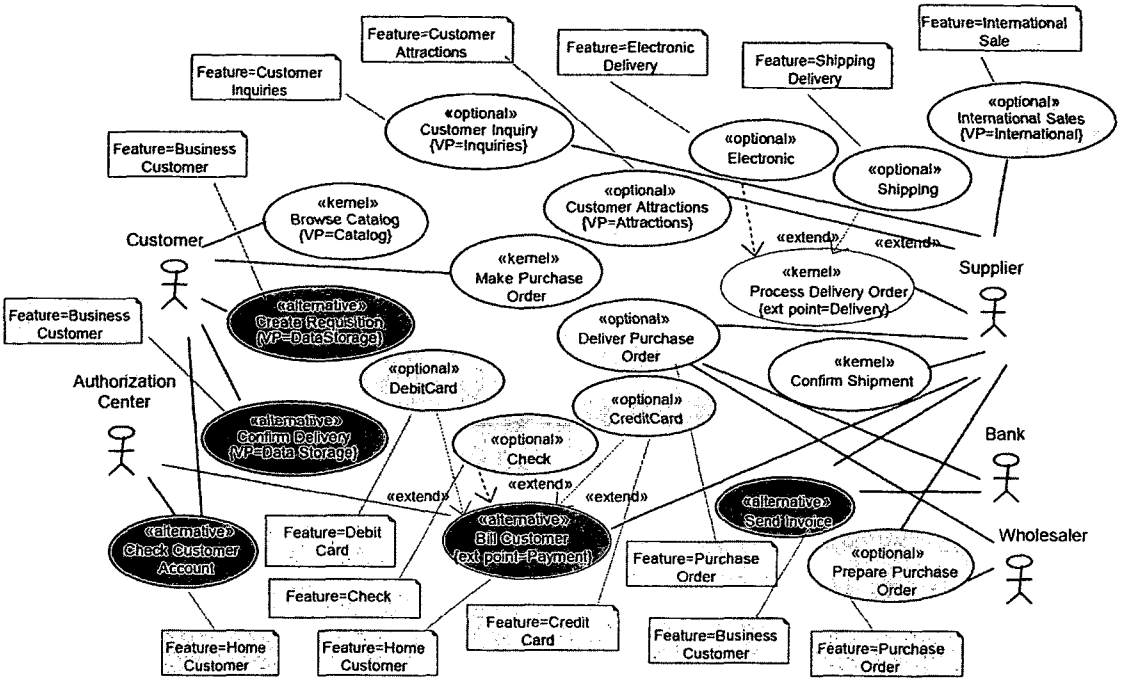


Figure 4.5: Use case model of the e-commerce SPL.

name rather than logical expressions (as in [CZA05b]).

A small feature can be represented as a variation point in a use case scenario. For example, the feature *DataStorage* is presented as a variation point in the use case *Create Requisition*. If the variation is rather complex, it may be represented with *extend* and *include* relationships between use cases. For example, the three optional features *CreditCard*, *DebitCard*, and *Check* are depicted through an *extend* relationship between use cases. In this case, both the base use case *Bill Customer* and the extending use cases *CreditCard*, *DebitCard*, and *Check* have to be selected in order to realize these three optional features.

In the second step, the class diagram for the e-commerce SPL is created as shown in Figure 4.2.a to 4.2.e. The object *CustomerInterface* behaves differently in B2B

systems than in B2C systems. Therefore, a generalization/specialization hierarchy is used to model the different behaviours of this class. The two subclasses *B2BInterface* and *B2CInterface* are used by B2B systems and B2C systems, respectively. The same happens with the superclass *SupplierInterface* shown in Figure 4.2.b, which is specialized into two variants *POSupplier* and *Supplier*. In the SPL class diagram, each *variant* and *optional* class is annotated with the feature name requiring it (given through tagged values).

Sequence diagrams are used to model the objects that participate in each use case scenario and the sequence of messages between them. Thus, in the third step, a sequence diagram is created for each scenario in each use case of interest. For instance, the *kernel* sequence diagram *BrowseCatalog* shown in Figure 4.6, realizes a scenario in the *kernel*

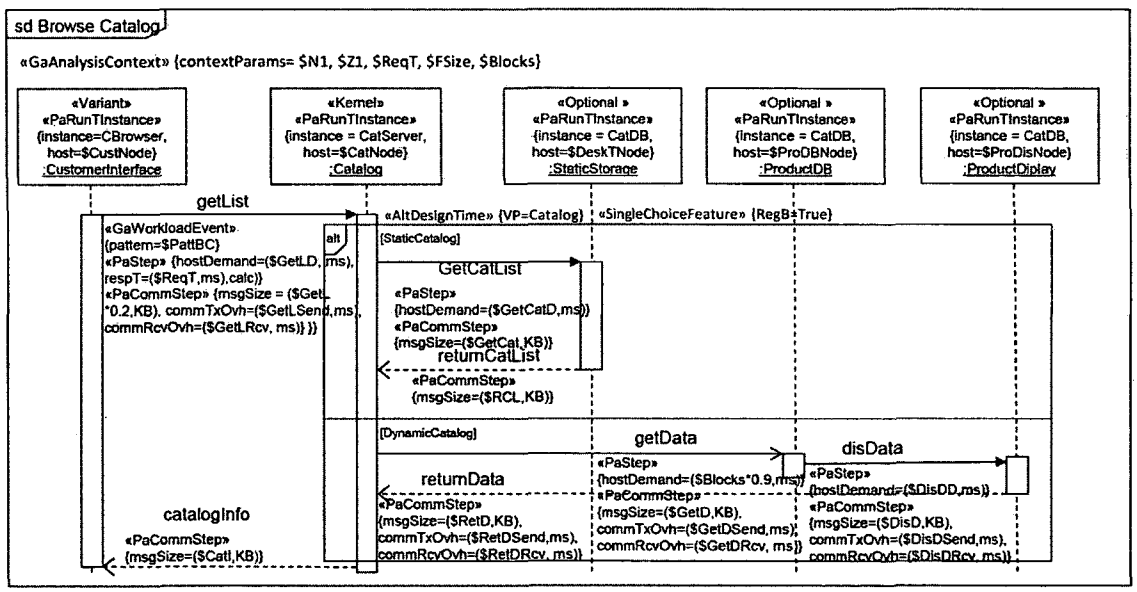


Figure 4.6: SPL Scenario Browse Catalog.

use case with the same name. It contains the variation point *Catalog* with two variants *StaticCatalog* and *DynamicCatalog*. The alternative *Combined Fragment* is used to represent the two alternative features, while the tagged value of the stereotype *«SingleChoiceFeature»* indicates a regular behavior in the derived product sequence diagram.

Figure 4.4 shows the sequence diagram *BillCustomer* which realizes a scenario in an *alternative* use case containing an extension point extended by three *optional* use cases. The extension point *Payment* is depicted as a stereotype on each of the three optional *Combined Fragments*; each such fragment contains an *Interaction Use* referring to the *Interaction* that models the respective extending optional use cases. The three optional *Combined Fragments* are annotated with *«SingleChoiceFeature»* with a tagged value indicating a regular behaviour at run-time in the product sequence diagram, if only one option is chosen during product derivation. However, if more than one option may be chosen during product derivation, the tagged value of the stereotype *«MultiChoiceFeature»* indicates an alternative behaviour.

The scenario *CreateRequisition* which corresponds to an alternative use case containing a variation point is illustrated in Figure 4.7. The alternative *Combined Fragment* whose choices are based on the value of the quality feature *DataStorage* (represented as a variation point in the scenario) is stereotyped with the SPLV stereotype *«AltDesignTime» {VP=DataStorage}*. Each operand of the alternative *Combined Fragment* corresponds to one of the alternative features (*Distributed* or *Centralized*).

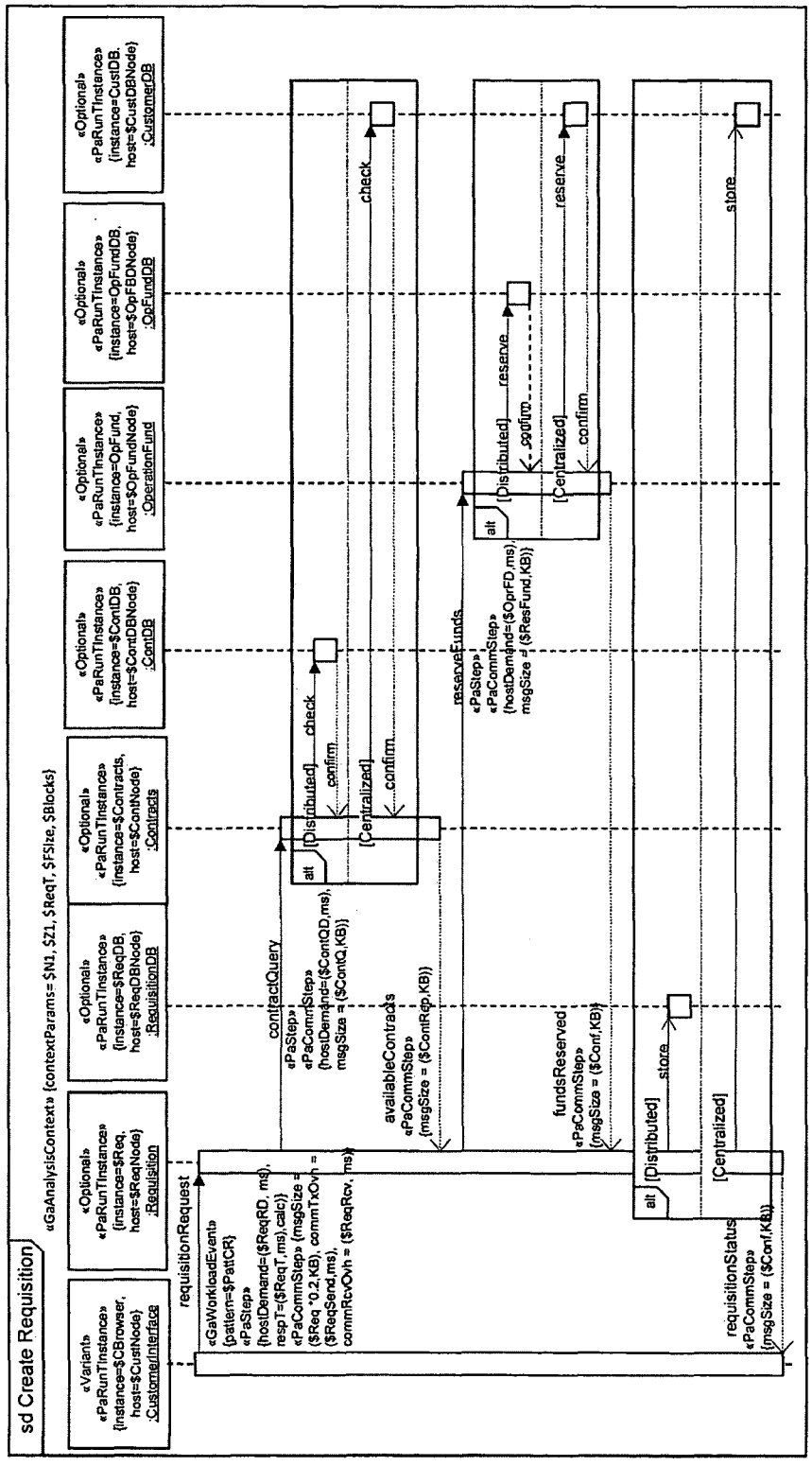


Figure 4.7: SPL Scenario Create Requisition.

The sequence diagram *ProcessDeliveryOrder* which realizes a scenario in a *Kernel* use case contains an extension point extended by two *optional* use cases. The extension point *Delivery* is depicted as a stereotype on each of the two optional *Combined Fragments* as shown in Figure 4.8. Within each one of the optional *Combined Fragment*, an *Interaction Use* refers to the *Interaction* that models the corresponding extending optional use case. The two optional *Combined Fragments* are annotated with *«SingleChoiceFeature»* with a tagged value indicating a regular behaviour in the product sequence diagram, if only one option chosen. However, if the two options are chosen during product derivation, the tagged value of the stereotype *«MultiChoiceFeature»* indicates two optional behaviours in the product scenario.

In the same way are created the other SPL sequence diagrams. Each sequence diagram has to be annotated with parametric MARTE performance specifications that can be reused for different product derivations. During the model transformation process, these generic annotations will be bound to concrete values.

4.3.4 MARTE Annotations

In order to understand what kind of performance annotations need to be added to UML models to enable performance analysis, one needs to look at the basic concepts contained in the performance domain model. As already mentioned, performance is determined by how the system behaviour uses system resources. Scenarios define execution paths with externally visible end points. Performance requirements (such as response time, throughput, probability of meeting deadlines, etc.) can be placed on scenarios. In

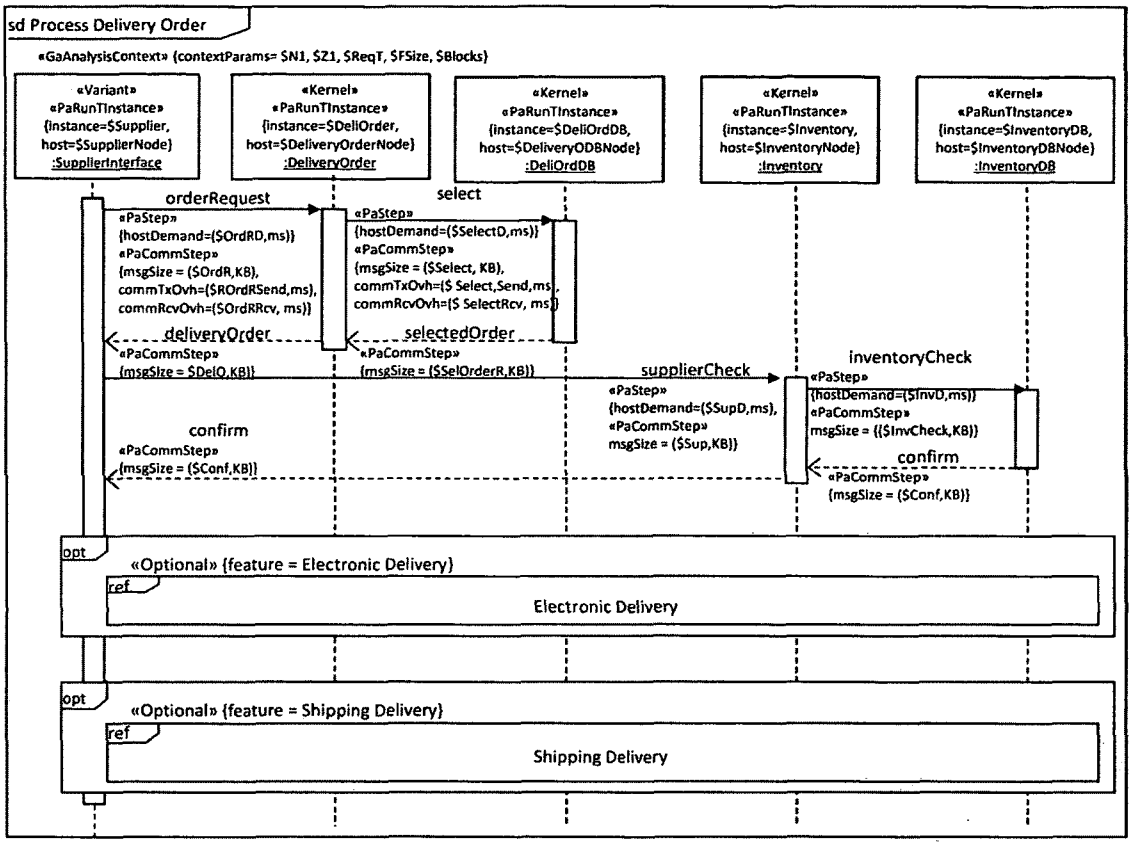


Figure 4.8: SPL Scenario Process Delivery Order.

MARTE the performance domain model describes three main types of concepts: *resources*, *scenarios*, and *workloads*.

The resources used by the software can be active or passive, logical or physical software or hardware. Some of these resources belong to the software itself (e.g., critical section, software server, lock, buffer), others to the underlying platforms (e.g., process, thread, processor, disk, communication network). Each scenario is composed from scenario steps joined by predecessor-successor relationships, which may include fork/join, branch/merge and loops. A step may represent an elementary operation or a whole sub-scenario. Quantitative resource demands for each step must be given in the

performance annotations. Each scenario is executed by a workload, which may be open (i.e., requests arriving in some predetermined pattern) or closed (a fixed number of users or jobs in the system). The performance annotations used in our case study are explained in this subsection.

MARTE annotations have the capability to express parameters in the form of variables which can be assigned or can be bound to "concrete values". By concrete values, we don't mean only literal values, but also expressions in function of other annotation variables. By convention, annotation variables start with '$' to avoid confusing them with other identifiers from the UML model. The capability of using variables and expressions in MARTE annotations raises their level of abstraction and makes the annotations more reusable.

An interaction diagram representing a scenario to be considered for performance analysis is stereotyped itself as *«GaAnalysisContext»*, as shown in Figure 4.9. This stereotype may have a set of annotation parameters defining global properties of this analysis context, held by its tag *{contextParams= $N1, $Z1, $ReqT, $FSize, $Blocks}*.

Properties of the workload, behaviour, and resources may be defined as functions of such global variables.

The workload of a scenario is defined as a stream of events driving the system; a workload may be open or closed. A closed workload with a number of users *$N1* and user think time for a user *$Z1* is specified as follows:

«GaWorkloadEvent» {pattern= (closed (population=$N1), (extDelay=$Z1))}

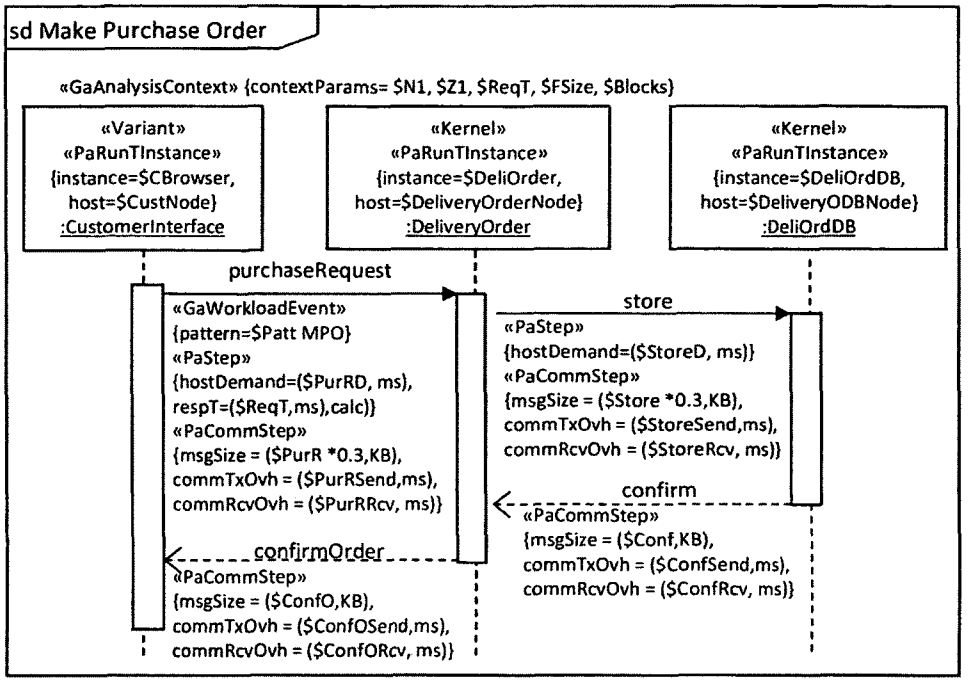


Figure 4.9: SPL Scenario Make Purchase Order.

An example of open workload with an inter-arrival time *$ATime* (ms) is expressed as:

«GaWorkloadEvent» {pattern= (open (IntArrTime=$ATime,ms))}

Each lifeline whose role is an active object is stereotyped as *«PaRunTInstance»*, providing an explicit connection at the annotation level between a role in a behavior definition (a lifeline) and a run-time instantiation of a process or thread (active object) playing that role. For example, the tag *{instance= $CBrowser}* indicates the name of run-time instance of a process executing the lifeline role, while the tag *{host= $CustNode}* indicates the physical node on which the instance is running. In this case, the attributes *instance* and *host* are given by the generic parameters *$CBrowser* and *$CustNode*, which will be bound later to a concrete run-time instance and host name, respectively.

Conceptually, a scenario represented by a UML sequence diagram is composed of units of execution named steps. MARTE defines two kinds of steps for performance analysis: execution step (stereotyped *«PaStep»*) and communication step (stereotyped *«PaCommStep»*). *«PaStep»* may be applied to an *ExecutionOccurence* (represented as thin rectangle on the lifeline) or to the message that triggers it. For instance, in Figure 4.9, the message *purchaseRequest* is stereotyped as an execution step:

«PaStep» {hostDemand=($PurRD, ms), respT=(($ReqT, ms), calc)}

where the tag *hostDemand* indicates the execution time required by the step, which is given by the variable *$PurRD* in time units of milliseconds. The attribute *respT* corresponds to the response time of the scenario starting with this step; the variable *$ReqT* will save the calculated response time in milliseconds. The same message *purchaseRequest* is also stereotyped as a communication step:

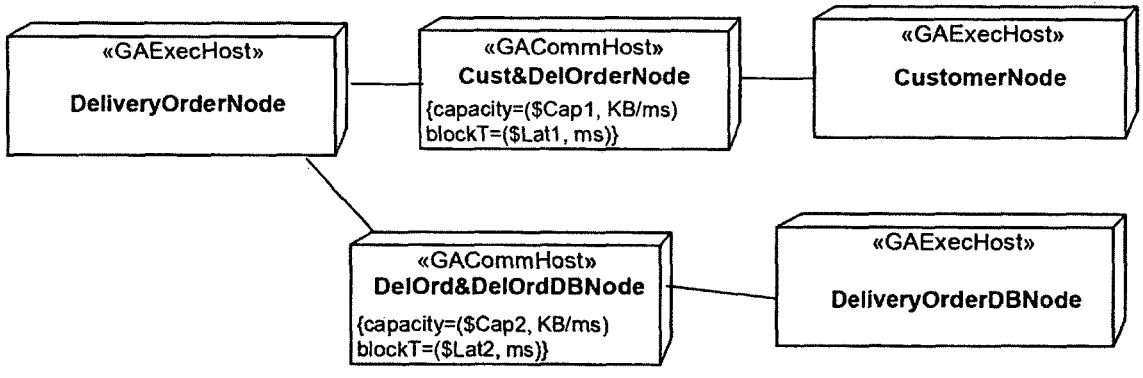


Figure 4.10: Part of a product deployment diagram.

«PaCommStep» {msgSize = ($PurR,KB), commTxOvh = ($PurRSend, ms),

commRcvOvh = ($PurRRcv, ms)}}

where the message size is given by the variable *$PurR* in KiloBytes. The overheads for sending and receiving this particular message are the variables *$PurRSend* and *$PurRRcv*, respectively, in milliseconds. We propose to annotate each communication step (which corresponds to a logical communication channel) with the CPU overheads for transferring the respective message: *commTxOvh* for transmitting (sending) the message and *commRcvOvh* for receiving it. Eventually, these overheads will be added in the performance model to the execution demands of the two execution hosts involved in the communication (one for sending and the other for receiving the respective message).

Every processing node in the deployment diagram is stereotyped as an execution host as shown in Figure 4.10. In performance modeling, *«GaExecHost»* can be any device which executes behavior, including storage and peripheral devices. Each type of physical communication channel is stereotyped *«GaCommHost»* with two different attributes *capacity* and *blockT* that represented the capacity and latency for the physical communication channel, respectively.

4.4 Application Engineering Phase: Concrete Product Model

The application engineering process which is the focus of this research uses the results (domain assets) from the domain engineering process to generate product family members (concrete products). Our first model transformation approach takes as input the SPL source model created during the domain engineering process and generates a product PIM with generic performance annotations, as explained in subsection 4.4.2. The other model transformations for generating parameter spreadsheets and deriving a product PSM are explained in chapter 7. The next subsection presents the target models.

4.4.1 Target Models

The target model of the first model transformation is a UML product PIM with generic MARTE annotations, where the variability expressed in the SPL model has been analyzed and resolved to a specific product. Therefore, a product model does not contain any more SPL-related stereotypes, tagged values and constraints. However, the product model contains generic performance annotations.

The target model for a specific product PIM consists of the following:

1. A use case view for the specific product
2. A product class diagram
3. A product sequence diagram for each scenario in each selected use case

The second model transformation for generating parameter spreadsheets takes as input the generated product PIM along with the PC-feature model and the deployment diagram provided by the user. The third transformation takes as source models both the generated product PIM and the product deployment diagram as well as the spreadsheets after the user provides concrete values. The target of this model transformation is a UML product PSM, where the performance annotations have been bound to concrete values corresponding to the respective product model element, as indicated by the user and the software components have been also allocated to actual physical devices according to the user choice.

The target model for a product PSM consists of the following:

1. A use case view for the specific product
2. A product class diagram

3. A product sequence diagram for each scenario in each selected use case with concrete performance annotations

4. A deployment diagram of the product with concrete performance annotations

4.4.2 Derivation Process of Product PIM

Our model transformation approach takes as input the SPL source model created during the domain engineering process in section 4.3.3 and generates a product target model for a given member of the SPL.

The derivation of a concrete UML product platform specific model (PSM) with concrete performance annotations from the SPL model with generic annotations requires three model transformations as shown in Figure 1.1:

1) Transforming the SPL model to a product platform independent model (PIM) with generic performance annotations.
2) Generating spreadsheets containing the generic parameters and guiding information for the given product PIM. The user will provide concrete values for binding on these spreadsheets.
3) Performing the actual binding by using the concrete values provided by the user to generate a product platform specific model (PSM).

The transformations handle two kind of generic parametric annotations:

a) Product-specific parameters due to the variability expressed in the SPL model.
b) Platform-specific parameters due to device choices, network connections, middleware, and run time environment.

This subsection describes the first model transformation for instantiating a product PIM with generic performance annotations from the SPL model and handling the product-specific parameters. The other model transformations are explained in chapter 7.

In general, there are two different techniques of deriving a product model that corresponds to a given feature configuration from a SPL model:

1) *Positive variability* [VOE07] or *additive* notion [HEI07] where one starts by copying the minimal core representing common SPL artifacts to the target model, and then selectively adds additional product-specific model elements according to the selected features.
2) *Negative variability* or *subtractive* technique, where SPL artifacts related to unselected features are removed from the SPL model.

Several works apply the negative variability concept to their derivation approaches such as [CZA05b], [HEI08], [STO09], and [VOE07] where a superimposition of all variant products of a family is the source model, from which model elements that do not correspond to any of the selected features for the intended product are removed during the model transformation process that generates the product. The positive variability technique is used in general in conjunction with AOM, where the common part of a SPL is represented as the base model while, the variable parts are aspect models that are woven into the base model to generate the product model, as in [VOE07]. In [JAY07], each variant feature is modeled separately and combined with the design model realizing kernel features to generate the product model by using a UML graph transformation approach.

Our model transformation approach applies the concept of positive variability where we start by selecting and copying the model elements that represent *kernel* features to the target model, then selectively add other elements realizing the desired *optional* and *alternative* features.

The proposed derivation approach is based on mapping features from the feature model to model elements realizing them in the SPL model (both structural and behavioral views). The feature model in our approach is represented as a feature dependency class diagram, as shown in Figure 4.1. A feature configuration is a legal combination of features which specifies a particular product. The derivation process is initiated by specifying a given product through its feature configuration. The selected features are checked for consistency against the feature dependencies and constraints in the feature model created in section 4.3 to identify any inconsistencies between features. An example is checking to ensure that no two mutually exclusive features are chosen. The feature configuration is considered a parameter for the transformation, which should be set without editing the program [BEZ05]. The steps of the proposed model transformation algorithm are presented in Figure 4.11.

The second step in the derivation process is to select the use cases realizing the chosen features. All the *kernel* use cases are copied to the product use case diagram, since they represent functionality provided by every member of the SPL. If a chosen feature is realized through *extend* or *include* relationships between use cases, both the base use case and the *included* or *extending* use cases have to be selected, as well. A use case containing in its scenario variation point(s) required to realize the selected feature(s) has to be chosen, too. The *optional* and *alternative* use cases are selected and copied to the

target use case diagram if they are mapped to a feature from the feature configuration. The interpretations of other non-annotated elements will be explained in the description of the transformation rules. Finally, the use case diagram for the product is developed after all the SPLV stereotypes were eliminated.

The third step is to derive the product class diagram by selecting first all the *kernel* classes from the SPL class diagram. In the SPL class diagram, each *optional* and *variant* class is annotated with the feature(s) requiring it. *Optional* and *variant* classes needed for the desired product are selected next. Moreover, superclasses of the selected *optional* or *variant* classes have to be selected as well. Then, the SPLV stereotypes are removed. An association between two classes is copied to the target model if and only if both classes are selected. If only one is selected, the association is not copied.

The fourth step in our transformation approach is to generate the sequence diagrams corresponding to different scenarios of the chosen use cases. Each scenario of a chosen use case is realized by a sequence diagram which has to be selected from the source model and copied to the target one.

The SPLV stereotypes are eliminated after binding the generic roles associated to the life-lines of each selected sequence diagram to specific roles corresponding to the chosen features. For instance, the sequence diagram *BrowseCatalog* has the generic *alternate* role *CustomerInterface* which has to be bound to the concrete role *B2CInterface* to realize the feature *HomeCustomer* or to *B2BInterface* to realize the feature *BusinessCustomer*. However, the selection of the *optional* roles is based on the corresponding features. For instance, the sequence diagram *BrowseCatalog* has the

generic *optional* role *StaticStorage* which is selected if the feature *StaticCatalog* is chosen.

Variability within a sequence diagram represented through the extended optional and alternative *Combined Fragments* is also bound to a specific behaviour in the derived product scenario. For instance, the sequence diagram *BrowseCatalog* contains the alternative *Combined Fragment* that represents the two alternative features *StaticCatalog* and *DynamicCatalog*. One of these operands is selected and represented as a regular behaviour in the derived product scenario.

The model transformation that derives a specific product model is also responsible for the binding of the generic performance annotations to concrete values. Some of these generic annotations are attributes of the design model, others of the underlying platforms. Before binding, we need to specify a PC-feature configuration for some model elements (such as messages). This is done through the generation of binding directive spreadsheets, asking the user to define specific PC-feature configurations and to enter concrete values for all generic performance annotations and platform allocations for the given product. The process for generating spreadsheets and performing the actual binding of performance specifications will be explained in detail in chapter 7, which describes the mapping between the PC-features and the affected model elements.

The next chapter explains how the transformation rules navigate the source model from annotated to non-annotated elements, taking into account the UML metamodel and its constraints.

```
BEGIN
1. Select the desired features for the product and check
   them for consistency

2. Select use cases realizing the chosen features
   from the SPL use case diagram
   SWITCH
      CASE 1: Feature realized as a use case
            BEGINCASE
               Select the respective use case;
            ENDCASE
      CASE 2: Feature realized as "extend" or "include"
            relationships between use cases
            BEGINCASE
               Select the base use case and the
               included or extended use cases;
            ENDCASE
      CASE 3: Feature realized as a variation point
            within a scenario realizing the use case;
            BEGINCASE
               Select the respective use case;
            ENDCASE
   ENDSWITCH

3. Derive the product class diagram from the SPL
   class diagram.
         Select "Kernel" classes;
         Select "Optional" or "Variant" classes
         corresponding to the chosen features;

4. FOR each scenario of the selected use cases
      Choose the corresponding sequence diagram;
      FOR each chosen sequence diagram
         Bind each generic role associated to a life-
         line to the desired role for handling the
         chosen feature;
      ENDFOR
   ENDFOR
END
```

Figure 4.11: Steps of model transformation algorithm.

Chapter 5: ATL Transformation for Generating a Product PIM

This chapter presents the model transformation approach introduced in section 4.4.2 for deriving automatically a product PIM out of a SPL model. The transformation is implemented in the Atlas Transformation Language (ATL) [ATL], which is specialized for model transformations. The source model is the SPL model described in section 4.3 with two profiles applied, MARTE and SPLV. The target model generated by the transformation is the model of a particular product, as presented in section 4.4. The transformation rules presented here handle the implicit and explicit mapping of features to SPL design model described in section 4.2.

The transformation is illustrated with the e-commerce case study, more specifically with the derivation of a product model consisting of use case, class, and sequence diagrams. A few examples of ATL transformation rules are given, which contain extensive comments in natural language. Please see section 3.3.2 for a brief description of the ATL syntax.

A preliminary step for the model transformation generating the PIM is to select the desired features for the respective product and to construct the feature configuration. The next section describes how the feature configuration is given to the ATL transformation without hard-coding it in the program.

5.1 Passing Feature Configuration to the Transformation

The feature configuration is considered an input parameter for the model transformation and can be given as a mark model. The mark model concept has been introduced in the OMG MDA guide [OMG04] as a means of providing concrete parameter values to a transformation. This capability of allowing transformation parameterization through mark model instances makes the transformation generic and more reusable in different contexts.

The mark model is considered as another input model specified in the ATL *header* ,which is used to declare general information such as the module name, the source and target metamodels and imported libraries. The *header* below shows the mark model *FC*.

```
module SPLModel2Product;
create OUT : UML from SPL : UML, FC : UML;
```

The ATL transformation reads the values from the marc model through a *Helper*, as explained later in this section.

As described in section 4.3.2, the feature model is represented as a UML class diagram whose nodes are stereotyped either as "features" or "feature groups". Each class representing a feature has a tagged value (i.e., property) "selected" indicating whether the respective feature is selected for a given feature configuration or not. After the user selects the desired features for the product, the feature model is given to the transformation as a separate input model.

The following helper first collects the names of all the selected features from the input model *featureModel* and then returns *True* if model element for which the helper is invoked is annotated with a feature included in the set of selected features.

```
-- This helper returns "true" if the respective model element is annotated with a
selected feature
helper context Pro!Element def:selectedElement(): Boolean = let
values : Sequence(String)=Pro!Class.
         allInstancesFrom('featureModel')->select
         (class|class.getTagValue('Feature','selected')
         = 'True') -> collect(c|c.name) in
      if   (values->includes(self.getTagValue
                              ('Alternative','feature')))
         or (values -> includes(self.getTagValue
                              ('Optional','feature')))
         or (values -> includes(self.getTagValue
                              ('Variant','feature')))
        or (self.hasStereotype('Kernel'))
      then
            true
      else
            false
      endif;

-- This helper returns "true" if the respective model element is stereotyped with the
-- stereotype name given as a parameter
helper context Pro!Element def:  hasStereotype
(stereotype:String):Boolean = self.getAppliedStereotypes()
-> exists(c|c.name.startsWith(stereotype));

-- This helper returns the tagged value of a stereotype's property
-- (both the stereotype and property name are given as parameters)
helper context UML!Element def : getTagValue
(stereotype:String, tag:String): UML!Element=
      if
            self.getAppliedStereotypes()->select
            (e|e.name=stereotype)->notEmpty()
      then
            self.getValue(self.getAppliedStereotypes()
            ->select(e|e.name=stereotype)-> first(), tag)
      else
            OclUndefined
      endif;
```

5.2 Deriving the Target Root

The first ATL rule creates a model element that represents the root of the target model, which corresponds to the root of the source model. Since both the source and the target are in our case UML models, the root of each of them is an instance of the metaclass Model, which is a container that packages all the other model elements. The list of contained objects is then referenced by the *packagedElement* attribute. At the beginning, this list is empty. The model will carry the same name as the source model proceeding with the word *ProductModel.*

```
rule Model {
 from
      s : Pro!Model
   using {
  usecases : Sequence(Pro!UseCase)=
  s.packagedElement->select(e|e.oclIsTypeOf(Pro!UseCase));
  actors : Sequence(Pro!Actor)=
  s.packagedElement->select(e | e.oclIsTypeOf(Pro!Actor));
  classes : Sequence(Pro!Class)=
  s.packagedElement->select(e | e.oclIsTypeOf(Pro!Class));
  associations : Sequence(Pro!Association)=
  s.packagedElement->
        select(e|e.oclIsTypeOf(Pro!Association));
  collaborations : Sequence(Pro!Collaboration)=
  s.packagedElement->
        select(e|e.oclIsTypeOf(Pro!Collaboration));
  sendOperationEvents : Sequence(Pro!SendOperationEvent)=
  s.packagedElement->
        select(e|e.oclIsTypeOf(Pro!SendOperationEvent));
  receiveOperationEvents:
                        Sequence(Pro!ReceiveOperationEvent)=
  s.packagedElement->
```

```
        select(e|e.oclIsTypeOf(Pro!ReceiveOperationEvent));
          }
-- Creating the root element of the target model
-- packageImport identifies a package whose elements are imported by the model
-- namespace and allows for the use of unqualified names
-- packagedElement is also created to specify the packgeable elements which are
-- owned by this package
 to
       t : Pro!Model ( name<-'ProductModel' + '-' + s.name,
          packageImport <- s.packageImport,
          packagedElement <- usecases ->
                collect(e|thisModule.resolveTemp(e,'use')),
          packagedElement <- actors ->
                collect(e|thisModule.resolveTemp(e,'act')),
          packagedElement <- classes ->
              collect(e|thisModule.resolveTemp(e,'class')),
          packagedElement <- associations ->
                collect(e|thisModule.resolveTemp(e,'ass')),
          packagedElement <- collaborations->
                collect(e|thisModule.resolveTemp(e,'col')),
          packagedElement <- sendOperationEvents->
                collect(e|thisModule.resolveTemp(e,'soe')),
          packagedElement <- receiveOperationEvents->
               collect(e|thisModule.resolveTemp(e,'roe')))}
```

The model will contain model elements of type *UseCase, Actor, Class, Association, Collaboration, SendOperationEven,* and *ReceiveOperationEvent.*

The derivation of the use case, class diagram, and sequence diagrams will be explained in the following sections.

5.3 Use Case Diagram Derivation

The SPL use case diagram represents all possible use cases that may be needed by any member of the product family. Thus, the derivation can be done by selecting and copying the use cases from the SPL use case diagram to the product use case diagram. Since the concept of positive variability is applied to the proposed transformation approach, the derivation starts by copying the mandatory core representing common use cases to the target model, then selectively adds additional use cases based on the chosen features.

We need to create in the target model all the model element types that compose a use case diagram according to the UML metamodel: *UseCase*, *Actor*, *Association*, *Property*, *ExtentionPoint*, *Extend*, and *Include* [OMG07a]. Since *optional* and *alternative* use cases are annotated with the features requiring them, the *Use Case* element is selected if and only if the feature given in its annotation is present in the feature configuration. The following rule is applied to each model element of type UseCase from the source model, checking whether to select and copy it to the target model. A use case is selected if it is stereotyped as *kernel,* or if it is *optional* or *alternative* and is explicitly annotated with a feature from the feature configuration for the product.

```
-- Rule UseCase checks each model element of UseCase type whether to select and
-- copy it to the target model or not by calling the helper selectedElement()
rule UseCase{
     from
          s : Pro!UseCase (s.selectedElement()))

-- The extensionPoint owned by the use case is copied by calling the lazy rule
-- ExtensionPoint
     to
          use : Pro!UseCase ( name <- s.name,
          extensionPoint <- s.extensionPoint ->
                  collect(e| thisModule.ExtensionPoint(e)),

-- Check if the use case is an extending use case by navigating from it to its base use
```

```
-- case through  extendingUseCase.extend.extendedCase
            extend <-
      if (s.extend->collect(extended|extended.extendedCase)

-- Checking whether the base use case is selected or not by calling the helper
-- selectedElement()
  ->select(currentUsecase|currentUsecase.selectedElement())
  ->notEmpty())
-- Whenever the base use case is selected, the extending use case is copied to the target
-- model and the Extend relationship owned by the extending use case through extend is
-- copied as well by calling the lazy rule Extend
      then
            s.extend -> collect(e|thisModule.Extend(e))
      else
            OclUndefined
      endif,

-- The included use case is selected and copied to the target model  in the same manner
            include <-
      if (s.include->collect(included|included.extendedCase)
  ->select(currentUsecase|currentUsecase.selectedElement())
  ->notEmpty())
      then
            s.include -> collect(e|thisModule.Include(e))
      else
            OclUndefined
      endif
                  )}

-- This lazy rule is executed when called by the previous rule to copy the
-- ExtensionPoint to the target model
lazy rule ExtensionPoint{
         from
           s : Pro!ExtensionPoint
         to
           t : Pro!ExtensionPoint ( name <- s.name) }

-- This lazy rule is executed when called by the rule UseCase to copy the extend
-- relationship to the target model
lazy rule Extend{
   from
     s : Pro!Extend
   to
     t : Pro!Extend ( name <- s.name) }

-- This lazy rule is executed when called by the rule UseCase to copy the include
-- relationship to the target model
```

```
lazy rule Include{
   from
     s : Pro!Include
   to
     t : Pro!Include ( name <- s.name) }
```

Actor is a non-annotated element associated to one or more use cases, so its implicit mapping is evaluated as follows. Consider an example of a use case diagram fragment containing a use case *UCaseC* with an actor *ActorA* connected to it by an association *AC, which* owns two properties, *Pro1* and *Pro2,* referencing the use case *UCaseC* and the actor *ActorA,* respectively. Figure 5.1.a shows an abstract syntax fragment for the example use case model, which conforms to the UML metamodel [OMG07a]. Figure 5.2.a shows the navigation between annotated and non-annotated model elements during the transformation. First, we need to identify both the use case and actor connected to this association *AC.* This is done through the attribute *memberEnd,* which collects all the properties related to the association (see Figure 5.2.a, step 1). Since the *type* of the property refers to the end of the association [OMG07a], we can navigate to the use case and the corresponding actor through this attribute. Step 2 in Figure 5.2.a shows the navigation from *Pro1* to *UCaseC,* and step 3 from *Pro2* to actor *ActorA.* If *UCaseC* is selected, the actor *ActorA* and the association *AC* are selected as well. This implicit mapping is encoded in the following rules.

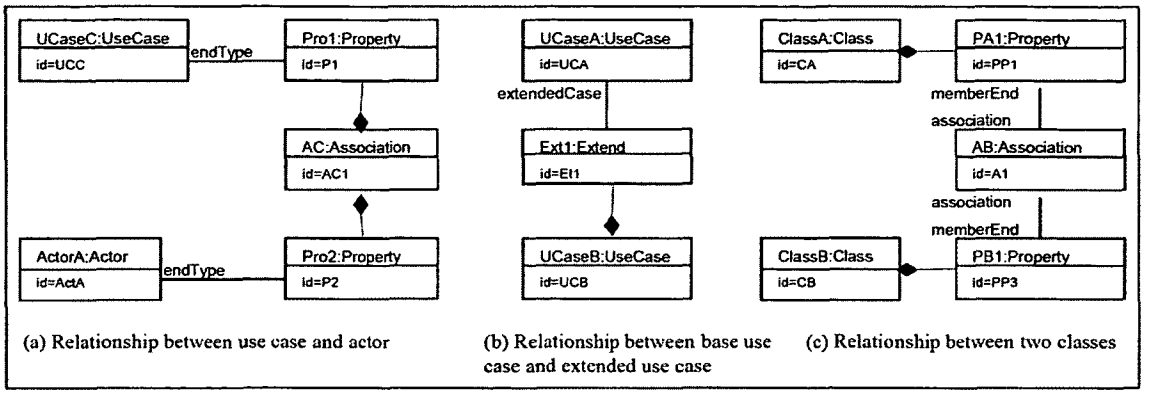


Figure 5.1: Abstract syntax fragments for use cases and classes.

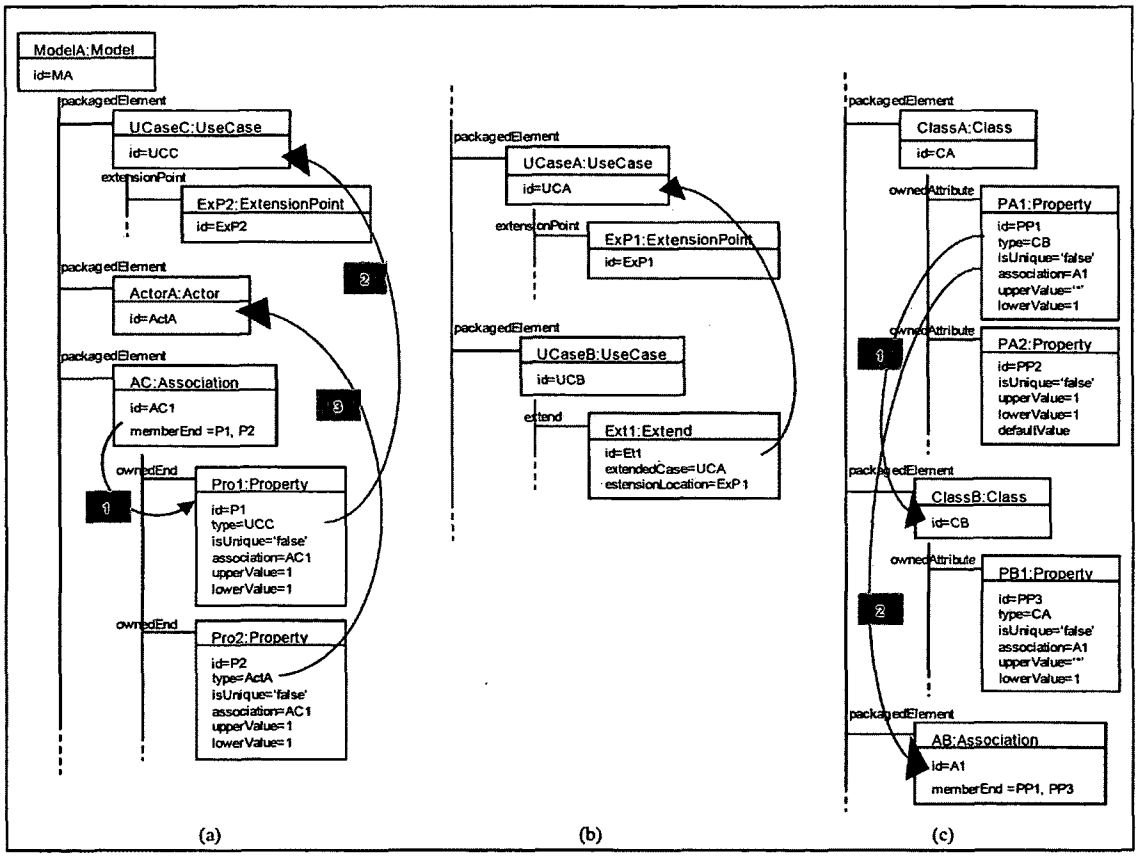


Figure 5.2: Navigation between annotated and non-annotated elements for use cases and classes.

```
-- Rule Association checks each model element of this type whether to select and copy
-- it to the target model or not by calling the helper isUseCaseAssociation ()
rule Association{
      from
            s : Pro!Association (s.isUseCaseAssociation())
      to
            ass : Pro!Association (name <- s.name,

-- The property owned by the association through ownedEnd is copied as well by
-- calling the lazy rule Property
ownedEnd <-s.ownedEnd-> collect(e|thisModule.Property(e)))}

-- This helper returns "true" if the attached use case is selected
helper context Pro!Element def : isUseCaseAssociation(): Boolean
= if
        self.refImmediateComposite().oclIsTypeOf(Pro!Model)

-- Navigating from an association to both of its ends through
-- association.memberEnd.type
      then if self.memberEnd->collect
             (currentProperty|currentProperty.type.
              oclIsTypeOf(Pro!UseCase))->notEmpty())

-- Checking whether the attached use case is selected or not by calling the helper
-- selectedElement()
      then self.memberEnd->collect
          (currentProperty|currentProperty.type)->select
          (currentUsecase|currentUsecase.selectedElement())
           -> notEmpty()
      else
            false
      endif
      else
            false
      endif;

-- This lazy rule is called by the previous rule to copy a property with its upper and
-- lower multiplicity values to the target model
lazy rule Property{
      from
            s : Pro!Property
      to
            t : Pro!Property(name <- s.name,
            upperValue <-thisModule.
                        LiteralUnlimitedNatural(s.upperValue),
            lowerValue <- thisModule.
                        LiteralInteger(s.lowerValue),
```

```
-- Copy to target model the actor referenced by this property through property.type by
-- calling the lazy rule Actor
        type <- thisModule.Actor(s.type))}

-- This lazy rule is executed when called by the previous rule to copy the actor to the
-- target model
unique lazy rule Actor{
    from
        s : Pro!Actor
    to
        act : Pro!Actor (name <- s.name) }
```

An *extending* or *included* use case is selected if and only if its base use case is selected. In order to check that, consider the following example: *UCaseA* is a base use case, and *UCaseB* is extending it; the abstract syntax is given in Figure 5.1.b and navigation in Figure 5.2.b. First, through the *Extend* relationship owned by *UcaseB,* where the attribute *extendedCase* of *Extend* references the base *UCaseA*, we can identify if the base *UCaseA* is selected. If this is the case, then the extending *UCaseB* is selected too. The *Included* use case is treated in a similar way. The rule *UseCase* interprets this implicit mapping by navigating from an extending use case to the base use case.

Whenever a use case has an *ExtesionPoint*, *Extend*, or *Include* relationships according to the containment hierarchies of the UML metamodel, they are automatically selected if their container is selected. The same concept applies to properties owned by an association.

5.4 Class Diagram Derivation

The class diagram is generated in a similar manner to the use case diagram. Given that the SPL class diagram represents the union of all possible product class diagrams, the derivation can be done by selecting and copying the classes from the SPL class diagram

to the product class diagram. The *kernel* classes are copied first, and then the *optional* and *variant* classes are selected and copied if and only if the features given in their annotations are presented in the feature configuration.

The following types of model element are non-annotated: *Property*, *Operation*, *Generalization*, and *Association*. Therefore, their implicit mapping has to be found from the related neighbouring elements. We need to distinguish between a property representing an attribute (related to the class by *ownedAttribute*) and a property representing an association end (related to an association by *memberEnd)*. A property representing an attribute has to be selected if its container is selected. However, the one representing an association end is selected if and only if its class container and the related association are selected.

In order to select and copy to the target model only the associations between selected classes as well as their *memberEnds*, we have to navigate from the property of a selected class that represents an association end to the other end of the association and check whether the class on this end is selected or not. Assume that there are two classes: *ClassA* and *ClassB* connected with an association *AB* as shown in the abstract syntax in Figure 5.1.c. *ClassA* owns a property *PA1* that has an attribute *type* referencing the other end of the association, *ClassB*, as shown in Figure 5.2.c step 1. In turn, *PA1* has an attribute *association* referencing the association *AB* as shown in Figure 5.2.c step 2. The rule that interprets this implicit mapping navigates from the selected *ClassA* to the other end *ClassB* though the attribute *type* of the property *PA1* and checks whether *ClassB* is selected or not. If *ClassB* is selected, property *PA1* is selected as well. Last step is to

navigate through the attribute *association* of property *PA1* to the association *AB*, and to copy it to the target model.

```
-- Rule Class checks each model element of this type whether to select and copy it
-- to the target model or not by calling the helper selectedElement()
rule Class{
     from
          s : Pro!Class (s.selectedElement())
     using {
          ownedAttributes : Sequence(Pro!Property)=
                    s.ownedAttribute->
                    select(e|e.oclIsTypeOf(Pro!Property));
          ownedOperations : Sequence(Pro!Operation)=
                    s.ownedOperation->
                    select(e|e.oclIsTypeOf(Pro!Operation));
          generalizations : Sequence(Pro!Generalization)=
               s.generalization->
               select(e|e.oclIsTypeOf(Pro!Generalization));
           }

-- Copying the class, the property representing an association end, and the property,
-- the operation, and the generalization owned by the class
     to
          class : Pro!Class (name <- s.name,
               ownedAttribute <- ownedAttributes->
               collect(e|thisModule.resolveTemp(e,'pro')),
               ownedOperation <- ownedOperations->
                collect(e|thisModule.resolveTemp(e,'ope')),
               generalization <- generalizations->
                 collect(e|thisModule.resolveTemp(e,'gen'))
                                          )
     }

-- This rule copies an ownedAttribute property with its upper, and
-- lower multiplicity values to the target model
rule PropertyClass{
     from
          s : Pro!Property (s.isClassProperty())
     to
          pro : Pro!Property (name <- s.name,
          owningAssociation <- s.owningAssociation,
          type <- s.type,
          upperValue <-
          thisModule.LiteralUnlimitedNatural(s.upperValue),
          lowerValue <-
          thisModule.LiteralInteger(s.lowerValue)
                              )
```

```
}

-- This helper checks for each property representing an association end
--  whether the class on the other end of the association is selected and
--  returns "true" if both classes are selected
helper context Pro!Element def:isClassProperty(): Boolean =
        if self.class.oclIsTypeOf(Pro!Class)
             then if self.type.oclIsTypeOf(Pro!Class)
                     then if self.class.selectedElement()
                             then self.type.selectedElement()
                             else false
                            endif
                       else
                              false
                    endif
             else
                  false
        endif;

-- Rule Operation checks each model element of this type whether to select and copy it
-- to the target model by checking whether the class owned this operation is selected
-- or not
rule Operation{
      from
            s : Pro!Operation (s.class.selectedElement())
      to
            ope : Pro!Operation (name <- s.name)}

-- Rule Generalization checks each model element of this type whether to select and
-- copy it to the target model by checking whether the class owned this generalization
-- is selected or not
rule Generalization{
      from
        s : Pro!Generalization(s.specific.selectedElement())
      to
        gen : Pro!Generalization (general <- s.general)}

-- This rule copies the Association
-- and its ownedEnd and memberEnd to the target model
rule Association{
      from
            s : Pro!Association (s.isClassAssociation())
      to
            ass : Pro!Association (
                  name <- s.name,
                  ownedEnd <- s.ownedEnd,
                  memberEnd <- s.memberEnd )}
```

```
-- This helper checks for each association whether the classes on both end of the
-- association is selected and returns "true" if both classes are selected
helper context Pro!Element def : isClassAssociation():
    Boolean = if
        self.refImmediateComposite().oclIsTypeOf(Pro!Model)
            then if self.memberEnd->
                collect(currentProperty|currentProperty.
                type.oclIsTypeOf(Pro!Class))->notEmpty()
                 then self.memberEnd->
                collect(currentProperty|currentProperty.
                type)->select(currentClass|currentClass.
                selectedElement())->size()=2
                 else
                        false
                  endif
            else
                  false
            endif;
```

Properties related to a class (attributes), generalizations, and operations are elements contained into a class, so according to the UML containment hierarchies they are selected whenever their container is selected.

5.5 Sequence Diagram Derivation

The SPL Sequence diagrams represent all possible behaviours that may be supported by any member of the product family. A SPL sequence diagram (SD) is generic, which is used to describe the objects that participate in each use case scenario and the sequence of messages between them, and may contain different objects corresponding to *optional* or *variant* classes.

Thus, the derivation can be done by selecting and copying to the product model only SD that realized a selected use case scenario. We need to create all the model element types that compose a SD according to the UML metamodel: *Lifeline*, *Message*, *Message Occurrence Specification*, *Behaviour Execution Specification*, *Execution*

Occurrence Specification, and *Combined Fragment* realizing behaviour at run-time [OMG07a].

We introduce the following notation:

- SD_T : the set of all scenarios in all use cases
- SD_S : the set of selected sequence diagrams, where $SD_S \subseteq SD_T$

An interaction is selected if it is realizing a selected scenario.

```
-- Rule Interaction checks each model element of this type whether to select and copy it
-- to the target model or not by calling the helper selectedSD()
-- The Interaction contains model elements of types Message, Lifeline,
-- ExecutionOccurrenceSpecification, BehaviorExecutionSpecification,
-- MessageOccurrenceSpecification and CombinedFragment
rule Interaction {
      from
            s : Pro!Interaction (s.selectedSD())
      using {
            messages:Sequence(Pro!Message)=s.message->
                        select(e|e.oclIsTypeOf(Pro!Message));
            lifelines:Sequence(Pro!lifeline)=s.lifeline->
                        select(e|e.oclIsTypeOf(Pro!Lifeline));
            executionOccurrenceSpecifications:Sequence
                        (Pro!ExecutionOccurrenceSpecification)=
                        s.fragment ->    select(e| e.oclIsTypeOf
                        (Pro!ExecutionOccurrenceSpecification));
            behaviorExecutionSpecifications:Sequence
                        (Pro!BehaviorExecutionSpecification)=
                        s.fragment -> select(e| e.oclIsTypeOf
                        (Pro!BehaviorExecutionSpecification));
            messageOccurrenceSpecifications:Sequence
                        (Pro!MessageOccurrenceSpecification)=
                        s.fragment -> select(e| e.oclIsTypeOf
                        (Pro!MessageOccurrenceSpecification));
            combinedFragments:Sequence(Pro!CombinedFragment)=
                        s.fragment -> select(e| e.oclIsTypeOf
                        (Pro!CombinedFragment));
            executionOccurrenceSpecificationsInFrag:Sequence
                        (Pro!ExecutionOccurrenceSpecification)=
        if not(s.fragment.oclIsTypeOf(Pro!CombinedFragment))
            then s.fragment ->select(e | e.oclIsTypeOf
                        (Pro!ExecutionOccurrenceSpecification))
            else
                  s.fragment.operand.fragment->select(e|e.
          oclIsTypeOf(Pro!ExecutionOccurrenceSpecification))
```

```
        endif;
            messageOccurrenceSpecificationsInFrag:Sequence
                      (Pro!MessageOccurrenceSpecification)=
        if (s.fragment.oclIsTypeOf(Pro!CombinedFragment))
            then s.fragment.operand.fragment->select(e|e.
            oclIsTypeOf(Pro!MessageOccurrenceSpecification))
            else
              s.fragment->select(e|e.oclIsTypeOf
             (Pro!MessageOccurrenceSpecification))
        endif;
                  }
    to
         int : Pro!Interaction ( name <- s.name,
                                    message <- messages ->
                 collect(e|thisModule.resolveTemp(e,'msg')),
                                  lifeline <- lifelines ->
                 collect(e|thisModule.resolveTemp(e,'lif')),
                fragment<-executionOccurrenceSpecifications->
                 collect(e|thisModule.resolveTemp(e,'eos')),
                fragment<-behaviorExecutionSpecifications->
                 collect(e|thisModule.resolveTemp(e,'bes')),
                fragment<-messageOccurrenceSpecifications->
                 collect(e|thisModule.resolveTemp(e,'mos')),
                fragment<-combinedFragments->
                 collect(e|thisModule.resolveTemp(e,'cf'))))
```

```
-- This helper returns "true" if the respective model element of type Interaction is
-- selected by checking whether its name is in the list of the selected
-- use case names
helper context Pro!Element def : selectedSD(): Boolean =
      let usecases : Sequence(String) =
Pro!UseCase.allInstances()-> select(uc|uc.selectedElement())->
collect(c|c.name) in
if (usecases -> includes(self.name ))
         then true
         else false
       endif;
```

A *lifeline* is selected if it is required by a selected feature and not contained in an unselected VCF or if it is contained in a selected VCF and required a selected feature. Figure 5.3 shows an example of the unselected lifeline *C*, since it is contained in the unselected optional *Combined Fragment*. We mean by contained in VCF that the lifeline is created and destroyed in the VCF as the lifeline *C*.

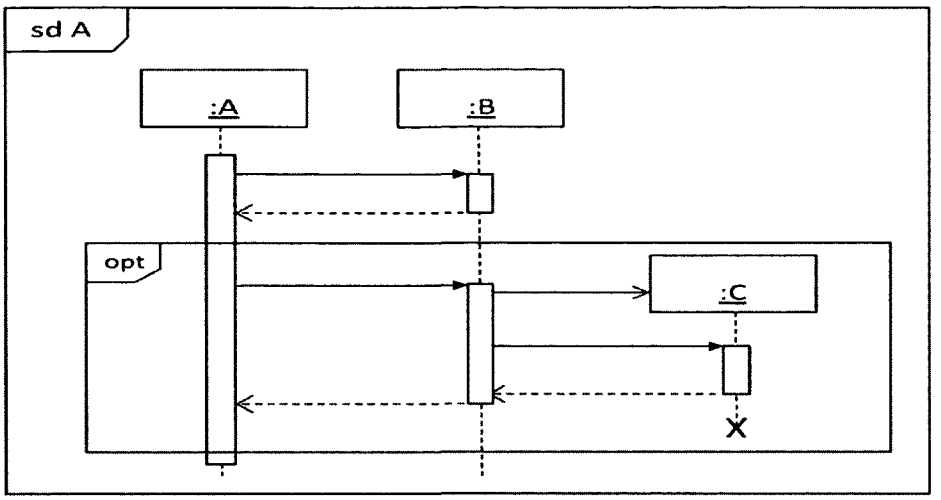


Figure 5.3: Example of Lifeline contained in Combined Fragment.

The following constraint is used to identify a selected *lifeline*.

$$((l_s \in L_S \wedge l_s \notin (VCF_T - VCF_S)) \vee (l_s \in L_S \wedge l_s \in VCF_S))$$

The following ATL rule represents this constraint.

```
-- Rule Lifeline checks each model element of this type whether to select and copy it
-- to the target model or not by calling the helpers selectedElement()
rule Lifeline {
      from
            s : Pro!Lifeline (s.interaction.selectedSD() and
                  ((s.selectedElement() and
                              (not s.isContainedInFragment()))
                     Or (s.selectedElement() and
                           (s.isContainedInSelectedFragment()))))
      to
            lif : Pro!Lifeline (name <-
                  if s.name = 'CustomerInterface'
                        then
                              s.getTagValue('Variant','feature')
                        else
                              s.name

                  endif,
                              coveredBy <- s.coveredBy)}
```

In order to select and copy to the target model only *Messages* between selected *Lifelines*, we have to navigate from the message to the *Message Occurrence Specifications*, then to both lifelines and check whether these lifelines are selected or not. Assume that there is a message *MessageA* that its sender lifeline is *LifeLineA* and its receiver is *LifeLineB*. *MessageA* has two attributes *sendEvent* and *receiveEvent* referencing two model elements of type *Message Occurrence Specification*, *MOSA* and *MOSB*, as shown in Figure 5.4.a. In turn each one of *MOSA* and *MOSB* has an attribute *covered* referencing the lifeline *LifeLineA* and *LifeLineB*, respectively. The rule that interprets this implicit mapping navigates from *MessageA* to *MOSA* though the attribute *sendEvent* and to *MOSB* through *receiveEvent* as shown in Figure 5.4.a step 1, then from *MOSA* to *LifeLineA* and from *MOSB* to *LifeLineB* though the attribute *covered* as shown in Figure 5.4.a step 2, and checks whether *LifeLineA* and *LifeLineB* are selected or not.

A *Message* is selected if both sender and receiver (lifelines) are selected and not contained in an unselected VCF or contained in a selected VCF.

Assume the following notation:

- $l_d(m)$: the sender lifeline of message m
- $l_r(m)$: the receiver lifeline of message m
- $l_c(mos)$: the lifeline covered by Message Occurrence Specification mos
- M_T : the set of all messages in a sequence diagram
- M_S : the set of selected messages in a sequence diagram, where $M_S \subseteq M_T$

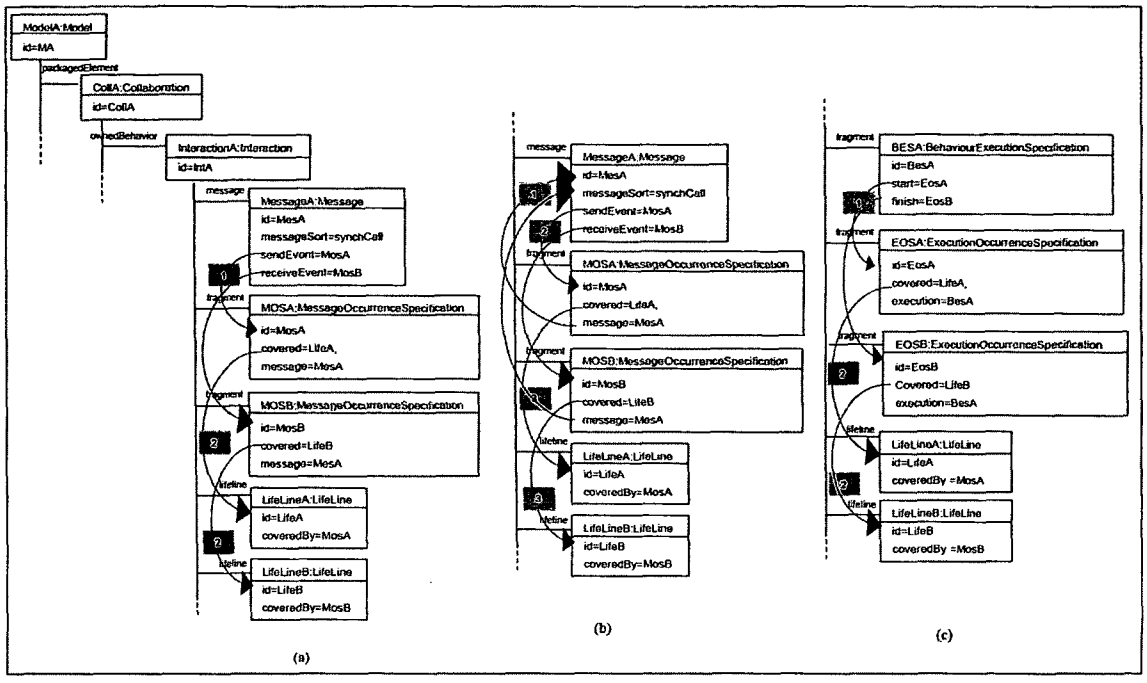


Figure 5.4: Navigation between elements explicitly annotated with features and non-annotated in sequence diagrams.

The condition for selecting a message is as follows:

$\forall m_s \in M_S : ((\ l_d(m) \in L_S \wedge\ l_r(m) \in L_S \wedge m_s \notin (\ VCF_T - VCF_S\ )) \vee (\ l_d(m) \in L_S \wedge\ l_r(m) \in L_S \wedge m_s \in VCF_S\ ))$

The following ATL rule represents this constraint.

```
rule Message {
     from
          s : Pro!Message (((s.sendEvent.covered->
              select(l|l.selectedElement())->size()=1) and
                        (s.receiveEvent.covered->
              select(l|l.selectedElement())->size()=1) and
                        (not s.isContainedInFragment()))
          Or            ((s.sendEvent.covered->
              select(l|l.selectedElement())->size()=1) and
                        (s.receiveEvent.covered->
              select(l|l.selectedElement())->size()=1) and
                       (s.isContainedInSelectedFragment())))
     to
          msg : Pro!Message ( name <- s.name,
```

```
        messageSort <- s.messageSort,
        sendEvent <- s.sendEvent,
        receiveEvent <- s.receiveEvent )}
```

Message Occurrence Specification is a non-annotated element that specifies the occurrence of events. It is a kind of message end, thus its implicit mapping is evaluated as follows. Consider an example of a *Message Occurrence Specification, MOSA* that represents the sending end of the message *MessageA* and *MOSB* represents the receiving end of the same message *MessageA* as shown in Figure 5.4.b. To ensure that *MessageA* is selected, we first navigate to *MessageA* through the attribute *message*, step 1 in Figure 5.4.b, then from *MessageA* back to *MOSA* and *MOSB* through the attributes *sendEvent* and *receiveEvent*, respectively as shown in Figure 5.4.b step 2. Through the attribute *covered*, we navigate to *LifeLineA* and *LifeLineB*, step 3 in Figure 5.4.b and check whether they are selected or not.

Assume the following:

- MOS_T : the set of all *Message Occurrence Specification* in a sequence diagram
- MOS_S : the set of selected Message Occurrence Specification in a sequence diagram, where $MOS_S \subseteq MOS_T$

The condition for selecting a MOS and its ATL implementation are as follows:

$\forall mos_s \in MOS_S : ((\ l_c(mos) \in L_S \wedge mos_s \notin (\ VCF_T - VCF_S\ )) \vee (l_c(mos) \in L_S \wedge mos_s \in VCF_S\ ))$

```
rule MOS {
     from
          s : Pro!MessageOccurrenceSpecification
                    (((s.message.sendEvent.covered->
               select(l|l.selectedElement())->size()=1)   and
                    (s.message.receiveEvent.covered->
               select(l|l.selectedElement())->size()=1)   and
```

```
                not (s.isContainedInFragment()))
    or           ((s.message.sendEvent.covered->
        select(l|l.selectedElement())->size()=1) and
                  (s.message.receiveEvent.covered->
        select(l|l.selectedElement())->size()=1) and
                 (s.isContainedInSelectedFragment())))
to
     mos : Pro!MessageOccurrenceSpecification
          (name <- s.name,
            covered <- s.covered,
            event <- s.event,
            message <- s.message,
            enclosingInteraction <- if
                    s.isContainedInSelectedFragment()
        then Sequence{s.enclosingOperand.
      refImmediateComposite().enclosingInteraction}->
      collect(e|thisModule.resolveTemp(e,'int'))
      ->first()
        else  Sequence{s.enclosingInteraction}->
          collect(e|thisModule.resolveTemp(e,'int'))
          ->first()
     endif  )}
```

Behaviour Occurrence Specification is another element not explicitly mapped with the features requiring it, which represents the execution of a unit of behavior within the *Lifeline*. It is represented by two *Execution Occurrence Specifications*, the start and the finish [OMG07a], thus its implicit mapping is evaluated as follows. Consider an example of a *Behaviour Occurrence Specification, BESA* with the attributes *start* and *finish* referencing the two *Execution Occurrence Specifications, EOSA* and *EOSB*, respectively. The rule that interprets this implicit mapping navigates from *BESA* to *EOSA* and *EOSB* though the attributes *start* and *finish*, respectively, as shown in Figure 5.4.c step 1, then from *EOSA* to *LifeLineA* and from *EOSB* to *LifeLineB* though the attribute *covered* as shown in Figure 5.4.c step 2.

Assume the following notation:

- BES_T : the set of all Behaviour Execution Specification in a sequence diagram

- BES_S : the set of selected Behaviour Execution Specification in a sequence diagram, where $MOS_S \subseteq MOS_T$

The condition for selecting a *Behaviour Occurrence Specification* and its ATL implementation are as follows:

$\forall bes_s \in BES_S : ((l_c(mos) \in L_S \wedge bes_s \notin VCF_T - VCF_S)) \vee (l_c(mos) \in L_S \wedge bes_s \in VCF_S))$

```
rule BES {
     from
          s : Pro!BehaviorExecutionSpecification
                    (((s.finish.covered->
               select(l|l.selectedElement())->size()=1) and
                         (s.start.covered->
               select(l|l.selectedElement())->size()=1) and
                        (not s.isContainedInFragment()))
     or                  ((s.finish.covered->
               select(l|l.selectedElement())->size()=1) and
                        (s.start.covered->
               select(l|l.selectedElement())->size()=1) and
                        (s.isContainedInSelectedFragment())))
to
          bes : Pro!BehaviorExecutionSpecification
                   (name <- s.name,
                      start <- s.start,
                      finish <- s.finish,
                      enclosingInteraction <- if
                          s.isContainedInSelectedFragment()
          then     Sequence{s.enclosingOperand.
           refImmediateComposite().enclosingInteraction}->
           collect(e|thisModule.resolveTemp(e,'int'))
           ->first()
          else    Sequence{s.enclosingInteraction}->
                 collect(e|thisModule.resolveTemp(e,'int'))
                 ->first()
          endif  )}
```

Execution Occurrence Specification is another element not explicitly mapped with the features requiring it, which represents a starting or ending event for actions or behaviors [OMG07a]. Consider the previous example in Figure 5.4.c, where we have two *Execution Occurrence Specifications, EOSA and EOSB* representing the starting and

finishing of the *Behaviour Occurrence Specification, BESA*. To ensure that *BESA* is selected, we first navigate to *BESA* through the attribute *execution*, then from *BESA* back to *EOSA* and *EOSB* through the attributes *start* and *finish*, respectively as shown in Figure 5.4.c step 1. Through the attribute *covered*, we navigate to *LifeLineA* and *LifeLineB*, step 2 in Figure 5.4.c and check whether they are selected or not.

Assume the following notation:

- EOS_T : the set of all Execution Occurrence Specification in a sequence diagram
- EOS_S : the set of selected Execution Occurrence Specification in a sequence diagram, where $EOS_S \subseteq EOS_T$

Therefore:

$\forall eos_s \in EOS_S ((l_c(mos) \in L_S \wedge eos_s \notin VCF_T - VCF_S)) \vee (l_c(mos) \in L_S \wedge eos_s \in VCF_S))$

```
rule EOS {
     from
          s : Pro!ExecutionOccurrenceSpecification
                    (((s.execution.start.covered->
               select(l|l.selectedElement())->size()=1) and
                    (((s.execution.finish.covered->
               select(l|l.selectedElement())->size()=1) and
                    (not s.isContainedInFragment()))

       or           ((s. execution.start.covered->
               select(l|l.selectedElement())->size()=1) and
                    (((s.execution.finish.covered->
               select(l|l.selectedElement())->size()=1) and
                    (s.isContainedInSelectedFragment())))
     to
          eos : Pro!ExecutionOccurrenceSpecification
                   (name <- s.name,
                     covered <- s.covered,
                     execution <- s.execution,
                     enclosingInteraction <- if
                        s.isContainedInSelectedFragment()
          then  Sequence{s.enclosingOperand.
              refImmediateComposite().enclosingInteraction}
              ->collect(e|thisModule.resolveTemp(e,'int'))
              ->first()
```

```
    else  Sequence{s.enclosingInteraction}->
          collect(e|thisModule.resolveTemp(e,'int'))
          ->first()
endif )}
```

Combined Fragments are extended to model variability within a sequence diagram during the construction of the SPL model at design time. However, during the product derivation process, a choice must be made based on the characteristics of the selected feature (for instance, some features can allow multiple run-time simultaneous choices for the same product, others do not). Each operand in a VCF needs to be mapped to a specific behaviour in the product SD. Thus, two different stereotypes are introduced to define the specific run-time behaviour in the product model based on the selected feature(s). The stereotype *«SingleChoiceFeature»* has two attributes *RegB* and *OptB* referring to the regular and optional behaviours, respectively, while *«MultiChoiceFeature»* has three attributes *RegB*, *OptB*, and *AltB* referring to regular, optional, and alternative behaviours, respectively.

We define several types of feature groups as shown in table 5.1, where each type is annotated with different stereotypes to distinguish between different run-time behaviours. For instance, the At-Least-One-of feature group is represented as VCFs with *«OptDesignTime»* and annotated with *«SingleChoiceFeature» {RegB=True}* as shown in Figure 5.5.a. There are two possibilities for the selected operand: 1) to become a part of

Table 5.1: Variability Combined Fragments and Intended Run-Time Behaviour.

Type of Feature Group	Type of Variability Combined Fragment at Design Time	Intended Run-Time Behaviour of the Generated Product	Stereotype and Tagged Value	Mapping Constraints
Exactly-One-of Feature group	*Alt* VCF «AltDesignTime» {VP}	Optional behaviour	«SingleChoiceFeature» {OptB=True}	∃vcfs: EO(vcfs) ∧ OptSingle(vcfs) ⇒ OPT(product)
		Regular behaviour	«SingleChoiceFeature» {RegB=True}	∃vcfs: EO(vcfs) ∧ RegSingle(vcfs) ⇒REG(product)
Zero-or-one-of Feature group	*Alt* VCF «AltDesignTime» {VP}	Optional behaviour	«SingleChoiceFeature» {OptB=True}	∃vcfs: ZO(vcfs) ∧ OptSingle(vcfs) ⇒ OPT(product)
		Regular behaviour	«SingleChoiceFeature» {RegB=True}	∃vcfs: ZO(vcfs) ∧ RegSingle(vcfs) ⇒REG(product)
At-Least-One-of Feature group	*Opt* VCF «OptDesignTime» {VP}	Optional behaviour	«SingleChoiceFeature» {OptB=True}	∃vcfs:AO(vcfs) ∧ OptSingle(vcfs) ⇒ OPT (product)
		Optional behaviour	«MultiChoiceFeature» {OptB=True}	∃vcfs:AO(vcfs) ∧ OptMulti(vcfs)⇒ OPT(product)
		Regular behaviour	«SingleChoiceFeature» {RegB=True}	∃vcfs:AO(vcfs) ∧ RegSingle(vcfs) ⇒ REG(product)
		Regular behaviour	«MultiChoiceFeature» {RegB=True}	∃vcfs:AO(vcfs) ∧ RegMulti(vcfs) ⇒ REG(product)
		Alternative behaviour	«MultiChoiceFeature» {AltB= True}	∃vcfs:AO(vcfs) ∧ AltMulti(vcfs) ⇒ ALT(product)
Zero-or-more-of Feature group	*Opt* VCF «OptDesignTime» {VP}	Optional behaviour	«SingleChoiceFeature» {OptB=True}	∃vcfs:ZM (vcfs) ∧ OptSingle(vcfs) ⇒ OPT(product)
		Optional behaviour	«MultiChoiceFeature» {OptB=True}	∃vcfs:ZM (vcfs) ∧ OptMulti(vcfs) ⇒ OPT(product)
		Regular behaviour	«SingleChoiceFeature» {RegB=True}	∃vcfs:ZM(vcfs) ∧ RegSingle(vcfs) ⇒ REG(product)
		Regular behaviour	«MultiChoiceFeature» {RegB=True}	∃vcfs:ZM (vcfs) ∧ RegMulti(vcfs) ⇒ REG(product)
		Alternative behaviour	«MultiChoiceFeature» {AltB= True}	∃vcfs:ZM (vcfs) ∧ AltMulti(vcfs)⇒ ALT(product)
More-than-One-of Feature group	*Opt* VCF «OptDesignTime» {VP}	Optional behaviour	«MultiChoiceFeature» {OptB=True}	∃vcfs:MO (vcfs) ∧ OptMulti(vcfs) ⇒ OPT(product)
		Regular behaviour	«MultiChoiceFeature» {RegB=True}	∃vcfs:MO (vcfs) ∧ RegMulti(vcfs) ⇒ REG(product)
		Alternative behaviour	«MultiChoiceFeature» {AltB= True}	∃vcfs:MO (vcfs) ∧ AltMulti(vcfs)⇒ ALT(product)

the regular behaviour of the product (so it will be added to the regular behavior); 2) to represent an optional behaviour at run-time (so it shall be represented as optional *Combined Fragment*). Assuming operand *"I"* is selected, it will be a part of the regular run-time behaviour in the product model as shown in Figure 5.5.b since the attribute *RegB* is set to *True*. Figure 5.5.c shows another example of the Exactly-One-of feature group, which represented as a VCF with *«AltDesignTime»* and annotated with *«SingleChoiceFeature» {OptB=True}*. The selected operand *"II"* will be represented as optional *Combined Fragment* in the product model, as shown in Figure 5.5.d. The More-

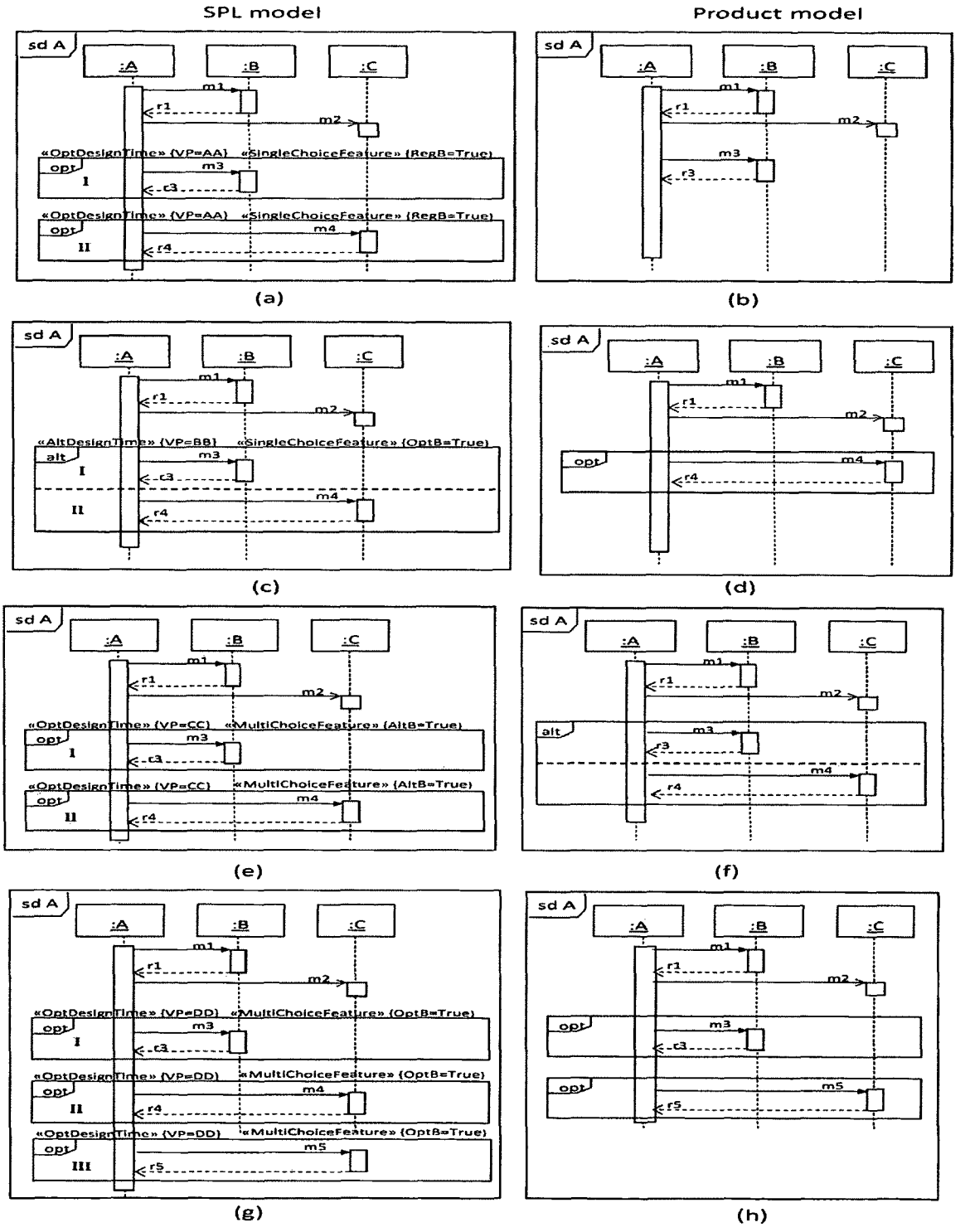


Figure 5.5: Examples of Variability Combined Fragments and Intended Run-Time Behaviour.

than-one-of feature group is represented in Figure 5.5.e, 5.5.g. In Figure 5.5.e, the SPL model contained two optional VCFs stereotyped with *«MultiChoiceFeature» {AltB=True}*. Since the attribute *AltB* is set to *True*, the two operands *"I","II"* are represented as two operands of an alternative *Combined Fragment* in the product model, as shown in Figure 5.5.f . However, in Figure 5.5.g, the attribute *OptB* is set to *True*, thus each one of the two selected operand *"I", "III"* is represented as an optional *Combined Fragment* in the product model, as shown in Figure 5.5.h.

We introduce the following notation:

EO(vcf_s) : a selected VCF corresponding to an Exactly-One-of feature group

ZO(vcf_s) : a selected VCF corresponding to a Zero-or-One-of feature group

AO(vcf_s) : a selected VCF corresponding to an At-Least-One-of feature group

ZM(vcf_s) : a selected VCF corresponding to a Zero-or-more-of feature group

MO(vcf_s) : a selected VCF corresponding to a More-than-One-of feature group

RegSingle(vcf_s) : a vcf_s stereotyped as «*SingleChoiceFeature*» with tagged value *RegB* = *True*.

OptSingle(vcf_s) : a vcf_s stereotyped as «*SingleChoiceFeature*» with tagged value *OptB* = *True*.

RegMulti(vcf_s) : a vcf_s stereotyped as «*MultiChoiceFeature*» with tagged value *RegB* = *True*.

OptMulti (vcf_s) : a vcf_s stereotyped as «*MultiChoiceFeature*» with tagged value *OptB* = *True*.

AltMulti (vcf_s) : a vcf_s stereotyped as «*MultiChoiceFeature*» with tagged value *AltB* = *True*.

OPT(product) : an *opt* combined fragment generated in the product SD

ALT(product) : an *alt* combined fragment generated in the product SD

REG(product) : a part of the regular behaviour generated in the product SD

The following are some of the constraints used to define the mapping between different types of variability combined fragment at design time and the intended run-time behaviour of the generated product. All the mapping constrains are defined in table 5.1.

$\exists$ vcf_s : $EO(vcf_s) \wedge OptSingle(vcf_s) \Rightarrow OPT(product)$

$\exists$ vcf_s : $ZO(vcf_s) \wedge RegSingle(vcf_s) \Rightarrow REG(product)$

$\exists$ vcf_s : $AO(vcf_s) \wedge RegMulti(vcf_s) \Rightarrow REG(product)$

$\exists$ vcf_s : $ZM(vcf_s) \wedge OptMulti(vcf_s) \Rightarrow OPT(product)$

$\exists$ vcf_s : $MO(vcf_s \wedge AltMulti(vcf_s) \Rightarrow ALT(product)$

Chapter 6: Performance Completions

This chapter covers the variability space of the performance completions and represents it through so-called Performance Completion-feature model (PC-feature model). The PC-feature model explicitly captures the variability in the performance completions and expresses the dependencies and relationships between them.

The concept of platform independent models (PIM) and platform specific models (PSM) was introduced by OMG's Model Driven Architecture (MDA). A PIM addresses the system independently of supporting platform details such as the middleware and implementation. This makes the model portable and reusable. PSM gives a more detailed lower level view of the system and takes the underlying platform into account. The transformations from platform independent models to platform specific models and then to code are introduced in the literature through different techniques.

Performance is a run-time property of the deployed system and depends on two types of factors: some are contained in the design model of the product (obtained from the SPL model) while others characterize the underlying platforms and run-time environment are external to the design model. Performance models need to reflect both types of factors. Woodside et al. proposed the concept of performance completions to close the gap between abstract design models and external factors [WOO02]. As reviewed in section 3.1, several approaches were proposed to model the impact of the

platform/middleware on the architecture and the performance of software systems [VER05], [COR07], [BAL06], [BEC08], [HAP10].

Including the platform overheads into software models can be achieved in two different ways [VER05], [MEN04]:

1) Combining the platform overheads and adjusting the performance attributes of the existing model elements of the systems based on these overheads. Adding no components would make the models simple and would still allow estimation of the system performance.
2) Modeling the platform and its overheads by adding new, middleware-specific components to the system model. These components represent the changes caused by using the middleware and the impacts on the overall system performance. Although this way makes the system models to be more complex, it provides more fine-grained model and sufficient details to generate more accurate performance estimates and identify the exact cause of the performance problems.

In this research, we propose to include the performance impact of underlying platforms into the UML+MARTE model of a product as aggregated platform overheads, expressed in MARTE annotations attached to existing processing and communication resources in the generated product model. This will keep the model simple and still allow us to generate a performance model containing the performance effects of both the product and the platforms. At the same time, this way gives designers the opportunity to rapidly model the system using different types of middleware, without having to explore the internals of all those different middleware types. The obtained models can then be

Table 6.1: Mapping of PC-features to affected performance attributes.

PC-feature	Affected Performance Attribute	MARTE Stereotype	MARTE Attribute
SecureCommunication	Comm. overhead	PaCommStep	commRcvOvh commTxOvh
ChannelType	Channel Capacity Channel Latency	GaCommHost	capacity blockT
DataCompression	Message size Comm. overhead	PaCommStep	msgSize commRcvOvh commTxOvh
ExternalDeviceType	Service Time	PaStep	extOpDemand
MessageType	Comm. overhead	PaCommStep	commTxOvh
Retrieved Document	Service Time	PaStep	prob

used to compare the system performance using the different types of middleware and make a well-founded decision about which middleware to use.

As already mentioned, performance completions provide a means to extend the modeling constructs of a system by including the influence of the underlying platforms and execution environments in performance evaluation models. In this chapter, the variability space of the performance completions is covered and represented through a Performance Completion-feature model (PC-feature model). Each feature from the PC-feature model may affect one or more performance attributes. For instance, data compression reduces the message size and at the same time increases the processor communication overhead for compressing and decompressing the data. Thus, it is mapped to the performance attributes message size and communication overhead through the MARTE attributes *msgSize*, *commTxOvh* and *commRcvOvh*, respectively. The mapping here is between a PC-feature and the performance attribute(s) affected by it,

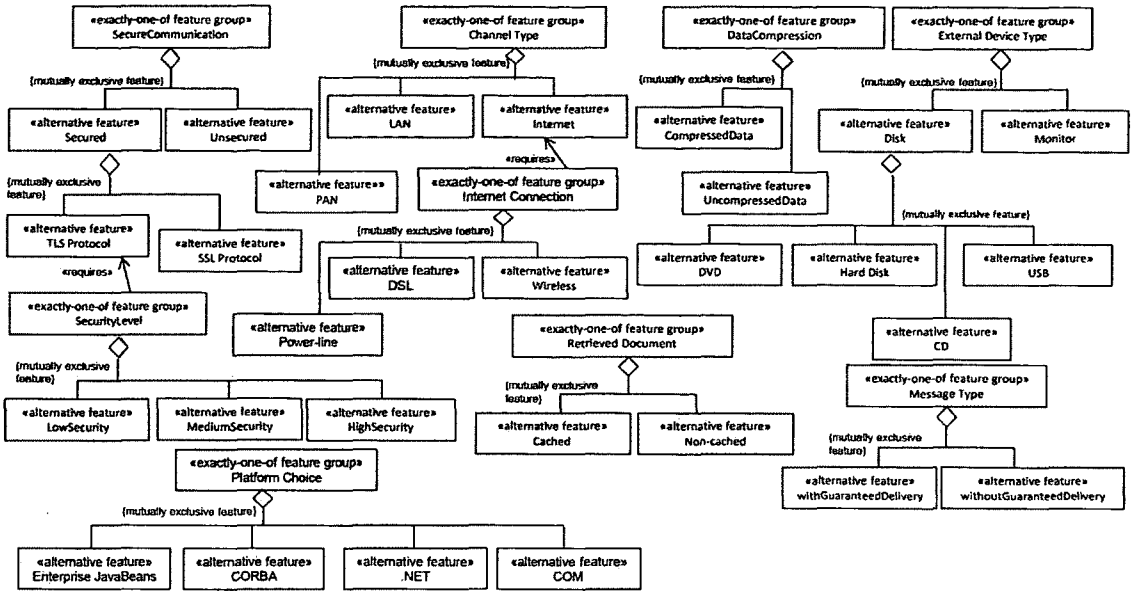


Figure 6.1: Part of the Performance-Completion feature model of the e-commerce SPL.

which are represented as MARTE stereotype attributes associated to different model elements. Table 6.1 illustrates this type of mapping between PC-features and the design model with MARTE annotations.

Please note that the PC-feature model is not related to a specific application or product, as opposed to the regular feature model which is product-line specific. The PC-feature model is generic and can be used with different applications. It allows developers to easily select a specific configuration of platform/environment on which an application can be run. Furthermore, the PC-feature model gives developers the opportunity to add extra PC-features related to a specific application and to include more detailed ones without affecting the proposed model transformation approach. Figure 6.1 illustrates an example of PC-feature model.

Adding security solutions requires more resources and longer execution times, which in turn has a significant impact on system performance. We introduce a PC-feature

group called *SecureCommunication* that contains two alternative features *Secured* and *Unsecured*. The *Secured* feature offers two security protocols: Secure Socket Layer (SSL) and Transport Layer Security (TLS) that can be augmented to the applications. Furthermore, we introduce three security level alternatives depending on the size of the key used in the handshake phase and on the strength of the encryption and message digest algorithms used in the data transfer phase, as proposed in [MEN04]. Each security level requires different overheads for sending and receiving secure messages. These overheads are mapped to the communication overheads through the attributes *commRcvOvh* and *commTxOvh*, which represent the host demand overheads for receiving and sending messages, respectively. Since not all the messages exchanged in a product need to have the same communication overheads, we propose to annotate each individual message stereotyped as *«PaCommStep»* with the processing overheads for the respective message: *commTxOvh* for transmitting (sending) it and *commRcvOvh* for receiving it. In fact, these overheads correspond to the logical communication channel that conveys the respective message. Eventually, the logical channel will be allocated to a physical communication channel (e.g., network or bus) and to two execution hosts, the sender and the receiver. The *commTxOvh* overhead will be eventually added in the performance model to the execution demands of the sender host and *commRcvOvh* to that of the receiver host.

Each type of physical communication channel stereotyped *«GaCommHost»* has different capacity for the amount of information that can be transmitted over it. As the channel's capacity increases, the latency time for transmitting data over this channel decreases. Our example provides three different communication channels with three alternative connections for the Internet. The capacity and latency for each physical

channel type are respectively mapped to the attributes *capacity* and *blockT* of the stereotype «GaCommHost» stereotyping each communication node in the deployment diagram.

Data compression requires extra operations that increase the processing time, but at the same time compression helps reducing the use of resources, such as hard disk space or communication channel bandwidth. Data compression/decompression is adding an overhead when sending and receiving a message, which is mapped to the attributes *commTxOvh* and *commRcvOvh*, respectively. However, compression also reduces the amount of data to be transferred and thus decreases the delivery time (e.g., a compression algorithm may reduce the size of data to 60% [HAP10]). Thus, the amount of compressed data transmitted over a physical channel is mapped to the attribute *msgSize* of the stereotype *«PaCommStep»* annotating each communication step in the sequence diagram.

Another communication mechanism that affects the delivery time of a message due to the access to additional resources is whether the communication is with or without guaranteed delivery (e.g., the delivery time with guaranteed delivery increased by 25% [HAP10]); the effect is mapped to the *commTxOvh* attribute.

The cache mechanism has a significant impact on the performance of real time application. Caching a fraction of data in the main memory of the application server reduces the access to the back-end database server. Thus, it reduces the I/O activity to the database server (e.g., caching may reduce the response time especially at high loads to three times lower and may increase the throughput by 56% [MEN07]). The probability of the step for retrieving the data from the database when a cache is missed is mapped to the

attribute *prob* of the stereotype *«PaStep»* attached to the steps that perform database access.

The PC-feature group *PlatformChoice* includes different types of middleware such as CORBA, Web-services, etc., which will affect also the communication overheads. We may either map their effect to the *commTxOvh* and *commRcvOvh* attributes, or may use MARTE external operations described below. Another communication overhead caused by different types of communication protocols such as SOAP and RMI are presented in the PC-feature model. Each realization has a different impact on performance (e.g., the overhead caused by SOAP protocol is larger than the RMI [HAP10]).

Another solution for representing platform resources in the performance models is by using the so-called external operations. MARTE provides the concept of "external operation calls" to represent resource operations that are not explicitly modeled within the UML design model, but can be added in the performance model. The stereotype *«PaStep»* has two attributes: a) *extOpDemands*, an ordered set of identifiers for operations by external services which are demanded by this Step, in a form understood by the performance environment, and b) *extOpCount*, an ordered set of number of requests made for each external operation during one execution of the Step, listed in the same order as the demands. Examples of such external calls are middleware operations or disk operations hidden in database calls. Different types of external devices are represented in the PC-feature model by the feature *ExternalDeviceType*. Each device offers different operations with different execution times. The invocation of such operations is represented by the attributes *extOpDemands* and *extOpCount* of an execution step.

It is important to note that the MARTE annotations contain both performance-affecting attributes of the product we want to analyze, as well as environment/platform characteristics. For instance, the CPU execution times of different scenario steps are indicated by the attributes *hostDemand* of *«PaStep»*. The size of a message from a sequence diagram represented by the attribute *msgSize* is a property of the software product, which may be modified by properties of the communication channel, such as compression/decompression. The product model obtained by the proposed transformation approach includes both the performance attribute contained directly in the design model and the platform/environment factors corresponding to PC-features.

In order to automate the whole process of generating a performance model for a specific product, the mapping between the PC-features and the performance attributes they affect needs to be specified. Thus, each stereotyped class representing a feature in the PC-feature model has a tagged value indicating the list of the attributes it affects. For instance, the PC-feature *DataCompression* affects the attribute list {*msgSize*, *commTxOvh*, *commRcvOvh*}. Another tagged value is defined to include guidance information associated with the PC-feature. The next chapter explains how we automate this mapping during the process of generating a user-friendly representation of the generic MARTE parameters that need to be bound to concrete values.

It is worth mentioning that the PC-feature model helps in addressing the problem of software to hardware allocation arising when an existing product runs on a new platform or uses a different type of communication realization. In chapter 8, we explain how this allocation problem can be propagated to a performance model where software developers can easily assess the impact of using a certain type of middleware on the

system performance and can be able to detect possible performance problems as early as possible in the development process.

Chapter 7: Handling Performance Parameters

This chapter explains the automatic generation of a user-friendly representation with generic parameters and guiding information. The mapping between the PC-features and the performance attributes they affect is illustrated. The actual binding realized by model transformation is described as well.

In SPL, different members may vary from each other in terms of their functional requirements, quality attributes, platform choices, network connections, physical configurations, and middleware. Many details contained in the system that are not part of its design model but of the underlying platforms and environment, do affect the run-time performance and need to be represented in the performance model. Performance completions, as proposed by Woodside [WOO02], provide a general concept to include low-level details of execution environment/platform in performance models. The variability space of the performance completions is represented through the Performance Completion feature model (PC-feature model) described in the previous chapter.

7.1 Requirements and Proposed Solution

In general, a SPL model contains a very large number of performance annotation parameters. For each generated product model, these parameters need to be bound to concrete values provided by the performance analyst that will be used for performance

analysis. A simple way to perform the binding is to ask the user to provide a list of parameter-value couples. As such a manual operation is user-unfriendly and error-prone, we would like to provide semi-automated support according to the following requirements:

a) Avoid asking the user to inspect all the model diagrams and to extract manually the parameters that need to be bound.

b) Avoid expecting the user to do the mapping between a PC-feature and the model element it affects.

c) Provide valid choices for the allocation of software to hardware.

d) Generate spreadsheets with the data that can be extracted automatically (such as the parameters that need to be bound). Choose a presentation format that facilitates the manipulation of data (search, filter, sort, etc.).

e) Avoid as much as possible to generate a large number of rows in the spreadsheets, which depends on the number of annotated model elements, the average number of parameter per model element and the number of PC-features.

f) Provide a flexible mechanism for guidance to the user related to the range of values for each PC-feature. Such guiding information can be edited for every application by the performance analyst.

g) Provide sufficient information to facilitate the binding process.

The following solution is proposed to address each of the previous requirements.

Requirements (a) and (b) are satisfied by proposing to automate the process of collecting all generic parameters that need to be bound to concrete values from the

product model and to associate each PC-feature from the PC-feature model to the model element(s) it may affect.

Requirement (c) is resolved by automating the collection of all the hardware resources from the deployment diagram provided by the user and associating their list to each lifeline in the spreadsheets. Then, the user decides on the actual allocation by choosing a processor from this list.

In order to satisfy requirement (d), we choose to present the data to the user in a spreadsheet format that can be opened by Microsoft Excel. We generate a spreadsheet that can be sorted, filtered, searched per UML diagram.

Requirement (e): in order to minimize the number of rows and consequently the number of questions to the user, we propose to navigate the UML model when possible to answer some of these questions. For instance, rather than asking the user to specify for each message its physical communication channel, its sender, and its receiver, we navigate the sequence diagram to get the sender and receiver lifelines for each message. From the actual nodes that deployed these lifelines, we can get the communication node that conveyed the respective message in the deployment diagram.

Requirement (f) is answered by associating each PC-feature in the PC-feature model with guiding information that helps the user in providing concrete binding values. In the generated spreadsheets, each PC-feature is automatically mapped to the corresponding guidance. The guidelines can be adjusted by the performance analyst for a given SPL and a known execution environment.

Requirement (g) is satisfied by providing the user with the information needed to make the binding decision. For example, each message is associated with the name of the *Interaction* contained it and the name of the sender and receiver lifelines.

The following section addresses the proposed solution in more details.

7.2 Generating User-Friendly Representation

The generic parameters of a product model derived from the SPL model are related to different kind of information:

a) Product-specific resource demands such as execution times, number of repetitions and probabilities of different steps.

b) Software-to-hardware allocation such as component instances to processors.

c) Platform/environment-specific performance details (also called performance completions).

The user (i.e., performance analyst) needs to provide concrete values for all generic parameters; this will transform the platform-independent product model into a platform-specific model describing the run-time behaviour of the product for a specific run-time environment.

Choosing concrete values to be assigned to the generic performance parameters of type (a) is not a simple problem. In general, it is difficult to estimate quantitative resource demands for each step in the design phase, when an implementation does not exist and cannot be measured yet. Several approaches are used by performance analysts to come up with reasonable estimates in the early design stages: expert experience with previous versions or with similar software, understanding the complexity of the algorithms,

measurements of reused software, measurements of existing libraries, or using time budgets. As the project advances, early estimates can be replaced with measured values for the most critical parts. Therefore, it is helpful for the user of our approach to keep a clearly organized record for the concrete values used for binding in different stages of the project. For this reason, we proposed to automate the collection of the generic parameters from the model on spreadsheets that will be provided to the user.

The parameters of type (b) are related to the allocation of software components to processors available for the application. The user provides a deployment diagram that represents the actual hardware platform of the given product. The software components of that product need to be allocated to the actual processing nodes in that deployment diagram. The MARTE stereotype *«RunTInstance»* annotating a lifeline in a sequence diagram provides an explicit connection between a role in the behaviour model and the corresponding run-time instance of a component. The attribute *host* of this stereotype indicates on which physical node from the deployment diagram the instance is running. Using parameters for the attribute *host* enable us to allocate each role (a software component) to an actual hardware resource. The transformation collects all these hardware resources and associates their list to each lifeline in the spreadsheets. The user decides on the actual allocation by choosing a processor from this list.

The performance effects of variations in the platform/environment factors (such as network connections, middleware, operating system and platform choices) are included into our model by aggregating the overheads caused by each factor and by attaching them via MARTE annotations to the affected model elements. As already mentioned, the variations in platform/environment factors are represented in our

approach through the PC-feature model. A specific run-time instance of a product is configured by selecting a valid PC-feature combination from the PC-feature model. We define a PC-feature configuration as a complete set of choices of PC-features for a specific model element. For instance, a PC-feature configuration for a given message could be {*MediumSecurity*, *Compressed*, *CORBA*, *withoutGuaranteedDelivery*}.

It is interesting to note that a PC-feature has impact on a subset of model elements in the model, but not necessarily on all model elements of the same type. For instance, the PC-feature *SecuredCommunication* affects only certain communication channels in a product model, not all of them. Hence, a PC-feature needs to be associated to certain model element(s), not to the entire product. This mapping is set up through the MARTE performance specifications annotating the affected model elements in the product model, as described in previous chapter.

Dealing manually with a huge number of performance annotations by asking the developer to inspect every diagram in the generated product model, to extract the generic parameters and to match them with the PC-features is an error-prone process. As already mentioned, we propose to automate the process of collecting all generic parameters that need to be bound to concrete values from the product model and to associate each PC-feature to the model element(s) it may affect, then present the information to the developer in a user-friendly format. We generate a spreadsheet per diagram, indicating for each generic parameter some guiding information that helps the user in providing concrete binding values.

An example of such guiding information is the different overheads for sending and receiving secure messages. For instance, in [MEN04] overhead data is provided for

three security levels (low, medium and high): the handshake overhead is (10.2ms, 23.8 ms, 48.0 ms), and the data transfer overhead per KB of data is (0.104 ms, 0.268 ms, 0.609 ms). For instance, we used this data as guiding information in Figure 7.2. For instance, the overhead for sending a message with low security level is (5.1+0.052*msgsize) and for receiving is (5.1+0.052*msgsize). For a given SPL, the performance analyst may tailor the guiding information to the platform and environment intended for performance analysis. In general, the guidelines can be adjusted by the performance analyst for a given SPL and a known execution environment.

The process of generating the spreadsheets takes place after a given product PIM is derived from the SPL model. Due to the large semantic gap between the source and target models of the transformation, we follow the example "Microsoft Office Excel Extractor" [BRU05] from the Eclipse/ATL website, which applies the principle of separation of concerns and breaks the transformation into a series of simpler transformations. This transformation series enables us to get some control over the order in which to navigate the ATL source models. The process is composed of four different model transformations:

a) From a specific UML product model into a Table model that contains several tables, one for each sequence diagram; each parametric performance annotation is represented as a table row.

b) From the Table model into a SpreadsheetMLSimplified model that represents (as the name says) a simplified subset of the spreadsheetML XML used by Microsoft to import/export Excel workbook's data from/to XML.

c) From the SpreadsheetMLSimplified into an XML model.

d) The XML model created in the previous step is re-written as an XML file which can be directly opened by Microsoft Excel.

The mapping between PC-features and the corresponding performance attributes takes place during the transformation (a). Each MARTE attribute gets the name of the PC-features that have an impact on this attribute attached to it. For instance, the attribute *msgSize* is associated with the PC-feature *DataCompressed*. Another association is between the MARTE attribute *host* annotating a model element of type *lifeline* and the list of all available deployment nodes from the deployment diagram.

The transformation handles differently the context analysis parameters, which are usually defined by the modeler to be carried without binding throughout the entire transformation process, from the SPL model to the performance model for a product. These parameters can be used to explore the performance analysis space and will be given concrete values during performance analysis. A list of the context analysis parameters are provided to the user, who will decide whether to bind them now to concrete values, or to use them unbound in MARTE expressions.

The four transformations are implemented in the Atlas Transformation Language (ATL) [ATL]. The rules of the first transformation handle the generation of the Table model from the UML product model. A few examples of helpers and rules of this transformation are given bellow, with extensive comments in natural language.

```
-- Rule Interaction2Table transforms each SD in UML model to a table in Table model
rule Interaction2Table {
     from
          interaction : UML!Interaction
           (interaction.hasStereotype('GaAnalysisContext'))

-- Define the headers' names
```

```
        using {
          titles_name : Sequence(String) =
            Sequence{'Element_Type','Stereotype_Name',
                     'Attribute_Name','Element_Name',
                    'PC-feature_Name','Guideline for Value',
                    'Generic_Parameter', 'Concrete_Value'};
                }
        to
              table : Table!Table(
              name <- interaction.name,
              rows <- Sequence{title_row, blank_row,

-- Create a row for each attribute
                          Sequence{UML!Message.allInstances()->
                             collect(e |thisModule.resolveTemp
                                       (e,'hostDemand_row'))},
                          Sequence{UML!Message.allInstances()->
                             collect(e | thisModule.resolveTemp
                                       (e,'msgSize_row')) },
                          Sequence{UML!Message.allInstances()->
                             collect(e | thisModule.resolveTemp
                                       (e,'commTxOvh_row')) },
                          Sequence{UML!Message.allInstances()->
                             collect(e | thisModule.resolveTemp
                                      (e,'commRcvOvh_row'))}}),

-- Create the title row
                    title_row : Table!Row(
                          cells <- Sequence{ title_cols }),
                    title_cols : distinct Table!Cell
                          foreach(name in titles_name)
                          (content <- name),

-- Create a blank row
                    blank_row : Table!Row(
                            cells <- Sequence{ blank_cols }),
                    blank_cols : Table!Cell(
                          content <- ''  )}

-- Rule Message2Rows collects all the generic tagged values of the stereotypes
-- «PaStep» and «PaCommStep» extending model elements of type message
-- and transforms them to a row in a table
rule Message2Rows {
     from
          message : UML!Message
            using {  hostDemand_name : Sequence(String) =
                  Sequence{'Message','PaStep','hostDemand',
                    message.name,'Application-Annotation',
                 message.getTagValues('PaStep','hostDemand')};
```

```
                msgSize_name : Sequence(String) =
                        Sequence{'Message','PaCommStep',
                                'msgSize', message.name,
```

-- Call helper "pcFeatureName" to get the PC-feature affects attribute msgSize

```
                message.pcFeatureName('PaCommStep','msgSize'),
```

-- Call helper "getTagValues" to get the generic attribute value

```
                message.getTagValues('PaCommStep','msgSize')};
               commTxOvh_name : Sequence(String) =
               Sequence{'Message','PaCommStep','commTxOvh',
               message.name,
              message.pcFeatureName('PaCommStep','commTxOvh'),
              message.getTagValues('PaCommStep','commTxOvh')};
                  commRcvOvh_name : Sequence(String) =
              Sequence{'Message','PaCommStep','commRcvOvh',
                             message.name},
            message.pcFeatureName('PaCommStep','commRcvOvh'),
            message.getTagValues
                             ('PaCommStep','commRcvOvh')};}
    to
                hostDemand_row : Table!Row(
                     cells <- Sequence{hostDemand_cols}),
               hostDemand_cols : distinct Table!Cell
                    foreach(name in hostDemand_name)
                    content <- name),
              msgSize_row : Table!Row(
                    cells <- Sequence{ msgSize_cols }),
               msgSize_cols : distinct Table!Cell
                    foreach(name in msgSize_name) (
                    content <- name),
              commTxOvh_row : Table!Row(
                    cells <- Sequence{ commTxOvh_cols}),
               commTxOvh_cols : distinct Table!Cell
                    foreach(name in commTxOvh_name) (
                    content <- name),
               commRcvOvh_row : Table!Row(
                    cells <- Sequence{ commRcvOvh_cols}),
               commRcvOvh_cols : distinct Table!Cell
                    foreach(name in commRcvOvh_name) (
                    content <- name)}
```

-- This helper returns the tagged value of the stereotype's attribute both stereotype
-- and attribute name are given as parameters

```
helper context UML!Element def :
               getTagValues(stereotype:String,tag:String) :
               UML!Element =
    if self.getAppliedStereotypes()->
          select(e | e.name =stereotype)->notEmpty()
```

```
        then
             self.getValue(self.getAppliedStereotypes()
             ->select(e|e.name=stereotype )->first(),tag)
             ->first()
        else ''
    endif;
```

```
-- This helper returns the PC-feature name affecting the respective attribute; both
-- stereotype and attribute name are given as parameters
helper context UML!Element def :
        pcFeatureName(stereotype:String, name:String):
        String =
    if   self.getAppliedStereotypes() ->
          select(e | e.name = stereotype)->notEmpty()
        then
             UML!Class.allInstances() ->
             select(class|class.getTagValues
             ('pc-feature','AttList')=name) ->
             collect(c|c.name)->first()
        else ''
    endif;
```

The generated spreadsheet presents a user-friendly format for the users of the transformation who have to provide appropriate concrete values for binding the generic performance annotations. Being automatically generated, they capture all the parameters that need to be bound and reduce the incidence of errors.

7.3 User-Interaction

The process of binding the generic parameters of a product model is a decision-making process that requires humans to select a specific PC-feature configuration for each model element and - based on this selection - to enter a concrete value for each parameter. The user also has to provide the actual hardware platform for the given product.

The scenario *DeliveryPurchaseOrder* for a specific product is shown in Figure 7.1. A part of the generated spreadsheet for that scenario *DeliveryPurchaseOrder* is

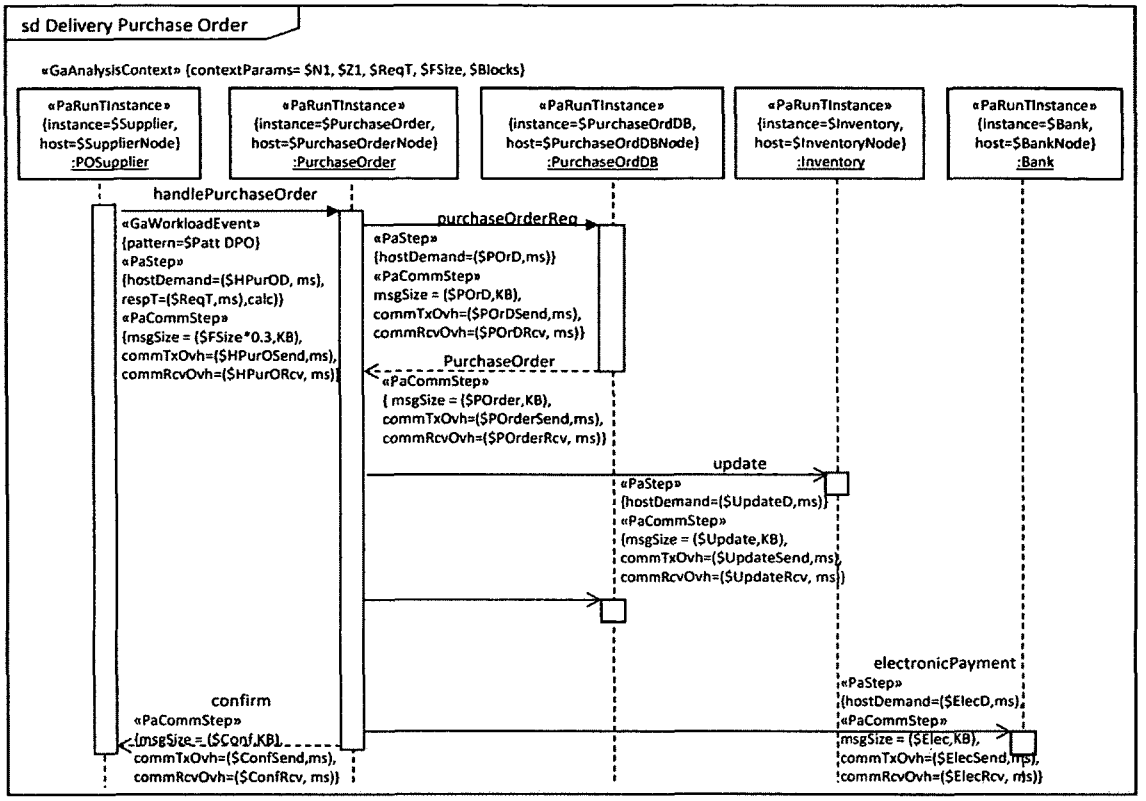
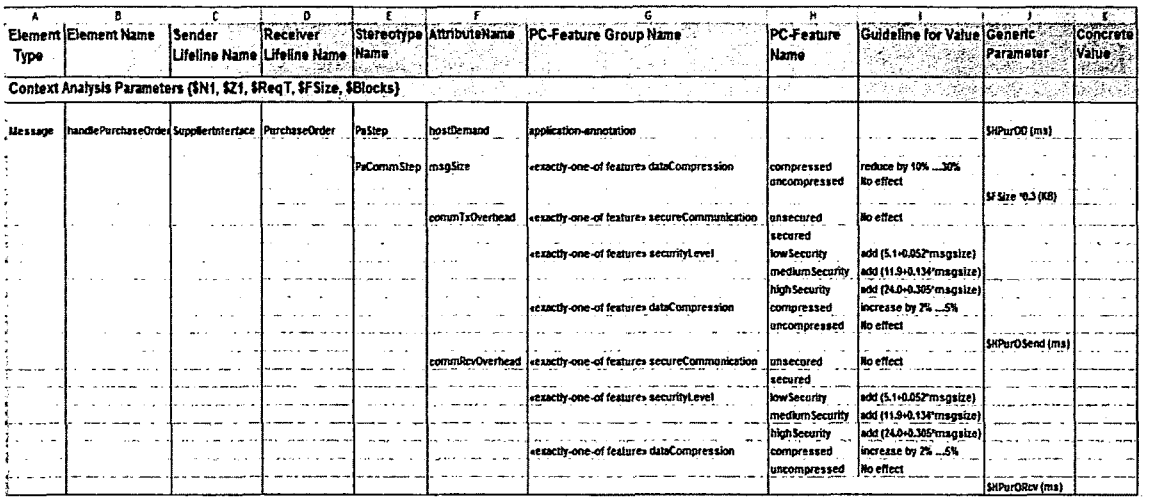


Figure 7.1: Scenario Delivery Purchase Order for a specific product.

shown in Figure 7.2. The row below the title row indicates a list of all the context analysis parameters. The user will decide whether to bind them now to concrete values, or to use them unbound in MARTE expressions. The column titled *Concrete Value* is designated for the user to enter appropriate concrete value for each generic parameter, while the column *Guideline for Value* provides a typical range of values to guide the user.

For instance, the attribute *hostDemand* is a type of product-specific resource demands. For this type of generic performance parameter, the user needs to estimate a quantitative resource demand and enters a concrete value.

The PC-feature *DataCompression* is mapped to the MARTE attribute *msgSize* annotating a model element of type *message*. The user has to choose whether the data



A	B	C	D	E	F	G	H	I	J	K
Element Type	Element Name	Sender Lifeline Name	Receiver Lifeline Name	Stereotype Name	AttributeName	PC-Feature Group Name	PC-Feature Name	Guideline for Value	Generic Parameter	Concrete Value
Context Analysis Parameters {$N1, $Z1, $ReqT, $FSize, $Blocks}										
Message	handlePurchaseOrder	SupplierInterface	PurchaseOrder	PaStep	hostDemand	application-annotation			$HPurOO (ms)	
				PaCommStep	msgSize	«exactly-one-of feature» dataCompression	compressed	reduce by 10% ...30%		
							uncompressed	No effect	$FSize *0.3 (KB)	
					commTxOverhead	«exactly-one-of feature» secureCommunication	unsecured	No effect		
						«exactly-one-of feature» securityLevel	secured low Security	add (5.1+0.052*msgsize)		
							medium Security	add (11.9+0.134*msgsize)		
							high Security	add (24.0+0.305*msgsize)		
						«exactly-one-of feature» dataCompression	compressed	increase by 2% ...5%		
							uncompressed	No effect		
					commRcvOverhead	«exactly-one-of feature» secureCommunication	unsecured	No effect	$HPurOSend (ms)	
						«exactly-one-of feature» securityLevel	secured low Security	add (5.1+0.052*msgsize)		
							medium Security	add (11.9+0.134*msgsize)		
							high Security	add (24.0+0.305*msgsize)		
						«exactly-one-of feature» dataCompression	compressed	increase by 2% ...5%		
							uncompressed	No effect		
									$HPurORcv (ms)	

Figure 7.2: Part of the generated Spreadsheet for the scenario Delivery Purchase Order.

transmitted over this channel is compressed or uncompressed. Assuming that the choice is "compressed", then the concrete value for the attribute *msgSize* is reduced by 10% to 30%.

As the value of the attribute *msgSize* is an expression *$FSize*0.2*$GetL* in function of the context analysis parameter *$FSize*, it is the user's choice to bind it at this level or keep it as a parameter in the output it produces.

The PC-features *SecureCommunication*, *SecurityLevel*, and *DataCompression* are mapped to the MARTE attributes *commTxOvh* and *commRcvOvh* annotating a model element of type *message*. For instance, if the PC-selection features chosen by the user are "secured" with "low security level", the concrete value entered by the user is obtained by evaluating the expression (5.1+0.052*msgSize), assuming that the user follows the provided guideline. Since the choice for the PC-feature *DataCompression* is "compressed", the user may decide to increase by 4% the existing overhead due to compression feature.

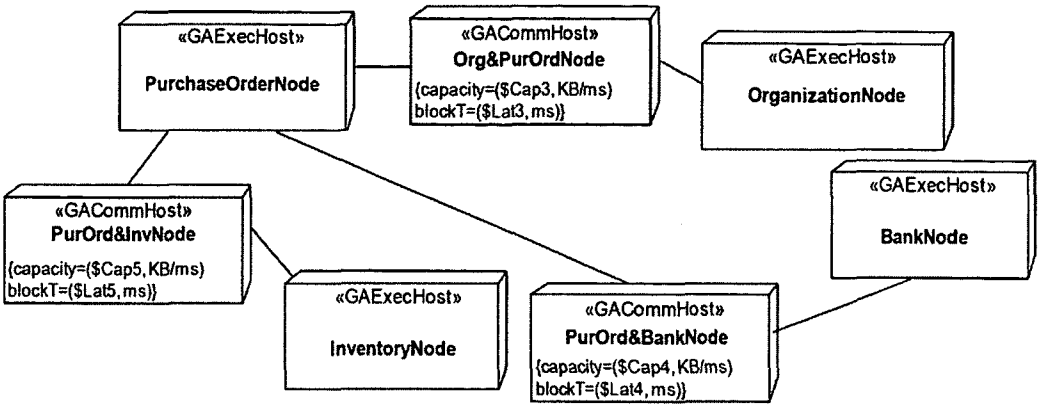


Figure 7.3: Part of the product deployment diagram.

After the user selects a PC-feature combination for each model element, he/she can delete the remaining unselected PC-features from the spreadsheet, ending up with a small set of rows containing annotations that need to be bound to concrete values.

Another type of generic performance parameters that need to be bound to concrete values are related to software-to-hardware allocation. Figure 7.3 shows a part of the product deployment diagram provided by the user to be used for the scenario *DeliveryPurchaseOrder*. Figure 7.4 shows another part of the generated spreadsheet for the scenario *DeliveryPurchaseOrder* that addresses this type. The column title *List of Nodes* provides a list of all the available hardware nodes.

For instance, the attribute *host* of the stereotype *«PaRunTInstance»* annotating a lifeline in a sequence diagram needs to be bound to a concrete physical node from the deployment diagram. The user has to choose a processor from the given list.

7.4 Performing the Actual Binding

After the user selects an actual processor for each lifeline role provided in the spreadsheets and enters concrete values for all the generic performance parameters, the

A	B	C	D	E	F	G
Element Type	Element Name	Stereotype Name	AttributeName	List of Nodes	Generic Parameter	Concrete Value
Lifeline	SupplierInterface	PaRunTInstance	host	PurchaseOrderNode	$SupplierNode	
				OrganizationNode		
				BankNode		
				InventoryNode		
				CustomerNode		
				RequisitionNod		
				OperationFundNode		
				ContractsNode		
Lifeline	Inventory	PaRunTInstance	host	PurchaseOrderNode	$InventoryNode	
				OrganizationNode		
				BankNode		
				InventoryNode		
				CustomerNode		
				RequisitionNod		
				OperationFundNode		
				ContractsNode		

Figure 7.4: Another part of the generated Spreadsheet for the scenario Delivery Purchase Order.

next model transformation takes as input these generated spreadsheets along with its corresponding product PIM, and binds all the generic parameters to the actual values provided by the user. The outcome of the transformation is a product PSM with concrete performance annotations, which can be further transformed into a performance model.

In order to automate the actual binding process, the generated spreadsheets with concrete values are given as a mark model to the binding transformation. The mark model concept has been introduced in the OMG MDA guide [OMG04] as a means of providing concrete parameter values to a transformation, which makes the transformation generic and more reusable in different contexts.

To consider the spreadsheets as a mark model for the transformation, we apply the same principle of separation of concerns and break the transformation into a series of

simpler transformations as in [BRU05]. Three extra model transformations have to be done before performing the actual binding:

a) From the spreadsheets (XML file) to an XML model.

b) From XML model to the required syntax in Ecore-based format.

c) XML with required syntax is further extracted as an XML file that can be accepted by ATL.

The following rule is an example from the model transformation that transforms the XML model to the required syntax that can be accepted as a separate input by the ATL transformation program.

```
-- Rule Data stores the word 'name' in the attribute name whenever the helper
-- isName() returns True, otherwise stores the word 'value'. The value of a string data
-- is stored in the value attribute by calling the helper getStringDataValue()
rule Data {
      from
            esd : Ecore!Element (esd.name='Data' and
                  esd.getStringAttrValue('ss:Type')='String')
      to
            data : Ecore!Attribute (
            parent <-Sequence{esd.parent.parent}->collect(e |
                  thisModule.resolveTemp(e,'row'))->first(),
            name <- if esd.isName()
                        then 'name'
                        else 'value'
                     endif,
            value <- esd.getStringDataValue())}

-- This helper returns the value of a string attribute. It returns an empty string if the
-- attribute doesn't exist. The attribute name is given as parameter
helper context Ecore!Element def :
            getStringAttrValue(attrName : String) : String =
      let   eltC : Sequence(Ecore!Attribute) = self.children
                  ->select(a|a.oclIsTypeOf(Ecore!Attribute)
                        and a.name = attrName)->asSequence()
      in
            if eltC -> notEmpty()
                  then
                        eltC -> first().value
                  else
```

```
                    ''
        endif;
```

-- This helper returns the value of a string data. It returns an empty string if the
-- value doesn't exist.

```
helper context Ecore!Element def :
                              getStringDataValue() : String =
    let eltC : Sequence(Ecore!Text) = self.children->
      select(d|d.oclIsTypeOf(Ecore!Text))->asSequence()
    in
        if eltC -> notEmpty()
              then
                  eltC -> iterate(txt; res : String = '' |
                           res.concat(txt.value))
              else
                    ''
        endif;
```

-- This helper returns the value of the startLine

```
helper context Ecore!Element def:
                               getStartLineValue():String =
    let  eltC : Sequence(Ecore!Text) = self.children ->
      select(d | d.oclIsTypeOf(Ecore!Text))->asSequence()
    in
        if eltC -> notEmpty()
              then
                   eltC -> first().startLine.toString()
              else
                    ''
        endif;
```

-- This helper returns *TRUE* whenever the respective element has a startLine value
-- equal the startLine value of the parent of its parent +1

```
helper context Ecore!Element def: isName() : Boolean =
    if self.getStartLineValue().toInteger()=
      self.parent.parent.getStartLineValue().toInteger()+1
        then
              true
        else
              false
    endif;
```

The main transformation to perform the actual binding takes place now, after the mark model is ready to be injected into the model transformation as an XML file with the required syntax. A few examples of helpers of this transformation are given bellow. As

an example, the following helper is called by different rules to get the value of an attribute.

```
-- This helper returns the value of the attribute 'value' and gets as a parameter the value
-- of the attribute 'name' by checking all elements in mark model 'parameters'
helper def : getParameter (variable : String) : String =
            XML!Element.allInstancesFrom
            ('parameters')->select(m|m.name = 'Row')->
            select(a|a.getStringAttrValue('name')= variable)
            ->first().getStringAttrValue('value');

-- This helper is called by the previous one to return the value of a string attribute.
-- It returns an empty string if the attribute doesn't exist.
helper context XML!Element def: getStringAttrValue(attrName :
String) : String =
      let attX :Sequence(XML!Attribute)= self.children
         ->select(a|a.oclIsTypeOf(XML!Attribute) and a.name=
                                    attrName)->asSequence()
      in
            if  attX -> notEmpty()
                  then attX ->first().value
                  else ''
            endif;
```

The process of assessing the impact of using a certain type of middleware/platform on the performance of a given product is facilitated by the mapping between the PC-features and the model elements they affect. Each time, the developers want to evaluate the effects of a certain platform and/or a specific execution environment, a PC- feature configuration has to be selected from the PC-feature model during the generation of the spreadsheets. Then, a performance model is obtained for the chosen PC-feature configurations where possible performance problems can be detected as early as possible in the development process. In the next chapter, it is shown how different PC-feature configurations affect the corresponding LQN performance models.

Chapter 8: Performance Analysis

After a product PSM with concrete performance annotations is generated, it is further transformed into a LQN performance model using the PUMA transformation approach [WOO05] [WOO08]. This chapter presents some performance analysis experiments conducted with the LQN models obtained for different generated product models.

During the development process of a distributed real-time application such as an e-commerce system, developers need to assess the impact of different design alternatives and different platform/environment choices on the system performance. This will enable them to detect performance problems as early as possible in the development process.

Different questions are raised up that need to be answered. For instance, what is the performance price when adding security features to a communication channel in order to secure the data transmitted over it? Data allocation is another important issue, what are the performance and security impacts when using different architectures designs? In this chapter, a LQN model is manually derived from the generated product model and the impact on its performance for different architectures designs and levels of secure communication channels is analyzed by using the LQN model.

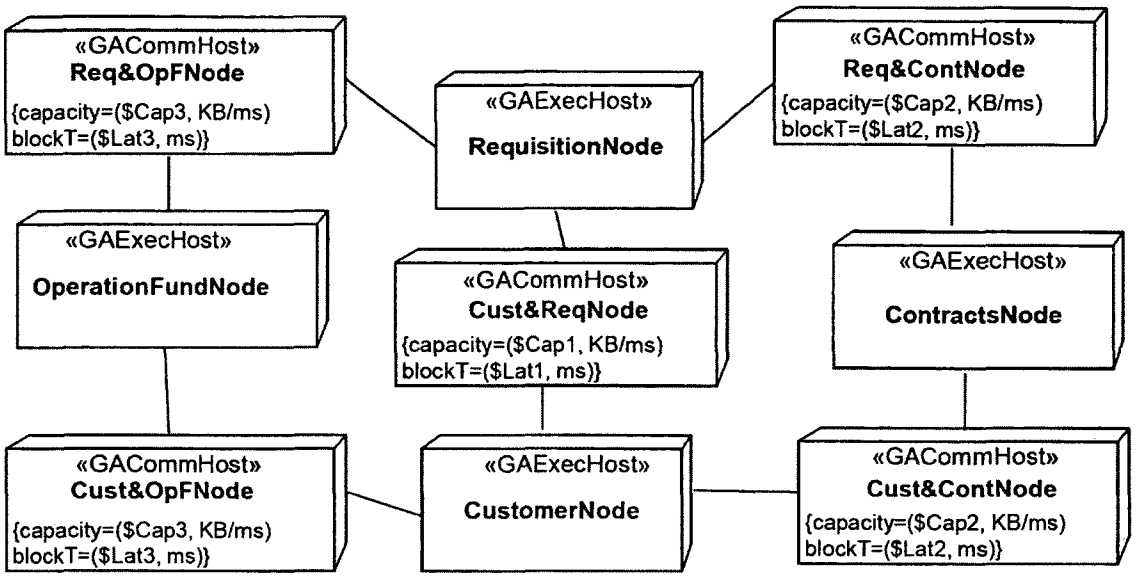


Figure 8.1: Part of the product deployment diagram.

8.1 Centralized versus Distributed Architecture

In the e-commerce application, it is important where the data is located in order to fulfill performance and security requirements. This location problem is examined in two different architectures: 1) distributed and 2) centralized.

In the centralized architecture, all customer data is contained in one database. For instance, in Figure 8.1 the customer database is located in one node *CustomerNode* for the centralized design. The centralized architecture has the advantage that updating and maintaining the data consistency is easier, but has the disadvantage of becoming the system bottleneck for large system sizes (when the number of customers and the amount of data go up). A distributed architecture represents a solution where several databases divide the data and the work among them. It has potential for faster response

A	B	C	D	E	F	G	H	I	J	K
Element Type	Element Name	Sender Lifeline Name	Receiver Lifeline Name	Stereotype Name	AttributeName	PC-Feature Group Name	PC-Feature Name	Guideline for Value	Generic Parameter	Concrete Value
Context Analysis Parameters {$N1, $Z1, $ReqT, $FSize, $Blocks}										
Message	requisitionRequest	CustomerInterface	Requisition	PaStep	hostDemand	application-annotation			$ReqR0 (ms)	18.4 ms
				PaCommStep	msgSize	«exactly-one-of feature» dataCompression	compressed	reduce by 10% ...30%		
							uncompressed	No effect	$Req *0.2 (KB)	377.6 KB
					commTxOverhead	«exactly-one-of feature» secureCommunication	unsecured	No effect		0.0 ms
							secured			
						«exactly-one-of feature» securityLevel	lowSecurity	add (5.1+0.052*msgsize)		
							mediumSecurity	add (11.9+0.134*msgsize)		
							highSecurity	add (24.0+0.305*msgsize)		
						«exactly-one-of feature» dataCompression	compressed	increase by 2% ...5%		
							uncompressed	No effect		0.0 ms
									$ReqSend (ms)	0.052 ms
					commRcvOverhead	«exactly-one-of feature» secureCommunication	unsecured	No effect		0.0 ms
							secured			
						«exactly-one-of feature» securityLevel	lowSecurity	add (5.1+0.052*msgsize)		
							mediumSecurity	add (11.9+0.134*msgsize)		
							highSecurity	add (24.0+0.305*msgsize)		
						«exactly-one-of feature» dataCompression	compressed	increase by 2% ...5%		
							uncompressed	No effect		0.0 ms
									$ReqRcv (ms)	0.052 ms

Figure 8.2: Part of the generated Spreadsheet for the scenario Create Requisition.

times and improved performance, but makes the updates and keeping data consistency more difficult.

To enable the derivation of a performance model, the user needs to choose a PC-feature configuration for each model element of a given product model. Figure 8.2 shows a part of the generated spreadsheet for the scenario *CreateRequisition* where the user bound the generic performance annotations to concrete values based on the chosen PC-feature configuration. The user also needs to provide the actual hardware platform for the given product as shown in Figure 8.1. In order to evaluate the performance impact of a specific system runs on two different architectures (centralized and distributed), the user provides the actual allocation of the software components for the two architecture

A	B	C	D	E	F	G
Element Type	Element Name	Stereotype Name	AttributeName	List of Nodes	Generic Parameter	Concrete Value
Lifeline	CustomerInterface	PaRunTInstance	host	PurchaseOrderNode OrganizationNode BankNode InventoryNode CustomerNode RequisitionNod OperationFundNode ContractsNode	$CustNode	CustomerNode
Lifeline	Requisition	PaRunTInstance	host	PurchaseOrderNode OrganizationNode BankNode InventoryNode CustomerNode RequisitionNod OperationFundNode ContractsNode	$ReqNode	RequisitionNode
Lifeline	CustomerDB	PaRunTInstance	host	PurchaseOrderNode OrganizationNode BankNode InventoryNode CustomerNode RequisitionNod OperationFundNode ContractsNode	$CustDBNode	CustomerNode

Figure 8.3.a: Part of the generated Spreadsheet for the scenario Create Requisition for Centralized design.

designs. Figures 8.3.a and 8.3.b show other parts of the generated spreadsheets for allocating the centralized and distributed architecture, respectively.

The key performance scenario *CreateRequisition* is transformed into two LQN models shown in Figures 8.4.a and 8.4.b to represent the two different architectures; centralized and distributed, respectively. In the centralized architecture, all customer database is allocated to the node *CustomerNode* while, in the distributed architecture, the customer information is distributed over the three nodes *RequisitionNode*,

A	B	C	D	E	F	G
Element Type	Element Name	Stereotype Name	AttributeName	List of Nodes	Generic Parameter	Concrete Value
Lifeline	RequisitionDB	PaRunTInstance	host	PurchaseOrderNode	$ReqDBNode	RequisitionNode
				OrganizationNode		
				BankNode		
				InventoryNode		
				CustomerNode		
				RequisitionNod		
				OperationFundNode		
				ContractsNode		
Lifeline	ContDB	PaRunTInstance	host	PurchaseOrderNode	$ContDBNode	ContractsNode
				OrganizationNode		
				BankNode		
				InventoryNode		
				CustomerNode		
				RequisitionNod		
				OperationFundNode		
				ContractsNode		
Lifeline	OpFundDB	PaRunTInstance	host	PurchaseOrderNode	$OpFBDNode	OperationFundNode
				OrganizationNode		
				BankNode		
				InventoryNode		
				CustomerNode		
				RequisitionNod		
				OperationFundNode		
				ContractsNode		

Figure 8.3.b: Part of the generated Spreadsheet for the scenario Create Requisition for Distributed design.

ContractsNode, and *OperationFundNode*. We solved the LQN models for different numbers of users and compared the two systems, which differ only in the choice of the feature *DataStorage*: *Centralized* or *Distributed*. The effects of the two architecture choices on the response time "R" are compared in Figure 8.5.

The LQN results show that the feature *DataStorage* has a considerable effect on performance as shown in Figure 8.5, as the response time for the centralized architecture is significantly higher than for the distributed architecture.

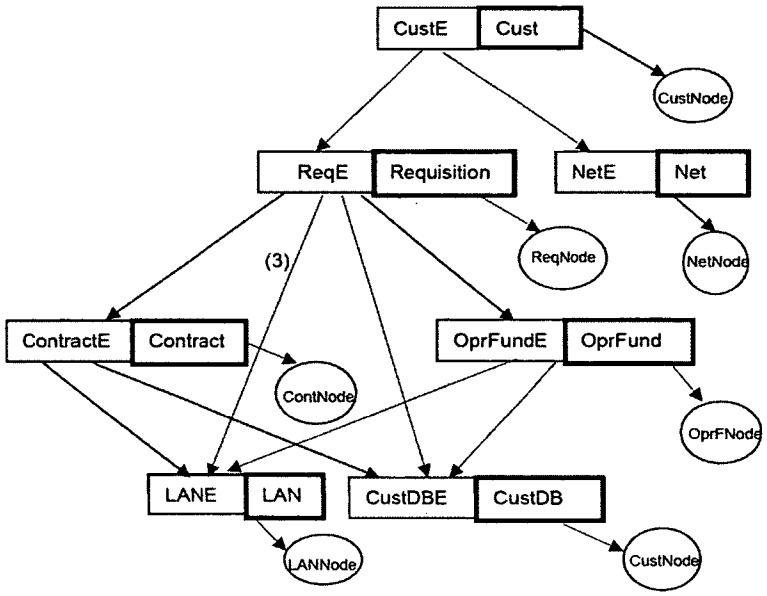


Figure 8.4.a: Centralized LQN model.

This brief example illustrates the potential for performance analysis in early development stages, by allowing the developers to compare the performance effects of different design alternatives.

8.2 Performance Effects of Security Feature

Web-based applications, such as an e-commerce system that contains sensitive data and has many customers, require securing the data transmitted over certain communication channels. However, adding security may include a performance price. System designers need to make choices between different security levels and to make security/performance trade-offs. In order to illustrate the impact on performance of a secure communication channel between the browser and the webServer, a performance analysis experiment

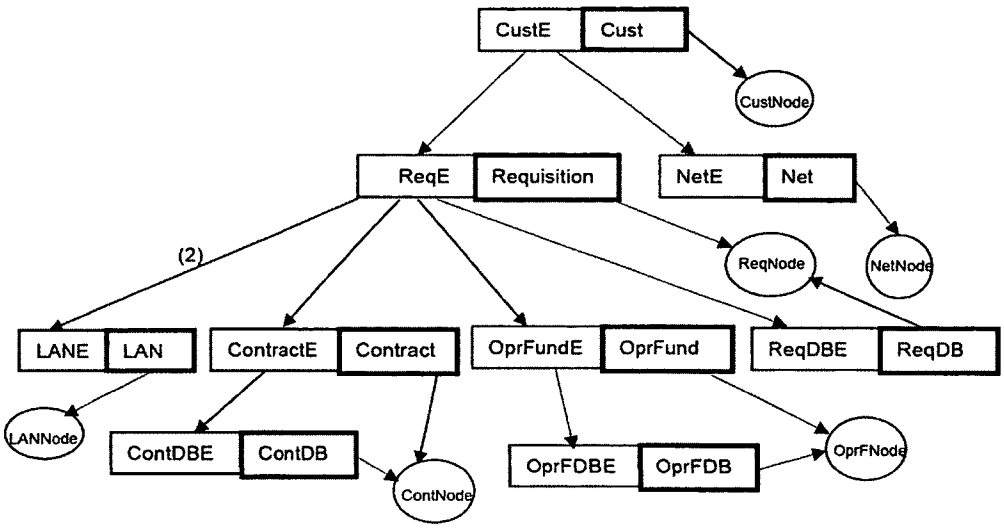


Figure 8.4.b: Distributed LQN model.

based on LQN models derived for a specific system with different security levels is presented in this section.

When a given system is generated, a specific configuration has to be selected from the PC-feature model. The key performance scenario *CheckCustomerAccount* shown in Figure 8.6 is transformed into the LQN models used for experiments shown in Figure 8.7. The following configuration was chosen. Each of the roles *CustumerAccount* and *Billing* is running on different nodes *CustAccNode* and *BillingNode*, respectively. Furthermore, these nodes are linked to the *CustNode* through a DSL Internet channel with 100 ms latency. A similar connection is set up between the *BillingNode* and the *AuthCenterNode*. The data is transmitted uncompressed with an average message size of 377.6 KB. The *CustumerAccount* database accesses an external device (hard disk) with an average read/write time of 77.1 ms.

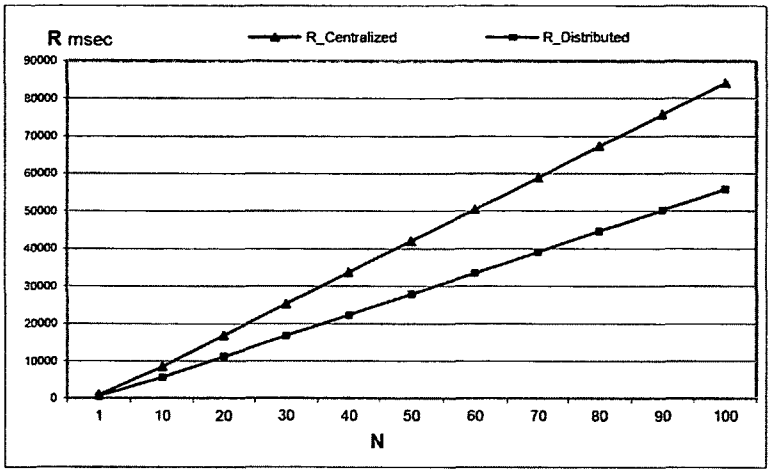


Figure 8.5: Response time in function of the number of users.

All communication channels in the unsecure system include no security solution, while the secure system contains certain secure channels using the TLS protocol. TLS has two phases: the handshake phase is used by the browser and webServer to exchange secrets and to generate a confidential symmetric key that is used for data exchange during data transfer, the second phase of the protocol. The public key encryption in the handshake phase may use keys of different lengths; a longer key provides a higher level of security, but the performance overhead increases. The strength of the symmetric encryption key and message digest algorithms used by technology to exchange data may also vary, using strong encryption and authentication algorithms providing higher security. These algorithms are computationally intensive and add different performance overheads to the system. We used the data provided in [MEN04] for three levels of security: the handshake overhead is of 10.2ms, 23.8 ms, 48.0 ms, and the data transfer overhead per KB of data is of 0.104 ms, 0.268 ms, 0.609 ms. The fourth case is for an unsecure system. The LQN performance model is analyzed for different numbers of users

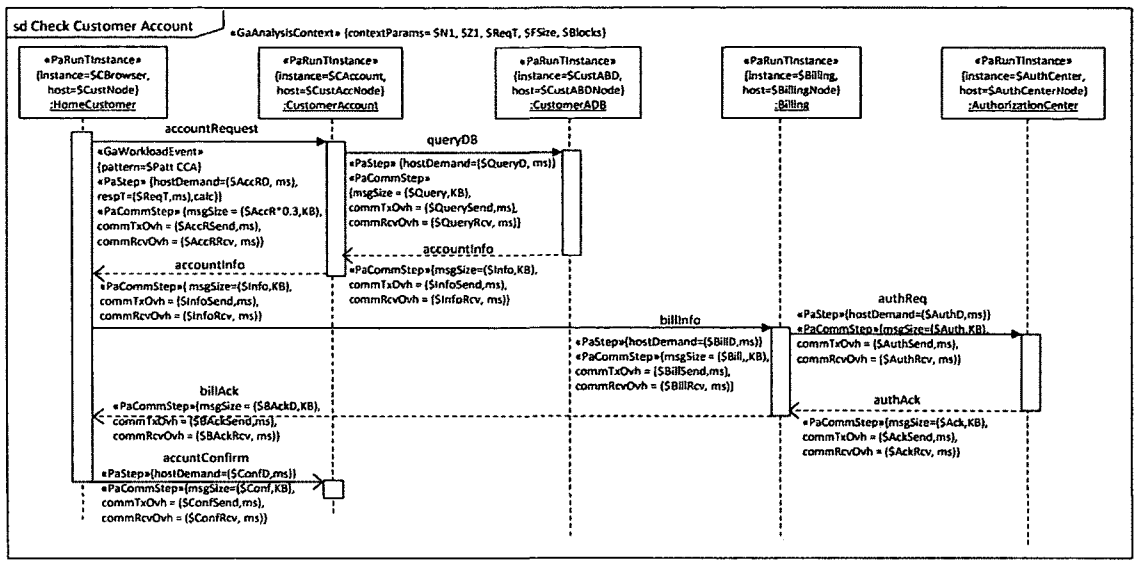


Figure 8.6: Scenario Check Customer Account for a specific product.

with an existing solver [FRA20]. Figure 8.8.a shows the response time of a user requesting an account for different system choices (unsecure system and three security levels), while Figure 8.8.b shows the throughput, both in function of the number of simultaneous users executing the same scenario.

The LQN results show that the secure system has a considerable effect on performance as shown in Figures 8.8.a and 8.8.b, as the response time for the secure system is much higher than for the unsecure system. As the number of users increases, the response time increases significantly due to the competition for resources. On the other hand, the throughput for the unsecure system is the highest and that for the most secure system the lowest. Throughput represents the number of requests completed per unit of time, so for low workloads the throughput increases with the number of users until the system approaches saturation and cannot serve more simultaneous requests regardless the number of users.

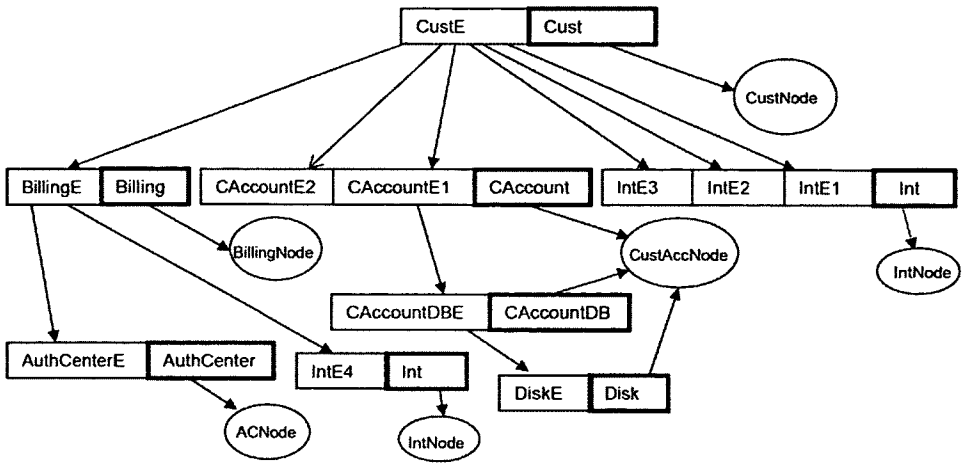


Figure 8.7: LQN model for Scenario Check Customer Account.

This brief example illustrates the potential for performance analysis in early development stages, by allowing developers to analyze trade-off between two non-functional requirements, performance and security, and to compare the impacts of different security levels on the system performance.

Another performance analysis experiment was conducted on different architectures (centralized and distributed) and with different levels of security. The results show that the response time for the centralized architecture is significantly higher than for the distributed architecture for all levels of security. This experiment allows developers to analyze trade-off between two non-functional requirements, performance and security, and at the same time to compare the impacts of different design alternatives on performance.

In general, a quantitative performance model helps the analyst to verify whether a system has the capacity to meet its performance requirements. It also helps identifying

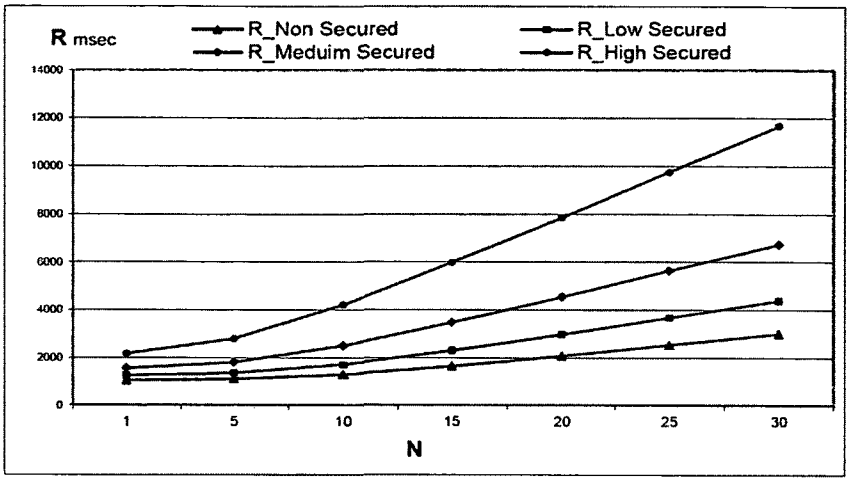


Figure 8.8.a: Response time in function of the number of users for unsecure system and three security levels.

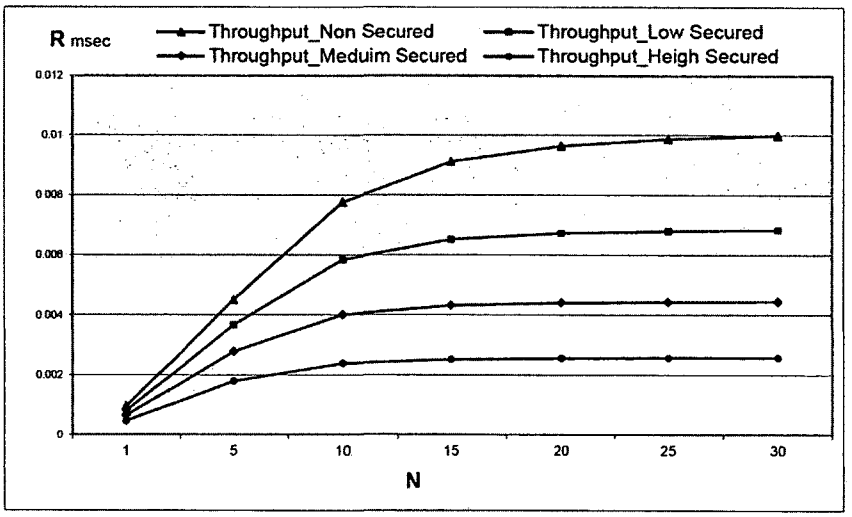


Figure 8.8.b: Throughput in function of the number of users for unsecure system and three security levels.

the performance "hot spots" (e.g., the resources that will saturate first) and provides guidance for design or configuration changes in order to solve or mitigate the problems.

Chapter 9: Verification and Validation

The proposed model transformation approach for deriving a given UML product model from a SPL deals with several models including feature model, feature configuration, SPL model, and the generated product model. In order to verify the transformation, the correctness of all these participating models has to be ensured [HEI09].

There are three levels of verification and validation:

- The feature model and feature configuration (verified as described below)
- The transformations implemented in ATL. (There is recent work in formal verification of model transformation, but cannot be applied yet to transformations as complex as ours, so we validated the implemented ATL transformation through test cases, as described below).
- The validation of the entire approach from the SPL model to the performance model (part of the larger project, so it is out of the research scope as shown in Figure 1.1).

Heidenreich [HEI09] proposes different verification approaches for checking the well-formedness of all participating models of a SPL. Well-formedness of a model means that the model conforms to its metamodel language and satisfies its multiplicities and constraints. Feature models are presented in [HEI09] as cardinality-based feature models (CBFM) where different verification constraints are proposed to ensure their correctness.

Examples of such constraints are: the root feature is mandatory, the cardinality of a child feature must conform to the cardinality of its parent feature, and referenced features must exist. The feature configuration which is a subset of the CBFM has to satisfy other constraints such as: a) when a child feature is selected, its parent feature has to be selected; b) if a feature is selected, all its mandatory child features must be selected too; c) if an alternative feature is chosen, at most one of its child features must be selected.

The feature model is represented in our approach as a dependency class diagram where the relationships between features are specified through several dependency constraints such as *requires*, *mutually includes*, *mutually exclusive*, and *more-than-one-required* as well as feature groups that impose other constraints on a group of related features such as *exactly-one-of* and *at-least-one-of*. Similar to [HEI09], we introduce a number of constraints that have to be satisfied by the feature model in our approach to ensure its correctness:

- No *kernel* feature requires an *optional* or *alternative* feature because the *kernel* feature has to be selected so it cannot depend on the selection of any *optional* or *alternative* feature.
- No *kernel* feature has a *mutually includes* relationship with other *optional* or *alternative* feature, as mention above because *kernel* feature cannot depend on the selection of any *optional* or *alternative* feature.
- The *more_than_one_required* constraint has to references more than one feature.
- The *requires* and *mutually exclusive* constraints do not contradict each other.
- A feature group has to contain more than one feature.

A feature model represents a set of all possible combinations of features for products of a family; a specific product is configured by selecting a valid feature combination from the feature model and producing a feature configuration based on the product's requirements. In order to ensure the validity of the feature configuration, we propose other constraints that have to be enforced in each feature configuration:

- The feature configuration must contain all the *kernel* features from the feature model.
- The constraints defined in feature groups have to be followed.
- Both features referenced by the *mutually includes* constraint have to be included.
- If a feature requires the selection of other features, they have to be selected too.
- If a selected feature excludes the selection of other features, the other features must not be included.
- If a feature references several features through *more-than-one-required* constraint, at least two of these referenced features have to be included in the feature configuration.

Verifying that all product models instantiated from a SPL model conform to the well-formedness rules of the language is not feasible in practice because of the large number of possible product models generated from a SPL.

Heidenreich [HEI09] distinguishes between three types of constraints that have to be enforced in a SPL model: a) the multiplicity of a model element must match with the multiplicities specified in its metamodel language; b) the generated model must conform to its modeling language's type; c) the generated model must conform to specific semantic constraints depicted by the modeling language.

Czarnecki et al. [CZA06] present different constraints that have to be enforced in a superimposition model of all variant products called model template that is similar to our SPL model. The paper explains how to interpret the OCL rules in a way to express the well-formedness constraints for a model template with respect to the UML metamodel. As an example, it illustrates how to ensure the multiplicity of a binary association. To our knowledge, these are the only works addressing the problem of verifying a template model for SPL.

In our approach, these well-formedness constraints are enforced during the product derivation process. Our approach ensures that the generated models satisfy the well-formedness constraints of the UML metamodel during their construction since the proposed mapping technique used during the model transformation process guarantees that each model element selected from the source model and copied to the target model conforms to the specifications of the UML metamodel. The interpretation of the feature mapping to model elements realizing the features is based on the navigation between annotated and non-annotated model elements with respect to their relationships as specified in the UML metamodel. For example, a binary association in the target model is guaranteed by construction to be connected to both of its classifiers. The association is copied to the target only if both of its classifiers are selected and copied to the target model; from these classifiers, we navigate to the association connected them in order to decide whether to copy it or not. Furthermore, the containment hierarchy of the target model is ensured during the automatic model transformation, since the implicit mapping of the contained elements is inferred from the selection of their containers.

Another example of well-formedness constraint enforced by construction is that the property of a class from the target model that connects it to an association end is copied to the target model if and only if the respective association is also copied. If the class is selected for the target but the association is not, then the class will not have a hanging property.

As mentioned before, our model transformation approach applies the concept known as positive variability [VOE07] where we start with the minimal core that presents common model elements in SPL and selectively add additional product-specific elements according to the selected features. We believe that we use the positive variability in a way that makes it less error-prone, since every time a new element is added to the target model, its consistency with the neighboring target element is verified.

9.1 Test Cases

The verification of our proposed model transformation is also done with the help of several test cases. Each test case was designed to cover a number of specific issues, as laid out in the checklist from Table 9.1 to Table 9.4. In order to verify that our approach derives a product model that contains all the common parts in the SPL model and only the selected *optional* and *alternative* elements according to the feature configuration of the given product, several test cases are developed for each type of diagram.

The approach is verified against its completeness, correctness and containment conformance. Completeness means that the derived product model contains all the selected model elements based on its feature configuration, there is no missing element. Correctness means that the model doesn't contain any unselected model elements.

Containment conformance means that the derived product model conforms to the containment hierarchy defined in the UML metamodel. For instance, a class contains its properties.

Verifying the derived product model for completeness is done by defining a test case for each type of model element. Checking the correctness of the derived model against the absence of the unselected model elements is inferred from these test cases by detecting any unselected element. Several test cases are designed where the simple ones invoke

Table 9.1: Test Cases for the root of the target model.

	Model	Package Import	Use Case	Actor	Class	Association	Collaboration	SendOperationEvent	ReceiveOperationEvent
TC #1	X								
TC #2		X							
TC #3			X						
TC #4				X					
TC #5					X				
TC #6						X			
TC #7							X		
TC #8								X	
TC #9									X

only one of the transformation rules described in chapter 5, while the complex test cases involve more than one rules.

Assuming that the proposed derivation approach is applied to the e-commerce case study to derive a specific product model consisting of the use case, class, and sequence diagrams from the e-commerce SPL model. Section 9.1.1 describes the test cases for the model itself. Section 9.1.2 and 9.1.3 describe the test cases applied to the

use case and class diagrams, respectively, while section 9.1.4 describes the test cases applied to sequence diagrams.

9.1.1 Test Cases for Model

The SPL model is the root of the source UML model. It is an instance of the metaclass *Model*, which is a container that packages all the other model elements. The list of contained objects is then referenced by the *packagedElement* attribute. Model elements of type *UseCase*, *Actor*, *Class*, *Association*, *Collaboration*, *SendOperationEvent*, and *ReceiveOperationEvent* have to be contained in the model. Test cases shown in Table 9.1 verify the containment of the derived product model against the containment hierarchy of the SPL model.

9.1.2 Test Cases for Use Case Diagram

The SPL use case diagram is shown in Figure 9.1. It consists of all model element types that compose a use case diagram according to the UML metamodel and all the stereotypes and tagged values from the applied SPLV profile. The SPL use case diagram contains *use cases*, *actors*, and *associations* where a use case with extension point includes the name of the extension point, while an extending use case includes the name of the base use case. Each association contains the names of its role ends, while each property has two attributes *Literal Unlimited Natural* and *Literal Integer* for specifying its multiplicity.

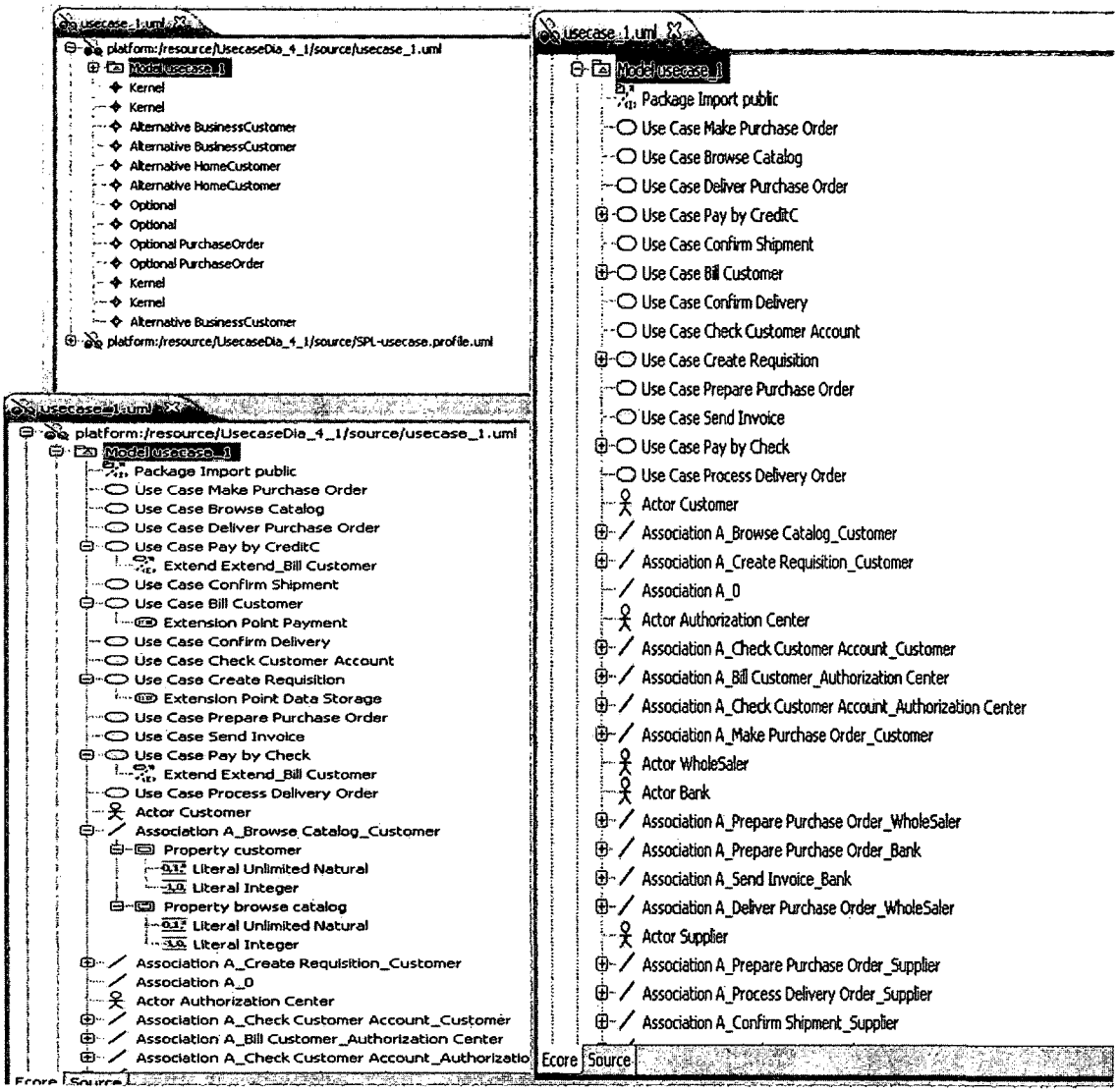


Figure 9.1: Use Case Diagram of the e-commerce SPL.

In order to check both the completeness and correctness of the derived model, the product model shown in Figure 9.2 has to include only the *kernel* use cases and the *optional* and *alternative* use cases annotated with the selected feature. The actors attached to the selected use cases and their associations have to be included as well. Test Cases #1 - #8 in Table 9.2 are simple ones; they verify the presence of the selected model elements, along with the absence of the unselected ones. Test Cases #9 – #13 are more complex, they verify the containment conformance of the product model by involving combinations of model elements. Test Case #9 verifies that all the selected model elements are included inside the model. Test Cases #10 and #11 ensure that each use case with extension point contains the correct name of the extension point and each extending use case contains the correct name of the base use case, respectively. Test Case #12 verifies that the associations contain their properties, while Test Case #13 verifies that these properties contain their upper and lower attributes.

The correctness of the product model is also inferred by checking that the model doesn't contain other model elements realizing unselected features as well as the stereotypes or tagged values related to the SPLV profile are not included any more.

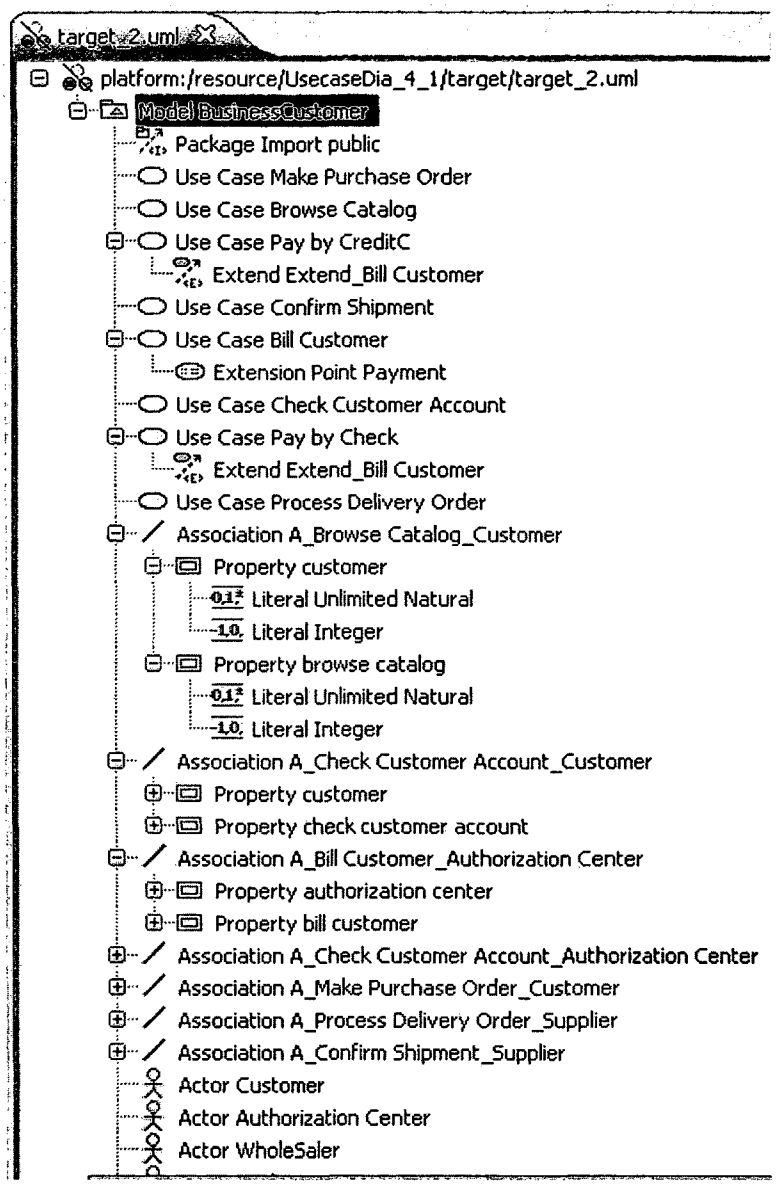


Figure 9.2: Use Case Diagram of a specific Product.

Table 9.2: Test Cases for Use Case Diagram.

	Use Case	Actor	Association	Extension Point	Extended Use Case	Property	Literal Integer	Literal Unlimited Natural
TC #1	X							
TC #2		X						
TC #3			X					
TC #4				X				
TC #5					X			
TC #6						X		
TC #7							X	
TC #8								X
TC #9	X	X	X	X	X	X	X	X
TC #10	X				X			
TC #11	X			X				
TC #12			X			X		
TC #13						X	X	X

9.1.3 Test Cases for Class Diagram

The SPL class diagram shown in Figure 9.3 consists of all model element types that compose a class diagram according to the UML metamodel and all the stereotypes and tagged values from the applied SPLV profile. The SPL class diagram contains *Class* and *Association*. Each class contains two types of properties and operations. The sub-class contains a generalization/specialization relationship. The property related to an association has two attributes *Literal Unlimited Natural* and *Literal Integer* for specifying its multiplicity while, the property related to a class has three attributes *Literal Unlimited*

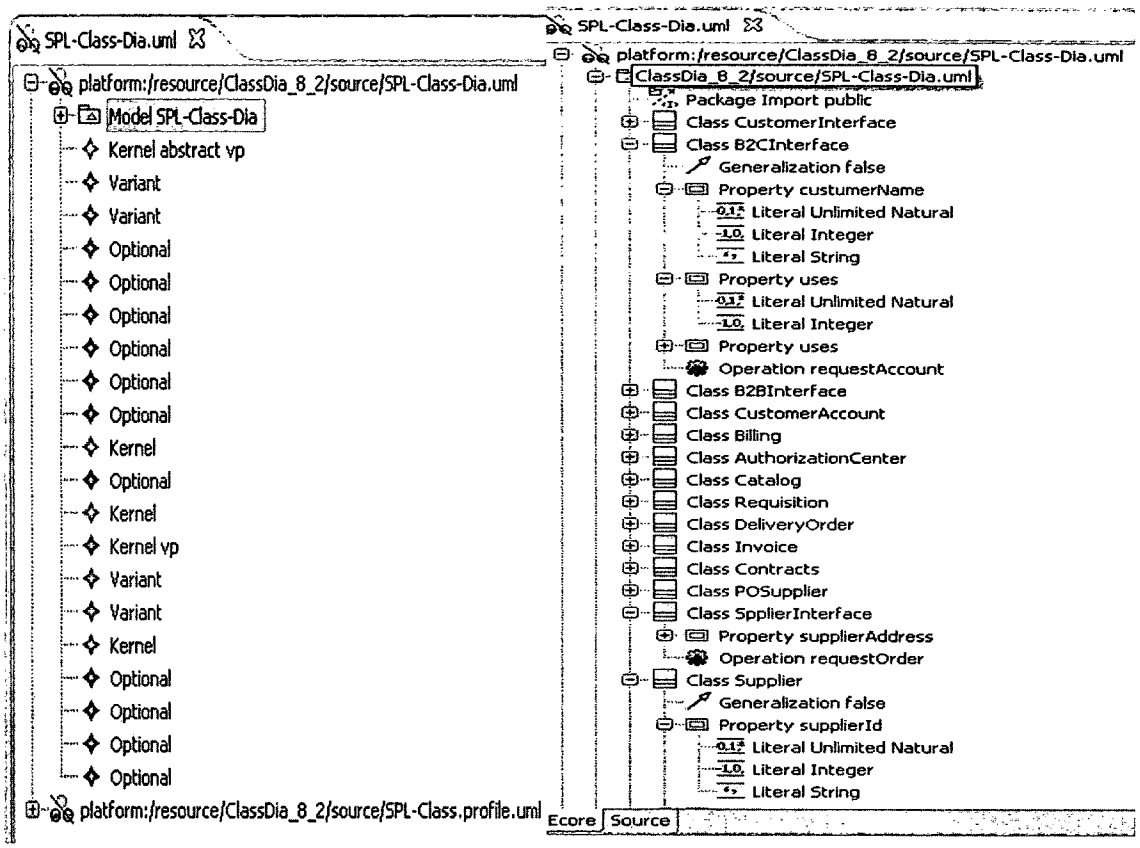


Figure 9.3: Class Diagram of the e-commerce SPL.

Natural, Literal Integer, and Literal String for specifying its multiplicity and default value.

The completeness and correctness of the class diagram are verified in a similar manner to the use case diagram. The product model shown in Figure 9.4 has to include only the *kernel* classes and the *optional* and *variant* classes mapping to the selected features. Simple and complex test cases have been applied to the model as shown in Table 9.3. For instance, Test Cases #12 and #13 verify that the attribute related to a class contains three attributes to specify its multiplicity and default value while the property related to an association contains only two attributes to specify its multiplicity.

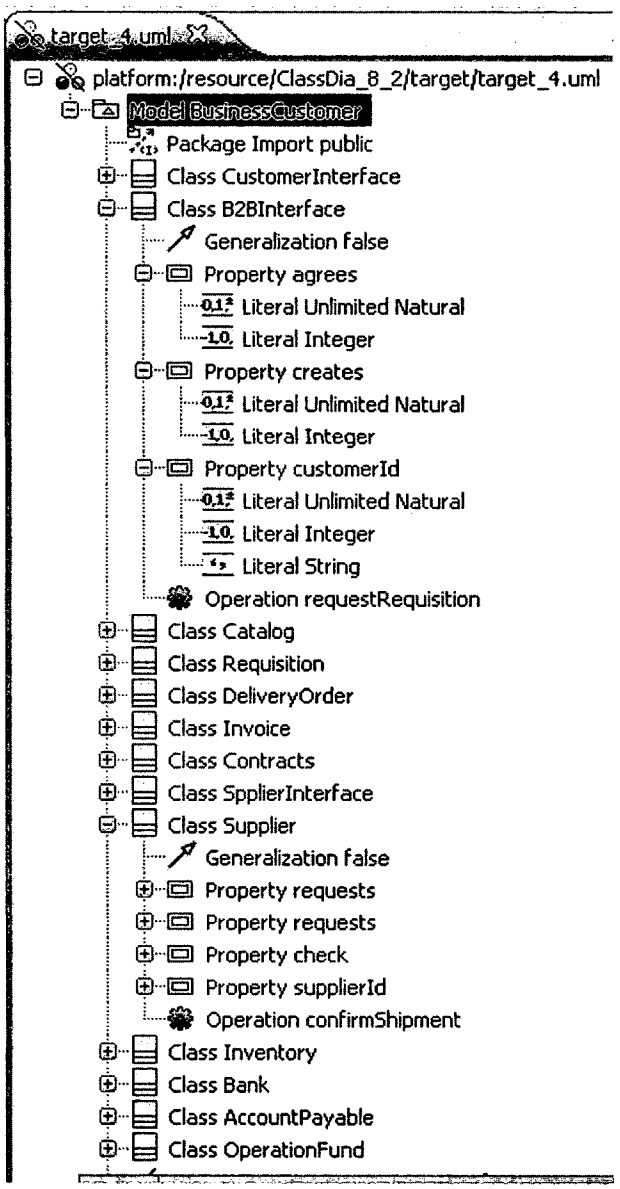


Figure 9.4: Class Diagram of a specific Product.

Table 9.3: Test Cases for Class Diagram.

	Class	Association	Operation	Generalization	Attribute	Property	Literal String	Literal Integer	Literal Unlimited Natural
TC#1	X								
TC#2		X							
TC#3			X						
TC#4				X					
TC#5					X				
TC#6						X			
TC#7							X		
TC#8								X	
TC#9									X
TC#10	X	X	X	X	X	X	X	X	X
TC#11	X		X	X	X	X			
TC#12					X		X	X	X
TC#13						X	X	X	

9.1.4 Test Cases for Sequence Diagram

The SPL sequence diagram contains a collaboration which may include more than one interaction. The interaction contains model elements of type *Lifeline*, *Message*, *Message Occurrence Specification*, *Behaviour Execution Specification*, *Execution Occurrence Specification*, and *Combined Fragment.* The *Combined Fragment* may contain an *Interaction Use* or other model elements. The SPL sequence diagram also contains all the stereotypes and tagged values from the applied SPLV profile as well as the stereotypes and the generic tagged values from the MARTE profile.

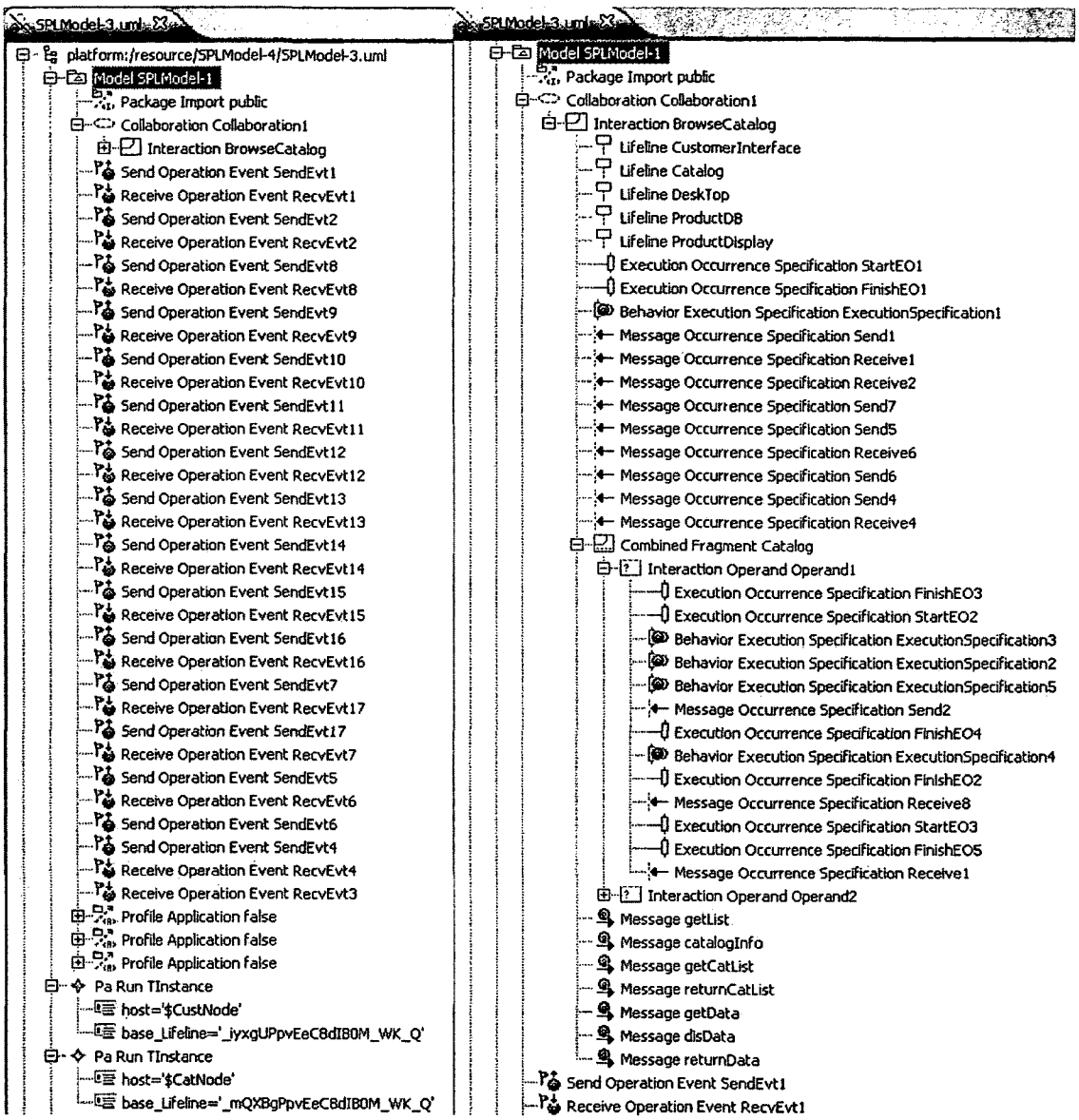


Figure 9.5: SPL Scenario Browse Catalog.

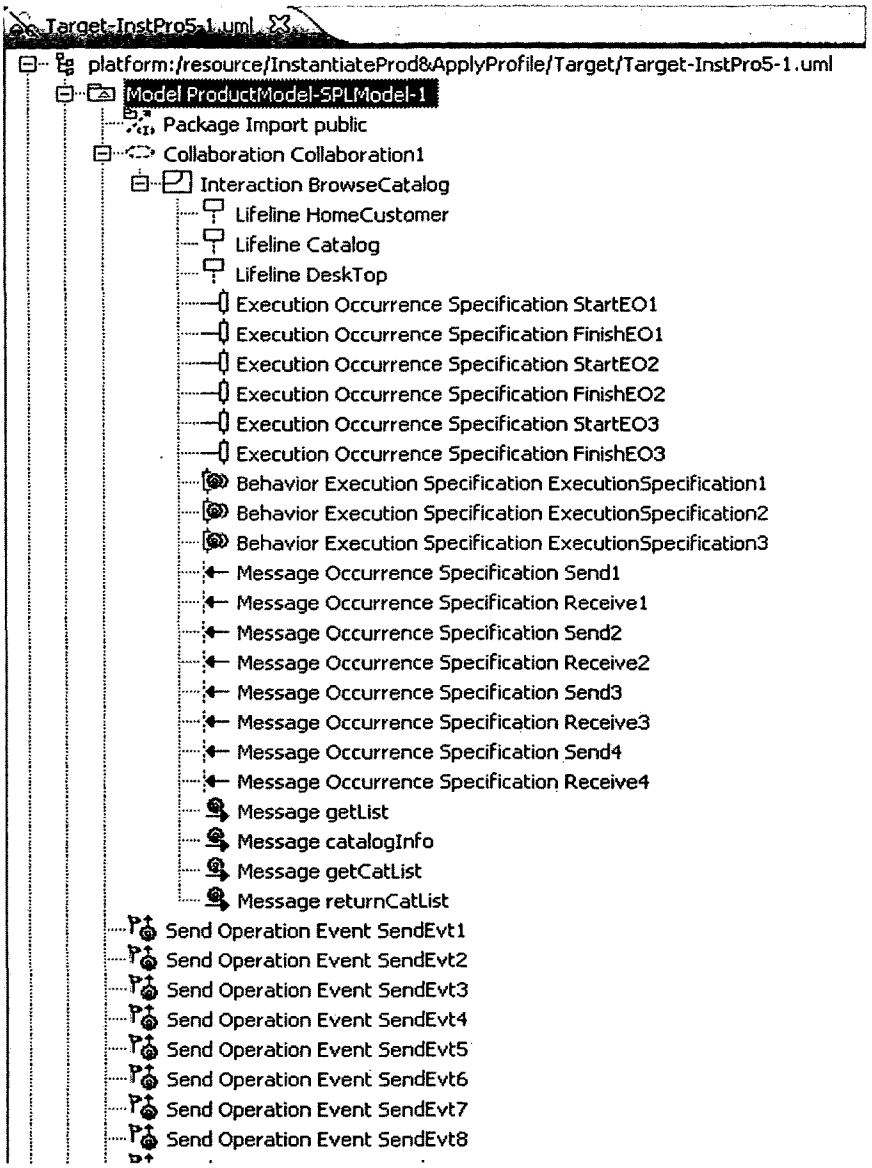


Figure 9.6: Scenario Browse Catalog for a specific Product with feature StaticCatalog.

For instance, Figure 9.5 shows the SPL scenario *BrowseCatalog* where collaboration "*Collaboration1*" contains the interaction *BrowseCatalog*. The figure also shows some of the MARTE annotations applied to the *lifelines*. The scenario *BrowseCatalog* contains an *Alt Combined Fragment* "*Catalog*" which contains two *Operands*.

Table 9.4: Test Cases for Sequence Diagram.

	Collaboration	Interaction	Lifeline	Message	Message Occurrence Specification	Behaviour Execution Specification	Execution Occurrence Specification	Combined Fragment
TC #1	X							
TC #2		X						
TC #3			X					
TC #4			X	X				X
TC #5				X	X			
TC #6			X			X		X
TC #7						X	X	
TC #8								X
TC #9		X	X	X	X	X	X	X

In order to check both the completeness and correctness of the derived model, the product model shown in Figure 9.6 has to include only the *kernel* lifelines and the *optional* and *variant* lifelines corresponding to the selected feature configuration. The messages between two selected lifelines that are not contained in unselected *Combined Fragments* have to be included as well. For each selected message, both the sending and receiving ends of the message represented by *Message Occurrence Specifications* have to be selected. The unit of behaviour execution within a selected lifeline that is not contained in unselected *Combined Fragments* should be selected. The start and finish *Execution Occurrence Specifications* for each selected *Behaviour Occurrence Specification* have to be included as well. Test Cases #1 - #7 in Table 9.4 verify the presence of the selected model elements, along with the absence of the unselected ones.

For instance, the feature *StaticCatalog* is selected in the product scenario *BrowseCatalog* shown in Figure 9.6. Test Case #8 verifies the absent of the *Combined Fragment Catalog*, along with the presence of the model elements contained in the selected *Operand*. Test Case #9 verifies the containment conformance of the product

model. For instance, the model elements contained in a selected *Operand* are moved to be directly contained in the *Interaction*.

Chapter 10: Conclusions

This chapter concludes the thesis document by presenting the accomplishments of the work that have been performed in this study to achieve the research goal, along with a section discussing lessons learned from using different open source tools. The study's limitations and possible directions of future work are presented.

10.1 Contributions

This section summarizes the work presented in this research, and gives the overall conclusions. The most significant contributions are as follows:

- Proposing and implementing a model transformation algorithm that derives a specific product model from a UML software product line model. The SPL model is extended with two profiles: a) SPLV (defined in the thesis) for expressing product line variability, and b) MARTE (standardized by OMG) for performance annotations. The transformation algorithm handles both types of annotations. The generated product model will be transformed by the previously proposed PUMA transformation into a performance model.
- Proposing an efficient mapping technique between features and the model element realizing them, which aims to minimize the amount of explicit feature annotations in the UML design model of SPL. Implicit feature mapping is inferred during product

derivation from the relationships between annotated and non-annotated model elements as defined in the UML metamodel. The mapping technique is implemented in the model transformations for deriving use case, class and sequence diagrams for a specific product based on the selected feature configuration.

- Separate the platform independent and platform specific concerns during product derivation. Performance completions, which are platform aspects, execution environments, different types of communication realizations and other external factors that have an impact on performance, are added to the platform-independent model of the product. The variability space of performance completions is represented by a separate feature model, Performance Completion-feature model (PC-feature model).
- Proposing a user-friendly semi-automatic technique for handling large numbers of generic parameters that need to be bound with concrete values provided by the user. This technique reduces the burden of the user by automating the collection of generic parameters from different diagrams of the model, presenting them in context, giving guidelines regarding the range of values that can be chosen and providing support for selecting PC features for different model elements.

A more detailed list of the work performed during the thesis research and practical contributions is given bellow:

- Introducing a UML profile SPLV based on Gomaa's methodology PLUS [GOM05], to represent variability in SPL artifacts and describe the mapping between the feature model and design models. SPLV contains a subprofile for the SPL behavioural view

to map the variability at design time during product derivation to the variability at run-time in the generated product model.

- The mapping between PC-features and performance attributes they affect is specified. Every possible PC-feature choice is mapped to certain MARTE annotations corresponding to UML model elements in the product model. This mapping is realized by the transformation generating the parameter spreadsheets, which provides the user with mapping information in order to put the annotation parameters needing to be bound to concrete values into context.
- Handling the allocation of the software components of a given product PIM to the actual processing nodes in the deployment diagram provided by the user. The transformation that generating the spreadsheets collects all these hardware resources and associates their list in the spreadsheets for the user to decide on the actual allocation by choosing a processor from this list.
- A model transformation for transforming the generated spreadsheets to the required syntax accepted by the ATL program is accomplished. The spreadsheets can now be read as input to the binding transformation.
- Performing the actual binding of performance annotations to transform the generated product PIM to a PSM.
- The source model to our model transformation is created, which consists of: a) a multi-view UML design model for a family of products, which represents a superimposition of all variant products, called SPL model in our work; b) generic MARTE annotations to represent performance specifications; c) SPLV profile to represent variability in SPL artifacts.

- An e-commerce case study product line that contains 27 features is created by analyzing the requirements in the domain and extracting the common part from the variable ones.
- The feature model of the e-commerce SPL is created as an extended UML class diagram, where the features and feature groups are modeled as stereotyped classes and the dependencies and constraints between features as stereotyped associations.
- The feature configuration is given to the ATL transformation program as an input parameter without hard-coding it.
- Some performance analysis experiments conducted with the LQN models obtained for different generated product models are presented.
- The verification of the well-formedness of the participating models in our derivation process is ensured.
- Several test cases are developed to check the derived use case, class, and sequence diagrams against its completeness, correctness, and containment conformance.

Table 10.1 shows the number of rules and helpers per ATL transformation. In order to reduce the number of model transformation rules to raise the efficiency of our ATL programs, we tried to build more general rules from which different cases can be derived instead of using many specific rules for specific types of model elements. For example, instead of having three specific rules for base, extending, and included use cases, we have only one general rule that can derive these three types of use cases with the help of additional constraints. This will raise the efficiency of our transformation since the ATL engine will navigate the source model once instead of three times.

Table 10.1: Number of Rules and Helpers.

Transformation	# of rules	# of helpers
Deriving product PIM from SPL model	34	14
Generating a Table model from UML model	6	7
Transforming the generated spreadsheets to required syntax	5	5
Performing the actual binding to derive product PSM	34	22

10.2 Out of the Thesis Scope

We consider the following tasks out of the scope of the thesis:

- This research does not cover the entire domain engineering phase of SPL development process, our focus being on the application engineering phase where product models are derived. However, we present two domain engineering activities, the construction of the e-commerce case study and the source model of SPL.
- The automatic transformation of the outcome of the proposed approach (a given UML product model with concrete performance annotations) by the PUMA toolset to a Layered Queueing Network (LQN) performance model is not covered in the thesis, as shown in Figure 1.1.

10.3 Limitations

There are some limitations with the proposed model transformation of this thesis such as:

- The derivation approach is concerned with the diagrams needed for generating a LQN performance model, not with all the diagrams for generating a complete product model.

- This work does not pay attention to detailed issues related to the generation of a product implementation. For instance, methods in the class diagram are not mapped to specific features; also, we do not verify whether each message in the sequence diagram matches a method in the class diagram.
- Limitation in verification and validation (as described in chapter 9):
    - We validated only the model transformations that were designed and implemented in the thesis, but did not validate the transformation from software to performance model which is part of the larger research project (as shown in Figure 1.1)
    - The model transformation validation was done through test cases rather than formally.

10.4 Lessons Learned

Since model transformation is a new emerging field, two new open source tools under development by two different research groups from France are used in this research implementation; the Papyrus modeling tool and the rule-based Atlas transformation tool. The lessons learned from employing these tools and the challenge faced for integrating them are as follows:

- Being a relatively new language, transformation development in ATL has a longer start-up time than development in general-purpose languages because the education, time, and skills necessary to properly utilize ATL are not as common.

- Since these tools are still under development, every new updated version of one tool or the other raised compatibility problems as some features and plug-ins are deprecated and no longer supported.
- The only support or help is from two specific newsgroups associated with each of the two tools; each question takes time to be answered and there is a lot of trial and error.
- However, there are obvious advantages in using a UML modeling tool such as Papyrus which provides advanced visual tools for building UML diagrams, allowing the creation of the SPL model by drawing (drag and drop) without worrying about the XML version or the validation against the UML metamodel.
- Using a model transformation language such as ATL, the transformation can be implemented straightforward and without worrying about the details of navigating the UML model.

10.5 Future Work

Although the proposed model transformation approach for deriving a UML+MARTE model of a given product from a UML+SPLV+MARTE model of SPL is successfully realized in this research, we propose several directions for future work:

- Completing the integration with the PUMA toolset (which is being re-implemented for the latest MARTE version) and experimenting with performance models in order to validate them and to bring recommendation for improvement to the designers.
- Integrating aspect-oriented modeling (AOM) technique to our derivation approach to insert new requirements (features) into a specific product model. These new features are not realized in the SPL model. Thus, they are going to be presented as concerns

that have to be weaved to specific locations in the product model. Each new feature has structural and behaviour views annotated with generic performance properties. We believe that this new technique for inserting a new feature is more efficient than changing the entire SPL model to include new features. When a new feature becomes stable, it is possible to weave it into the SPL model - which is part of the SPL model evolution.

- In our approach the performance impact of underlying platforms is included into the UML+MARTE model of a product as aggregated platform overheads, expressed in MARTE annotations attached to existing processing and communication resources in the generated product model. This will keep the model simple and still allow us to generate a performance model containing the performance effects of both the product and the platforms. In the future, we propose to employ aspect-oriented modeling to include the impacts of underlying platforms by presenting each PC-feature in the PC-feature model as a generic aspect model that can be reused with different applications. The PC-aspect models are composed to a platform independent model of the generated product from a SPL model to obtain a context-specific model for further generation of a performance model. This will provide more fine-grained model and sufficient details to generate more accurate performance estimates and identify the exact cause of the performance problems.
- Integrating a feature modeling tool such as pure:variants [PVA06] or featureplugin [ANT04] into our approach for providing a valid feature configuration by automatically checking the validity of selected features from the feature model, identifying constraint violations, as well as resolving dependency conflicts.

- Developing an automatic mapping tool (similar with FeatureMapper [HEI08]) for our mapping technique from features to model elements, to generate a mapping while the developer is modeling the realization of a feature. Thus, any changes to the model can be tracked back and mapped to a previously selected feature.
- Applying the proposed approach to large software product line applications to verify its scalability.

References

[ANTH08] P. Anthonysamy and S. Some, "Aspect-Oriented Use Case Modeling for Software Product Lines", In Proceedings of the 2008 Aspect-oriented software development (AOSD) workshop on Early aspects, Brussels, Belgium, 2008.

[ANT04] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: Feature modeling plug-in for Eclipse", In OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, pp.67-72, New York, NY, USA, 2004.

[ATL] Atlas Transformation Language (ATL), http://www.eclipse.org/m2m/atl

[AVI07] O. Avila-García, A. E. García, and E. V. S. Rebull, "Using Software Product Lines to Manage Model Families in Model-Driven Engineering", in ACM symposium on Applied Computing, pp. 1006-1011, Seoul, Korea, 2007.

[AZE10] S. Azevedo, R. Machado, A. Braganca, H. Ribeiro, "Support for Variability in Use Case Modeling with Refinement", Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software MOMPES '10, 2010

[BAB10] M. Babar, L. Chen, F Shull, "Managing Variability in Software Product Lines", IEEE Software, vol. 27, no. 3, pp. 89-91, 2010.

[BAL04] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey", in IEEE Transactions on Software Engineering, vol. 30, N.5, pp.295-310, 2004.

[BAL06] S. Balsamo, M. Marzolla, and R. Mirandola, "Efficient Performance Models in Component-Based Software Engineering", Proc. of the 32nd EUROMICRO Conference on Software Engineering and Advanced, 2006.

[BAR09] J. Barreiros and A. Moreira, "A Model-Based Representation of Configuration Knowledge", In Proceedings of the 1st International workshop on Feature-Oriented Software Development (FOSD'09), pp. 43-48, Denver, Colorado, USA, 2009.

[BART09] J. Bartholdt, M. Medak, R. Oberhauser, "Integrating Quality Modeling with Feature Modeling in Software Product Lines", Proc. of the 4th International Conference on Software Engineering Advances (ICSEA2009), pp.365-370, 2009.

[BEC08] S. Becker, "Coupled Model Transformations", Proceedings of the 7th international workshop on Software and performance, Princeton, NJ, USA, pp. 103-114, 2008.

[BEL10] L. Belategi, G. Sagardui, L. Etxeberria, "Variability Management in Embedded Product Line Analysis", Proc. Of the 2nd International Conference on Advances in System Testing and Validation Lifecycle (VALID'10), pp. 69-74, Nice, France, 2010.

[BEL10a] L. Belategi, G. Sagardui, L. Etxeberria, "MARTE mechanisms to model variability when analyzing embedded software product Lines", Proc. Of the 14th International Conference on Software Product Line (SPLC'10), pp.466-470, 2010.

[BEL11] L. Belategi, G. Sagardui, L. Etxeberria, "Model based analysis process for embedded software product lines", Proc of the 2011 Int Conference on Software and Systems Process (ICSSP '11), 2011.

[BEN05] D. Benavides, P. Trinidad, and A. Ruiz-Cort´es, "Automated Reasoning on Feature Models", Proc. of the 17th International Conference on Advanced Information Systems Engineering (CAiSE), 2005.

[BER02] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML sequence diagrams and statecharts to analysable Petri net models", In Proceeding of the 3rd Int. Workshop on Software and Performance (WOSP02), pp. 35-45, Rome, July 2002.

[BEZ05] J. Bezivin, F. Jouault, and J. Palies, "Towards Model Transformation Design Patterns", In the First European Workshop on Model Transformation (EWMT 05), 2005.

[BIL04] A. Billig, S. Busse, A. Leicher, and J. G. SuB, "Platform Independent Model Transformation Based on TRIPLE", Proc. of the 5th ACM/IFIP/USENIX international conference on Middleware, LNCS 2004, Vol 3231, pp.493—511, Toronto, Canada, 2004.

[BOO99] G. Booch, J. Rumbaugh, and I. Jacobson, "The Unified Modeling Language User's Guide", New York: Addison-Wesley, 1999.

[BOT07] G. Botterweck, L. O'Brien, and S Theil, "Model-driven derivation of product architecture", In 22nd IEEE/ACM International Conference on Automated Software Engineering, pp. 469-472, Atlanta, Georgia, USA, 2007.

[BOT09] G. Botterweck, K. Lee, and S. Thiel, "Automating Product Derivation in Software Product Line Engineering", In Proceedings of Software Engineering 2009 (SE09), pp 177-182, Kaiserslautern, Germany, 2009.

[BRA05] A. Braganca and R. J. Machado, "Deriving Software Product Line's Architectural Requirements from Use Cases: an Experimental Approach", In the 2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, Rennes, France, 2005.

[BRA06] A. Braganca and R. J. Machado, "Extending UML 2.0 Metamodel for Complementary Usages of the «extend» Relationship within Use Case Variability Specification", SPLC 2006, Baltimore, Maryland, 2006.

[BRA07a] A. Braganca and R. J. Machado, "Adopting Computational Independent Models for Derivation of Architectural Requirements of Software Product Lines", In the 4th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES'07, pp. 91-101, 2007.

[BRA07] A. Braganca and R. J. Machado, "Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines", In the 11th International Software Product Line Conference (SPLC), Kyoto, Japan, 2007.

[BRU05] H. Bruneliere, ATL Transformation Example: Microsoft Office Excel Extractor, 2005. http://www.eclipse.org/m2m/atl/atlTransformations/MSOfficeExcelExtractor/ExampleMicrosoftOfficeExcelExtractor[v00.01].pdf

[BUH05] S. Buhne, K. Lauenroth, and K. Pohl, "Modelling Requirements Variability Across Product Lines", In Proceedings of the 13th IEEE International Conference on Requirements Engineering, pages 41–50, 2005.

[CAV04] C. Cavenet, S. Gilmore, J. Hillston, L. Kloul, and P. Stevens, "Analyzing UML 2.0 activity diagrams in the software performance engineering process", In Proceedings of the 4th International Workshop on Software and Performance (WOSP 2004), pp. 74-83, Redwood City, CA, Jan 2004.

[CHE09] L. Chen, M. A. Babar, N. Ali, "Variability Management in Software Product Lines: A Systematic Review", In Proceedings of the 13th International Software Product Line Conference (SPLC), San Francisco, CA, USA, 2009.

[CHO05] Y. Choi, G. Shin, Y. Yang, and C. Park, "An approach to extension of UML 2.0 for representing variabilities", Computer and Information Science, Fourth Annual ACIS International Conference, INSPEC Accession 3, pp.258–261, 2005.

[CHU09] L. Chung and J.C. sampaio do prado Leite, "On Non-Functional Requirements in Software Engineering", Conceptual Modeling: Foundations and Applications, pp. 363-379, 2009.

[CLA01] M. Clauss, "Modeling variability with UML", GCSE 200–Young Researchers Workshop, September 2001.

[CLA01a] M. Clauss, "Generic Modeling using UML extensions for variability", in Workshop on Domain Specific Visual Languages at OOPSLA, Tampa Bay, FL, USA, 2001.

[CLE01] P. Clements, and L. Northrop, "Software Product Lines: Practice and Patterns", pp.608, Addison-Wesley, 2001.

[COR00] V. Cortelessa and R Mirandola, "Deriving a Queueing Network based Performance Model from UML Diagrams", In Proceedings of the 2nd ACM Workshop on Software and Performance, pp.58-70, Ottawa, Canada, 2000.

[COR07] V. Cortellessa, A. Di Marco, and P. Inverardi, "Non-Functional Modeling and Validation in Model-Driven Architecture", Proc of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA07), pp. 25, Mumbai, 2007.

[CZA00] K. Czarnecki and U.W. Eisenecker, "Generative Programming: Methods, Tools, and Applications", Addison-wesley, Boston, MA, 2000.

[CZA02] K. Czarnecki, T. Bednasch, and P. Unger, U.W. Eisenecker, "Generative Programming for Embedded Software: An Industrial Experience Report", In Proceeding of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, 2002.

[CZA04] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged configuration using feature models", In Proceedings of the R. L. Nord (ed.), Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, 2004.

[CZA05] K. Czarnecki, M. Antkiewicz, C.H.P. Kim, S. Lau, and K. Pietroszek, "Model-Driven Software Product Lines", in OOPSLA, San Diego, California, 2005.

[CZA05a] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization", Software Process Improvement and Practice, pp. 7–29, 2005.

[CZA05b] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants", In Generative Programming and Component Enginering (GPCE), volume 3676 of LNCS, pp. 422–437, Springer-Verlag, 2005.

[CZA05c] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multi-level configuration of feature models", Software Process Improvement and Practice, pp. 143–169, 2005.

[CZA06] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness OCL constraints", In Proceedings of the 5th international conference on Generative programming and component engineering, pp. 221-220, Portland, Oregon, USA, 2006.

[DEE03] S. Deelstra, M. Sinnema, J. van Gurp, and J. Bosch, "Model driven architecture as approach to manage variability in software product families", In Proceedings of the Workshop on Model Driven Architectures: Foundations and Applications, pp. 109–114, 2003.

[DEE05] S. Deelstra, M. Sinnema, and J. Bosch, "Product derivation in software product families: A case study", in Journal of Systems and Software, vol. 74, pp. 173-194, 2005.

[DEE09] S. Deelstra, M. Sinnema, J. Bosch, "Variability assessment in software product families", Information and Software Technology, pp. 195-218, 2009.

[ERI05] M. Eriksson, J. Borstler, and K. Borg, "The PLUSS Approach – Domain Modeling with Features, Use Cases and Use Case Realizations", In Proceedings of the 9th Int. Software Product Line Conference (SPLC), pp. 33-44, Rennes, France, 2005.

[ERI09] M. Eriksson, J. Borstler, and K. Borg, "Managing requirements specifications for product lines - An approach and industry case study", The Journal of Systems and Software 82 (2009), pp. 435–447, 2009.

[EST] A. Estevez, J. Padron, V. Sanchez, and J. L. Roda, "ATC: A low-level model transformation language", In MDEIS 2006: Proceedings of the 2nd International Workshop on Model Driven Enterprise Information Systems, 2006.

[FAN04] Fantechi, A., S. Gnesi, G. Lami, and E. Nesti, "A Methodology for the Derivation and Verification of Use Cases for Product Lines", In SPLC 2004, Springer, 2004.

[FIL10] S. Filho, H. Mariano, U. Kulesza, T. Batista, "Automating Software Product Line Development: A Repository-Based Approach", Proceedings of the 36th EUROMICRO conference on Software Engineering and Advanced Applications (SEAA), pp. 141-144, 2010.

[FON00] S. Fong, C. Se-Leng, "Modeling personnel and roles for electronic commerce retail" Special Interest Group on Computer Personnel Research Annual Conference, In Proceedings of the 2000 ACM SIGCPR conference on Computer personnel research, Chicago, Illinois, United States, pp. 45-53, 2000.

[GOM02] H. Gomaa and M.E. Shin "Multiple-View Meta-Modeling of Software Product Lines", In Proceedings of the 8th International Conference on Engineering of Complex Computer Systems (ICECCS 2002), IEEE Computer Society 2002, pp. 238-246, 2002.

[GOM04] H. Gomaa and D. L. Webber, "Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model", In Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04), pp. 90268c - Track 9, 2004.

[GOM04a] H. Gomaa and M. E. Shin, "A Multiple-View Meta-Modeling Approach for Variability Management in Software Product Lines", In: Bosch, J., Krueger, C. (eds.) ICOIN 2004 and ICSR 2004. LNCS, vol. 3107, pp. 274–285. Springer, Heidelberg, 2004.

[GOM05] H. Gomaa, "Designing Software Product Lines with UML: From Use Cases to Pattern-based software Architectures", Addison-Wesley Object Technology Series, 2005.

[GOM05a] H. Gomaa, "Architecture-Centric Evolution in Software Product Lines", Workshop on Architecture-Centric Evolution, Glasgow, UK, 2005.

[GOM06] H. Gomaa and M. Saleh, "Feature Driven Dynamic Customization of Software Product Lines", In Proceedings of the 9 International Conference on Software Reuse, Springer Verlag LNCS 4039, pp. 58-72, Torino, Italy, 2006.

[GOM07] H. Gomaa and M. Hussein, "Model-Based Software Design and Adaptation", International Conference on Software Engineering Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 7, 2007.

[GOM07a] H. Gomaa and M. E. Shin, "Automated Software Product Line Engineering and Product Derivation", In Proceedings of the 40th Hawaii International Conference on System Sciences, 2007.

[GOM07b] H. Gomaa, "Feature Dependent Coordination and Adaptation of Component-Based Software Architectures", In Proceedings of the Workshop on Coordination and Adaptation Techniques, Berlin, Germany, 2007.

[GOM08] H. Gomaa and M.E. Shin "Multiple-View Modeling and Meta-Modeling of Software Product Lines", The Institution of Engineering and Technology 2008 (IET Softw.), vol.2, No.2, pp. 94-122, 2008.

[GRE04] J. Greenfield and K. Short, "Software Factories: Assembling Applications with Patterns", Models, Frameworks, and Tools. Wiley, Indianapolis, IN, 2004.

[GRO07a] I. Groher and M. Voelter, "Expressing feature-based variability in structural models", In Proceedings of the Workshop on Managing Variability for Software Product Lines, Kyoto, Japan, 2007.

[GRO07b] I. Groher, H. and M. Voelter, "XWeave – Models and Aspects in Concert", In Proceedings of the 10th international workshop on Aspect-oriented modeling, pp. 35-40, New York, NY, USA, 2007.

[GRO07] I. Groher, H. Papajewski, M. Voelter, "Integrating Model-Driven Development and Software Product Engineering", In Eclipse Summit '07: Proceedings of the Eclipse Modeling Symposium, Ludwigsburg, Germany, 2007.

[GRO08a] I. Groher, and M. Voelter "Using Aspects to Model Product Line Variability", In Proceedings of the 12th International Conference of Software Product Lines, SPLC 2008, pp. 89-95, Limerick, Ireland, 2008.

[GRO08] I. Groher, C. Schwanninger, M. Voelter "An Integrated Aspect-Oriented Model-Deriven Software Product Line Tool Suite" In Proceedings of the 30th International Conferenve on Software Engineering, pp. 939-940, 2008.

[GRO09] I. Groher and M. Voelter. "Aspect-Oriented Model-Driven Software Product Line Engineering"' Transactions on Aspect-Oriented Software Development (AOSD) VI, LNCS, pp. 111-152, 2009.

[GRU02] V. Gruhn, M. Mocker, L. Schiipe, "Development of an Electronic Commerce Portal System using a Specific Software Development Process" In Proceedings of 14th International Conference on Tools Pacific: Objects for internet, mobile and embedded applications, ACM, Vol. 21, Sydney, Australia, pp. 93-101, 2002.

[HAL03] G. Halmans and K. Pohl, "Communicating the variability of a software-product family to customers", Journal of Software and Systems Modeling, Springer, 2003.

[HAP10] J. Happe, S. Becker, C. Rathfelder, H. Friedrich, R. H. Reussner, "Parametric performance completions for model-driven performance prediction", Performance Evaluation, Volume 67 , Issue 8, pp. 694-716, 2010.

[HAU04] O. Haugen, B. MOller-Pedersen, and J. Oldevik, A. Solberg, "An MDA-based framework for model-driven product derivation", In: M. H. Hamza, editor, Software Engineering and Applications, pp. 709-714. ACTA Press, Cambridge, 2004.

[HEI07] F. Heidenreich and C. Wende, "Bridging the gap between features and models", In the 2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07) co-located with the 6th International Conference on Generative Programming and Component Engineering (GPCE'07), 2007.

[HEI08] F. Heidenreich, J. Kopcsek, C. Wende, "FeatureMapper: Mapping Features to Models", In Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08), pp. 943-944, New York, NY, USA, 2008.

[HEI08a] F. Heidenreich, I. Savga, and C. Wende, "On Controlled Visualisations in Software Product Line Engineering", In Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE'08), collocated with the 12th International Software Product Line Conference (SPLC'08), 2008.

[HEI09] F. Heidenreich, "Towards Systematic Ensuring Well-Formedness of Software Product Lines", In Proceedings of the 1st International workshop on Feature-Oriented Software Development (FOSD'09), pp. 69-74, Denver, Colorado, USA, 2009.

[HEI10] F. Heidenreich, P. Sanchez, J. Santos, S. Zschaler, M. Alferez, J. Araujo, L. Fuentes, U. Kulesza, A. Moreira, and A. Rashid, "Relating Feature Models to Other Models of a Software Product Line: A Comparative Study of FeatureMapper and VML", Transactions on Aspect-Oriented Software Development (TAOSD), 6210, 2010.

[JAY07] P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa, "Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis", In oDELS'07 Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems, LNCS, pp. 151–165, Nashville TN USA, 2007.

[JOH02] I. John, and D. Muthig, "Product Line Modeling with Generic Use Cases", SPLC-2 Workshop on Techniques for Exploiting Commonality Through Variability Management, Second Software Product Line Conference, San Diego, USA, 2002.

[KAH01] P. Kahkipuro, "UML-Based Performance Modeling Framework for Component-Based Distributed Systems", in: R. Dumke et al.(eds), Performance Engineering, LNCS Vol. 2047, Springer, Berlin Heidelberg New York pp167—184, 2001.

[KANG90] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study", Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[KANG98] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", Annals of Software Engineering, vol. 5, pp. 143 – 168, 1998.

[KIM07] Y. Kim, M. Moon, and K. Yeom, "An Aspect-Oriented Approach for Representing Variability in Product Line Architecture", In VaMoS'07: 1st International Workshop on Variability Modelling of Software-intensive Systems, 2007.

[KOR07] B. Korherr, and B. List, "A UML 2 Profile for Variability Models and their Dependency to Business Processes", Database and Expert Systems Applications DEXA '07, the 18th International Conference, Regensburg, Germany, pp. 829-834, 2007.

[LAH07] P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. J'ez'equel, "Introducing Variability into Aspect-Oriented Modeling Approaches", In Proceedings of MoDELS'07, LNCS, pp. 498–513, Nashville TN USA, Vanderbilt University, Springer-Verlag, 2007.

[LOP04] J. P. Lopez-Grao, J. Merseguer, J. Campos, "From UML Activity Diagrams to Stochastic Petri Nets: Application To Software Performance Engineering," In the 4th International Workshop on Software and Performance (WOSP 2004), Redwood City, CA, pp25—36, 2004.

[LOU05] N. Loughran, A. Sampaio, and A. Rashid. From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation. In MoDELS Satellite Events, pp. 262–271, 2005.

[MARTE11] Object Management Group, "UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE)", Version 1.1, OMG document formal/2011-06-02, 2011.

[MEI01] R. D. van der Mei, R. Hariharan, P.K. Reeser, "Web Server Performance Modeling"' Telecommunication Systems (TELSYS) 16(3-4), pp. 361-378, 2001.

[MEN04] D. Menasce, V. Almeida, L. Dowdy, "Performance by Design: Computer Capacity Planning by Example", Prentice Hall PTR, Upper Saddle River, NJ 07458, 2004.

[MEN07] D. Menasce and V. Akula, "Improving the Performance of Online Auctions Through Server-side Activity-Based Caching", World Wide Web Journal, Springer Verlag, Vol. 10, No. 2, pp. 181-204, 2007.

[MON02] L. Monestel, T. Ziadi, and J.-M. Jézéquel, "Product line engineering: Product derivation", In Workshop on Model Driven Architecture and Product Line Engineering, at the SPLC2 conference, San Diego, 2002.

[MOR07] B. Morin, O.Barais, J.M. Jézéquel, R. Ramos, "Towards a Generic Aspect-Oriented Modeling Framework", In Proceedings of the 3rd International Workshop on Models and Aspects (In conjunction of ECOOP'07), Berlin, Germany, 2007.

[MOR08] B. Morin, O.Barais, J.M. Jézéquel, "Weaving Aspect Configurations for Managing System Variability", In VaMoS'08: the 2nd International Workshop on Variability Modelling of Software-intensive Systems, 2008.

[MOR08a] B. Morin, J. Klein, O.Barais, J.M. Jézéquel, "A Generic Weaver for Supporting Product Lines", In EA@ICSE'08: Int. Workshop on Early Aspects, 2008.

[MOR08b] B. Morin, G. Vanwormhoudt, P. Lahire, A. Gaignard, O. Barais, and J.M. Jézéquel. "Managing variability complexity in aspect-oriented modeling", In Proceedings of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 08), Toulouse, France, 2008.

[MOR09] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J.M. Jézéquel, "Weaving Variability into Domain Metamodels", In ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09), Denver, Colorado, USA, 2009.

[NGU09] Q. Nguyen, "Non-Functional Requirements Analysis Modeling for Software Product Lines", Proc. of Modeling in Software Engineering (MISE'09), ICSE workshop, pp. 56-61, 2009.

[NOH11] A. Nohrer, and A. Egyed, "Optimizing User Guidance during Decision-Making", Proc. of the 15th Int Conference on Software Product Line (SPLC'11), Munich, Germany, 2011.

[OLD07] J. Oldevik, O. Haugen, "Higher-Order Transformations for Product Lines", In Proceedings of 11th Int. Software Product Line Conference (SPLC), pp. 243-254, Kyoto, Japan, 2007.

[OLI05] E. Olimpiew and H. Gomaa, "Model-based Testing for Applications Derived from Software Product Lines". In Proceedings of the Workshop on Advances in Model-Based Software Testing, International Conference on Software Engineering, St. Louis, MO, 2005.

[OMG04] Object Management Group, "MDA Guide Version 1.0.1", omg/03-06-01, 2003.

[OMG07] Object Management Group, "UML: Infrastructure specification", Version 2.1.2, OMG document formal/2007-11-04, 2007.

[OMG07a] Object Management Group, "UML: Superstructure specification", Version 2.1.2, OMG document formal/2007-11-02, 2007.

[OMG08] Object Management Group, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification", Version 1.0, OMG document formal/2008-04-03, 2008

[OMG09] Object Management Group, "Common Variability Language (CVL) Request For Proposal" OMG Document: ad/2009-12-03, December 2009.

[PVA06] Pure Systems website, Pure::Variants, http://www.pure-systems.com/, 2006.

[PER08] G. Perrouin, J. Klein, N. Guelfi, J. M. Jézéquel, "Reconciling Automation and Flexibility in Product Derivation", In Proceedings of the 12th International Software Product Line Conference (SPLC 08), PP. 339-348, 2008.

[PET02] D. C. Petriu, H. Shen, "Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications", In Computer Performance Evaluation (T. Fields, P. Harrison, J. Bradley, U. Harder,Eds.) LNCS 2324, pp. 159-177, Springer, 2002.

[PLE10] A. Pleuss, G. Botterweck, and D. Dhungana "Integrating Automated Product Derivation and Individual User Interface Design", Proceedings of the 4th International Workshop on Variability Modeling of Software-Intensive Systems (VAMOS 2010), pp. 69-76, January, 2010.

[POH05] K. Pohl, G. Böckle, and F. van der Linden, "Software Product Line Engineering: Foundations, Principles, and Technique", Springer-Verlag Berlin, Heidelberg, 2005.

[PRE02] W. Pree, M. Fontoura, and B. Rumpe. "Product line annotations with uml-f", In Gary J. Chastek, editor, Software Product Lines, In Proceedings of 2nd International Conference, SPLC2, San Diego, CA, USA, 2002, proceedings, LNCS vol 2379, Springer, 2002.

[RAA08] M. Raatikainen, E. Niemelä, V. Myllärniemi, and T. Männistö, "Svamp – An Integrated Approach for Modeling Functional and Quality Variability", 2nd International Workshop on Variability Modeling of Software-intensive Systems (VaMoS), 2008.

[RAB07] R. Rabiser, P. Grunbacher, and D. Dhungara "Supporting Product Derivation by Adapting and Augmenting Variability Models", In Proceedings of 11th International Software Product Line Conference (SPLC 07), pp. 141-150, 2007.

[ROB02] S. Robak, B. Franczyk, and K. Politowicz, "Extending the UML for Modeling Variability for System Families", Int. J. Appl. Math. Comput. Sci., Vol.12, No.2, pp. 285–298, 2002.

[SAN06] J. Sánchez Cuadrado, J. García Molina, and M. Menárguez Tortosa, "RubyTL: A Practical, Extensible Transformation Language", In Proceedings of 2nd European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, pp 158–172, 2006.

[SIN04] M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, "COVAMOF: A Framework for Modeling Variability in Software Product Families", In Proceedings of the 3rd Software Product Line Conference (SPLC 2004), Springer Verlag Lecture Notes on Computer Science Vol. 3154 (LNCS 3154), pp. 197-213, 2004.

[SIN06] M. Sinnema , S. Deelstra , J. Nijhuis , J. Bosch, "Modeling Dependencies in Product Families with COVAMOF", In Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, pp.299-307, 2006.

[SMI90] C.U. Smith, "Performance Engineering of Software Systems", Addison Wesley, 1990.

[STO07] R. Stoiber, S. Meier, and M. Glinz, "Visualizing Product Line Domain Variability by Aspect-Oriented Modeling", In Proceedings of the 2nd International Workshop on Requirements Engineering Visualization (REV'07), in conjunction with RE'07. New Delhi, India, 2007.

[STO08] R. Stoiber, T. Reinhard, M. Glinz, "Visualization Support for Software Product Line Modeling", In Proceedings of the 2nd International Workshop on Visualization in Software Product Line Engineering (ViSPLE'08), at SPLC'08. Limerick, Ireland, 2008.

[STO09] R. Stoiber,M. Glinz, "Modeling and Managing Tacit Product Line Requirements Knowledge", In Proceedings of the 2nd International Workshop on Managing Requirements Knowledge (MaRK'09), at RE'09, Atlanta, USA, 2009.

[STO10] R. Stoiber, M. Glinz, "Feature Unweaving: Efficient Variability Extraction and Specification for Emerging Software Product Lines", Proceedings of the 4th International workshop on Digital Object Identifier, Software Product Management (IWSPM), pp. 53-62, 2010

[STR06] J. Street and H. Gomaa, "An Approach to Performance Modeling of Software Product Lines", Workshop on Modeling and Analysis of Real-Time and Embedded Systems, Genova, Italy, October 2006.

[TAJ11] S.B. Tajali, V.D. Radonjic, and J.P. Corriveau, "Challenges of Variability in Model-Driven and Transformational Approaches: A Systematic Survey", In Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture, Boulder, Colorado, USA, June 2011.

[TRE03] G. W. Treese, L. C. Stewart, "Designing Systems for Internet Commerce", Addison Wesley, second edition, ISBN 0-201-76035-5,

2002.

[TRU09] S. Trujillo, A. Zubizarreta, X. Mendialdua, J. de Sosa, "Feature-Oriented Refinement of Models, Metamodels, Model Transformations", In Proceedings of the 1st International workshop on Feature-Oriented Software Development (FOSD'09), pp. 87-94, Denver, Colorado, USA, 2009.

[TRU10] S. Trujillo, J. Garate, R. Lopez-Herrejon, X. Mendialdua, A. Rosado, A. Egyed, C. Krueger, J. de Sosa, "Coping with Variability in Model-Based Systems Engineering: An Experience in Green Energy", In Proceedings of the 6th European conference on Modelling Foundations and Applications (ECMFA), pp. 294-304, France, 2010.

[VER05] T. Verdickt, B. Dhoedt, F. Gielen, and P. Demeester, "Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models", IEEE Trans. on Software Engineering, Vol. 31, No. 8, 2005.

[VOE07a] M. Voelter, and I. Groher, "Handling Variability in Model Transformations and Generators", In Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), 2007.

[VOE07] M. Voelter, I. Groher, "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development", In Proceedings of the 11th International Software Product Line Conference (SPLC), pp. 233–242, Washington, DC, USA, 2007.

[WEB03] D. Webber and H. Gomaa, "Modeling Variability with the Variation Point Model", Proc. 7 International Conference on Software Reuse, Springer Verlag LNCS 2319, pp. 109-122, Austin, Texas, 2003.

[WHI07] J. White, D. C. Schmidt, E. Wuchner, A. Nechypurenko "Automating Product-Line Variant Selection for Mobile Devices", In Proceedings of the 11th International Software Product Line Conference (SPLC 07), pp. 129-140, 2007.

[WOO95] C.M.Woodside, J.E. Neilson, D.C. Petriu, S. Majundar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software", In IEEE Trans. on Computers, vol.44, Nb.1, pp. 20-34, 1995.

[WOO02] M. Woodside, D. C. Petriu, and K. H. Siddiqui, "Performance-related Completions for Software Specifications", Proc. of the 22rd International Conference on Software Engineering, ICSE 2002, pp. 22-32, Orlando, Florida, USA, 2002.

[WOO05] C.M.Woodside, D.C. Petriu, D.B. Petriu, H.Shen, T.Israr. J. Merseguer, "Performance by Unified Model Analysis (PUMA)", Proc. of the 5th ACM Int.Workshop on Software and Performance WOSP'2005, pp. 1-12, Palma, Spain, 2005

[WOO08] C.M. Woodside, D.C. Petriu, J. Xu, T. Israr, J.Merseguer, "Methods and Tools for Performance by Unified Model Analysis (PUMA)", Technical Report SCE-08-06, Carleton University, Systems and Computer Engineering, pp. 35, 2008.

[ZIA02] T. Ziadi, L. Hélouët, and J.-M. Jézéquel, "Modeling behaviors in product lines", In Proceedings of REPL'02 Workshop on Requirements Engineering for Product Lines, pp. 33–38, Essen, Germany, 2002.

[ZIA03] T. Ziadi, L. Hélouët, and J.-M. Jézéquel, "Towards a UML Profile for Software Product Lines", In Software Product-Family Engineering, the 5th International Workshop, pp. 129–139, Springer, 2003.

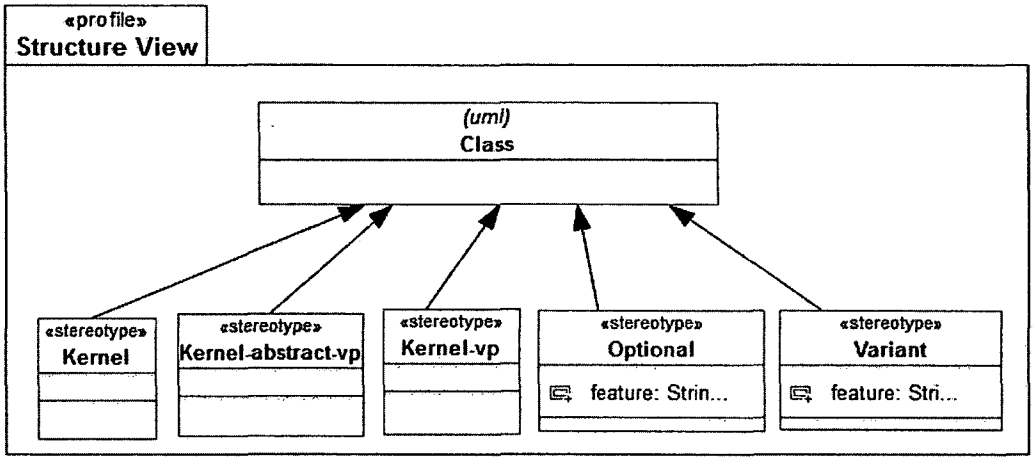
[ZIA03a] T. Ziadi, J.-M. Jézéquel, and F. Fondement, "Product line derivation with uml", In Jilles van Gurp and Jan Bosh, editors, Proceedings Software Variability Management Workshop, University of Groningen Department of Mathematics and Computing Science, pp. 94–102, 2003.

[ZIA04] T. Ziadi, L. Hélouët, and J.-M. Jézéquel, "Behaviors generation from product lines requirements", In Proceedings of the UML 2004 workshop on Software Architecture Description, 2004.

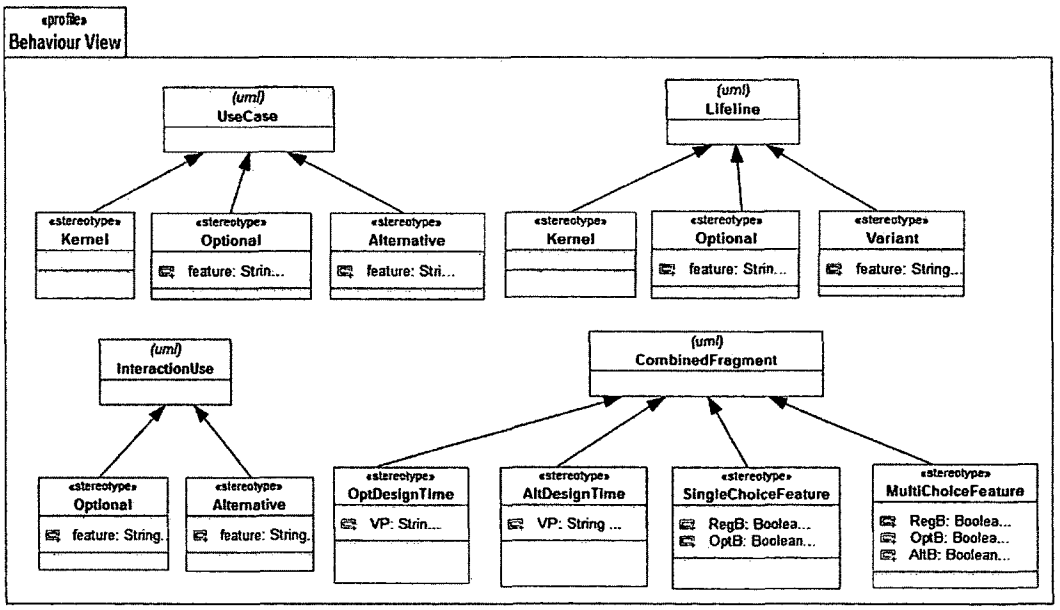
[ZIA06] T. Ziadi and J.-M. Jézéquel, "Software Product Lines, chapter Product Line Engineering with the UML: Deriving Products" pp. 557-586, Springer 2006.

[ZIA07] T. Ziadi and J.-M. Jézéquel, "Plibs: an eclipse-based tool for software product line behavior engineering" In Proceedings of the 3rd Workshop on Managing Variability for Software Product Lines, SPLC 2007, Kyoto, Japan, 2007.

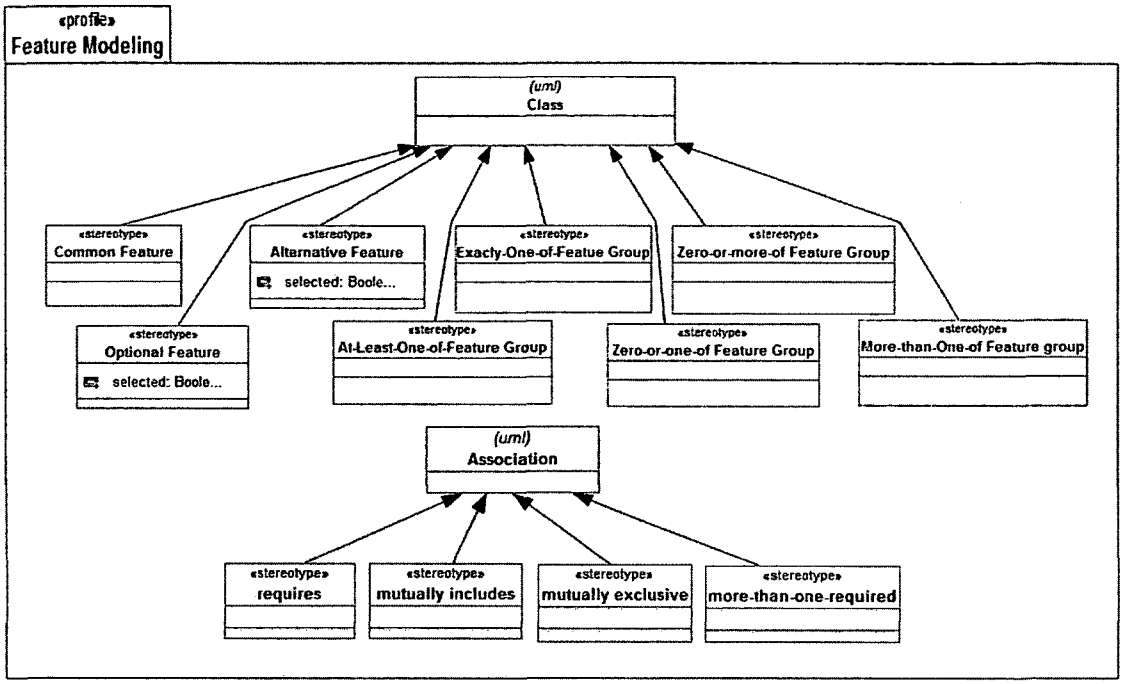
Appendix A: SPLV Profile



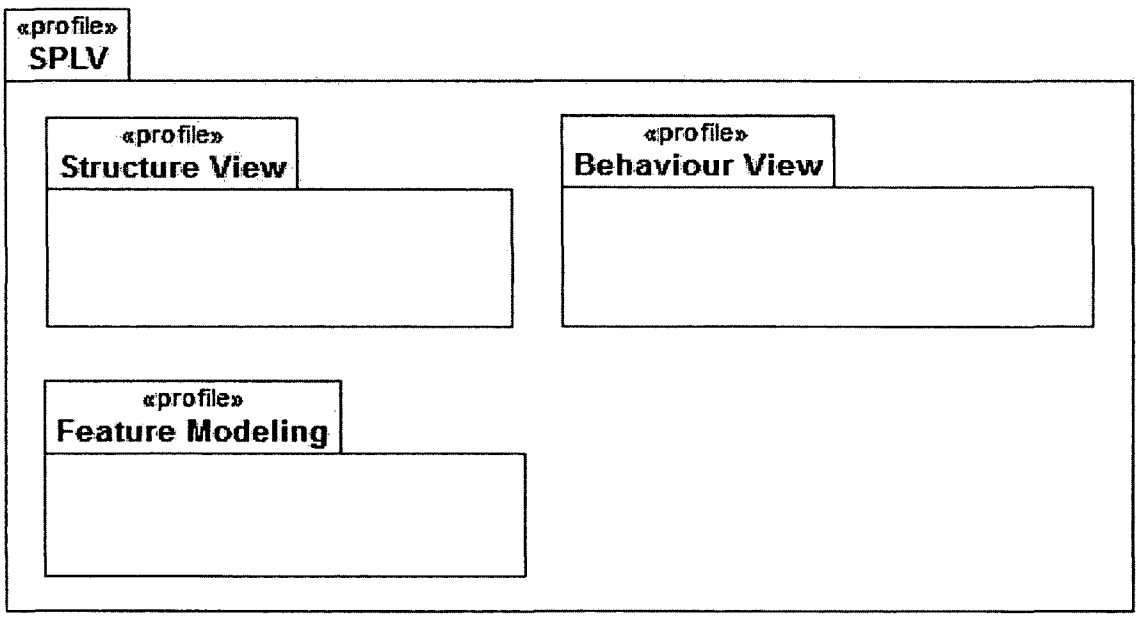
The Structure Package of the SPLV Profile.



The Behaviour Package of the SPLV Profile.



The Feature Modeling Package of the SPLV Profile.



The SPLV Profile Package.