UNIVERSITÉ DE BLIDA 1
Faculté des Sciences
Département d'Informatique

DOCTORAL THESIS
Option: Génie des Systèmes Informatiques

PLANNING AND DERIVATION OF PRODUCTS IN SOFTWARE PRODUCT LINE BASED ON SOFTWARE ARCHITECTURE

by
LAHIANI Nesrine

In front of a jury composed of:

Nadjia Benblidia	Professor, U.Blida1	President
Narhimene Boustia	Associate Professor, U.Blida1	Examiner
Nacim Chikhi	Associate Professor, U.Blida1	Examiner
Walid Hidouci	Professor, ESI Algiers	Examiner
Djamal Bennouar	Professor, U.Bouira	Supervisor

Blida, February 2019

ABSTRACT

Companies are more and more forced to customize their software products for completely different customers. In practice they often clone an existing system and adapt it to the customer's needs. In such scenarios software product lines promise benefits, for example, reduced maintenance effort, improved quality, and customizability. However, introducing new development processes into a company is risky and might not pay off . The other advantage is that this fairly recent software development paradigm allows companies to create efficiently a variety of complicated products with a short lead-time. This thesis focuses on product planning and derivation, which is the process of creating individual products (members) during application engineering using shared product family artifact.

Firstly, we propose to apply Model-Driven Engineering techniques to provide a systematization of the Domain Engineering therefore to enable the automation of the Application Engineering. Model-driven techniques commonly rely on the use of metamodeling as a means to automate model-to-text and model-to-model transformations. In this work, we use ATL as a model-to-model transformation language and Acceleo as a model-to-text transformation language.

Secondly, due to the fact that a generalization relation may also exist between product lines. We propose a new structure to represent a composite software product line that allows members of the composition to communicate and interact with each other. The hierarchical structure proposed is based on inheritance that provides an easily understandable representation. The aim of the representation is to derive multiple products using a simple but practical method.

Finally, we illustrate the application of our approaches in various case studies in the context of e-Government Product Lines, from the feature model of each different product line to the final application. By applying the proposed approach, it becomes feasible to derive a number of applications in a specific domain.

Keywords: Software Product Line, Product Derivation, Multi Software Product Lines, Software Architecture

ملخص

تضطر الشركات لتخصيص برمجيات مختلفة لزبائنها. وفي الممارسة العملية غالبا ما يستنسخ البرنامج ويكيف لاحتياجات العملاء. في مثل هذه السيناريوهات خطوط إنتاج البرمجيات لها فوائد عدة، على سبيل المثال: تقليص جهود الصيانةو تحسين نوعية والتخصيص. في معظم الحالات يتم تطبيق النهج الاستخراجي، والذي يتم إعادة هندسة الأنظمة القديمة لتنفيذ اعادة الاستخدام. ومع ذلك فإن إدخال عمليات إنمائية جديدة في الشركة أمر محفوف بالمخاطر وقد لا يؤتي ثماره. ولتفادي تنفيذ برنامج واحد، برزت خطوط منتجات البرمجيات كنهج إنمائي هام. ويجمع خط إنتاج البرمجيات بين هندسة البرمجيات وإعادة استخدام البرمجيات لبناء أنظمة معقدة وعالية الجودة. الميزة الأخرى هي أن هذا النموذج تطوير البرمجيات الحديثة يسمح للشركات تصميم مجموعة متنوعة من المنتجات المعقدة في مدة قصيرة. تركز هذه الأطروحة على تخطيط المنتجات واشتقاقها، وهي عملية إنشاء المنتجات الفردية (الأعضاء) خلال هندسة. أولا، نقترح تطبيق تقنيات هندسة نموذجية لتوفير نظام من هندسة المجال وبالتالي تمكين أتمتة هندسة التطبيقات. تعتمد التقنيات التي تعتمد على النماذج عادة على استخدام ميتامودلينغ كوسيلة لأتمتة التحولات من نموذج إلى نص ونموذج إلى نموذج. في هذا العمل، نستخدمَ ATL كلغة تحول نموذج إلى نموذج Acceleo كلغة التحول نموذج إلى نص. كلتا اللغتين تستند إلى قواعد، مما يبسط كل من مهمة استخراج المعلومات من ملف يمثل نموذجا ومهمة تحويل هذه المعلومات.

ثانيا، بما انه تجود علاقة التعميم أيضا بين خطوط الإنتاج. نقترح هيكلا جديدا لتمثيل خط منتج مركب البرمجيات التي تسمح لأعضاء التواصل والتفاعل مع بعضها البعض. ويستند الهيكل الهرمي المقترح إلى الميراث الذي يوفر تمثيلا يسهل فهمه. والهدف من التمثيل هو استخلاص منتجات متعددة باستخدام طريقة بسيطة ولكنها عملية. نوضح تطبيق نهجنا في مختلف دراسات الحالة في سياق خطوط المنتجات الحكومة الإلكترونية، من نموذج ميزة من كل خط إنتاج مختلفة إلى التطبيق النهائي. ومن خلال تطبيق النهج المقترح، يصبح من الممكن استخلاص عدد من التطبيقات في مجال معين.

الكلمات الرئيسية: خط برمجيات الانتاج، اشتقاق المنتج، تعدد خطوط برمجيات المنتج، هندسة البرمجيات .

RÉSUMÉ

Les entreprises sont de plus en plus obligées à personnaliser leurs produits logiciels pour des clients complètement différents. En pratique, ils clonent souvent un système existant et l'adaptent aux besoins du client. Dans de tels scénarios, les lignes de produits logiciels promettent des avantages, par exemple, une réduction de l'effort de maintenance, une qualité améliorée et une personnalisation. Cependant, l'introduction de nouveaux processus de développement dans une entreprise est risquée et pourrait ne pas compenser. L'autre avantage est que ce paradigme de développement de logiciels assez récent permet aux entreprises de créer efficacement une variété de produits compliqués avec un court délai. Cette thèse se concentre sur la planification et la dérivation des produits, qui est le processus de création de produits individuels (membres) lors de l'ingénierie des applications en utilisant des artefacts familiaux de produits partagés.

Tout d'abord, nous proposons d'appliquer les techniques d'ingénierie par les modèles (IDM) pour assurer une systématisation de 1er sous processus «l'ingénierie de domaine » afin de permettre l'automatisation de 2ème sous processus « la dérivation des produits». Les techniques axées sur les modèles reposent généralement sur l'utilisation du métamodèle comme moyen d'automatiser les transformations de modèle à texte et de modèle à modèle. Dans ce travail, nous utilisons ATL comme langage de transformation modèle-modèle et Acceleo comme langue de transformation modèle-texte.

Ensuite, du fait qu'une relation de généralisation peut également exister entre les lignes de produits. Nous proposons une nouvelle structure pour représenter une ligne de produits logiciels composites qui permet aux membres de la composition de communiquer et d'interagir les uns avec les autres. La structure hiérarchique proposée est basée sur l'héritage qui fournit une représentation facilement compréhensible. Le but de la représentation est de dériver de multiples produits en utilisant une méthode simple mais pratique.

Enfin, nous illustrons l'application de nos approches dans diverses études de cas dans le contexte des lignes de produits e-gouvernement , du modèle caractéristique de chaque ligne de produits différente à celle de l'application finale. En appliquant l'approche proposée, il devient possible de dériver un certain nombre d'applications dans un domaine spécifique

Mots-clés: Ligne de produits logiciels, Dérivation de produits, Multiple Lignes de produits logiciels, Architecture logicielle.

ACKNOWLEDGEMENTS

This thesis would not have been completed without the help of others. I would like to take this opportunity to express my gratitude towards them and acknowledge them.

Thanks to Allah for giving me this opportunity, the strength and the patience to complete my dissertation finally, after all the challenges and difficulties.

Next, all my gratitude goes to my advisor Pr. Djamal BENNOUAR. Working under your direction during all these years has been a great experience. Thanks for your guidance, your advice, your time and specially for being patient enough. In general, thank you for always having the right words and the good ideas to keep me motivated and in the good direction to get to the end of this research. It was an honor having Pr. Bennouar, an authority in the domains of software engineering and software product lines to be my advisor.

I am grateful to the jury members Pr. Nadjia BENBLIDIA, Pr. Walid HIDOUCI, Dr. Narhimene BOUSTIA and Dr. Nacim CHIKHI for having accepted to serve on my examination board and for the time they have invested in reading and evaluating this thesis.

Finally, I want to thank my family. First of all my parents, I would not be here without you. Thank you mom for supporting me all these years, for taking care of me .And to you dad, I know you would be proud of what we have achieved. Then of course, my brother and my sisters. I also would like to express my warmest and deepest appreciation to my husband, for his patience, assistance, continuous support and understanding in everything I done.; thank you all.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER1
INTRODUCTION

This chapter presents the context, motivation, objectives and scope of this work, as well as the thesis contributions.

The notion of software product lines (SPL) has received attention during the 1990s, and has proven itself in a large number of organizations. A software product line is a set of software- satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [1]. In a software product line context, software products are developed in two phases, i.e. a domain engineering process and an application engineering process. Domain engineering involves, amongst others, identifying commonalities and differences between product line members and implementing a set of shared software artifacts (e.g. components or classes) in such a way that the commonalities can be exploited economically, while at the same time the ability to vary the products is preserved. During application engineering individual products are derived from the product line, i.e. constructed using a subset of the shared software artifacts. If necessary, additional or replacement product specific assets may be created.

The process of creating these individual products during application engineering is known as Product Derivation. It is the process of constructing a product from a product line of software assets that is developed using shared product family artifact [2]. In a product line organization, the use of an effective product derivation process can help to ensure the return on investments required to develop the platform assets [3]. Product derivation is hence the focus of this dissertation, which reports on our work of investigating the state-of-the-art in this field and provides two industrials case studies which detail empirical evidences on how product derivation is performed in a real organization environment.

Product Derivation is the key activity in application engineering. In this context, this thesis proposes a model-driven product derivation approach based on Model-Driven Engineering principles [4, 5]. It addresses the construction of a concrete product from the product line, which includes the derivation of application artifacts from domain artifacts, for instance the derivation of Application Requirements from Domain Requirements, the derivation

of the Application Architecture from the Domain Architecture and the derivation of Application Components from Domain Components. Our big challenge is to produce adapted transformation programs to contain rules able to derive products with the desired user choices.

The remainder of this chapter describes the focus of this dissertation and starts by identifying the problems that motivate this research and our research goals in Section 1.1. Next, Section 1.2 explains the contributions described in this dissertation. In Section 1.3, we give a brief introduction to each of the chapters of the document. Finally, we list in Section 1.4 the publications made during the development of our work.

1.1 Problem Statement and Research Goals

The determination of the core assets of a software product line with the similarities and variability associated with this core represent a very important and strategic step in the process of implementing an SPL. Product planning and SPL in the derivation of product today seem not to have reached the highest degree of maturity hardware product lines (automotive, electronics).The heterogeneity of production plans would lead to a difficulty of building and composite product line severely limit interoperability. Also, the informal form of production plans, despite their accuracy for manipulating the human actors of SPL, makes it difficult to interpret in the context of automatic processing. However, product lines face a significant number of challenges.

These problems may be summarized in five points:

- How to specify a production strategy for an SPL?
- How to manage multiple and heterogeneous feature Models?
- How to define a generic and automatic derivation approach for SPL that best conform to stakeholders' wishes?
- How to structure, model and derive product from a composite SPls?
- How can several SPLs that belong to the same domain interact with each other?

To answers the questions asked above, this thesis focuses on reviewing current product derivation approaches, identifying their inadequacies and proposing more effective solutions. The integration of the concepts of software architecture and model-driven engineering in the process of raising the maturity level of planning and derivation is another aspect to be explored in the context of this work. The basic concepts of software architecture, namely component, connector and configuration concepts and model-driven engineering, should play an important role in the process of obtaining method, tools and

model that would make the approaches more effective Product scheduling and derivation specification and would increase the rate of automation of the derivation process.

The main objectives of this dissertation are summarized next:

- ✓ Plan a formal strategy for production in SPL.
- ✓ The integration of the concepts of software architecture and model-driven engineering in the process of raising the level of maturity of the planning and derivation.
- ✓ The introduction and support of component concept in Core Assets.
- ✓ Generic and automatic product derivation approach for SPL.
- ✓ Modeling composite SPLs and managing multiple feature modes.

1.2 Statement of the Contributions

As a result of this dissertation, the following contributions can be highlighted:

• **An analysis of the state-of-the-art of product derivation approaches:** This work presents an overview of related works and a framework to compare and classify Product Derivation approaches. Also, we give a review on current component-based software architecture approaches.

• **Model-Driven product derivation approach:** we apply Model-Driven Engineering techniques to provide a systematization of the Domain Engineering therefore to enable the automation of the Application Engineering. Model-driven techniques commonly rely on the use of metamodeling as a means to automate model-to-text and model-to-model transformations.

• **Composite software product lines case study**: a new structure to represent a composite software product line that allows members of the composition to communicate and interact with each other. The hierarchical structure proposed is based on inheritance that provides an easily understandable representation. The aim of the representation is to derive multiple products using a simple but practical method.

1.3 Dissertation Roadmap

The dissertation is divided in four parts. While this introductory chapter is part of the first part, the second one encloses the State of Art. The third part presents the contribution of this dissertation. Finally, the last part includes the conclusions and perspectives of this

dissertation. Below, we present an overview of the chapters that compose the different parts.

- Chapter 2 presents an overview on software product line engineering, its principles, foundations, architecture and adoption models;
- Chapter 3 depicts a survey on product derivation;
- Chapter 4 depicts a systematic review on component-based software architecture representing the current state-of-the-art in the area;
- Chapter 5 presents the proposed approach to derive product from an SPL based on the principals of MDA;
- Chapter 6 presents a new structure to represent a composite SPLs, to achieve the objectives and validate our results using an example that is a part of a composite e-Government Product Lines;
- Chapter 7 presents some concluding remarks about this work, its related work, and directions for future work.

1.4 Publications

We present below the list of research publications related to the work done while developing the approach described in this dissertation.

International Journal

- Lahiani, N., & Bennouar, D. A DSL-based approach to Product Derivation for Software Product Line. *Acta Informatica Pragensia*, 138-143. (2016). DOI: 10.18267/j.aip.90
- Lahiani, N., & Bennouar,D. A Brief Survey on Product Derivation Methods in Software Product. *mediterranean telecommunication journal*, Vol. 8, No 1 (2017).
- Lahiani, N., & Bennouar,D. Using inheritance to represent Hierarchical Software Product Lines. *Electronic Government an International Journal*. (2018). DOI: 10.1504/EG.2018.10015523
- Lahiani, N., & Bennouar,D. On the use of model transformation for the automation of product derivation process in SPL. *Acta Universitatis Sapientiae, Informatica,* 43−57 (2018). DOI: 10.2478/ausi-2018-0003

International Conferences

- Lahiani, N., & Bennouar, D, An MDA Based Derivation process for Software Product Lines, International Arab Conference on Information Technology (Acit2014), Oman, 2014.

- Lahiani, N., & Bennouar, D. Mapping Feature to IASA Architecture to derive product in Software product Line, International Conference on Advanced Communication Systems and Signal Processing (ICOSIP), Tlemcen, 2015.
- Lahiani, N., & Bennouar, D, A Model Driven Approach to Derive e-Learning Applications in Software Product Line, In the proceeding of Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication, Batna,2015. Doi:10.1145/2816839.2816850

CHAPTER2
WHAT IS SOFTWARE PRODUCT LINES?

2.1 Introduction

In this chapter, we discuss different domains and concepts applied in our proposal, including Software Product Line. The objective of this chapter is not to present an in-depth description of all the existing approaches and technologies surrounding these concerns, but to give a brief introduction to these concerns, used throughout the dissertation. This introduction aims at providing a better understanding of the background and context in which our work takes place, as well as the terminology and concepts presented in the next chapters.

The chapter is structured as follows; we present the main concepts of software product line engineering. Section 2.2 presents traditional software reuse vs. software product line. Section 2.3 describes the principles of SPL. Finally, Section 2.4 summarizes the ideas presented in this chapter.

2.2 Traditional Software Reuse vs. Software Product Line

Both traditional software development and software product line can rely on reuse of the assets. However, in traditional software development, reuse is not planned and traced. Despite the fact that, it may have reusable algorithms, components, methods used in the development which are stored in a repository, it can be longer to find and adapt a reusable asset from that repository than implementing a new functionality to the system.

Contrary to traditional software development, software product lines plan, manage and ensure reuse of assets. Decisions are made deliberately in a systematic way with a strategy. In addition to what has been said, according to the Software Engineering Institute's Framework 5.0 "When we speak of software product lines, we don't mean"

- fortuitous, small grained reuse
- Single system development with reuse
- just component-based or service-based development
- just a reconfigurable architecture
- releases and versions of single products
- just a set of technical standards

2.3 What is Software Product Lines?

There exist several models of software development processes, e.g., the V-model, the spiral model or the incremental model. Each of these models describes the different tasks or activities that take place during the process, required to build the final software. For instance, a traditional development process usually starts with the analysis of the customer's requirements, followed by several phases such as planning, implementation, testing and deployment. The aim of these processes is to develop one single software system at a time. By contrast, software product line engineering aims at building several similar software systems from a set of common elements.

A Software Product Lines can be defined as "is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way" [1]. It captures the commonalities between a set of products while providing for the differences between different instances of the product.

2.3.1 Principles and Benefits

Several benefits motivate the use of product line engineering to develop software systems. According to [6], the main motivations are:

- Reduction of development costs: An essential reason for introducing product line engineering is the reduction of costs [6]. When artifacts are reused in several different kinds of systems from the platform, rather than being developed from scratch to each product, implying in cost reduction. Thus, the fundamental idea of product lines is that it takes a family perspective instead of a single product perspective, which enables large scale reuse across the family members [7];
- Quality improvements: The product line adoption has also high influence on the quality of the resulting software [6] Since each product is resulted from a set of tested and reviewed assets, the number of defects expected to each new product can be considerably lower and consequently the quality level can be increased;
- Reduction of time-to-market: Several products are developed from a common set of assets, leading to significant reductions in both the development costs and time-to-market for individual products. However, product line engineering demands for a higher upfront investment, if compared to single-systems engineering, hence, the initial time-to-market may be higher since, in order to build the reusable platform it is necessary a long and dedicated time to the core assets development;

- Benefits for the customers: In a software product line, the products may be customized to specific customers in a simpler way, since variations among products are defined in anticipation, so that a same artifact may attend to different customer needs. Thus, customer can purchase products that fit their individual needs and wishes;
- Reduction of maintenance and evolution costs: When artifacts of the platform are changed (e.g. for the purpose of error correction) or new artifacts are added into it, these changes are propagated to all products derived from the platform. It usually leads to a simpler and cheaper maintenance and evolution, if compared to maintain and evolve a bunch of single products in a separate way;
- Improved cost estimation: When the reusable core assets are developed, the cost estimations for products from the product line are straightforward and do not include many risks. Consequently, the platform provides a sound basis for cost estimation.

2.3.2 Software Product Line Processes

Software product line engineering relies on a fundamental distinction of development *for reuse* and development *with reuse* as shown in Figure 2.1. In *domain engineering* (development for reuse) a basis is provided for the actual development of the individual products. As opposed to many traditional reuse approaches that focus on core assets, the product line infrastructure encompasses all assets that are relevant throughout the software development life-cycle [8].

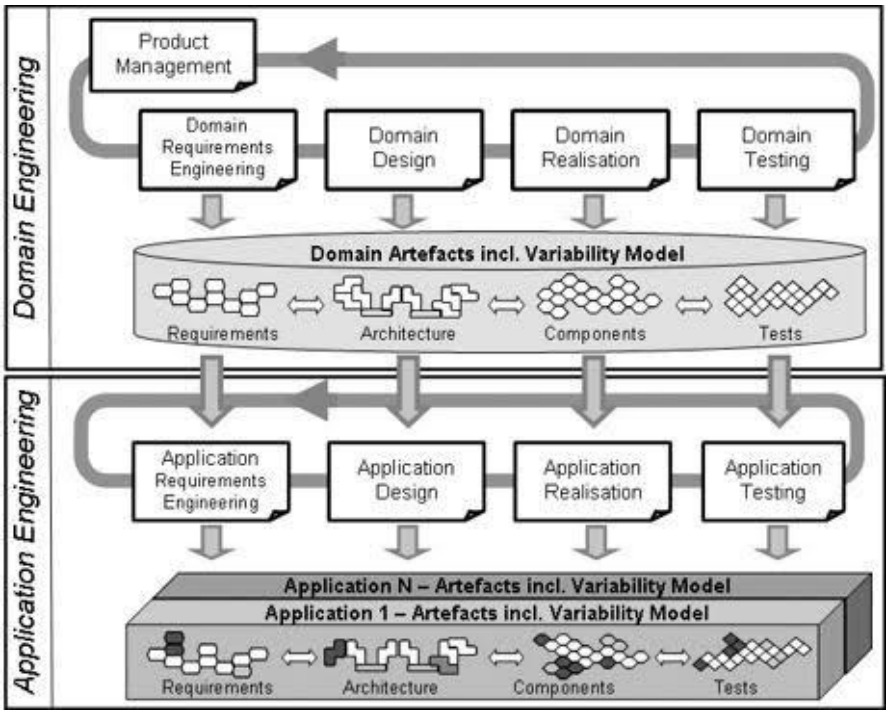

Figure 2.1: The software product line engineering framework**.**

Domain engineering focuses on the development of reusable assets that provide the necessary range of variability. As domain engineering continues as long as the product line exists, the underlying software development approach must be able to cope with long-term, highly complex system development.

According to [8] the activities within domain engineering are as follows:

- *Product management:* this activity aims to define the products that will constitute the product line as a whole. In particular, it aims at identifying the major commonalities and variabilities among the products. The major output of this activity is the product roadmap.
- *Domain requirements engineering*: this activity starts with the product roadmap and aims at a comprehensive analysis of the requirements for the various products in the product line. It captures these requirements, identifies commonalities and variabilities and constructs an initial variability model, which supports the further development steps.
- *Domain design*: starting from the requirements model, this activity aims at developing the product line architecture (or reference architecture).
- *Domain realisation*: this activity encompasses detailed design and implementation of the reusable software components. At this stage the planned variability which has

been expressed as a requirement must be realized with adequate implementation mechanisms.

- *Domain testing*: this aims at validating the generic, reusable components that were implemented as a result of the previous activity. Domain testing is much more difficult than testing in a single system context, mainly for two reasons: the implemented variability must be taken into account and there is no specific product which provides an integration context. In addition, domain testing also generates reusable test assets that can be reused in application testing.

As a result domain engineering sets up the common product line infrastructure, including all required variability.

Application engineering focuses on the development of the individual systems on top of the platform. As a large part of development effort and complexity is moved to domain engineering, this activity – and thus the underlying life-cycle model – will usually be profoundly different as it will not need to cope with so much complexity and the development will not span so much time. On the other hand, application engineering is directly involved with the customer and thus will often need to deal with much more rapid changes. As a consequence, a life-cycle model that is able to cope rapidly with changes is required.

According to [8] application engineering consists of the following activities:

- *Application Requirements Engineering*: This aims at identifying the specific requirements for an individual product. As opposed to single system requirements engineering, this starts from the existing commonalities and variabilities. It is thus the goal of this activity to stay as close as possible to the existing product line infrastructure.
- *Application Design*: This activity derives an instance of the reference architecture, which conforms to the requirements identified in the previous step. On top of this product-specific adaptations are built. Thus, as far as reusable components are concerned, the architecture is consistent with the reference architecture, enabling plug-and-play reuse.
- *Application Realisation*: Based on the available requirements and architecture, the final implementation of the product is developed. This includes reuse and configuration of existing components as well as building new components corresponding to product-specific functionality.

- *Application Testing*: In this step, the final product is validated against the application requirements. Similar to the previous steps, this builds on reusable assets from the corresponding domain activity.

While the details of the integration of domain engineering and application engineering will strongly depend on the situation, it is important to keep the two apart in terms of different types of activities that are typically performed with different quality criteria and objectives in mind.

2.3.3 Feature Modeling

When managing variability in a product line, Linden, F. at al. in [8] distinguish three main types:

1. *Commonality*: a characteristic (functionality or non-functional) can be common to all products in the product line. We call this a commonality. This is then implemented as part of the platform.
2. *Variability*: a characteristic may be common to some products, but not to all. It must then be explicitly modeled as a possible variability and must be implemented in a way that allows having it in selected products only.
3. *Product-specific*: a characteristic may be part of only one product – at least for the foreseeable future. Such specialties are often not required by the market per se, but are due to the concerns of individual customers. While these variabilities will not be integrated into the platform, the platform must be able to support them.

Feature modeling is a well-known technique for representing the concepts of a software domain. In fact, systematic reuse and domain driven approaches, such as Software Product Lines [1] and Generative Programming [9] rely on some kind of feature based notation.

Perhaps for that reason, several notations for feature modeling have emerged since it was introduced by [10]. Besides that, they are often used to represent which products belong to the SPL scope. In order to do that, a feature model describes the relevant features (or concepts) of a domain and details the constraints among those features. A valid member of an SPL satisfies all constraints defined in the corresponding feature model. For instance, consider the feature model of the *eShop Product Line* depicted in Figure 2.2. Note that it follows a tree-like notation where the parent-child relationships are categorized as:

- Mandatory relationships represent that whenever a parent feature is selected, the child feature must also be selected. The *Payment feature* on Figure 2.2 is mandatory.

- Optional relationships mean that a parent feature does not imply the child feature. The *Search feature* is optional.
- Inclusive or relationships define that at least one child of a parent feature must be select. For instance, a product might be configured with different payment methods.
- Alternative or relationships define that one, and only one child of a parent feature must be selected. For instance, only one of the available security options might be available for a given product.

Besides the parent-child relationships, feature models also have global constraints, such as (CreditCard implies High), stating that if the feature Credit Card is selected, High Security is also selected.

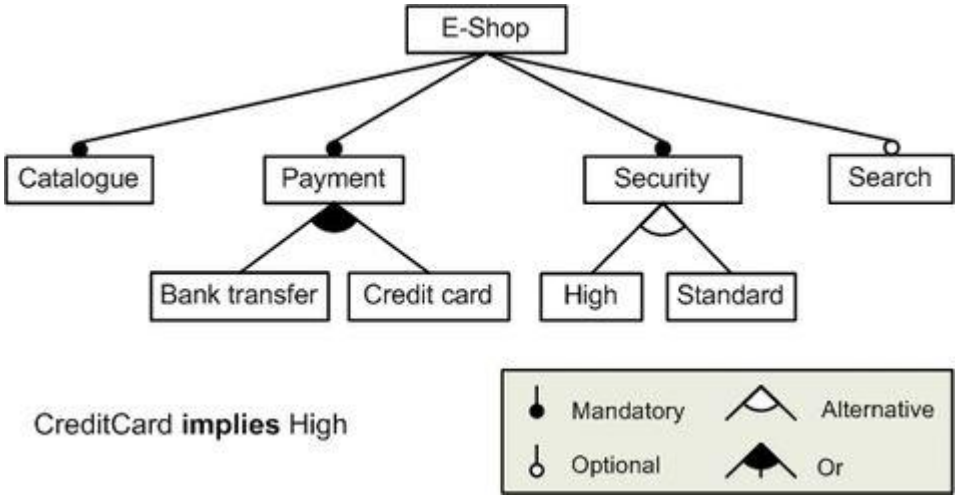

Figure 2.2: e-Shop Feature Model.

2.4 Chapter Summary

In this chapter, we have briefly introduced on software product line concepts. It included fundamental aspects of software product lines and some of motivations for applying it as a suitable software development strategy, highlighting its economic benefits that can be achieved across the large scale and planned reuse of family members. It is exactly these production economies that make software product lines attractive [1].

Next, we will present an overview on the product derivation area discussing their fundamental concepts and challenges. Besides, we present six product derivation methods and a comparison framework.

CHAPTER 3
PRODUCT DERIVATION

3.1 Introduction

Product Derivation (PD) is one of the central activities in Software Product Lines [11]. The PD is the process of constructing a product from a product line of software assets that is developed using shared product family artifacts [2]. In a product line organization, the use of an effective product derivation process can help to ensure the return of investment required to develop the platform assets [12]. Unfortunately, the existing product derivation approaches and tools have developed with different goals, for different purposes, and in different domains. Some approaches apply model-driven development techniques; while others boil down to a collection of guidelines or, in many cases, they provide a high-level methodology or process framework[12].The fact is that, although there are a considerable number of approaches, which consider product derivation, still there are many challenges to be overcome within product derivation field.

The remainder of this chapter is structured as follows: Section 3.2 provides background knowledge on product derivation. The challenges and difficulties are presented in Section 3.3. Then, the five product derivation methods are briefly presented in Section 3.4. Section 3.5 we introduce a comparative framework for evaluating product derivation methods and then are compared against the framework in Section 3.6. Section 3.7 summarizes the ideas presented in this chapter.

3.2 PRODUCT DERIVATION IN SPL

Rather than describing a single software system, the model of a software product line (SPL) describes the set of products in the same domain. This is done by distinguishing elements shared by all the products of the line, and elements that may vary from one product to another.

Software products are developed, in the context of product line engineering, according to two separate processes, namely domain engineering and application engineering. The former is dedicated to core asset development while the latter is aimed at yielding products.

We focus in this thesis at application engineering known also as product derivation (PD). PD has been defined in many different ways, McGregor in [13] defines it by "*Product derivation is the focus of a software product line organization and its exact form contributes heavily to the achievement of targeted goals*".

Deelstra et al. in [2] define product derivation by, "*A product is said to be derived from a product family if it is developed using shared product family artifacts. The term product derivation therefore refers to the complete process of constructing a product from product family software assets*".

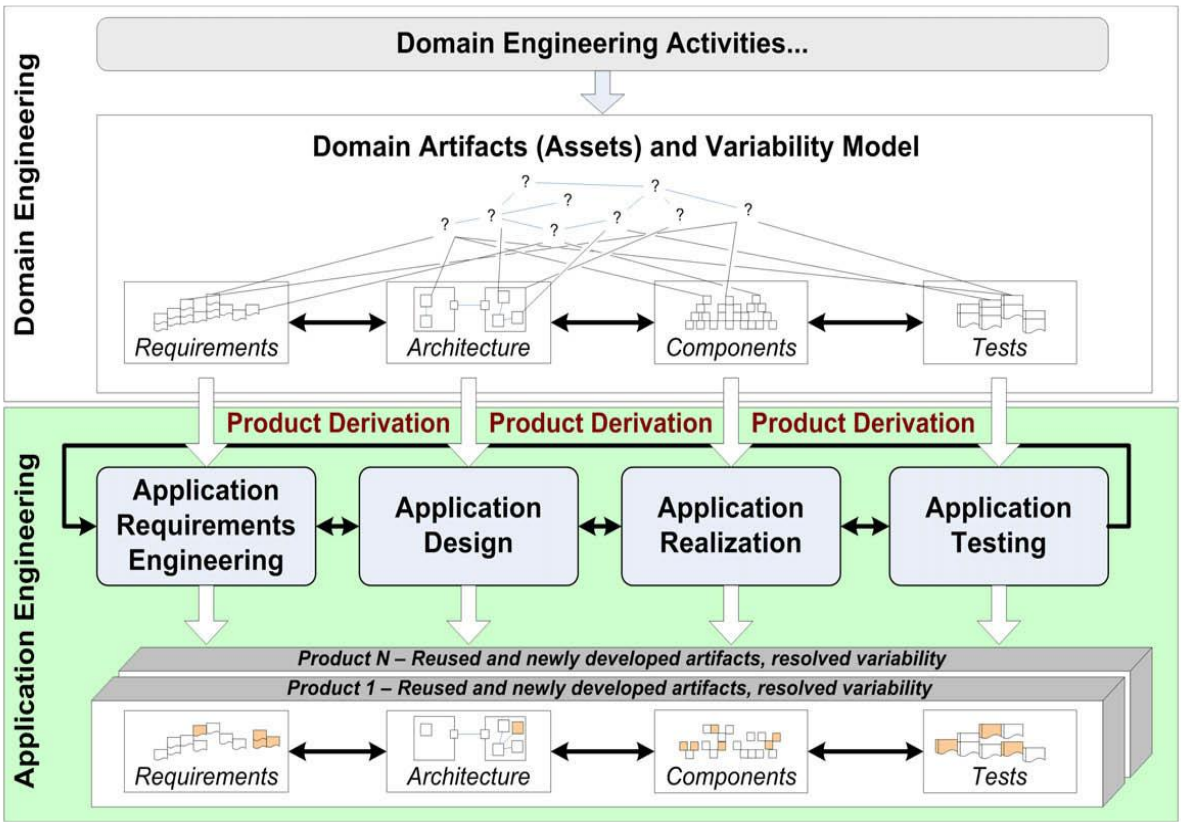

Figure 3.1 : SPLE processes.The upper white vertical arrows represent the product derivation process of selecting and customizing reusable assets during application engineering.

Figure 3.1 depicts a high-level application engineering process. The upper white vertical arrows represent the product derivation process of selecting and customizing reusable assets during application engineering. The lower white arrows indicate deployment activities necessary to arrive at a final product (e.g., deploying and integrating new components developed to address customer requirements with the existing derived components).

A well-defined domain engineering associated with a systematic product derivation process, where adequate tools support the activities can lead to automatic generation of products. On the other hand, even with all of these attributes, in many cases, the additional developments are necessary, since that, in practice, many requirements are not accounted for in the shared product family artefacts and can only be accommodated by adaptation of existing components and assets or even new development [14]. Thus, an effective product derivation approach must also consider those cases. Much of the SPL research to date has been focused on the domain engineering activities in a way that application engineers can derive the products with greater ease.

However, there are few research dedicated to the product derivation process [12]. In the same way, there are only few reports available about how the software development organizations derive their products from a product lines. These and other difficulties and challenges are discussed in next sections.

3.3 The Challenges and Difficulties

Deelstra et al. affirms “there is a lack of methodological support for application engineering and, consequently, organizations fail to exploit the full benefits of software product families” [2]. Rabiser et al. identified that in comparison with the great amount of research results on domain engineering activities, only few approaches and tools are available for product derivation. Much of the SPL research to date has been focused more on how to scope, define, and develop product lines rather than on how to effectively utilize them [15]. These issues demonstrate the real necessity of more researches on product derivation field.

Despite the importance of product derivation, there are difficulties associated with the process. Hotz et al. describes it as “slow and error prone, even if no new development is involved” [16]. Griss identifies the inherent complexity and the coordination required in the derivation process by affirming that “. . . as a product is defined by selecting a group of features, a carefully coordinated and complicated mixture of parts of different components are involved” [17]. Therefore, the derivation of individual products from shared assets is still a time-consuming and expensive activity in many organizations [2] Rabiser et al. enforce this point when they claim “guidance and support are needed to increase efficiency and to deal with the complexity of product derivation” [15].

According to Rabiser et al., the area of product derivation is still considered immature [15]. Existing approaches do not give detailed information on the strategies for product

customization, resolving variability, and database model derivation. Besides, the existing studies provide few details on the required steps, techniques and practices to a product derivation process. Additionally, there is little support for the derivation process

3. 4 Overview of Product Derivation Approaches

We first provide a brief overview of the six selected approaches and then show the results of our analysis and comparison in Section 3.5 based on the adapted evaluation framework introduced in Section 3.4.

PuLSE-I. PulSE (Product Line Software Engineering) is a full life-cycle product line [18]; it is centred on three main elements: the deployment phase, the technical components and the support components.

PulSE (Product Line Software Engineering) is a full life-cycle product line [14]; it is centered on three main elements: the deployment phase, the technical components and the support components.

The PuLSE-I is the application engineering process of PuLSE, it is centered on the instantiation of the product line infrastructure (the I in PuLSE-I stands for instantiation) [19].

PuLSE-I details how a single product can be built efficiently from the reusable product line infrastructure built during the other PuLSE activities. The trigger of PuLSE-I is a customer or the management having a product request that can be satisfied by the product line (i.e., the requested product is potentially in the scope of the product line).

Method Process. The method is divided into five main areas of activity, or phases:

- **Plan for the product line instance (the product):** Determine whether all characteristics of the required product are covered by the product line.
- **Create project plan:** Define what is product-specific and what can be fulfilled by the product line.
- **Instantiate and validate product line model:** Incrementally resolve decisions defined in the product line model (representing variation points).
- **Instantiate and validate reference architecture:** Instantiate variability to derive an ‘‘intermediate architecture’’ from the product line, validate, and then modify if necessary.
- **Product construction:** Lower level design, implementation, and testing based on reusable assets.

DREAM. Kim et al. in [20] proposed a methodology called DREAM a practical product line methodology stands for DRamatically Effective Application development Methodology. DREAM adopts the key activities of SPL and model transformation feature of MDA. The

methodology allows semi-automatically development of a large number of applications that vary on behaviour and implementation platform.

However, Kim et al. do not give any details about models used during phases nor the transformation tool. Moreover, a high level description of the activities and no guidance is provided on how the approach could be applied.

Method Process. The process consists of 9 phases, and each phase produced various artifacts with different characteristics. The derivation process starts after three phases of framework engineering; it used the model created during framework engineering.

- Application Requirement Analysis.
- Application Specific Design.
- Framework Instantiation.
- Model Integration.
- Application Detailed Design.
- Application Implementation.

KobrA. The KobrA method stands for Komponentenbasierte Anwendungsentwicklung that is German for “component-based application development” [21]. The approach has been developed by *Fraunhofer Institute for Experimental Software Engineering (IESE)* [22].

KobrA integrates two approaches component-based and software product line into a unified approach. It designed for modeling architecture, for developing both single and family system and to support a model driven architecture (MDA).

KobrA and PuLSE have a relationship between their activities which made KobrA an object-oriented customization of the PuLSE method. The approach does not provide any semi-automatic product derivation does not provide a process support, neither a support for the development of a specific product.

Method Process. Application engineering uses the framework built during framework engineering to construct specific applications in the domain covered by the framework. The application engineering process is split into two primary steps:

- **Context realization instantiation:** It starts when the software development organization has established an initial contact to a potential customer who is interested in a software system in the domain of one of the organization's frameworks. The outputs of this process are the context decisions and a concrete realization of the application's context.

- **Framework instantiation:** It starts when the application context realization is (partially) created and thus also the context decisions (partially) exist. The context decisions are used to initially instantiate the generic.

COVAMOF. Deelstra et al in [2] developed an approach called COVAMOF (Configuration in Industrial Product Families VAriability Modeling Framework).

COVAMOF is supported by a tool-suite, called COVAMOF- VS This tool-suite is implemented as a combination of Add-Ins for Microsoft Visual Studio .NET. It is designed for creating variability models of a product family, and using these models for configuration of individual products. It provides variation point and dependency views on variability models and allows defining, configuring, and realizing products following the COVAMOF derivation process.

Method Process. The COVAMOF derivation process is an instance of the generic derivation process except it does not focus on the difference between the initial and iterative phase, but it divided the process into four main steps as presented in [23]:

- **Product definition:** Defining customer and product name.
- **Product configuration:** Binding of variation points based on customer requirements.
- **Product realisation:** Tool-based translation of the configuration of the variability model to a configuration of an executable product.
- **Product testing:** Determining whether the product meets the customer requirements and deciding whether an additional iteration (product configuration/realisation/testing) is required.

Pro-PD. Through a series of research phases using sources in industry and academia, O'Leary in [3] has developed a process reference model for product derivation (Pro-PD) in Lero (the Irish Software Engineering Research Center) with a specific goal defining a process reference model for product derivation as a foundation for situation-specific process approaches to product derivation.

Method Process. Pro-PD is structured around five essential activities: Initiate project, identify and refine requirements, derive the product, develop the product, test the product, and management and assessment. Each of these activities contains roles, tasks and artefacts used to derive products from a software product line. Tasks are units of works that consume or produce one or more products, Pro PD groups related tasks into activities, each

activity has a specific goal, inputs and outputs. These tasks roles are assigned to human activity

DOPLER^{UCon}. (Decision-Oriented Product Line Engineering for effective Reuse: User-centered Configuration) has been developed by Mag. Rick Rabiser in [24] at Christian Doppler Laboratory for Automated Software Engineering. It is a tool-supported approach for product configuration with capabilities for adapting and augmenting variability models to guide sales people and application engineers through product derivation. Particular emphasis lies on support for requirements acquisition and management. The approach also aims to make variability models accessible to "non-technicians" such as sales people or customers to fully exploit the benefits of PLE.

Method Process. DOPLER^{UCon} process contains six different activities: (1) Domain expert prepare product configuration by creating the derivation model. (2) Users defined in the derivation model perform the actual product configuration by taking decisions visible to them. (3) In parallel, they capture arising product-specific requirements. (4) Based on these requirements developers conduct additional development. (5) Finally, engineers integrate new developments with the selected and customized product and deploy it for the customer.

3.5 Evaluation Framework

An evaluation framework is introduced in Table 3.1. To evaluate each approach we identify a set of criteria. We start the evaluation from the element of the problem situation, i.e. the approach context (input requirement and output product of the method). The second category is the problem solving process, i.e. what does the method itself contain (process, artifacts, tool...). The last category brings the two previous categories together through self-evaluation to evaluate the method output.

The goal of our evaluation is to provide an overview of current product derivation in SPL and find out if - and how - the methods differ from their each other. With various questions this study tries to address, e.g. maturity, practicality and scope of the methods to find differences. On the other hand the goal was to study if the methods really have what it takes to call them a derivation method. These elements were considered in the category of 'contents' by questioning if the methods satisfy the definition of a method. Framework elements were refined to cover features special for product line methods (e.g. modeling variability).

We briefly define each element framework next:

- Requirement specification: customer's requirements must be specified before using it as input, the specification could be textual, model…
- Final Product: the product at the end of the process could be an executable application, a hardware …
- Method Process: the activities that must be followed to derive a product that meets customer's requirements.
- Artifacts: During the derivation process each method produces a set of artifacts (reference architecture, test, documents...).
- Tool used: some sub-process need tools as eclipse or to model UML.
- Modelling variability is to efficiently describe more than one variant of a system. Variability can be expressed in stand-alone models, such as feature diagrams.
- Method maturity validates a method by a case study that could be industrial or academic show how the method is mature or not.

A more complete summary of this section is given in Table 3.1.

Table 3.1: The categories and the framework elements for Characterization and Comparison of product derivation methods.

Framework element	**Description**
Requirement specification	What is the trigger of the approach?
Final Product	What is expected at the end?
Method Process	What are the activities needed to derive a product?
Artifacts	What are the artifacts created and used during derivation process?
Tool used	What are the tools supporting the method?
Modeling variability	What kind of model does the approach use to modeling variability?
Method maturity	Has the method been validated in practical industrial cases?
Framework element	Description

Next we present the six product derivation methods and how this framework can be used to compare each method against a criterion.

3.6 Analysis and Discussion

The purpose of this discussion is to offer guidelines related to the selection of the most suitable method for product derivation. We selected six approaches for a product derivation in software product line, not to rate which one is the best but to characterize, to compare and to see how they differentiate and resemble. The purpose of this discussion is to analyse the results obtained.

We summarize the results of our analysis and comparison of PuLSE-I, DREAM, KobrA, Pro-PD, and DOPLER^{UCon} using the evaluation framework first introduced in Section 4 as it is shown in Table 3.2.

Some observations are evident first the context of the approaches; for specific goal, we can identify in all approaches a collective goal, which is “satisfy customer’s needs”. Then same for the input we can observe that to start the process each method needs “customer’s requirements” which is almost the same for all methods, it’s differ in the format e.g. Pro-PD translates this requirement before using it. Likewise the output of each approach is almost the same which is the final product that meets customer's requirements.

Moreover, the contents of the approaches, at this stage we can see how the approaches differ even though the goal is the same, but no method is similar to another. Some are abstract and difficult to apply e.g. dream that did not give any guidance or process support, moreover any details about tools used during the process.

The last criterion was validation, Pulse-I and Dream do not give any details about how they validate their processes. For the others, they have been validated in practical industrial case studies, academic or both as DOPLER^{UCon} did.

Finally, in practice how to know which method is the most suitable this is not the purpose of our research, but to select the right one each criterion must fits with problem context.

Table 3.2: Analysis and comparison of Product Derivation Approaches.

Approaches	Framework Element
	How does the method specify customer's requirements?
PuLSE-I-	A detailed project plan that considers the set of characteristics upon which the customer (or the marketing) and the developers have agreed. Each required characteristic is specified by a detailed description of how it will be supported.
DREAM	Translated customer requirement created based on customer requirements and a glossary.
KobrA	Decision models are used to capture the variabilities within applications in the product line.
COVAMOF	The engineer creates a new Product entity in the variability model. The properties of Product entities are the customer, a unique name for the product, and variation points that have been bound for the product
Pro-PD	Translate the Customer Requirements into the internal organisational language. The Product Analyst used a customer terminology Glossary
DOPLER^{UCon}	Domain experts create a derivation model which can have a name, description, and purpose. It is further on used to present decisions in the variability model to decision-makers during product configuration, to store their decisions, and to capture product-specific requirements.
	What are we expected at the end?
PuLSE-I-	The delivery process depends on the market size, For the mass market, the system must first be packaged together with an installation guide to enable any customer to install it at his/her machine. For small markets, like for individual systems for single customers, the system is installed by the developers at the customer's site.
DREAM	Produce executable application code and associated implementations.
KobrA	The final results are the application decisions consisting of the context decisions and the Komponent hierarchy decisions, together with the application realization and the application tree.
COVAMOF	A configuration of an executable product that satisfies all customer requirements.
Pro-PD	Product Release that satisfy all customer requirement.
DOPLER^{UCon}	Product delivered and/or to be installed for the customer
	What are the artifacts created and used during derivation process?
PuLSE-I-	Project plan
	Domain decision model instance

	Architecture decision model instance
	Low level configuration
	Code
	Test results
DREAM	Application Analysis Model
	Application Specific Design
	Instantiated Framework
	Integrated Application Model
	Detailed Design Model
	Application Code
KobrA	Context decisions
	Instance of generic Komponent hierarchy
COVAMOF	Product entity
	XML-based feature models
	C++/C# source files
Pro-PD	Two different kind of artifact:
	Software artifact.
	Documentation artifact.
DOPLER^{UCon}	Derivation model
	Code source files
	Specific–asset
	What are the tools supporting the method?
PuLSE-I-	**DIVERSITY/CDA** visualizes the impact of decision made which enable an immediate validation.
DREAM	Not reported
KobrA	Commercial UML tool
COVAMOF	**COVAMOF-VS** provides two main graphical views, i.e. the variation point view, and the dependency.
Pro-PD	Eclipse Process Framework(EPF) researcher used it to model the derivation product process and create a formalised version of Pro-PD

DOPLER^{UCon}	**ProjectKing** The configuration preparation activity of DOPLER^{UCon} is supported by the tool ProjectKing. Domain experts like project and sales managers use the tool to create derivation models based on variability models. **ConfigurationWizard** Domain experts and (software) engineers use the tool to review and communicate available variability, resolve variability, customize assets by taking decisions, generate configurations, capture product-specific requirements, relate product-specific requirements to the existing variability, and generate requirements specifications
	What kind of model does the approach use to modeling variability?
PuLSE-I-	**PuLSE-CDA** (Customizable Domain Analysis)[6].
DREAM	Not specified. Any reasonable representation scheme may be used.
KobrA	**UML** in KobrA, each Komponent (KobrA component) in the framework is described by a suite of UML diagrams as if it were an independent system in its own right [17].
COVAMOF	**COVAMOF** a variability modeling framework that models variability in terms of Variation Points and Dependencies [11].
Pro-PD	Not reported
DOPLER^{UCon}	**DOPLER^{VM}** the approach was influenced by earlier work by [18] and the Synthesis project [19] and comprises two main elements: asset and decision.
	Has the method been validated in practical industrial cases?
PuLSE-I-	The PuLSE approach has been applied in case studies, for example [20] and [10]
DREAM	Not reported
KobrA	The domain of Enterprise Resource Planning. It used in the development of the KobrA workbench.
COVAMOF	• Deelstra, S., Sinnema, M., Bosch, J., 2005. Product derivation in software product families: a case study [3] • Industrial validation of COVAMOF. J. Syst. Software [21].
Pro-PD	• Industrial case study : ROBERT BOSCH GMBH • further validation of Pro-PD through an inter-model evaluation with the SEI Product Line Practice Framework
DOPLER^{UCon}	• Siemens VAI : The case study focuses on SVAI's automation software for the continuous casting technology. • Business Software BMD.

3.7 Chapter Summary

In this chapter we have shown the real level of the research at this moment, in this domain, by presenting and discussing six of the most representative product derivation methods. We presented a framework that focused on comparing and evaluating different product derivation approaches.

Next, we will present a reference framework for software component models, which defines (and explains) terms of reference that we will use throughout this thesis. The definitions are general, and should therefore be universally applicable. Also, we will present six current software-Component models in details.

CHAPTER 4
COMPONENT-BASED SOFTWARE DEVELOPMENT

4.1 Introduction

The concept of component has been around in the computer hardware industry for a long time. To build a computer, hardware engineers no longer design tiny, basic elements from scratch. They simply plug off the-shelf components such as chips, boards, or cards together. Component-based development has brought a number of benefits to hardware engineers such as reusability, maintainability, flexibility, and integration readiness. Due to the constraint of time and budget, software engineers have sought similar techniques for software development leading in recent years to various techniques for building software from components. Component models like COM and CORBA (Common Object Request Broker Architecture) allow software engineers to plug together components in different languages and platforms. End-users are also benefiting from these technologies: for example, spreadsheet, word processing, drawing and database applications often use a component model to embed editable data from one application into the files created and managed by another [27,28 and 29].

The remainder of this chapter is organized as follows: Section 4.2 defines CBD and present a reference framework for software component models. Section 4.3 gives an overview of six current software component models. We summarize the ideas presented in this chapter in Section 4.4.

4.2 Component-Based Development

Component-based approach has in last year's shown considerable success in many application domains. CBD is a concept well known (almost inevitable) and proven in development of hardware systems. It is based on developing complex systems out of smaller, well defined components. Although the use of CBD principles in developing software has advanced in the last decade, Component-based software engineering (CBSE) still has a lot of room for improvement.

Main goals of CBD are [29]:

- reuse of components, thus shortening time-to-market of new systems;

- making the systems easier to maintain and upgrade by making their components easily replaceable and deployable;
- making the system development easier and more reliable by predicting system properties from the properties of its components.

In CBD systems are built by combining components using their interfaces. To connect interfaces of two components, contracts of those interfaces must be satisfied. To ensure that the components can be deployed to a component framework that will support them at run-time, and that they can interact with each other, component models are used.

4.2.1 What is a Component?

The word “component” is used very broadly and often loosely throughout the software industries. Generically, a component is defined as a computational unit [31]. Components can be things like clients and servers, databases, filters, and layers in a hierarchical system.

One of the most popular definitions of a component was offered by a working group at ECOOP (the European Conference on Object-Oriented Programming) [32]:

“*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties”*.

This definition emphasizes component composition. As a unit of composition, each component has its specified interface that determines how it can be composed with other components.

Sterling in [33] extended the above definition then distinguished three aspects of a component:

- A specification that describes what the component does and how it should be used.
- A packaging perspective in which a component is considered as a unit of delivery.
- An integrity perspective in which a component is considered as an encapsulation boundary.

They then defined a component simply as:

“A software package which offers services through interfaces”.

Although these definitions differ in detail, their proposers would probably agree that a component is an independent software package that provides functionality. Moreover, they

all emphasize the importance of well-defined interfaces. The interface could be an export interface through which a component provides functionality to other components or an import interface through which a component gains services from other components. All these definitions also emphasize the “black box” nature of a component: that is, a software engineer can use one to create a larger system without any knowledge of how it is implemented.

Most important characteristic of components are [30]:

- Reusability, the ability of a component to be used in different systems;
- Usability by third parties;
- Seamless replacement of a component with newer or different components.

To achieve this it is necessary to separate the implementation of a component from its interface.

4.2.2 What is an Interface?

Interfaces can be viewed as access points through which the components can exchange information and cooperate with other components or the component framework.

Interfaces can generally be divided into two groups: provided and required interfaces. Provided interfaces define services that a component can provide to other components or the framework. Required interfaces define services that the component requires from other components or the framework.

As already stated, it is essential that the component's interfaces are separated from the implementation of the component's services. They are used just to list and describe those services. By having the services listed and described, it is much easier to combine components into complex systems. The separation of interface and implementation enables a component to be replaced with new component that provides the same interface.

In most modern component models (e.g. COM, JavaBeans, .NET) interface's description gives just syntactical information. In many cases this information is not sufficient and the need of contractual definition of interfaces arises.

4.2.3 Contracts

Interface semantics can be viewed as contracts between the provider and the user of the interface [34]. Through a contract, user of the interface obliges to constraints

(preconditions) that the provider sets, and in respect to that, provider of the interface guarantees some functional and/or non-functional properties.

Hierarchically, we can divide contract definition in four levels [29]:

- **Level 1:** Syntactic interface. A list of operations including types of their inputs and outputs. Using the knowledge about types of inputs and outputs, type safety can be established. Type safety ensures that no run-time error will occur from usage of operation with wrong type of object.
- **Level 2:** Constraints on values of parameters and of persistent state variables. These constraints can be viewed as pre- and post- conditions for an operation. An example would be a range for the value of variable.
- **Level 3:** Synchronization between different services and method calls. This level describes the ordering between different interactions at the component interface. It also enables the interaction between the component and its environment to be non-atomic. Automata, temporal logic, process algebras or sequence diagrams can be used for the description.
- **Level 4:** Extra-functional properties. Level 4 describes properties like latency, worst case execution time, memory usage, reliability, robustness and availability. These properties are of great importance in real-time, embedded and safety-critical domains. By knowing the extra-functional properties of the components that will be used in the system, some properties of the system can be derived before it is actually built.

4.2.4 Component Model

Component model imposes a set of conventions that the components using that model must adhere to. With that conventions, component model ensures that the components can be deployed to the component framework and interact with each other.

Types of rules that component models define:

- Types of components that can be used;
- How the components interact with each other;
- How the components bind resources.

In a way, component models define the architecture of systems. That limits the flexibility of the system, but also speeds up the process of the development because new architecture does not have to be created.

4.2.5 Component framework

Component framework is a run-time infrastructure that upholds the component model [29]. It manages resources for components and supports component interaction.

Component framework can be viewed as an operating system for the components [29]. From that viewpoint, components are to framework what processes are to the operating system. The difference is that the component frameworks are more compact than operating systems. They are specialized to support a limited range of component types and interactions between those types. By limiting the diversity, component composition becomes simpler, more robust and more predictable. Another difference between component frameworks and operating systems is that the implementation of the framework does not have to be completely separated from the components. It is possible that a part of the framework is implemented by components themselves.

4.3 Current Software Component Models

In this section we describe current software component models. We have selected 6 component models that we encountered in the research literature and in practice.

4.3.1 CCM (CORBA Component Model)

CORBA Component Model [35] evolved from CORBA object model and it was introduced as a basic model of the OMG's component specification. The CCM specification defines an abstract model, a programming model, a packaging model, a deployment model, an execution model and a metamodel.

Component There are two levels of components [35]: *basic* and *extended*. Both are managed by component homes, but they differ in the capabilities they can offer. Basic components essentially provide a simple mechanism to "componentize" a regular CORBA object. Extended components, on the other hand, provide a richer set of functionality. A basic component is very similar in functionality to an EJB as defined in the Enterprise JavaBeans 1.1 specification. This allows much easier mapping and integration at this level.

Ports Components support a variety of surface features through which clients and other elements of an application environment may interact with a component. These surface features are called *ports*. The component model supports four basic kinds of ports [35]:

- Facets, which are distinct, named interfaces provided by the component for client interaction.
- Receptacles, which are named connection points that describe the component's ability to use a reference supplied by some external agent.
- Event sources, which are named connection points that emit events of a specified type to one or more interested event consumers, or to an event channel.
- Event sinks, which are named connection points into which events of a specified type may be pushed.
- Attributes, which are named values exposed through accessor and mutator operations. Attributes are primarily intended to be used for component configuration, although they may be used in a variety of other ways.

Basic components are not allowed to offer facets, receptacles, event sources and sinks. They may only offer attributes. Extended components may offer any type of port.

Components and Facets A component can provide multiple object references, called *facets*, which are capable of supporting distinct (i.e., unrelated by inheritance) IDL interfaces. The component has a single distinguished reference whose interface conforms to the component definition. This reference supports an interface, called the component's *equivalent interface* that manifests the component's surface features to clients. The equivalent interface allows clients to navigate among the component's facets, and to connect to the component's ports. Basic components cannot support facets; therefore attempts to navigate to other facets will always fail. The equivalent interface of a basic component is the only object available with which a client may interact. The other interfaces provided by the component are referred to as *facets*. Figure 4.1 illustrates the relationship between the component and its facets.

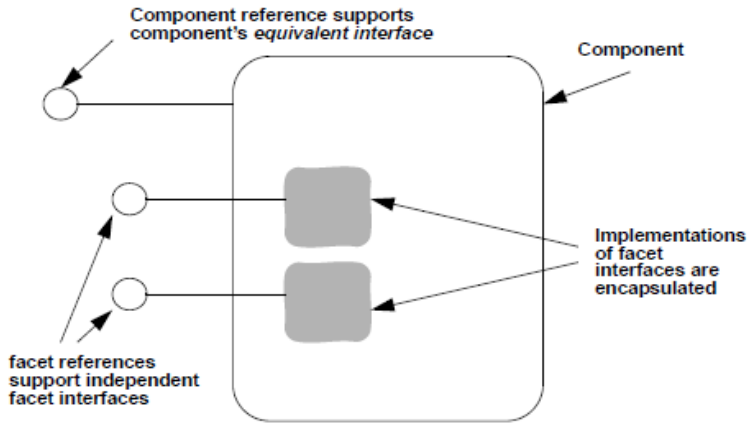

Figure 4.1: Component Interfaces and Facets.

The relationship between the component and its facets is characterized by the following observations [35]:

- The implementations of the facet interfaces are encapsulated by the component, and considered to be “parts” of the component. The internal structure of a component is opaque to clients.
- Clients can navigate from any facet to the component equivalent interface, and can obtain any facet from the component equivalent interface.
- Clients can reliably determine whether any two references belong to the same component instance.
- The life cycle of a facet is bounded by the life cycle of its owning component.

Component Identity A component instance is identified primarily by its component reference, and secondarily by its set of facet references (if any). The component model provides operations to determine whether two references belong to the same component instance, and operations to navigate among a component’s references.

Component Homes is meta-type that acts as a manager for instances of a specified component type. Component home interfaces provide operations to manage component life cycles, and optionally, to manage associations between component instances and primary key values. A component home may be thought of as a manager for the extent of a type (within the scope of a container). A home must be declared for every component declaration.

4.3.2 Koala

Koala is a component model developed by Philips for building software for consumer electronics. Koala components are units of design, development and reuse. Koala has a set of modeling languages: Koala IDL is used to specify Koala component interfaces; its Component Definition Language (CDL) is used to define Koala components, and Koala Data Definition Language (DDL) is used to specify local data of components. Koala components communicate with their environment or other components only through explicit interfaces statically connected at design time. Koala targets C as implementation language and uses source code components with simple interaction model. Koala pays special attention to resource usage such as static memory consumption.

Koala Components are units of design, development, and—more importantly—reuse. Although they can be very small, the components usually require many person-months of development effort. A component communicates with its environment through interfaces.

As in COM and Java, a Koala interface is a small set of semantically related functions. A component provides functionality through interfaces, and to do so may require functionality from its environment through interfaces. In our model, components access all external functionality through requires interfaces—even general services such as memory management [36].

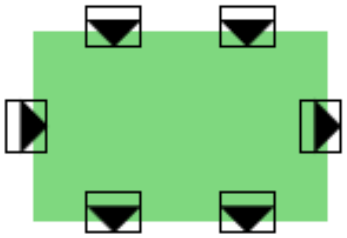

Figure 4.2: Koala Component.

In Koala, a component is represented in Figure 4.2. Interfaces is represented as squares with triangles, the tip of triangle represents the direction of function call. A Koala component's interface specifies the signature of a set of functions implemented by the component.

Koala Interface is a small set of semantically related functions (like in COM). To be more precise (again), an interface type is a syntactic and semantic description of an interface, and an interface instance is an interface occurring in a component. An interface type is described in an interface description language. As it is shown a simple IDL is used , resembling COM and Java interface descriptions, in which the function prototypes in a C syntax is listed [36].

```
Interface VolumeControl {

Void setVolume(Volume v);

Volume getVolume(void);}
```

4.3.3 JavaBeans

Developed by Sun Microsystems is based on Java programming language. In the JavaBeans specification a bean is defined by [37] as:

"A Java Bean is a reusable software component that can be manipulated visually in a builder tool".

Programming a Java component requires definition of three sets of data: i) properties (similar to the attributes of a class); ii) methods; and iii) events which are an alternative to method invocation for sending data. JavaBeans was primarily designed for the construction of graphical user interface. The model defines three types of interaction points, referred to as ports: (i) methods, as in Java, (ii) properties, used to parameterize the component at composition time, (iii) event sources, and event sinks (called listeners) for event-based communication.

There are a range of different kinds of JavaBeans components [37]:

1. Some JavaBean components will be used as building blocks in composing applications. So a user may be using some kind of builder tool to connect together and customize a set of JavaBean components s to act as an application. Thus for example, an AWT button would be a Bean.
2. Some JavaBean components will be more like regular applications, which may then be composed together into compound documents. So a spreadsheet Bean might be embedded inside a Web page.

Here is a simple code to write a *SimpleBean* Bean component:

```
import java.awt.*;

import java.io.Serializable;

public class SimpleBean extends Canvas

                implements Serializable{

 //Constructor sets inherited properties

 public SimpleBean(){

  setSize(60,40);

  setBackground(Color.red); }
```

SimpleBean extends the java.awt.Canvas component. *SimpleBean* also implements the java.io.Serializable interface, a requirement for all Beans. Setting the background color and component size is all that *SimpleBean* does.

4.3.4 Fractal

The FRACTAL component model [38] is a general component model that is intended to implement, deploy, and manage (i.e. monitor, control, and dynamically configure) complex software systems, including in particular operating systems and middleware. This motivates the main features of the model.

- Composite components (components that contain sub-components), in order to have a uniform view of applications at various levels of abstraction.
- Shared components (sub-components of multiple enclosing composite components), in order to model resources and resource sharing while maintaining component encapsulation.
- Introspection capabilities, in or der to monitor and control the execution of a running system.
- Re-configuration capabilities, in order to deploy and dynamically configure a system.

To allow programmers to tune the control of reflective features of components to the requirements of their applications, FRACTAL is defined as an extensible system. Control features of components are not predetermined in the model, rather the model allows for a continuum of reflective features or levels of control, ranging from no control (black-boxes, standard objects) to full-fledged introspection and intercession capabilities (including, e.g., access and manipulation of component contents, control over components' life-cycle and behavior, etc.) [39].

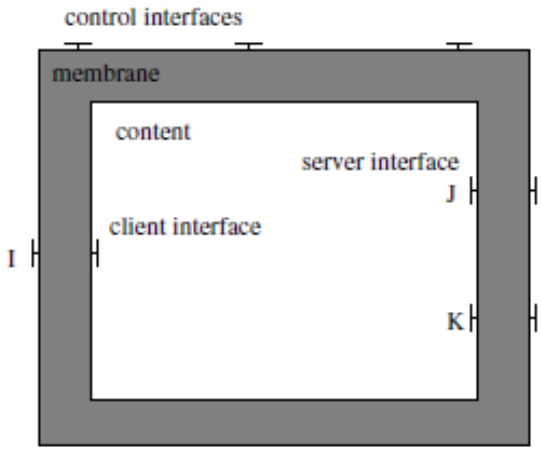

Figure 4.3: A Fractal Component.

As Figure 4.3 shows A *FRACTAL component* can be understood generally as being composed of a membrane , which supports interfaces to introspect and reconfigure its

internal features, and a content, which consists of a finite set of other components (called sub-components). The membrane of a component can have external and internal interfaces. External interfaces are accessible from outside the component, while internal interfaces are only accessible from the component's sub-components.

The membrane of a component is typically composed of several controllers. Typically, a membrane can provide an explicit and causally connected representation of the component's sub-components and superpose a control behavior to the behavior of the component's sub-components, including suspending, check pointing, and resuming activities of these sub-components. Controllers can also play the role of interceptors. Interceptors are used to export the external interface of a subcomponent as an external interface of the parent component. They can intercept the incoming and outgoing operation invocations of an exported interface and they can add additional behavior to the handling of such invocations (e.g. pre- and post-handlers). Each component membrane can thus be seen as implementing a particular semantics of composition for the component's sub-components. Controllers can be understood as meta-objects or meta-groups as they appear in reflective languages and systems [39].

4.3.5 IASA

IASA (Integrated Approach Software Architecture) is an approach to software architecture that allows the specification of aspect-oriented software architectures [40]. IASA defines a set of concepts that allows the specification of software architecture in a very flexible way with a high degree of freedom from any software mechanism constraint, allowing the specification of software architecture in a way that approaches the mental model of the architect. The full description of the approach IASA is detailed in [41].

The IASA component model defines a specific organization either for the external view applicable to any component (primitive, composite, COTS, legacy code) or for the internal view [42]. The external view is represented by the concept of envelope. The internal view consists of two parts as shown in Figure 4.4:

The operative part: appears at the top of the IASA component graphics notation, contains the components achieving the objectives of the core business aspect. Any component which has an internal structure different from the IASA internal organization of component is said a primitive component. COTS, legacy code and component written in a programming language are examples of primitive components.

The control part: appears at the bottom of the IASA component graphics not action, is composed of a controller, which is a specific component dedicated to control the operative part and a number of components handling the technical aspects). The controller is a mandatory component of the control part. The components handling technical aspects are usually called aspect components.

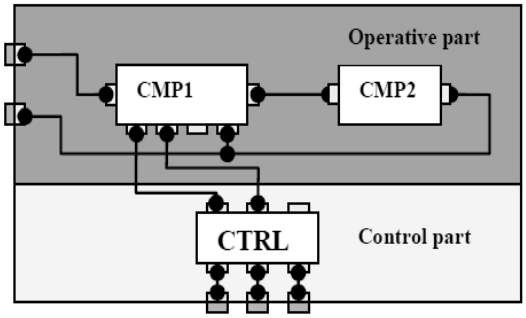

Figure 4.4: The IASA Component Model [41].

In IASA, a component interacts with the external world through a set of ports. A port has a structure made of access points and a behavior. The instantiation of a component is realized in the context of the envelope concept. An envelope is used to isolate the pure instance of a component from its operating environment by avoiding it with the necessary elements for the operation of the proceeding. The main graphic notations used by IASA are presented in Figure 4.5.

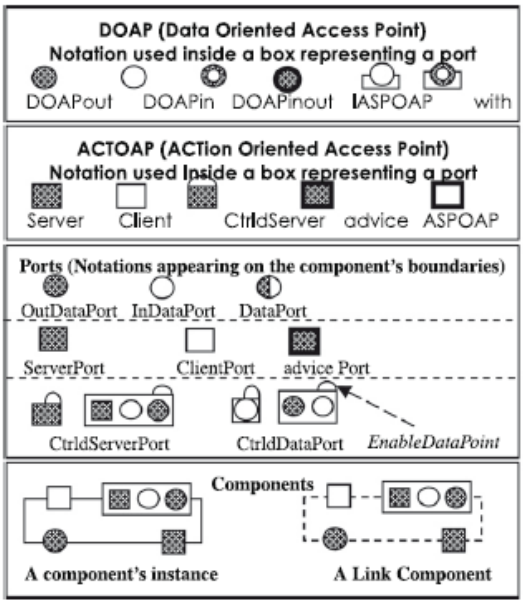

Figure 4.5: The main graphic notations used by IASA [40].

4.3.6 UML 2.0

UML 2 builds on the already highly successful UML 1.x standard, which has become an industry standard for modeling, design and construction of software systems as well as more generalized business and scientific processes. UML 2 defines 13 basic diagrams Types.

Component diagrams are used to model higher level or more complex structures, usually built up from one or more classes, and providing a well-defined interface. The component diagram's main purpose is to show the structural relationships between the components of a system. In UML 1.1, a component represented implementation items, such as files and executable. Unfortunately, this conflicted with the more common use of the term component, which refers to things such as COM components. Over time and across successive releases of UML, the original UML meaning of components was mostly lost. UML 2 officially changes the essential meaning of the component concept; in UML 2, components are considered autonomous, encapsulated units within a system or subsystem that provide one or more interfaces. Although the UML 2 specification does not strictly state it, components are larger design units that represent things that will typically be implemented using replaceable modules. But, unlike UML 1.x, components are now strictly logical, design-time constructs. The idea is that you can easily reuse and/or substitute a different component implementation in your designs because a component encapsulates behavior and implements specified interfaces.

In UML 2 they define a component as:

*"**Component** is an encapsulated unit within a system which provides one or more interfaces. When using components to model the **logical** architecture (solely in component diagrams) of a system the term 'component' refers to collection of classes which can be reused and replaced as a whole, when a single logical component can scattered around multiple physical nodes. When using components to model the **physical** architecture of a system (usually in deployment diagrams, but some people that are still custom to UML 1.x still use it in component diagrams) the term 'component' refers to dll, or some executable".*

In UML 2, a component is represented as rectangle with optional compartments stacked vertically. A high-level, abstracted view of a component in UML 2 can be modeled as just a rectangle with the component's name and the component stereotype text and/or icon.

Figure 4.6 shows different ways a component can be represented using the UML 2 specification.

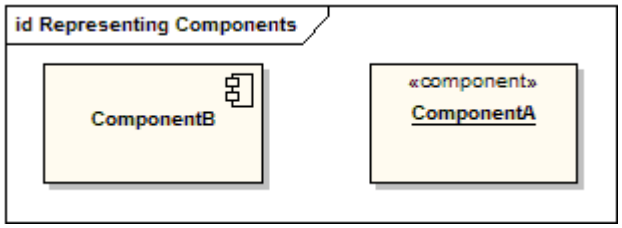

Figure 4.6: The UML 2 Component representation.

The assembly connector bridges a component's required interface (Component1) with the provided interface of another component (Component2); this allows one component to provide the services that another component requires as it is shown in Figure 4.7.

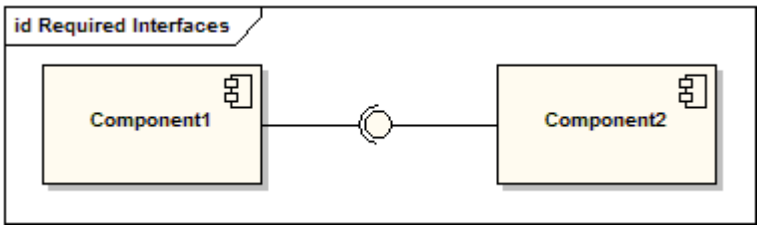

Figure 4.7: Required Interfaces in UML 2.

Using Ports with component diagrams as it is shown in Figure 4.8 allows for a service or behavior to be specified to its environment as well as a service or behavior that a component requires. Ports may specify inputs and outputs as they can operate bi-directionally.

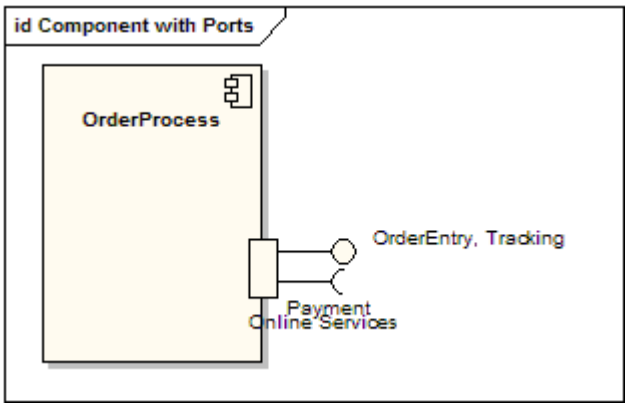

Figure 4.8: Components with Ports in UML 2.

In the next Chapter, we are going to use UML 2.0 as a Software component model to represent architecture references and product architecture.

4.4 Chapter summary

In this chapter, we have presented a survey of Software Component Models. We start by defining the principles that are in every component model.

The next chapter presents our proposal of Model-Driven product derivation approach including the mechanisms we developed for sorting out the drawbacks found in related work.

CHAPTER 5
MODEL-DRIVEN PRODUCT DERIVATION APPROACH

5.1 Introduction

In the previous part (Stat-of-art), we have presented a background on what we need in our work. We have also discussed the current product derivation approaches and shown also the limitations of each one and how are dedicated to a specific domain, in aim to propose a generic product derivation.

This chapter first introduces in Section 5.1 the production planning we proposed to produce and derive product from SPL. Then, we present in Section 5.2 presented how MDE can be used to enhance SPL Engineering. We show that MDE can be used to support the derivation of product line members. Section 5.3 describes in details the Model-Driven Product Derivation Approach proposed. At the end, we summaries the main ideas of this chapter.

5.2 Our Production Planning

In this section we propose a planning to produce product in SPL. We introduce in this section the production planning that we need before the development of a product derivation approach.

Each SPL organization has to follow a strategy to build products in efficient way as it is shown in Figure 5.1.

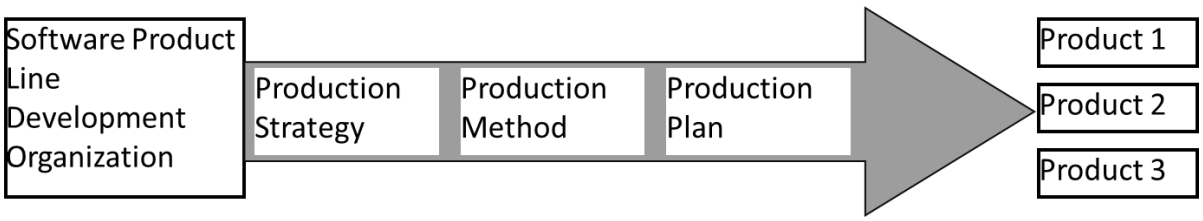

Figure 5.1: Production Planning.

We ask the following questions:

1. How can product development satisfy the organization's goals for the software product line?
2. What processes, models, and technologies can be used to ensure consistency across the core assets?
3. What does the product developer need to know to effectively utilize the core assets to develop products?

The production strategy is a high-level answer to the first question. The production strategy coordinates the design and use of the core assets. It begins as an informal notion, evolves concurrently with the core assets, and is ultimately documented in the production plan. The production strategy is based on the product line goals and influenced by the technologies to be used during production. This strategy specifies techniques and conditions for product development that support those goals.

Our main Goals are: *(1) reduced development time* and *(2) improved quality,* so our strategy is used automation wherever possible.

The Production method is the answer to the second question. A production method specifies a complete set of processes, models, and technologies to be utilized in product production and that satisfies the production strategy.

To satisfy the production strategy we are going to combine Model-Driven Engineering and Software Product Line in aim to automate the product derivation.

Finally, ***the production plan*** answers the last question. Each core asset has an attached process that is created by the core-asset developer and that describes how the core asset is used in product production. The production plan is a description of how the attached processes cooperate to yield a product.

The production plan specifies the following:

- inputs needed to build a product;
- activities that result in a completed product ;
- roles and responsibilities of the product developers;
- interactions needed with other groups in the organization;
- schedule and resources associated with building the product.

The product developer needs the production plan to be:

- **Efficient.** All activities in the production process are required to produce the specific product being developed;

- **Complete**. All information that is needed is in the production plan;
- Understandable. The information in the production plan is usable without outside assistance;
- Usable. The product developer is able to locate needed information quickly and easily.

5.3 Model-Driven Software Product Line

Model Driven Development (MDD) is a relatively new paradigm where models are central in the development. Model Driven Architecture (MDA) is a framework for software development proposed by the Object Management Group (OMG) in 2001[43] (i.e., MDA is a concrete realization of MDD). The notion of Model Driven Engineering (MDE) emerged later as a paradigm generalizing the MDA approach for software development [44].

5.3.1 Definition

Model-driven is a paradigm where models are used to develop software. This process is driven by model specifications and by transformations among models. It is the ability to transform among different model representations that differentiates the use of models for sketching out a design from a more extensive model-driven software engineering process where models yield implementation artifacts. Model Driven Architecture provides specific means for using models to accomplish the understanding, design, construction, deployment, maintenance and modification of software.

MDA's modeling techniques distinguish between business and technical aspects. This advocates that the designer must first capture the business concerns of the system in a model, called the Platform-Independent Model (PIM), while abstracting away technical details. Then, the PIM is transformed into a Platform-Specific Model (PSM) by introducing technical aspects of the target platform (in an MDA context). In general, a key challenge is the transformation of these models.

This transformation is usually specified by a set of precise mapping rules (more shortly). Finally, the resulting PSM can be used to generate implementation code

5.3.2 Why MDE?

Model-Driven Engineering (MDE) aims at reducing the accidental complexity associated with developing complex software-intensive systems [45]. In general, model-driven is a paradigm to reuse specific patterns or domains of software development. This emerges

through the extensive use of models, which replaces cumbersome (and usually repetitive) implementation activities. In this way, model-driven approaches improve development practices by accelerating them. According to Koniti, specific benefits of MDE are [46]:

1. productivity,
2. reduce cost,
3. portability,
4. reduced development time,
5. improved quality.

Overall, the main economic reason behind model-driven is the productivity gain achieved, which is reported by some studies [47].

5.3.3 Combination MDE-SPL

The main idea is to represent all the 4 phases of application engineering by MDA model as shown in Figure 5.2. The requirements for the system are modeled in a computation independent model, CIM describing the situation in which the system will be used [48]. After that, application design is transformed in a platform independent model, a PIM, is built. It describes the system, but does not show details of its use of its platform [48]. Then integrate both models into one PIM model. Application detailed design is modeled in PSM the platform specific model produced by the transformation is a model of the same system specified by the PIM; it also specifies how that system makes use of the chosen platform [48]. Finally, the PSM obtained contained all the information necessary to produce computer program code.

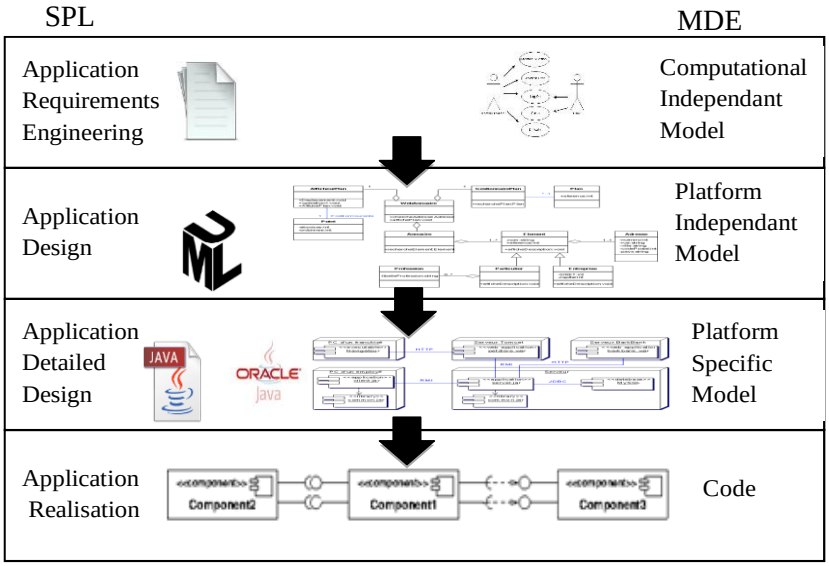

Figure 5.2: Combination of MDE and SPL.

This thesis adheres to the MDE principles, in particular for domain specific language design. We use the meta-modelling technique when addressing the definition of modelling languages. Moreover, the concept of model transformation is also extensively used. Hence, the notions of model, meta-model, model conformance and model transformation are major concerns on which this thesis relies on for the achievement of its contributions.

5.4 Our Model-Driven Product Derivation Approach

Our approach is founded on the principles and techniques of software product lines and model driven engineering. Figure 5.3 illustrates the main elements of our approach and their respective relationships.

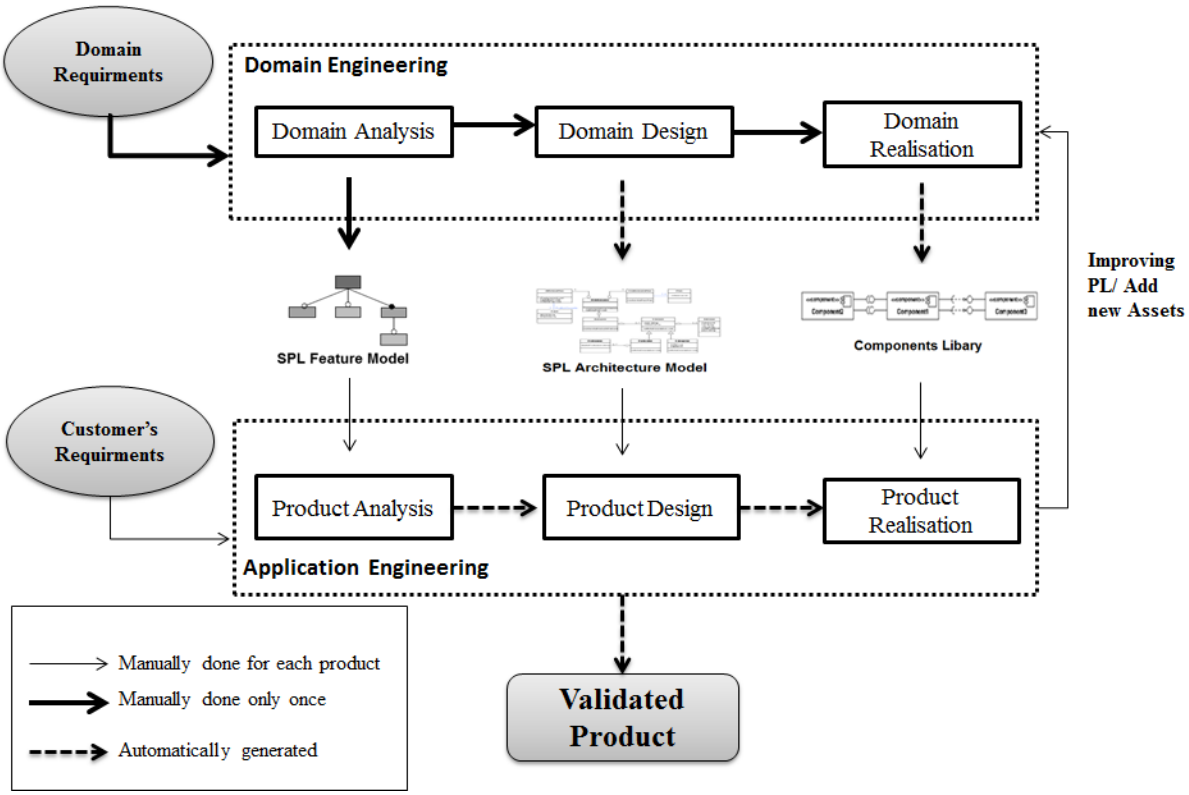

Figure 5.3: Overview of our approach.

5.4.1 Domain Engineering

***Domain Analysis*.** Domain analysis [49] or Feature modelling is the first activity to define the commonality and variability that can be expected to occur among the SPL members identified in the product line's scope. The main goals of domain requirements engineering are the development of common and variable domain requirements. We use feature model

[50] to present the similarities and variations among the products identified in the product line's scope that can be expected to occur.

To build our metamodel we modify the metamodel proposed by Czarnecki et al. [50] below by adding operation as class for the purpose of the transformation rules; we depict it in Figure 5.4. All Features in the Feature Model have distinct names and may have composing members.

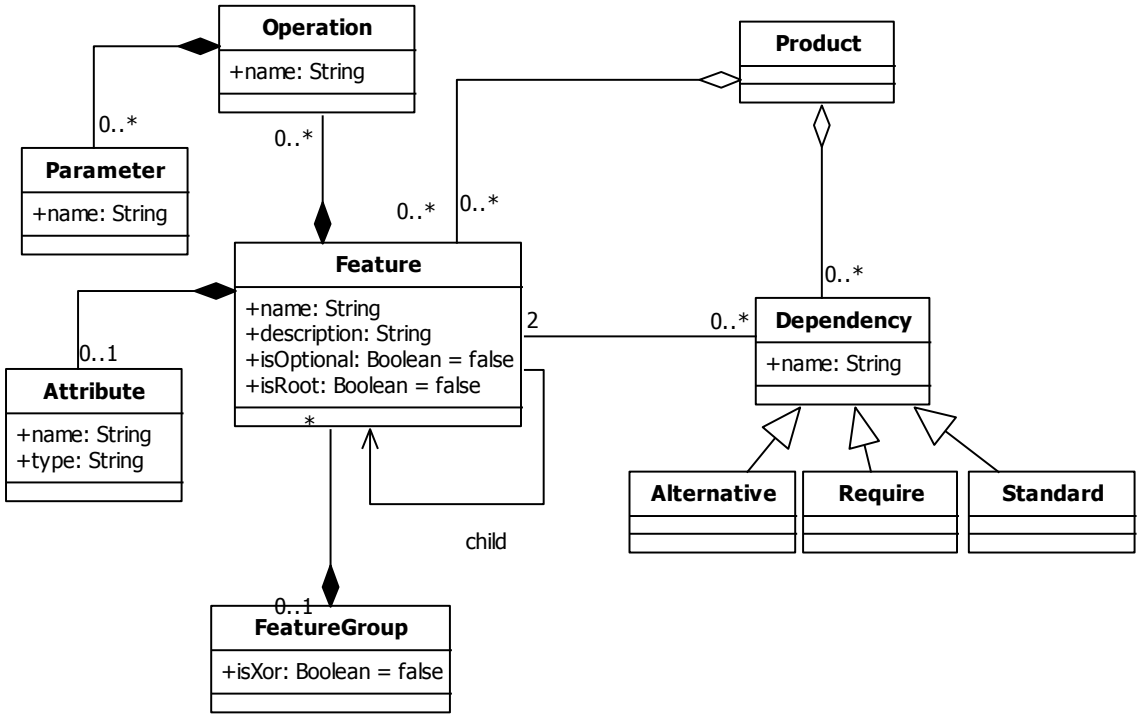

Figure 5.4: UML metamodel for feature models.

As an example, we define the feature model for e-Health Product Line, as Figure 5.5 shown as a result of the metamodel shown in Figure 5.4 doctors could connect via the application to follow up (1) remote consultation (via phone/message) and (2) manage patient's accounts. Patient also must do (3) a registration so that he/she can consult and (4) pay using its own credit card or just by bank transfer which are alternative features only one could be chosen. Drug refill and offline consultation are two optional features that could be chosen or just left.

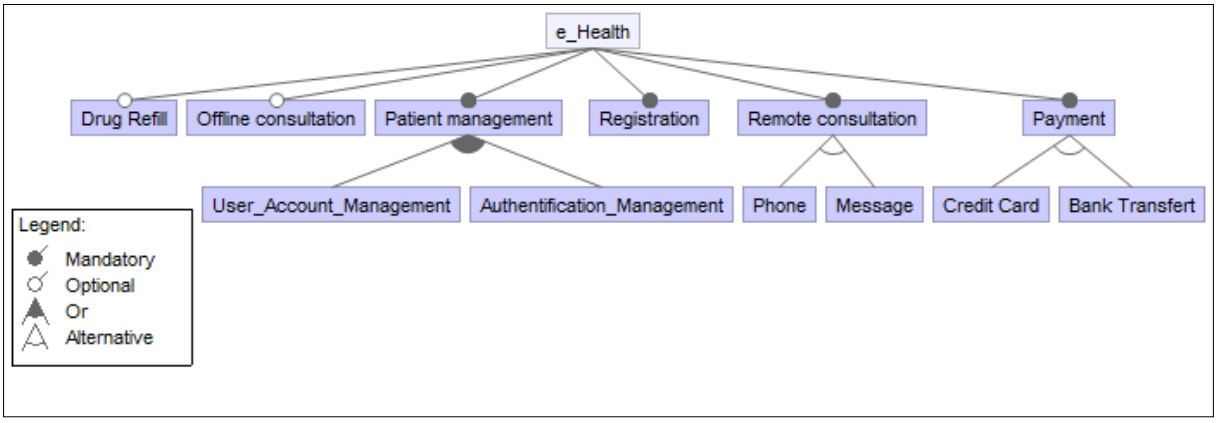

Figure 5.5: Example of for e-Health Feature Model.

The output of domain requirements engineering provided to domain design encompasses all defined domain requirements including commonality and variability as well as the definition of the product line variability in the feature model.

***Domain Design*.** The main goal of the domain design sub-process is to produce the reference architecture, defining the main software structure and the texture. The architect determines how requirements, including variability, are reflected in the architecture. An important characteristic of this architecture is the ability to select and configure reusable software artefacts.

At this stage the proposed derivation approach uses the mapping technique [51] in aim to map features to architecture model. After that, feature model is considered as an input parameter and then is processed by a model-to-model (M2M) transformation written in ATL (Atlas Transformation Language) [52] that creates an Architecture Model which composed of a set of rules and helpers. The rules define the mapping between the source and target model. The helpers are methods that can be called from different points in the ATL transformation. This model describes all components that have to be included to implement this particular Application Feature Model. We need to create in the target model all the model element types that compose a component model as it's shown in Figure 5.6.

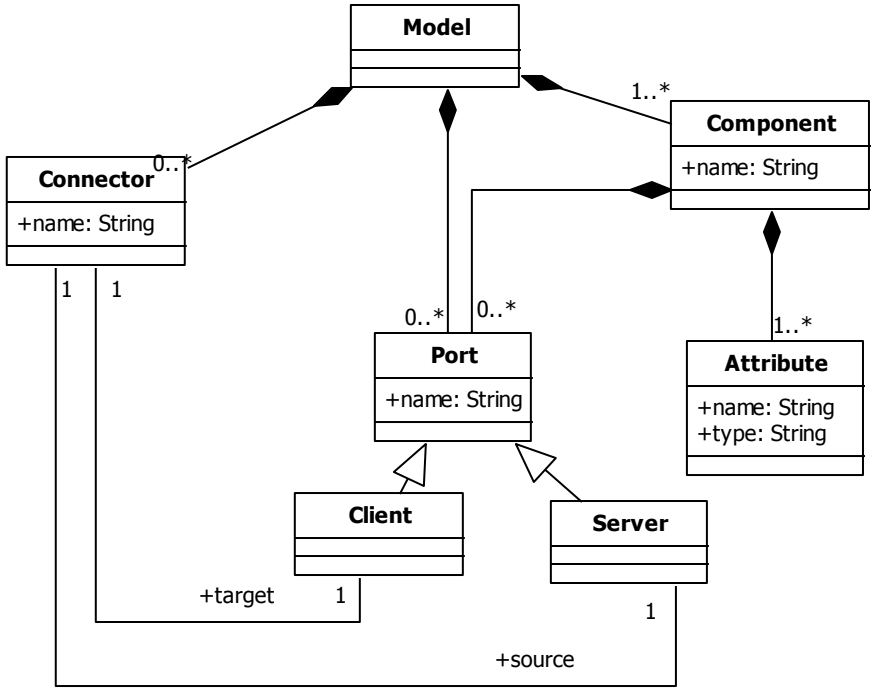

Figure 5.6: UML metamodel for Component models.

These are the rules to transform a Feature model to a Component model:

- For each feature instance, a component instance has to be created:
  - Their names have to correspond.
- For each feature Attribute instance, a component attribute instance has to be created.
- Their names have to correspond.
- Their Types have to correspond.
- The Classes have to correspond.
- For each dependency a connector has to be created
  - Their names have to correspond.
- For each operation a port has to be created
  - Their names have to correspond.
  - The Classes have to correspond.

Domain Realization. The goals of the domain realization sub-process are to provide the detailed design and the implementation of reusable software assets, based on

the Architecture Model obtained in the domain design. In addition, domain realization incorporates configuration mechanisms that enable application realization to select variants and build an application with the reusable artifacts. The model obtained in Domain Design is then processed by a model-to-text (M2T) transformation which generates an equivalent textual configuration implemented using Acceleo language [53] to promote the generation of Java. This tool specializes in the generation of text files (code, XML, documentation) starting from models. Using Acceleo we can generate the source code based on templates and models expressed with EMF [54].

5.4.2 Application Engineering

The main goal of application engineering is to derive a software product line application by reusing as many domain artefacts as possible. This is achieved by exploiting the commonality and the variability of the product line established in domain engineering. In this part we will show how the feature model is used in the application engineering sub-processes:

- Consider the commonality and the variability of the product line when defining the requirements for a specific application.
- Document the selected variants.
- Bind the selected variants from requirements to the architecture and to the components.

***Product Analysis*.** The main goal of product analysis is to document the requirements artifacts for a particular application and at the same time reuse, as much as possible, the domain requirements artefacts. Domain analysis creates the product analysis artefacts, which are reused for the application under consideration. The product analysis sub-process reuses the domain analysis artefacts to define the application requirements artefacts. The product analysis artefacts serve as a basis for application design.

A feature configuration is the production of this activity which is a legal combination of features that specifies a particular product. This activity uses feature models as input to select the feature relevant for customer's requirements to build the product and identify the specific-assets of the product. Once the selection is checked and validated by the product designer the output at this stage is a specialized version of feature model (application feature model).

We use FeatureIDE [55] an Eclipse plug-in for Feature-Oriented Software Development to create the feature configuration model defining the desired features in the new product being built; Figure 5.7 illustrates the selected features. We use a text-to-model transformation to obtain this model as an instance of the metamodel shown in Figure 5.4.

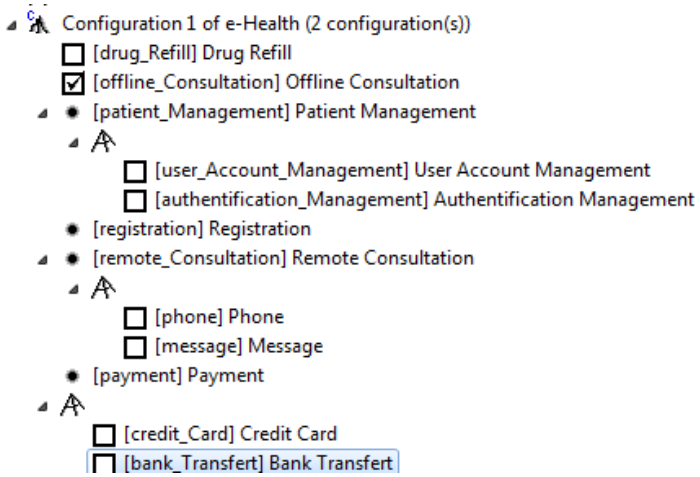

Figure 5.7: Feature Configuration Model for e-Health product Line.

Product Design. The main goal of the product design activity is to produce the product architecture model. The product architecture model is defined for the particular product being developed, considering its desired features defined in the feature configuration model. The product architecture is a specialization of the reference architecture developed in domain design.The application architecture is passed on to product realization where the reusable components and interfaces are assembled and where application-specific components and interfaces are developed.

During product design, the meta-transformation is used to generate from the feature-to-architecture transformation rule the Model Architecture artifact. This transformation is then applied to the feature configuration model to automatically generate the product architecture. The result is product architecture model generated by the rule shown in Figure 5.8, applied to the feature configuration model shown in Figure 5.7.

```
-- @path Feature=/Feature2Component\MetaModels
\FeatureMetamodel.ecore
-- @path Component=/Feature2Component\MetaModels
\ComponentMetamodel.ecore
create OUT: Components from IN: Features;
rule Componnent{
from
 e : Feature!Feature
to
 out : Component!Component (
            name <- e.name,
       )
}
rule Association{
from
 e : Feature!Dependency
to
 out : Component!Connector (
            name <- e.name,
       )
}
```

```
rule Attribute {
from
 e : Feature!Attribute
to
 out : Component!Attribute (
            name <- e.name,
            type <- e.type
      )
}
rule Port {
from
 e : Feature!Operation
to
 out : Component!Port (
            name <- e.name,
            type <- e.parameter->select(x|x.kind=#pdk_return)->
             asSequence()->first().type,
            parameters <- e.parameter->select(x|x.kind<>#pdk_return)->
            asSequence()
      )
}
```

Figure 5.8: Excerpt of Model Transformation Rules.

Product Realization. The goal of product implementation is to build the actual product, considering the architectural organization defined in the product architecture. The corresponding component implementations developed during the domain implementation must be used to obtain the implementation of the product. Product realization provides the detailed design and implementation of application-specific components and configures them with the right variants of the domain assets into applications. The main results of application realization are the application-specific components and interfaces, the selected variants of reused components, and the application configuration.

At this stage we write a program that generates Java code from our previously created architecture model using Acceleo. Our goal is to transform the features into java classes and Attribute into class properties, and finally generate set and get methods for class properties.

5.4.3 Feature-Component Mapping Technique

We use in the two sub-process defined previously (Domain engineering and Application Engineering) a mapping technique in aim to bind feature to component. We propose following the fourth next steps to map feature to component (figure 5.9):

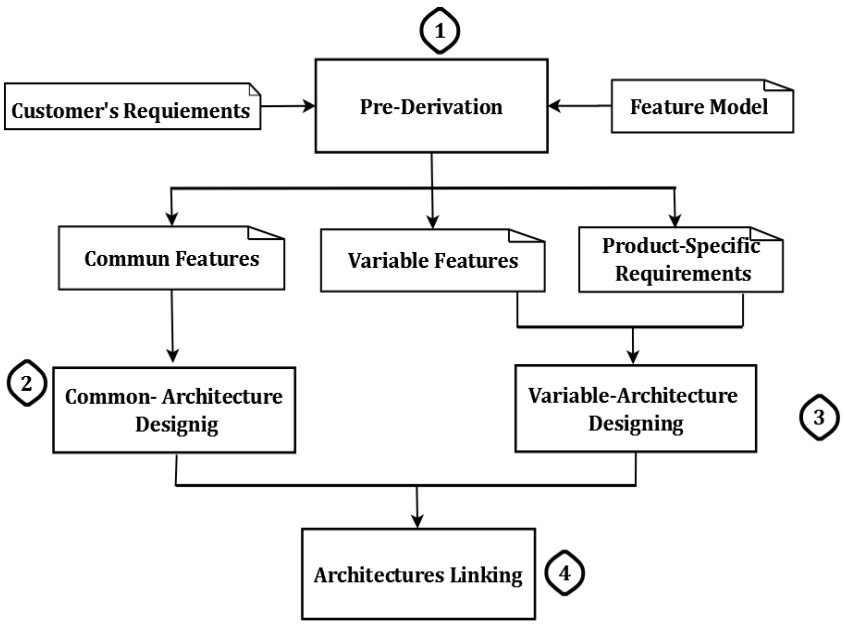

Figure 5.9: Process of features-architecture mapping

Step1 (Pre-Derivation): Initially step1 uses feature models as input to select the feature relevant for customer's requirements to build the product and identify the specific-assets of the product. Once the selection is checked and validated. The output at this stage is a specialized version of feature model (instance). Features are classified into two kinds: common features which are mandatory and variable features which make the difference between the product-line members, we need to distinct between features to facilitate the two next steps.

Step 2 (Common-Architecture Extraction)

To build the common architecture an extraction of common features is needed. We capture from feature model similarities which means all mandatory features that are common to every product. Once the set of common feature is obtained we create for each feature a component or a set of components combined in a specific way.

We subdivide our features into two kinds: (1) technique and (2) aspect to build the corresponding architectural model.

Step 3(Variable-Architecture Extraction) : Same as previous step we capture the variable features and also identifying customer's specific requirements if they exist. According to the final instance of feature model created on the first step (Pre-derivation) we map variable features to component. Then, in case of existing specific-requirements we based on information about their relationship with the available features, if it is possible we just modify an already existing components to adapt the new customer's wishes, else we

develop completely a new component from the scratch. At the end of this step we generate the variable architecture by creating for each feature a component.

Step 4 (Architectures Linking): Common and variable architectures are required to be linked each other and to do that connector provides the mechanisms for interconnecting the components and coordinating their interactions. Connectors between components are built according to dependencies that appear between features in feature model instance.

All connectors are created in the second step with the common architecture, when the variable component is added to the architecture we just activate the connector. The component is connected to another according to its interfaces that contain a set of required and/or provider services.

Some of them are exclusive; some can be active only if another specific feature is also active, and so on. These dependencies can be grouped as activation dependencies. A connector need to activate/deactivate the components mapped from the features.

5.5 Related Work

In this section, we cite the state-of-the-art related to product derivation approaches. Then, we compare our approach to the existing ones.

5.5.1 Background on Product Derivation by Transformation

Perovich et al. in [56] employ model-driven techniques to transform a feature model to specific product architectures. However, the domain design is specified in terms of ATL transformation rules, therefore the transformation processes is not completely automated. Such an approach is complex and makes the SPL architecture design process difficult.

An approach for deriving the architecture of a product by selectively copying elements from the SPL architecture based on a product-specific feature configuration is proposed in [57, 58]. The SPL architecture model contains variability to cover all products' aspects. This approach concerns only with the derivation of the high level product architecture. The mapping between features and the components realizing their implementation is done through an implementation model. A prototype that implements the derivation as a model transformation is described in the Atlas Transformation language.

Tawhid et al. in [59] proposed to derive an UML model of a specific product from the UML model of a product line based on a given feature configuration is enabled through the

mapping between features from the feature model and their realizations in the design model. The mapping technique proposed 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 and well-formed rules. The transformation is realized in the Atlas Transformation Language (ATL).

González-Huerta et al. in [60] presented a set of guidelines for the definition of pattern-based quality-driven architectural transformations in a Model-Driven SPL development environment. These guidelines rely both on a multimodel that represents the product line from multiple viewpoints as well as on a derivation process that makes use of this multimodel to derive a product architecture that meets the quality requirements.

5.5.2 Comparison

An evaluation framework is introduced in Table 5.1. To evaluate each approach we identify a set of criteria. The goal of our evaluation is to provide an overview of current product derivation in SPL and find out if - and how - the methods differ from their each other.

Table 5.1: the categories and the framework elements for Characterization and Comparison of product derivation methods

Framework element	**Description**
Requirement specification	What is the trigger of the approach?
Final Product	What is expected at the end?
M2M	Does the approach support M2M transformation?
M2T	Does the approach support M2Ttransformation?
Transformation language	In which language the transformation is written?
Automation	Is the process automatic?
Mapping technique	Does the approach use any mapping technique?

As it is shown in Table 5.2 we summarize the results of our analysis and comparison of related works and our approach using the evaluation framework.

Some observations are evident and we don't need to include in the comparison table first the context of the approaches; for the input we observed that to start the process each method needs "customer's requirements" which is almost the same for all methods, it's differ in the format. Likewise the output of each approach is almost the same which is the final product that meets customer's requirements. From the comparison, we observe that any works processed the model obtained in the M2M transformation by M2T transformation.

Table 5.2: Comparison of our approach with the related work.

Approaches	M2M	M2T	Mapping techniques	Automation	Transformation language
Botterweck et al. 2007	✓		✓	✓	ATL
Perovich et al. 2009	✓		✓		ATL
Tawhid et al 2011	✓		✓	✓	ATL
González-Huerta et al. 2014	✓			✓	QVT-Relations
Our Approach	✓	✓	✓	✓	ATL/Acceleo

5.6 Chapter Summary

Derivation of a product from an SPL seems to be an easy step since it's relied on reuse. Actually the product derivation represents one of the main challenges that SPL faces due to time-consuming.

Our primary goal is to make product derivation more efficient. To this end, we (1) integrate feature modelling to structure the SPL implementation to facilitate product derivation and (2) define a model-driven product derivation process, which transforms a feature configuration into an executable product. In this chapter, we detailed we intended to reduce the development time of a product by automating the derivation by generating some java code using Acceleo in conjunction with ATL.

The next chapter, we propose a new structure to represent composite SPLs. in order to better explain the general concepts, the problems related to this thesis and our approach to achieve the thesis objectives and validate our results, through this document we use an example that is a part of a composite e-Government Product Lines.

CHAPTER 6
HIERARCHICAL SOFTWARE PRODUCT LINES CASE STUDY FOR VALIDATION

6.1 Introduction

To avoid single software implementation Software Product Lines (SPLs) have emerged as an important development approach. A software product line combines software architecture and software reuse to build both complex and high-quality systems. The other advantage is that this fairly recent software development paradigm allows companies to create efficiently a variety of complicated products with a short lead-time.

Software product lines that belong to the same domain could have at least one characteristic in common or could also be inter-dependent. A generalization relation may also exist between product lines. The second contribution of this thesis is a new structure to represent a composite software product line that allows members of the composition to communicate and interact with each other.

Complex software-intensive systems comprise many subsystems that are often based on heterogeneous technological platforms. A composition of SPLs could be a solution for such large systems. For example, each subsystem could be built as a product line which is then combined with other product lines to form a product line of product lines or a composition of SPLs**.** In this context that, in this chapter, we explore the following three research questions:

RQ1: How can several SPLs that belong to the same domain interact with each other?

RQ2: Could one SPL acts as a master and manipulates other SPLs?

RQ3: What structure responds to both RQ1 and RQ2?

The remainder of this chapter is organized as follows. We discuss related work in Section 6.2. In Section 6.3 we sketch an outline of our approach. Section 6.4 we present the derivation process starting from modeling IHPL to deriving an architecture model. We conclude with a summary and discussion of future work in Section 6.5.

6.2 Inheritance and Hierarchical SPLs

Software Product Lines is traditionally defined as a set of software-intensive systems sharing a common, managed set of features that satisfy the specific wishes of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [1]. A simple software product line is represented by a *set* where the borders of the *set* are known as product lines scope. The scope of the product line is the collection of the products that form part of the product line. In other words, the scope encompasses all the products that the product line is capable of including [9]. This concept of a "simple software product line" is illustrated in Figure 6.1.

All the individual members of the *set* are distinguished from each other by the values of their variables. It is entirely possible that some members of the family may theoretically exist but not yet be built. Also, some features may not exist. In this case a specific feature must be built to adapt the new customer's requirements. Once the new feature is developed it will be added to the *set*. In case of unauthorized (or nonsensical) combinations of variables the SPL may be undefined at some points within the boundaries.

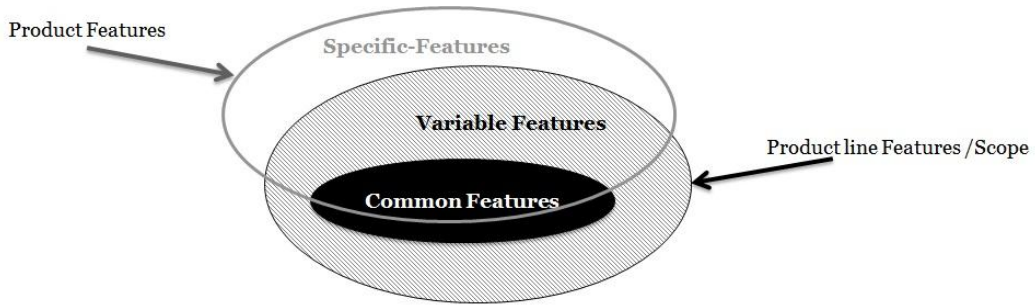

Figure 6.1: Simple Software Product Line.

While some domains need a simple SPL which is well understood and whereby products can be automatically generated others, such as complex systems, demand a larger variety of products, hence the composition of several SPLs is needed. Manipulating more than one SPL at the same time means using different feature models, which is more complicated. One way to reduce complexity is by using a top-down hierarchical structure as we argue in this thesis. We developed an inheritance and hierarchical product lines approach, which consists of representing composite SPLs in a multiple levels (hierarchical) where each SPL

(child) is able to inherit all the components of the super SPL. We call arbitrary the compositions of SPLs *IHPL* (Inheritance and Hierarchical Product Lines). Figure 6.2 shows the structure of our composite SPLs. In what follows we will detail each part of the figure.

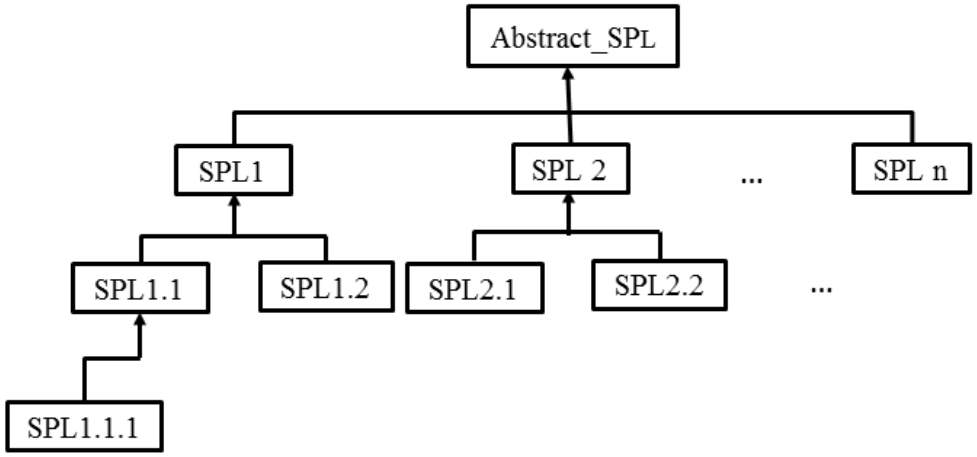

Figure 6.2: Hierarchical and Inheritance SPLs.

IHPL structure work well and could be implemented if the following conditions are met:

- The set of SPLs chosen to build the structure must belong to the same domain;
- At least one common feature is shared between SPLs;
- Variability must be present in each SPL member of the structure to be able to talk about large variety of products.

The proposed structure can be defined as a collection of SPLs starting at a root node (SPL master), where each node is an SPL consisting of a set of features (commons, variables) with the constraints that no SPL is duplicated. This organization allows decomposing higher-level SPLs into more detailed ones.

Likewise, the structure imposes hierarchical inheritance where the root serves as a super SPL (base SPL) which is an abstract SPL and its feature could be expressed as follows:

$$SPL_{root}.listF = SPL_1.listF \cap SPL_2.listF \cap \ldots \cap SPL_n.listF,$$

where SPL_x.listF is the list of features along with their type common (features that are part of each product) and variable (features that are only part of some products).

This expression is available for each parent to construct the list of common features of all its descendants. Each SPL is linked directly to only one parent (Single inheritance)

except the root, with constraints that where selecting a child implies selecting all features of its parent.

In order to obtain a larger scope we combine SPLs with other features from other SPLs from the same tree but only if the combination of variables is permissible and worthwhile. Therefore, the main role of the root node, which is an abstract SPL, is to carry out a product derivation through interaction between different SPLs and the combination of variables. This product derivation is detailed in the next section.

6.3 Modeling Features in IHPL

A feature is usually defined as “a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family” [9] or “a logical unit of behavior specified by a set of functional and non-functional requirements” [61]. Recently, Berger et al. in [62] presented a qualitative empirical study that answers the question “what is a feature?” and provides an in-depth analysis of 23 features in real-world by covering settings based on interviews investigating the practical use of features in three large companies.

Therefore, features modeling focus on identifying external visible characteristics of products in terms of commonality and variability, rather than describing all the details of products such as other modeling techniques. Feature models were first introduced in the FODA (Feature-oriented domain analysis) method [10], which also provided a graphical representation through feature diagrams.

To model a composition of SPLs several approaches can be applied as explained by [50] or [63] where a cardinality-based feature model is used to specify specializations and constraints in feature modeling. Whereas, *Invar* an approach proposed by Dhungana et al. in [64] allows the configuration of multi product lines by bridging heterogeneous variability modeling models, e.g., to configure a feature model, an OVM (Orthogonal Variability Model) model, and a decision model side-by-side.

Given that creating and maintaining a single feature model is not desirable for large systems [64, 65, and 66], approaches for separating feature models according to different views [67] or concerns [68] have been proposed. Likewise IHPL cannot be developed as a single SPL. Accordingly, we propose to model separately each SPL of the composition with the same variability modeling technique and decide not to use one single feature model for the whole composition. On the other hand we generate from several feature models a single hierarchical top-down composition model as shown in Figure 6.3.

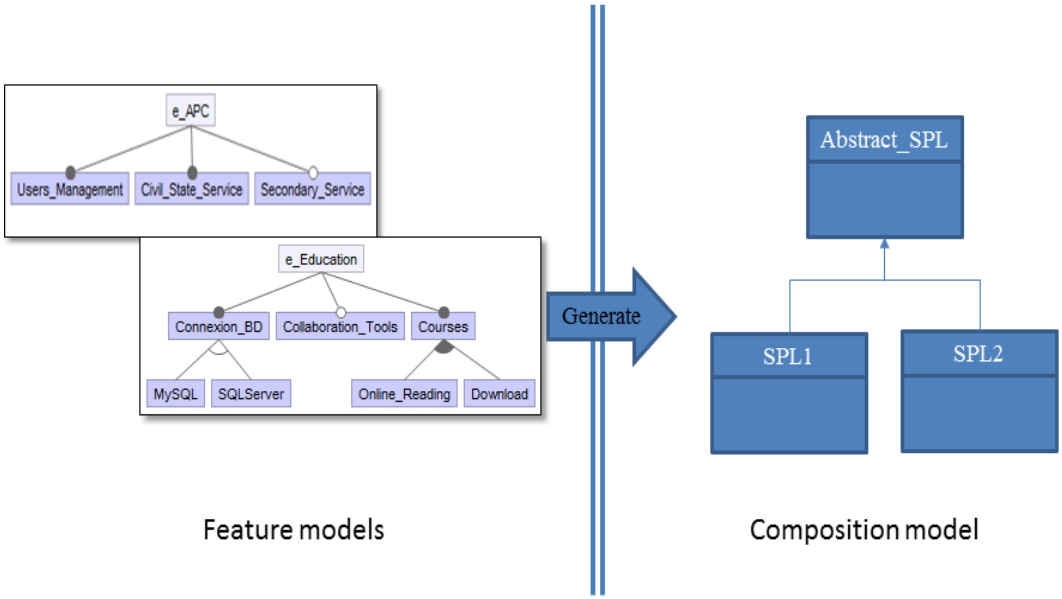

Figure 6.3: Generating composition model for IHPL.

To represent the relationship between SPLs the composition model uses the concept of inheritance of classes known from OOP (Object Oriented Programming) where each class represents an SPL. Each class of the model is represented in IHPL by a rectangle divided into three compartments .The first shows the name of the SPL, the second its features and the third constraints between features if they exist.

6.4 Product Derivation Process for IHPL

Product derivation represents a key element in software product line for its quality has a direct impact on software product costs and time-to-market. In order to derive products in IHPL we describe the required steps below.

6.4.1 Transformation Models

Figure 6.4 illustrates the main elements of our approach and their respective relationships.

Next we briefly explain the activities of second sub-process (Application Engineering) that we detailed in the previous chapter. In Figure 25 we show the transformation process.

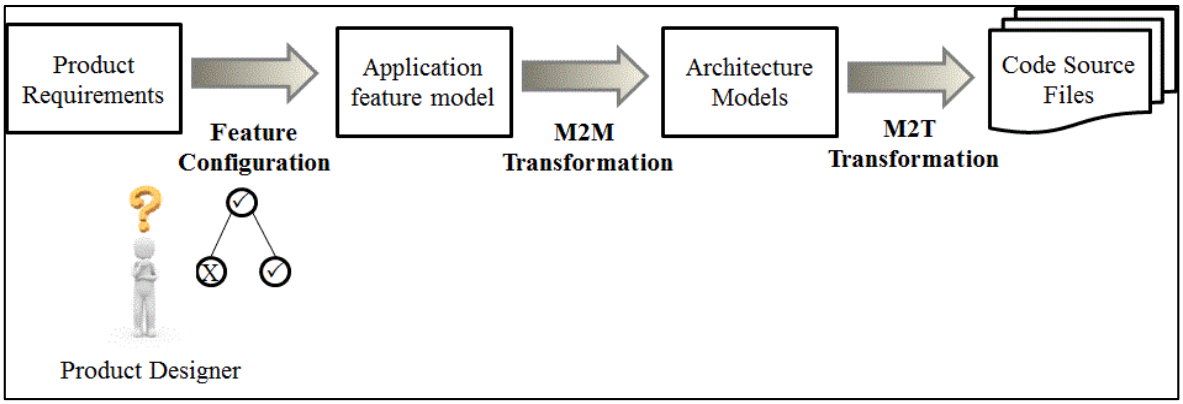

Figure 6.4: Transformation Models Process.

The first activity of our approach is the feature configuration. A feature configuration is a legal combination of features that specifies a particular product. Step1 uses feature models as input to select the feature relevant for customer's requirements to build the product and identify the specific-assets of the product. Once the selection is checked and validated by the product designer the output at this stage is a specialized version of feature model (application feature model). After that, application feature model is considered an input parameter and then is processed by a model-to-model (M2M) transformation written in ATL (Atlas Transformation Language) that creates an Architecture Model which composed of a set of rules and helpers. The rules define the mapping between the source and target model. The helpers are methods that can be called from different points in the ATL transformation. This model describes all components that have to be included to implement this particular Application Feature Model. The model is then processed by a model-to-text (M2T) transformation which generates an equivalent textual configuration implemented using Acceleo language to promote the generation of Java.

6.4.2 Feature-Component Mapping

Feature-Architecture mapping can be used to facilitate product derivation in software product line. Our proposal is a process for the generation of an architecture model for a specific product, starting from features that are organized as feature models according to the requirements of the domain. As we detailed in the previous chapter, our process of mapping consists of four steps. Before starting the fourth principal steps, two important inputs are required:

- **Customer's requirements:** to describe what customer needs, their requirements documented in natural language (textual requirements) or by conceptual models (model-based requirements).

- **Feature Model:** Our feature model focus on identifying external visible characteristics of products in terms of commonality and variability. Features can be common, optional, or alternative.

We propose the following four next steps to build the architecture:

Step 1 (Pre-Derivation): Initially step1 uses feature model as input to select the feature relevant for customer's requirements to build the product and identify the specific-assets of the product. Once the selection is checked and validated by the engineer the output at this stage is a specialized version of feature model (instance). Features are classified into two kinds: common features, which are mandatory and variable features, which differentiate between the product-line members for we need to distinguish between features to facilitate the next two steps.

Step 2 (Common-Architecture Extraction): To build the common architecture an extraction of common features is needed. We capture from feature model similarities which means all mandatory features that are common to every product will be considered. Once the set of common feature is obtained we create for each feature a component or a set of components combined in a specific way.

Step 3 (Variable-Architecture Extraction): Same as previous step we capture the variable features whilst at the same time taking into account customer's specific requirements where applicable. According to the final instance of feature model created on the first step (Pre-derivation) we map variable features to components. Then, in case where specific-requirements exist we use information about their relationship with the available features Wherever possible we just modify an already existing component to incorporate the new customer's requirements, otherwise we search for the feature required in other SPLs by navigating through the composition model generated from the set of feature models. For this navigation we first, traverse horizontally the composition model which is a ***local navigation*** with the aim to match each feature of its descendants with the desired specific feature. Second, in cases where the local navigation identified no matches we traverse this time the composition model vertically which is a ***global navigation***. The latter type of navigation provides access to others SPLs that allow us to search for the required feature. Finally, if the specific feature does not exist in any SPL we develop a completely new component from scratch. At the end of this step we generate the variable architecture by creating for each feature a component.

Step 4 (Architectures Linking): Common and variable architectures are required to be linked to each other. In order to do that a connector provides the mechanisms for

interconnecting the components and coordinating their interactions. Connectors between components are built according to dependencies that appear between features in the feature model instance.

All connectors are created in the second step with the common architecture, when the variable component is added to the architecture we just activate the connector. The component is connected to another according to its interfaces that contain a set of required and/or provider services.

Some features are exclusive; some can be active only if another specific feature is also active, and so on. These dependencies can be grouped as activation dependencies. A connector need to activate/deactivate the components mapped from the features.

6.5 Validation Case Study

In the context of an e-Government product lines, we present in this section a simple case study to illustrate the overall process, from the feature model of different product lines to the final architecture model. By applying the proposed approach, it becomes possible to derive a number of applications in a domain.

6.5.1 e-Government Product Line

To provide government services to citizens and business groups we develop different electronic applications, e-learning [69], e-APC, e-Health and adding e-Meeting [70] to our IHPL.

The first step is to define the commonality and variability that can be expected to occur among the SPL members identified in the product line's scope.

We will give a brief definition of each e-application along with their feature models in the following:

e-Learning platforms intended for learning online aims to ease and improve the teaching-learning process by means of taking advantage of internet technologies e.g. Moodle. E-Learning is just-in-time education integrated with high velocity value chains [71]. It is the delivery of individualized, comprehensive, dynamic learning content in real time, helping the development of knowledge communities, linking learners and practitioners with experts. Generally, e-learning improves the flexibility and quality of the education by [72]:

- providing access to a range of multimedia resources, such as, sounds, animations, videos and graphics;
- supporting the reuse of high quality and expensive resources;
- supporting increased communications between teachers and students and between students;
- enabling teachers to provide different materials to the students from different backgrounds;
- encouraging students to choose materials according to their own interests and to study at their own rhythm;
- helping students to take responsibilities for their own studies.

Figure 6.5 shows part of feature model we constructed for e-learning. This feature model specifies that e-learning applications must (1) connect to a database, (2) supply a HMI (Human Machine Interface), (3) use collaboration tool and (4) provide a different kind of Courses. The full description of the feature model is detailed in [69].

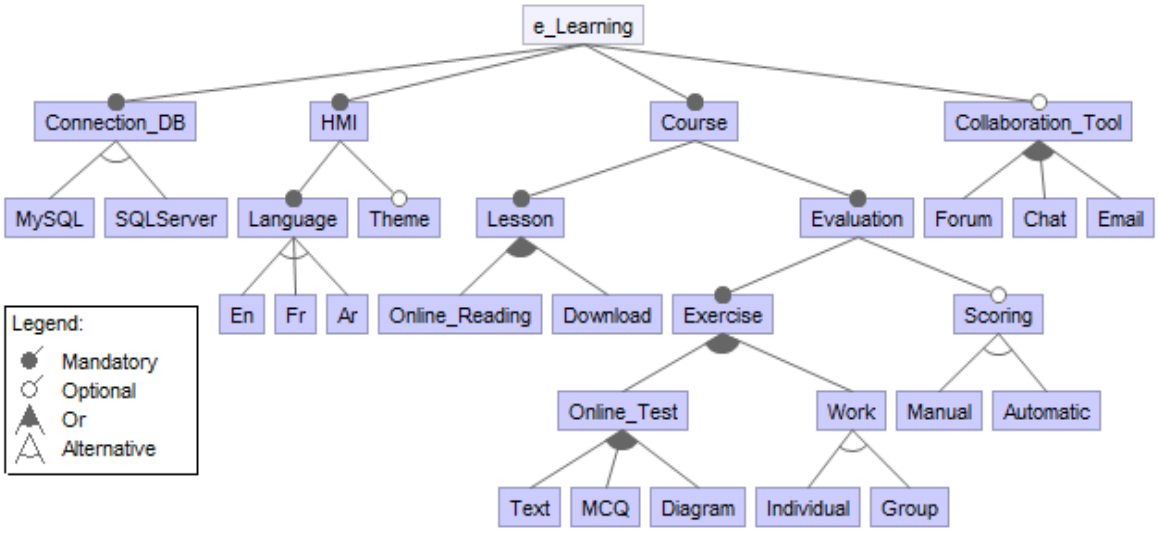

Figure 6.5: Feature Model for e-Learning applications.

e-APC The e-APC system aims to enable any citizen to access through the internet various services of a local government institution called APC (Assemblée Populaire Communale (in French), the local authority in Algeria). Most frequently required services by the citizen from the APC are the production of official documents like birth and marriage certificates. Civil State Service is a service inside the APC that delivers such official documents. Currently, a citizen requiring any of these certificates must present himself/herself to the

APC with the necessary proof documents and ask an APC officer to deliver him or her, the desired documents. The current process is time-consuming and lacks efficacity. Some key disadvantages include the long waiting time for the production of such documents (in some situation a day represents the time unit) and the high rate of errors since some operations are achieved manually and only some services are dealing with a software solution.

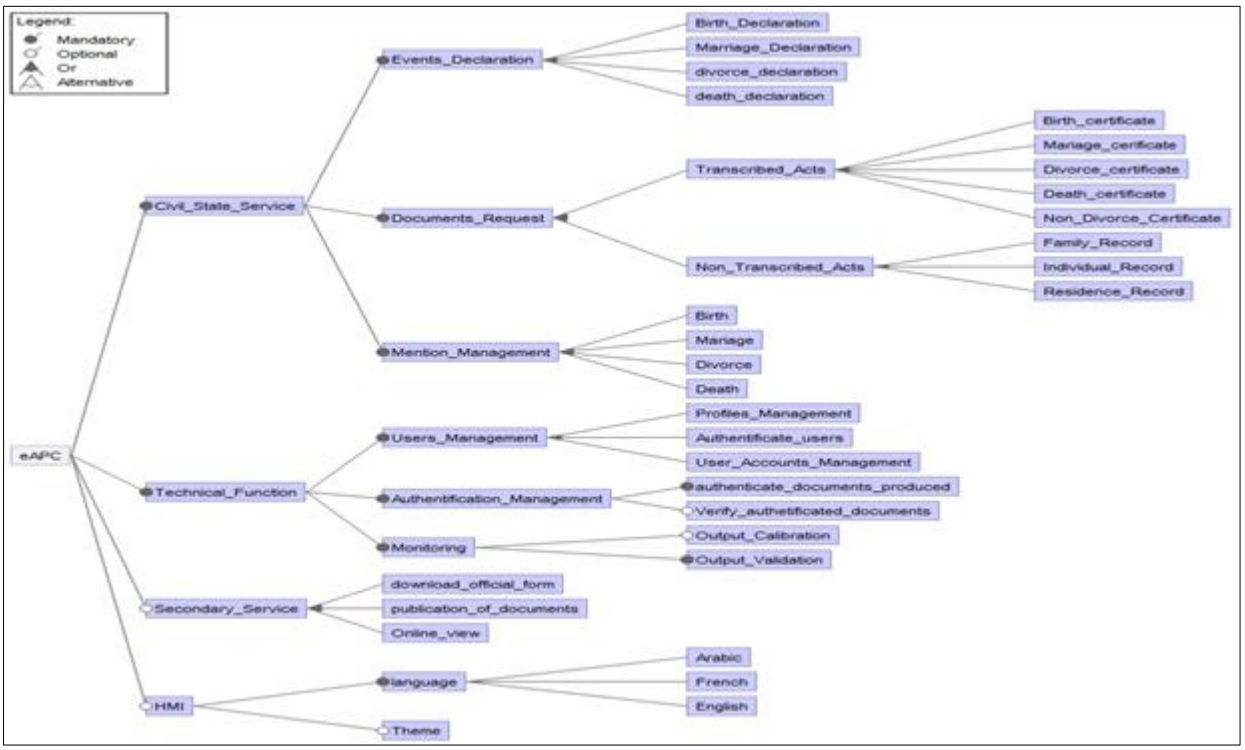

Figure 6.6: Generic Feature Model for e-APC applications.

We constructed a feature model for e-APC as is depicted in Figure 6.6. This feature model specifies that the e-APC application has four main features: (1) the kind of services that the application can provide (Civil state service); (2) the different functions that the application must contain (Technical function); (3) additional services that the application could have (Secondary service); and (4) the Human Machine Interface (HMI) that must have.

The HMI could contain or not Theme which is an optional feature, but must contain only one of three different languages (FR, EN, and AR) since these features are mutually exclusive alternatives. Also, Secondary service is an optional feature which contains three services that help citizens to download via internet: (1) an official template, (2) publish documents or just an online view of documents.

Two mandatory features must be used (1) civil state services which allow a citizen to declare different events, extract his/her civil documentation or manage mentions. (2) Technical functions for the second mandatory feature contains three other features that manage users, authentication and also monitoring the output (documents).

e-Health Health-related Internet technology applications delivering a range of clinical care, content, and connectivity, are referred to collectively as e-health. E-Health does not characterize only a technical development, but also a state-of-mind, a way of thinking, an attitude for networked, to ameliorate health care locally, regionally, and worldwide by using information and communication technology. The most remarkable attribute of e-Health is that it is enabling the transformation of the health system from one that is barely focused on curing diseases in hospitals by health professionals, to a system focused on keeping citizens healthy by affording them with information to take care of their health whenever the need arises, and wherever they may be. E-health is promoted as a mechanism to bring growth, gain, cost savings, and process improvement to health care.

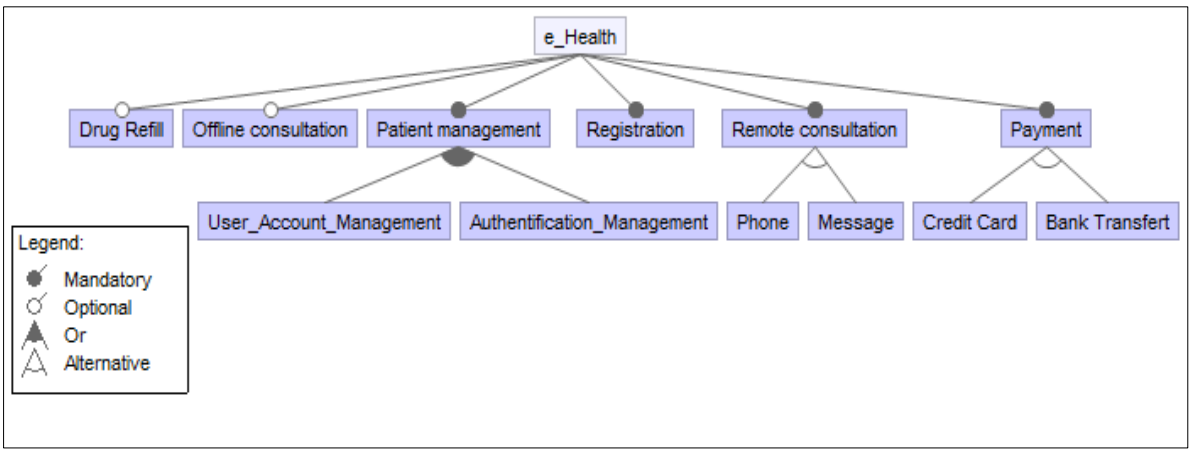

Figure 6.7: Feature Model for e-Health applications.

As Figure 6.7 shown doctors could connect via the application to follow up (1) remote consultation (via phone/message) and (2) manage patient's accounts. Patient also must do (3) a registration so that he/she can consult and (4) pay using its own credit card or just by bank transfer which are alternative features only one could be chosen. Drug refill and offline consultation are two optional features that could be chosen or just left.

e-Meeting Meeting can be found in various fields as in: meetings for year- end deliberation, meetings of a Scientific Council, meetings of business leader,… etc. Meeting over an electronic medium (through web-based software) aims to facilitate meetings without physically travelling to an agreed location and also avoid some problems that may occur when organizing face-to-face meeting as cost, availability...etc. Voice over Internet Protocol or VoIP is the most important aspect to the majority of web-based e-meeting application. VoIP allows voice transmission over the internet, which is the main to facilitating a real-time e-meeting. Some e-meeting applications also allows participants to create graphs and charts in real-time, likewise record and save the entire meeting so it can be reviewed at a later date.

Guendouz et al. have constructed feature model for e-Meeting and divided into three diagrams according to the type of features it includes: Business features, technical features, and implementation features.

We modify and simplify the feature model proposed by Guendouz et al. in [70].

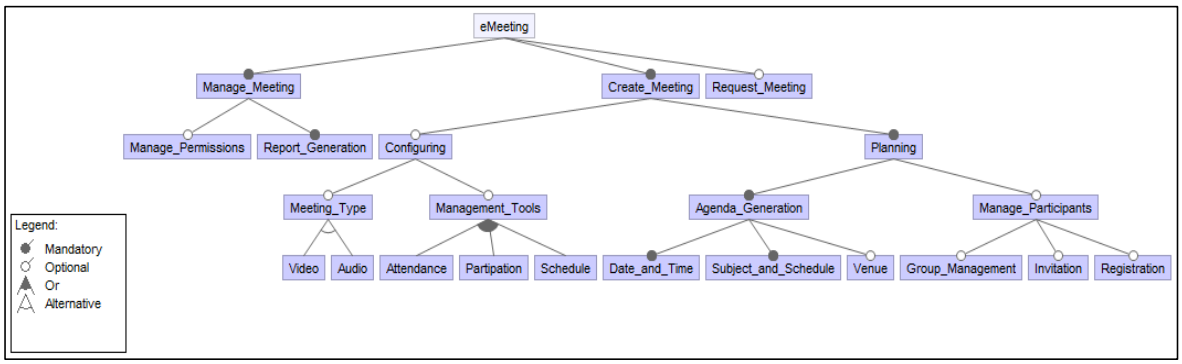

Figure 6.8: Feature Model for e-Meeting Applications.

As Figure 6.8 shows, the main features of an e-Meeting application are “Create meeting” and “Manage meeting”. Each e-Meeting application must allow at least the planning of a meeting and the generation of reports. However, in some cases a prior step can be needed before performing a meeting which is: the discussion of the meeting item, modeled by “Request_Meeting” in the Feature model. “Management of recurring items” feature can be included if the customer is interested to the history of previously treated items, there results, statistics about them and so on.

“Configuring” a Meeting is an optional feature that consist in preparing the application by fixing the meeting type (video, audio), in addition to the needed tools to run a meeting according to the user’s requirements. “Meeting tools” might include “Attendance

management", "Participation management", "Schedule manage"... the application can eventually be extended by new tools if required.

6.5. 2 Composition Model

Composition model uses the concept of inheritance of classes known from OOP to represent the relationship between SPLs. Based on the set of feature models we generate the composition model (UML Diagrams) from Java code in Eclipse using the ObjectAid UML Explorer for Eclipse. As shown in Figure 6.9 the classes of the composition model are subclasses of an abstract SPL class called e-Government and it contains common features as authentication, registration, security...etc.

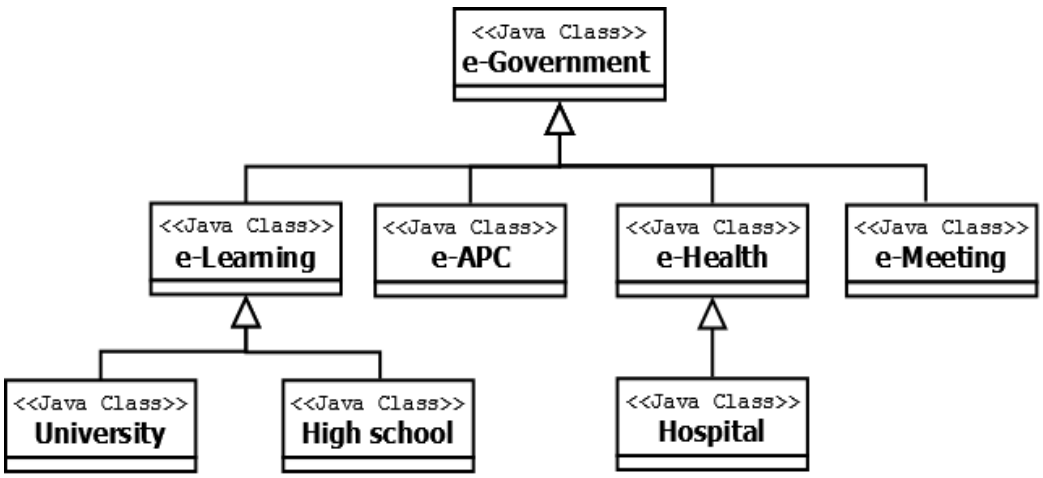

Figure 6.9: Composition Model of e-Government applications.

6.5. 3 Deriving Products

During product analysis we use FeatureIDE to create the feature configuration model defining the desired features in the new product being built; Figures (6.10, 6.11, and 6.12) illustrate the selected features. We use a text-to-model transformation to obtain this model as an instance of the metamodel.

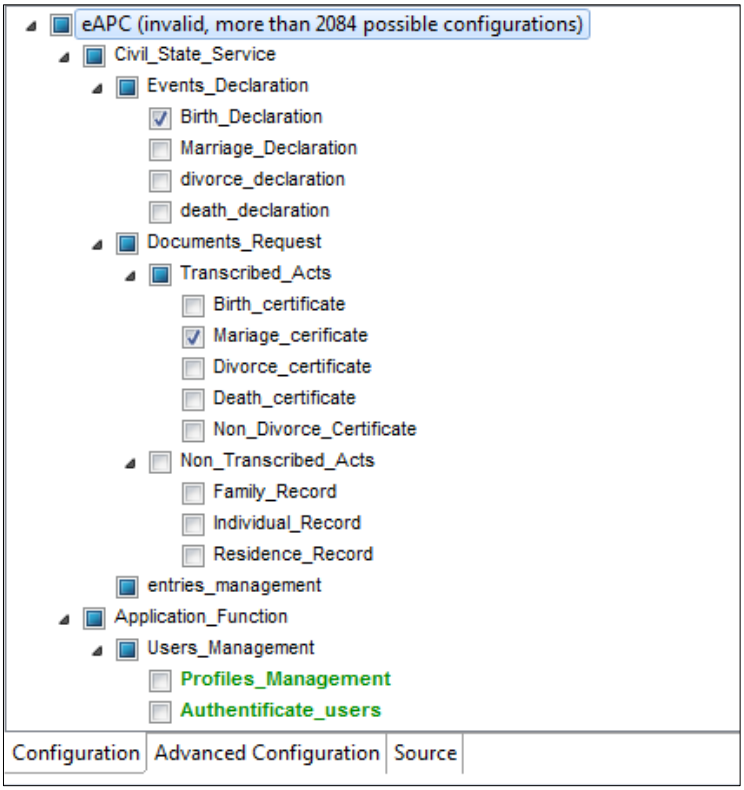

Figure 6.10: Feature Configuration Model for e-APC product Line.

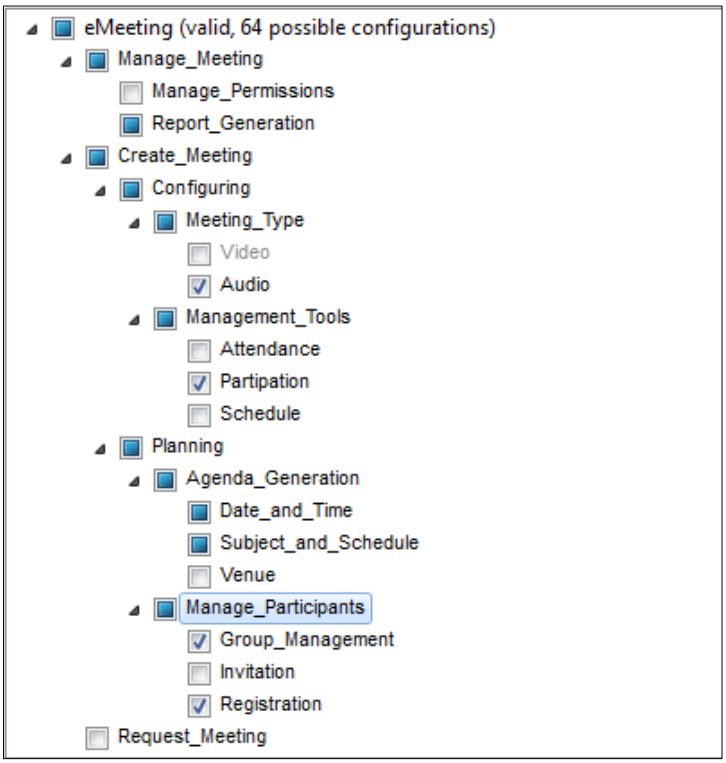

Figure 6.11: Feature Configuration Model for e-Meeting product Line

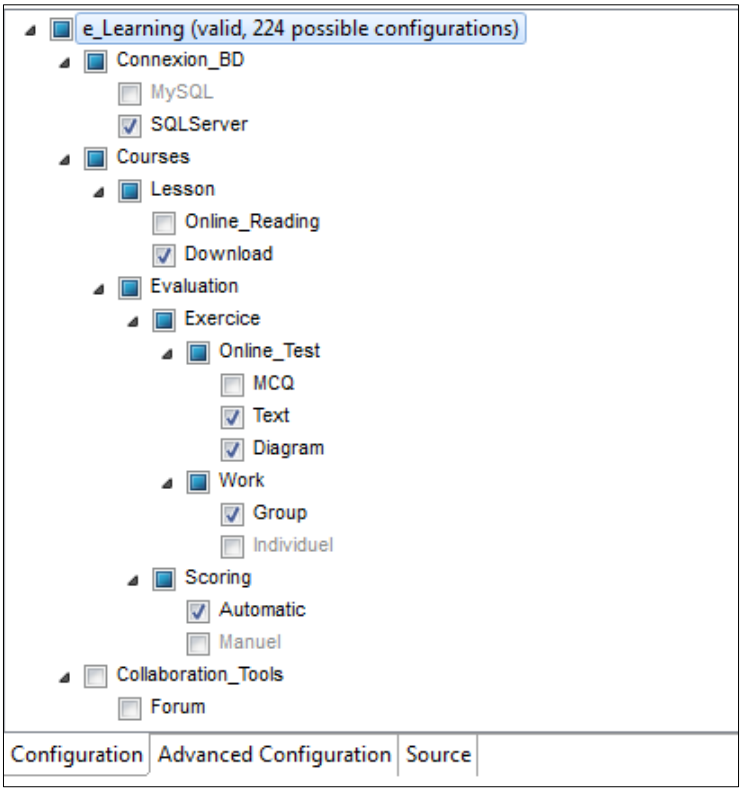

Figure 6.12: Feature Configuration Model for e-Meeting product Line.

During product design, the meta-transformation is used to generate from the feature-to-architecture transformation rule the Model Architecture artifact. This transformation is then applied to the feature configuration model to automatically generate the product architecture. The result is product architecture model generated by the transformation rules and applied to the feature configuration model shown in Figures (6.10, 6.11, and 6.12) separately.

As Figures (6.13 and 6.14) illustrate a fragment of the resulting PRODUCT ARCHITECTURE model generated by the rule, applied to the FEATURE CONFIGURATION MODEL. The e-Health and e-Learning product lines applications componenst composed by the subcomponents generated by the rules.

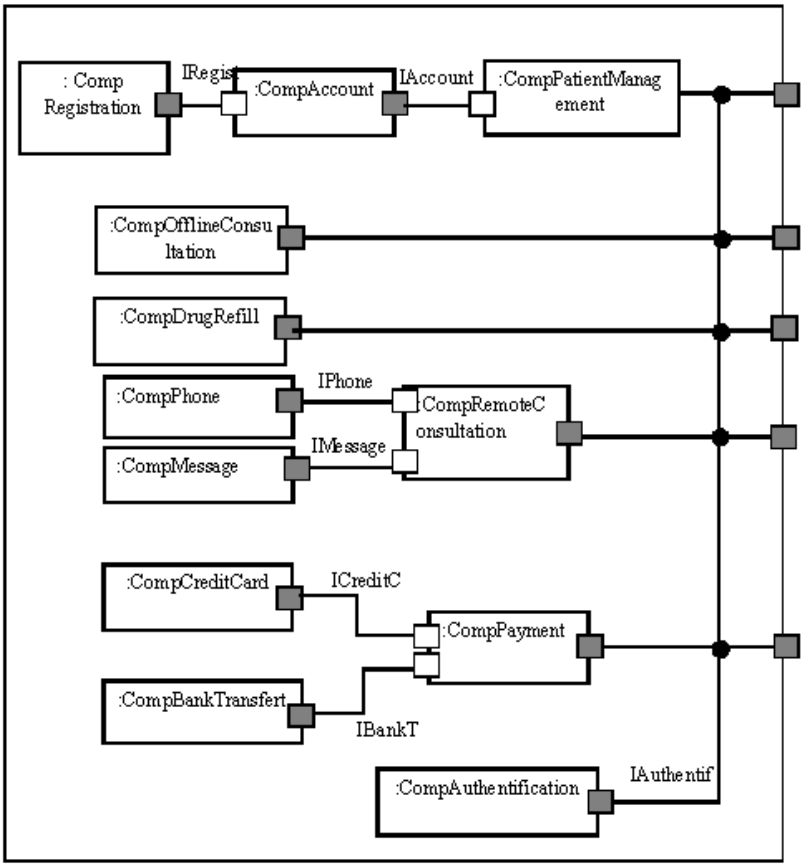

Figure 6.13: Component Model for e-Health product Line applications.

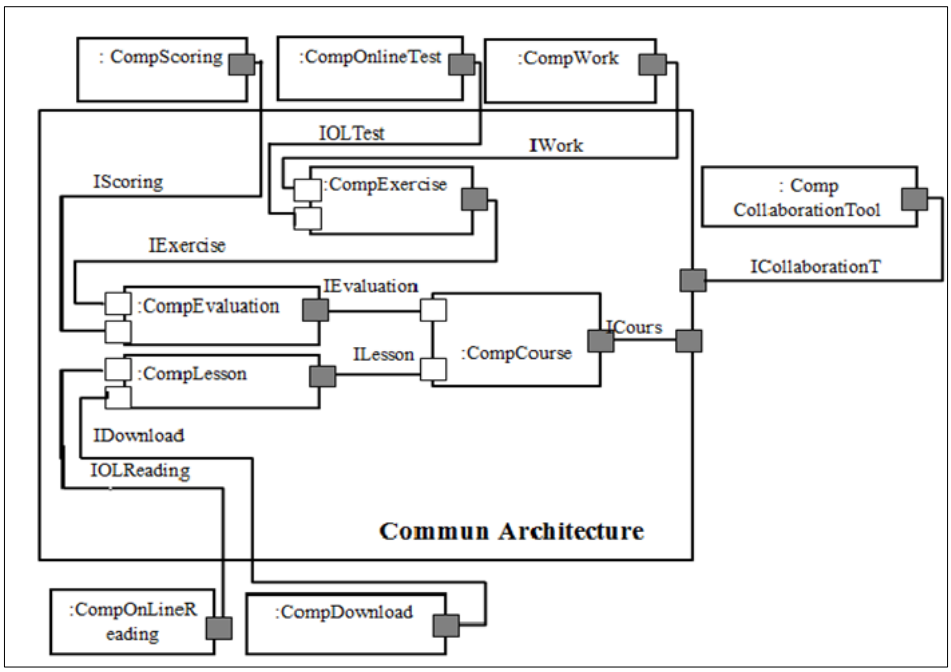

Figure 6.14: Component model for the e-learning Application SPL case study.

Final activity in application engineering is the application realization. At this stage we write a program that generates Java code from our previously created architecture model using Acceleo, which navigates the model and creates the source code (∗.java files for Java). Our goal is to transform the features into java classes and Attribute into class properties, and finally generate set and get methods for class properties. here is the code used to create a bean for each of the classes defined in our target model:

```
[comment encoding = UTF-8 /]
 [module generate('http://www.eclipse.org/uml2/3.0.0/UML')/]

 [template public generate(aClass : Class)]
 [file (aClass.name.concat('.java'), false)]
   public class [aClass.name.toUpperFirst()/] {
   [for (p: Property | aClass.attribute) separator('\n')]
     private [p.type.name/] [p.name/];
   [/for]

   [for (p: Property | aClass.attribute) separator('\n')]
     public [p.type.name/] get[p.name.toUpperFirst()/]() {
       return this.[p.name/];
     }
   [/for]

   [for (o: Operation | aClass.ownedOperation) separator('\n')]
     public [o.type.name/] [o.name/]() {
       // TODO should be implemented
     }
   [/for]
   }
 [/file]
     [/template]
```

6.6 Results and Discussion

In this section we report the results of the evaluation using the proposed approach and transformation rules. We carried out our experiment on a Toshiba Intel Core i5 4th Gen and 4 GB of RAM running Windows 7 Pro. We experimented the proposed derivation process on four deployment scenarios (e-University, e-Hospital …Etc.) using ATL/Acceleo. As it is shown in Table each e-Application used for the experiments had features and its number of valid configurations.

Table 6.1: number of features and configuration for e-Applications

	e-Learning	e-APC	e-Health	e-Meeting
Number of Features	29	43	22	12
Number of Configurations	244	2084	48	64

For each scenario, we used the automated techniques described in the previous chapter to transform Feature model into architecture model.
We tested the derivation of a product architecture for 4 random possible feature configurations with our e-Government case study. In all cases the product architecture was correctly derived. Furthermore, we also tested the detection of mapping errors. First, for each architectural variants with dependencies to other features, we tested the mappings of wrong feature dependencies. Second, we tested if any of the features described in the all e-Government system were missing during the mappings between these features and architectural elements. In all cases, our prototype detected the mapping error. One of the goal of our production strategy is to reduce the time of development. With respect to this purpose, a reduction of time and effort part has been clearly observed for all scenarios. The second goal is to use automation wherever possible, the degree is high, and the processes are automated as far as possible. This automation also takes care of the model consistency from the Feature Model through configuration to and the architecture model from which finally the code is generated.

We faced some problems during the experiments and that we could see it as a limitation of our approach. One of this problem is there is some missing information that is not yet provided by the feature model, for example the visibility of operations or attributes or special data types. Feature model still might not be capable of capturing really all the required information for the implementation of the domain. Moreover, it is important to provide support for test of final product against the customer's requirement.

6.7 Related Work

Every existing approach of modelling the integration of multiple feature modelsand structering multi product lines can be considered related work. So far, works about multiple SPls have been proposed ,we structure our discussion related work in the areas of Product Lines that supply other Product Lines,Automatic configuration of MPLs, velvet and Invar.

SPLs that supply other SPLs in a SOA environment are described by [73]. Their work focus was on modeling the interfacing between SPLs in a service-oriented environment. This includes service registration and service consumption.

Rosenmüller et al. presented in earlier work [74] an extension of existing modeling techniques design and configure MPLs. Therefore they use composition models that describe how an MPL is composed from multiple SPL instancesin aim to automate the

configuration of Multiple SPLs which is required to handle the resulting complexity. Then, recent work also by Rosenmüller et al. where they present a language for multi-dimensional variability modeling in [75] called *Velvet* that allows domain engineer to model each variability dimension of an SPL separately, then compose the separated dimensions and finally configure SPL. The syntax of Velvet uses parts of the syntax of TVL (text-based variability language) [76]. This improves reuse of FMs and supports independent modeling variability dimensions. Furthermore, Velvet combines feature modeling and configuration in a single language.

Invar (Integrated view on variability) an approach proposed by Dhungana et al in [78] and the tool prototype was presented in a short tool demonstration paper [64]. Invar allows to “plug-and-play” variability models. “Plugging” means simply adding new variability models to a shared repository. “Playing” means present the options to the end-user to allow her to configure the required product. The Invar prototype supports integration of three different variability modeling approaches, i.e., a feature modeling [77] , a decision modeling [78] , and an orthogonal variability modeling approach [79] .Invar enables the communication between different languages and tools for variability management. It eliminates the need to stick to one concrete variability modeling approach when designing multi product lines. A configuration front-end for end users transparently presents models created in different notations. Recently, [80] provide an extension to this previous work by extending Invar with different model enactment strategies allowing different orders in a configuration process based on multiple models.

In contrast to the approaches presented above, we propose to model SPLs separately and then generate a composite model for the whole IHPL. Our approach used an existing SPL modeling technique and we think that a single feature model for IHPL is not desirable for large systems. Therefore, a composition model is required to sufficiently model MPLs.

6.8 Chapter Summary

We presented in this chapter IHPL an approach that structures a set of SPLs that belong to the same domain. To facilitate communication between SPLs we model each SPL separately and then generate a composition model.

Further, we propose to derive an architecture model by mapping feature to component. The Feature-component mapping technique proposed is composed of four steps to sequentially produce a software architecture model. By instantiating the initial feature model, an instance of a feature model is constructed according to customer’s requirements.

Then, separate features are constructed into two types: common and variable. The main idea is to create for each feature a component or a set of components combined in a specific way. Linking these created components together based on the relationships among features in the feature model is the last step of our process.

CHAPTER 7
CONCLUSION

In this chapter, we summarize our thesis dissertation by summarizing the challenges and goals addressed, and we outline our contributions. Then, we discuss our perspectives related to the work presented in this dissertation.

7.1 Summary of the Dissertation

Product Derivation is the process of constructing a product from a product line of software assets [2]. An effective product derivation process within an organization brings the return of investment required for setting up the product line by allowing deriving customized products quickly and in an automated way [81]. However, when compared to the vast amount of research on software product lines, relatively few works has been dedicated to the process of product derivation [81].

A recent published report points out to an increasingly interest for product derivation related aspects, due to its importance for an organization [15]. However, this research field still lacks a set of improvements, especially in case of Software Product Line Engineering as described in [15]. Existing approaches do not give detailed information regarding strategies for product customization, variability resolution, or database model derivation. In addition, there is just a few existing approaches that provide support for the derivation process other than a high level description of the activities required.

In this work we presented a comparison framework for product derivation. To evaluate each approach we identify a set of criteria. We start the evaluation from the element of the problem situation, i.e. the approach context (input requirement and output product of the method). The second category is the problem solving process, i.e. what does the method itself contain (process, artifacts, tool...). The last category brings the two previous categories together through self-evaluation to evaluate the method output.

The goal of our evaluation is to provide an overview of current product derivation in SPL and find out if - and how - the methods differ from their each other. With various questions this study tries to address, e.g. maturity, practicality and scope of the methods to find differences.

Then, we proposed an approach for product derivation based on Model-Driven Engineering technology. To automate the process of Product Derivation we use the metamodelin. In this work, we use ATL as a model-to-model transformation language and Acceleo as a model-to-text transformation language.

Finally, a new representation of composite Software Product Lines is proposed in this work in aim to validate the product derivation approach proposed. We illustrate the application of our approaches in various case studies in the context of e-Government Product Lines, from the feature model of each different product line to the final application.

7.2 Research Contributions

The main contributions of this work can be split into the following aspects:

- **Comparison Framework:** an overview of related works and a framework to compare Product Derivation approaches based on a set of criteria.
- **A Model-Driven Product Derivation Approach:** Our approach is founded on the principles and techniques of software product lines and model driven engineering. This thesis adheres to the MDE principles, in particular for domain specific language design. We use the meta-modelling technique when addressing the definition of modelling languages. Moreover, the concept of model transformation is also extensively used.
- **A composite Software Product Lines:** a new structure to represent a composite software product line that allows members of the composition to communicate and interact with each other. The hierarchical structure proposed is based on inheritance that provides an easily understandable representation. The aim of the representation is to derive multiple products using a simple but practical method.

7.3 Perspectives

This work can be seen as an initial climbing towards an understanding of product derivation in software product lines, and interesting directions remain to improve what was started here and new routes can be explored in the future. Thus, the following issues should be investigated as future work:

- More Empirical Studies. This research presented the analysis, comparison, planning, derivation, structuring, and validation by e-Government case study. However, new studies in different contexts, including more companies and other

domains are still necessary in order to increment a set of empirical evidences extracted.

- Tool Support. The tool to support the product derivation process and implements the requirements defined in is still not complete. Moreover, it is important to provide support for test of final product against the customer’s requirement.

A List of Abbreviations and Acronyms

ATL	Atlas Transformation Language
CBD	Component-Based Development
CBSE	Component-based software engineering
CCM	CORBA Component Model
CIM	Computation Independent Model
CORBA	Common Object Request Broker Architecture
COVAMOF	Configuration in Industrial Product Families VAriability Modeling Framework
DREAM	DRamatically Effective Application development Methodology
DSL	Description Specific Language
EMF	Eclipse Modeling Framework
IASA	Integrated Approach Software Architecture
KobrA	Komponentenbasierte Anwendungsentwicklung
M2M	Model-to-Model
M2T	Model-to-Text
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
PD	Product Derivation
PIM	Platform-Independent Model
PSM	Platform-Specific Model
SPL	Software Product Line
UML	Unified Modeling Language
XML	Extensible Markup Languag

REFERENCE

1. Clements, P. and Northrop,L. ," Software Product Lines: Practices and Patterns", The SEI series in software engineering, Addison-Wesley, Boston, (2002).
2. Deelstra, S., Sinnema, M., and Bosch, J., "Product derivation in software product families: a case study", The Journal of Systems and Software, V.74, n° 2, (2005), 173-194.
3. O'Leary, P, "Towards a Product Derivation Process Reference Model for Software Product Line Organisations", Ph.D. thesis, University of Limerick, (2010).
4. Schmidt, D.C, "Guest editor's introduction: model-driven engineering", IEEE Comput, V.39, n°2, (2006),25–31
5. Stahl, T., Voelter, M. and Czarnecki, K., "Model-Driven Software Development Technology, Engineering, Management", John Wiley & Sons, (2006).
6. Pohl, K., Böckle, G.v.d. and Linden, F., "Software Product Line Engineering: Foundations, Principles, and Techniques", Springer Science & Business Media, (2005).
7. Schmid, K., "Planning Software Reuse - A Disciplined Scoping Approach for Software Product Lines", Ph.D. thesis, Fraunhofer IRB Verlag, (2003)
8. Linden, F. V. D., Schmid, K., and Rommes, E., "Software Product Lines in Action:The Best Industrial Practice in Product Line Engineering", Springer, Berlin, Heidelberg,(2007), 3-20.
9. Czarnecki, K., Eisenecker, U. W., Goos, G., Hartmanis, J., and van Leeuwen, J. Generative programming. Edited by G. Goos, J. Hartmanis, and J. van Leeuwen, 15, (2000).
10. Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S., "Feature-oriented domain analysis (FODA) feasibility study ",n°. CMU/SEI-90-TR-21), Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, (1990).
11. Botterweck, G., O'Brien, L., and Thiel, S, "Model-driven derivation of product architectures", In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ASE '07, New York, NY, USA, (2007), 469–472.
12. O'Leary, P., Rabiser, R., Richardson, I., and Thiel, S., "Important issues and key activities in product derivation: experiences from two independent research projects", In Proceedings of the 13th International Software Product Line Conference, SPLC '09, Pittsburgh, PA, USA. Carnegie Mellon University, (2009), 121–130.

13. McGregor, J., “Goal-driven Product Derivation”, Clemson University and Luminary Software LLC, U.S.A. Journal of object technology, (2009).
14. Wolter, K., Hotz, L., and Krebs, T., “Model-based configuration support for software product families”, Springer, Boston, MA, (2006), 43-61.
15. Rabiser, R., Grünbacher, P., and Dhungana, D., “Requirements for product derivation support: Results from a systematic literature review and an expert survey”, Information and Software Technology, V 52, n°3, (2010), 324-346.
16. Hotz, L., Günter, A., Krebs, T., and Fachbereich, H. C., “A knowledge-based product derivation process and some ideas how to integrate product development”. In in of Software Variability Management Workshop, (2003), 136–140.
17. Griss, M. L., “Implementing product-line features with component reuse”, In Proceedings of the 6th International Conference on Software Reuse: Advances in Software Reusability, London, UK. Springer-Verlag, (2000), 137–152.
18. Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., and DeBaud, J.M, “PuLSE: A methodology to develop software product lines”, In Proceedings of the Symposium on Software Reusability (SSR’99), (May 1999).
19. Bayer, J., Gacek, C., Muthig, D., and Widen, T, “PuLSE-I: deriving instances from a product line infrastructure”, in 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems, (2000), 237-245.
20. Kim, S.D., Min, H.G., Her, J.S., and Chang, S.H., “DREAM: A Practical Product Line Engineering using Model Driven Architecture”, in Proceedings of the Third International Conference on Information Technology and Applications (ICITA‟05), IEEE Computer Society: Washington, DC, USA, (2005), 70-75.
21. Atkinson, C., Bunse, C., and Bayer, J., “Component-based product line engineering with UML”, Addison-Wesley, London, New York, (2002).
22. Atkinson, C., Bayer, J., and Muthig, D., “Component-based product line development: the KobrA approach”, in Proceedings of the first conference on Software product lines : experience and research directions, (2000).
23. Sinnema, M., Deelstra, S., Nijhuis, J., and Bosch, J., “COVAMOF: A Framework for Modeling Variability in Software Product Families”, In: Proc. 3rd Int’l Conf. Software Product Lines (SPLC 04), San Diego, (2004).
24. Rabiser,R. , “A User-Centered Approach to Product Configuration in Software Product Line Engineering”, in Christian Doppler Laboratory for Automated Software Engineering, PhD Thesis, Institute for Systems Engineering and Automation, Johannes Kepler University, Linz, (2009).
25. Software Productivity Consortium, “Synthesis guidebook”, Technical report, SPC-91122-MC, Herndon, Virginia, (1991).

26. Sinnema, M. and Deelstra, S., “Industrial Validation of COVAMOF”, Journal of Systems and Software,V. 81, n°4, (2004), 584-600.
27. D’Souza, D. F. and Wills A.C., “Objects, Components, and Frameworks with UML– the Catalysis Approach”, Addison-Wesley, Reading, Mass, (1997).
28. Jacobson, I., Griss M., and Jonsson P., “Software Reuse, Architecture Process and Organization for Business Success”, ACM Press, Addison-Wesley Longman, (1997).
29. Szyperski C., “Component Software, Beyond Object-Oriented Programming”, ACM Press, Addison-Wesley, (1998).
30. Crnkovic, I., and Larsson, M. P. H., “Building reliable component-based software systems”. Artech House, (2002).
31. Shaw M. and Garlan D., “Software Architecture - Perspectives on an Emerging Discipline”, Prentice Hall, (1996).
32. Szyperski, Clemens, Gruntz, D., and Murer, S.,“Component software: beyond object-oriented programming”, (2002).
33. Sterling,L. , “Modeling with Interface”, In: Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1. ACM, (2002), 3-10.
34. Beugnard, A., Jézéquel, J. M., Plouzeau, N., and Watkins, D., “Making components contract aware”, Computer, V.32, n°7, (1999), 38-45.
35. OMG CORBA v 4.0 Available at: http://www.omg.org/spec/CCM/4.0/PDF/
36. Van Ommering, R., Van Der Linden, F., Kramer, J., and Magee, J., “The Koala component model for consumer electronics software”, Computer, V.33, n°3, (2000). 78-85.
37. JavaBeans specification, Sun Microsystemsé,, Avaliable http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html, 1997).
38. Bruneton, E., Coupaye, T., and Stefani, J. B., “The fractal component model. Draft of specification”, version, 2(3), (2004).
39. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J. B. “The fractal component model and its support in java”, Software: Practice and Experience, V.36, n°11, (2006), 1257-1284.
40. Bennouar,D. and and HENNI,A., “A Review of an Aspect Oriented Architecture Description Language”, The Meditaranen Journal of Computers and Networks, V.6, n°1,(2010),15-22.
41. Bennouar,D., “The Integrated Approach to Software Architecture”, PhD thesis, ESI, Oued Smar,Algies, (2009).

42. Bennouar,.D, Khammaci, T., and Henni A., “A New Approach To Component's Port Modeling In Software Architecture”, ACIT’2007, Lattikia, Syria, (December2007).
43. OMG. “MDA Guide version 1.0.1”. OMG document 2003-06-01, (2003).
44. Kent, S., “Model Driven Engineering”. In 3rd International Conference on Integrated Formal Methods (IFM 2002), Turku, Finland, (May2002),15-1.
45. Schmidt, D.C.: Guest editor’s introduction: “model-driven engineering”, IEEE Comput, V.39, n°2, (2006), 25–31.
46. Kontio, M., “Architectural Manifesto: The MDA Adoption Manual”, (2009)
47. Herst, D. and Roman, E., “Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) - Approach: A Productivity Analysis”, Technical report, TMC Research Report, (2003).
48. Mukerji, J., and Miller, J., "MDA Guide," (2003).
49. Northrop, L., Clements, P., Bachmann, F., Bergey, J., Chastek, G., Cohen, S.,and McGregor, J. A framework for software product line practice, version 5.0. SEI,((2007)
50. Czarnecki, K., Helsen, S., and Eisenecker, U., “Staged configuration using feature models”, In International Conference on Software Product Lines, Springer Berlin Heidelberg, (August, 2004), 266-283.
51. Lahiani, N., and Bennouar, D., “A Software Product Line Derivation Process Based on Mapping Features to Architecture”. In Proceedings of the International Conference on Advanced Communication Systems and Signal Processing, (November 2015).
52. “ATL Project”, [Online]. Available: http: //www.eclipse.org/atl/.
53. “Acceleo Project”, [Online]. Available: https: //eclipse.org/acceleo.
54. "Eclipse Modeling Framework," [Online]. Available: http://www.eclipse.org/emf/.
55. “FeatureIDE”,[Online]Available:http://wwwiti.cs.unimagdeburg.de/iti_db/research/featureide/#downloadK.
56. Perovich, D., Rossel, P. O., and Bastarrica, M. C., “Feature model to product architectures: Applying MDE to Software Product Lines”, In WICSA/ECSA (September 2009), 201-210).
57. Botterweck, G., O'Brien, L., and Thiel, S., “Model-driven derivation of product architectures”, In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (November 2007), 469-472.
58. Botterweck, G., Lee, K., and Thiel, S., “Automating product derivation in software product line engineering”, (2009).

59. Tawhid, R., and Petriu, D. C. , “Product model derivation by model transformation in software product lines”, In Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), (2011), 72-79.
60. González-Huerta, J., Insfran, E., Abrahão, S., and McGregor, J. D., “Architecture derivation in product line development through model transformations”, In Information System Development, Springer International Publishing. (2014), 371-384.
61. Bosch, J., “Design and Use of Software Architectures – Adopting and Evolving a Product-line Approach”, ACM Press, (2000).
62. Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., and Czarnecki, .K , “What is a feature?: a qualitative study of features in industrial software product lines”, Proceedings of the 19th International Conference on Software Product Line, (July 2015), 16-25.
63. Hwan, C., Kim, P., and Czarnecki, K., “Synchronizing cardinality-based feature models and their specializations”, In Model Driven Architecture–Foundations and Applications Springer Berlin Heidelberg, (November 2005), 331-348.
64. Dhungana, D., Seichter, D., Botterweck, G., Rabiser, R., Grunbacher, P., Benavides, D., and Galindo, J. A., “Configuration of multi product lines by bridging heterogeneous variability modeling approaches”, In Software Product Line Conference (SPLC), 15th International, (August 2011), 120-129.
65. Dhungana, D., Grünbacher, P., Rabiser, R., and Neumayer, T., “Structuring the modeling space and supporting evolution in software product line engineering”, Journal of Systems and Software, V.83, n°7, (2010), 1108-1122.
66. Hartmann, H., and Trew, T. “Using feature diagrams with context variability to model multiple product lines for software supply chains”, In Software Product Line Conference, SPLC'08,12th International, (September2008), 12-21.
67. Hubaux, A., Heymans, P., Schobbens, P. Y., Deridder, D. and Abbasi, E. K, “Supporting multiple perspectives in feature-based configuration”, Software & Systems Modeling, V.12, n°3, (2013), 641-663.
68. Acher, M., Collet, P., Lahire, P., and France, R. B., “Separation of concerns in feature modeling: support and applications”, In Proceedings of the 11th annual international conference on Aspect-oriented Software Development (March 2012), 1-12.
69. Lahiani, N., and Bennouar, D, A Model Driven Approach to Derive e-Learning Applications in Software Product Line. In Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication, (November 2015), 78-84.

70. Guendouz, A., Bennouar, D., “Component-Based Specification of Software Product Line Architecture”, In ICAASE, Algeria ,(2014), 100-107.
71. Drucker, P., “Need to know: Integrating e-learning with high velocity value chains”, A Delphi Group White Paper, (2000), 1-12.
72. Zhou, D., Cheng, X., and He, X. “The Development of a Customized E-Learning System”, Journal of Computational Information Systems, V.2, n°1, (2006), 211-216.
73. Trujillo, S., Kästner, C., and Apel, S., “Product lines that supply other product lines: A service-oriented approach”, In SPLC Workshop: Service-Oriented Architectures and Product Lines–What is the Connection, (September 2007).
74. Rosenmüller, M., and Siegmund, N., “Automating the Configuration of Multi Software Product Lines”, VaMoS, (2010), 123-130.
75. Rosenmüller, M., Siegmund, N., Thüm, T., and Saake, G., “Multi-dimensional variability modelling”, In Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems, (January 2011), 11-20.
76. Boucher, Q., Classen, A., Faber, P., and Heymans, P., “Introducing TVL, a text-based feature modelling language, In Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS’10), Linz, Austria, (January 2010), 27-29.
77. Trinidad, P., Benavides, D., Ruiz-Cortés, A., Segura, S. and Jimenez, A., “Fama framework”, In Software Product Line Conference, 12th International IEEE, (September 2012), 359-359.
78. Dhungana, D., Grünbacher, P., and Rabiser, R.. “The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study”, Automated Software Engineering,V.18, n°1,(2011), 77-114.
79. Roos-Frantz, F., Benavides, D., Ruiz-Cortés, A., Heuer, A., and Lauenroth, K., “Quality-aware analysis in product line engineering with the orthogonal variability model”, Software Quality Journal, V.20, n°3, (2012), 519-565.
80. Galindo, J. A., Dhungana, D., Rabiser, R., Benavides, D., Botterweck, G., and Grünbacher, P, “Supporting distributed product configuration by integrating heterogeneous variability modeling approaches”, Information and Software Technology, V.62, (2015), 78-100.
81. Rabiser, R., O’Leary, P., and Richardson, I., “Key activities for product derivation in software product lines”, Journal of Systems and Software, V.84, n° 2, (2011), 285-300.