

**BLIDA 1 UNIVERSITY**  
**Sciences engineering Faculty**  
Computer science Department

## **DOCTORATE THESIS**

Specialty: Computer Systems Engineering

TOWARDS A SOFTWARE PRODUCT LINE FOR E-GOVERNMENT

By

**Amina GUENDOZ**

In front of the jury composed of:

H. Abed	Professor U. Blida 1	President
N. Benblidia	Professor U. Blida 1	Examiner
N. Boustia	Professor U. Blida 1	Examiner
W. Hidouci	Professor ESI	Examiner
D. Bennouar	Professor U. Bouira	Supervisor

Blida, February 2021

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِیْمِ

و الصلّٰة و السلام علی أشرف المرسلین

## ملخص

مجال هندسة البرمجيات في تطور مستمر من أجل تطوير مناهج، طرق وأدوات جديدة تمكن من الرد بفعالية على الاحتياجات الجديدة، المعقدة و الديناميكية جداً بسبب انتشار البرمجيات في مختلف قطاعات الحياة البشرية. الوصول إلى هذا يتطلب الانتقال إلى حالة يكون فيها إنتاج برامج عالية الجودة ممكناً في وقت قصير جداً و بتكلفة جُذ منخفضة. ومع ذلك ، يجب أن تسمح عمليات الإنتاج بإمكانية إنشاء برامج قابلة للتكيف بحيث يمكن أن تلاءم الاحتياجات الفردية. في الوقت الحالي ، يبدو أن هناك نهجين رئيسيين واعدان للغاية لتحقيق هذه الأهداف بشكل مشترك: نهج البرمجيات المركبة وخطوط إنتاج البرمجيات

في حين أن خطوط إنتاج البرمجيات مستوحاة من خطوط الإنتاج الصناعية حيث يعتمد الإنتاج على تركيب قطع البرمجيات المعتمدة ، إذا قمنا بتحليل خطوط إنتاج البرمجيات الحالية ، نجد أنها تعتمد على مناهج البرامج التقليدية التي لا تتناسب مع المبادئ الأساسية لخطوط الإنتاج. نعتقد أن البرمجيات المركبة قد تمثل حلاً واعدًا لهذه المشكلة، و منه فإن القضية الأولى التي تمت معالجتها هي هذا العمل البحثي هو استكشاف إمكانية تزويد خطوط البرمجيات بنواة تستند أساساً إلى البرمجيات المركبة وإجراء التقييم في سياق الحكومة الإلكترونية. الحكومة الإلكترونية هي المشكلة الرئيسية الثانية التي يعالجها هذا العمل. في هذه الأطروحة ندرس إمكانية اشتقاق تطبيقات الحكومة الإلكترونية أو جزء مهم جداً منها من خط إنتاج بسيطاً كان أو مكوناً. الهدف من هذا العمل، إذن، هو وضع الأسس التكنولوجية والمنهجية لتطوير خط برمجيات للإنتاج السريع لتطبيقات الحكومة الإلكترونية وفقاً لمنهج تصميم معتمد على البرمجيات المركبة.

يمثل الحل المقترح خطوة أولى نحو تأسيس نهج يهدف إلى تطوير خطوط البرمجيات القائمة على البرمجيات المركبة وإدارة خطوط البرمجيات في المجالات المعقدة والمنتشرة مثل الحكومة الإلكترونية. يهدف النهج الذي نقترحه إلى إدارة إعادة الاستخدام داخل خطوط البرمجيات على مستويين من التجريد: بين التطبيقات ضمن نفس خط الإنتاج و عبر خطوط الإنتاج التي تنتمي إلى نفس المجال. من أجل إدارة إعادة الاستخدام الفعالة ضمن خطوط البرمجيات، نستحدث نهج "خط الإنتاج القائم على البرمجيات المركبة". بينما لإدارة إعادة الاستخدام بين خطوط البرمجيات، نقترح نهج خطوط المنتجات المتعددة القائمة على العناصر الجانبية. هذا الأخير هو نهج جديد لإدارة خطوط المنتجات المتعددة والذي يهدف بشكل أساسي إلى تنظيم إعادة الاستخدام عبر خطوط الإنتاج المنفصلة. تم التحقق من صحة النهج المقترح في سياق مجال الحكومة الإلكترونية.

**الكلمات المصنفة:** البرمجيات المركبة ، خط إنتاج البرمجيات ، الحكومة الإلكترونية ، الجانب ، التباين ، الاشتقاق الجزئي.

## ABSTRACT

Software engineering field evolves constantly to develop new approaches, methods, models and tools that enable the effective management of new and complex needs very dynamic due to the pervasiveness of software solutions in various human life sectors. Reaching that requires the transition to a situation where the production of quality software should be possible in a very short time at very low cost. Nevertheless, production processes must allow the possibility to create adaptable software which can fit the individual needs. Currently two main approaches seem to be very promising to jointly meet these objectives: the Software Architecture and the Software Product Line approaches.

While software product lines are inspired by industrial product lines where the production activity relies on assembling certified components, if we analyze actual software product lines we find that they relies on traditional modular and object software approaches which are not suitable to the basic principles of product lines. Believing that software architecture might represent a promising solution to this problematic, the first issue addressed in this research work is to explore the possibility of providing software product lines by a core based fundamentally on software architecture and to perform evaluation in the e-Government context. E-Government is the second major problem addressed by this work. In this thesis we study the possibility of deriving e-Government applications or very important part of them from a product line been simple or composed. The aim of this work is, then, to set up the technological and methodological bases to the development of a product line for the fast production of e-Government applications according to a component-oriented design approach.

The proposed solution presents a first step towards the foundation of an approach intended to the development of Component-Based Software product lines and the management of product lines in complex and pervasive fields like e-Government. The approach that we propose intends the management of reuse within software product lines at two abstraction levels: between the applications included in the same product line, and across product lines belonging to the same field. For the effective reuse management within software product lines, we introduce the "Component Based Product Line" approach. While, for managing reuse among product lines we suggest Aspect Multiple Product Line approach. The latter is a new multi product line management approach which aims mainly to systematize reuse across separated product lines. The proposed approach is validated in the context of e-Government field.

**Key words:** Software Architecture, Software Product Line, e-Government, Aspect, Variability, partial derivation.

# RÉSUMÉ

Le champ du génie logiciel est en évolution sans cesse croissante pour mettre sur pied de nouvelles approches, méthodes, modèles et outils qui permettraient la prise en charge efficace des nouveaux besoins complexes et très dynamiques dus à l'omniprésence des solutions logicielles dans les divers secteurs de la vie humaine. La prise en charge efficace de ces besoins impose le passage à une situation où la production d'un logiciel de qualité doit être possible dans des délais très courts à des coûts très réduits. Actuellement deux grandes approches semblent être très prometteuses pour répondre conjointement à ces objectifs: l'approche de conception de système logiciel par assemblage de composants, plus connue sous le nom d'architecture logicielle et l'approche de Ligne De Produits. Malgré que les lignes de produits logiciels sont inspirées des lignes de produits industrielles où l'activité de production repose sur l'assemblage de composants certifiés, si nous analysons les lignes de produits logiciels existantes, nous constatons qu'elles reposent sur des approches logicielles modulaires et objets traditionnelles qui ne sont pas adaptées aux principes de base des lignes de produits. Estimant que l'architecture logicielle pourrait représenter une solution prometteuse à cette problématique, la première problématique de ce travail de recherche est d'explorer la possibilité de fournir des lignes de produits logiciels par un noyau basé fondamentalement sur l'architecture logicielle et d'effectuer des évaluations dans le contexte de l'e-gouvernement. L'e-gouvernement est le deuxième problème majeur abordé par ce travail. Dans cette thèse, nous étudions la possibilité de dériver les applications d'e-gouvernement ou une partie très importante d'entre elles à partir d'une ligne de produits simple ou composée.

La solution proposée présente une première étape vers la fondation d'une approche destinée au développement de lignes de produits logiciels à base de composants et à la gestion de lignes de produits dans des domaines complexes et omniprésents comme l'e-gouvernement. L'approche que nous proposons vise la gestion de la réutilisation au sein des lignes de produits logiciels à deux niveaux d'abstraction: entre les applications incluses dans une même ligne de produits, et entre les lignes de produits appartenant au même domaine. Pour une gestion efficace de la réutilisation au sein des lignes de produits logiciels, nous introduisons l'approche «Component Based Product Line». Alors que, pour gérer la réutilisation entre les lignes de produits, nous suggérons l'approche Aspect Multiple Product Line. L'approche proposée est validée dans le contexte du domaine de l'e-gouvernement.

**Mots clés :** Ligne de produit, Architecture Logicielle, e-Gouvernement, Aspect, variabilité, dérivation partielle.

## **ACKNOWLEDGMENT**

I first thank Allah for giving me the strength to complete this humble work.

I would like to thank Mr. Bennouar for offering me the research subject treated in this thesis and also for their advices, support and patience.

My sincere thanks go also to the jury Members who agreed to read and evaluate the work presented in this thesis.

I express my deep gratitude to Madame Boumahdi as well as to the teachers of the computer science department and the members of the computerized systems research and development laboratory (LRDSI) for their advices, their suggestions and for the help they provided me.

Finally, but definitely not the least, I express my gratitude to my family for believing in me long after I'd lost belief in myself, and for sharing my wish to reach the goal of completing this task.

# CONTENT

<b>ABSTRACT</b>	<b>3</b>
<b>ACKNOWLEDGMENT</b>	<b>5</b>
<b>CONTENT</b>	<b>6</b>
<b>LIST OF ILLUSTRATIONS AND TABLES</b>	<b>7</b>
<b>INTRODUCTION</b>	<b>8</b>
<b>CHAPTER 1: SOFTWARE PRODUCT LINES</b>	<b>19</b>
1.1. Introduction	19
1.2. Core concepts of SPL	20
1.2.1. SPL definition, advantages and challenges	20
1.2.2. Managing Variability in SPLs	22
1.2.3. Feature models	24
1.3. SPL engineering	25
1.3.1. Domain engineering	27
1.3.2. Application engineering	28
1.4 Variability modeling techniques	28
1.4.1. Representing variability in separated models	29
1.4.2. Representing variability in artefacts	33
1.4.3. Discussion	36
1.5. Multiple Product Lines (MPLs)	37
1.5.1. MPLs meaning, benefits and issues	37
1.5.2. Overview of approaches related to MPLs	40
1.6 Conclusion	43
<b>CHAPTER 2: SOFTWARE ARCHITECTURE and SOFTWARE PRODUCT LINES</b>	<b>45</b>
2.1. Introduction	45
2.2. Core concepts of Software Architecture	46
2.2.1. Definition and advantages	46
2.2.2. Components	48
2.2.3. Connectors	50
2.2.4. Configuration	50
2.2.5. Component based development process	51
2.2.6. The ADLs	52
2.3. The IASA approach	53
2.3.1. The IASA component model	54
2.3.2. The IASA connector model	55
2.3.3. The IASA access points	56
2.3.4. The IASA ports	57
2.3.5. The SEAL action language	57

2.4. Component based product lines	58
2.4.1. State of the art	59
2.4.2. Discussion	64
2.5. Conclusion and recommendations	67
<b>CHAPTER 3: COMPONENT BASED SPLS (CBPL)</b>	<b>69</b>
3.1. Introduction	69
3.2. Composition oriented FM	70
3.3. Component Based Product Line Engineering	72
3.3.1. CBPL domain engineering	73
3.3.2. CBPL application engineering	73
3.4. Variability Modeling In CBPLs	75
3.4.1. Architecture design with IASA	75
3.4.2. Variability modeling extension for IASA	76
3.4.3. Mapping features to the architecture	78
3.5. Case study: e-meeting CBPL	79
3.5.1. Domain analysis	79
3.5.2. Product-line architecture design	81
3.6. Conclusion	83
<b>CHAPTER 4: ASPECT MULTI PRODUCT LINES (AMPL)</b>	<b>84</b>
4.1. Introduction	84
4.2. Crosscutting Reuse within MPLs: e-Government example	85
4.2.1. E-Admin SPL	86
4.2.2. E-Education SPL	89
4.2.3. Comparison and results	90
4.3. Aspect Multiple Product Lines (AMPL)	93
4.3.1. Separation of concerns in MPLs	93
4.3.2. Aspect SPLs	94
4.3.3. Partial derivation	95
4.4. AMPL Engineering	96
4.4.1. ASPLs engineering	97
4.4.2. Sub-SPLs Engineering	98
4.5. Discussion	102
4.6. Conclusion	104
<b>CHAPTER 5: PARTIAL DERIVATION AND COMPOSITION</b>	<b>106</b>
5.1. Introduction	106
5.2. Partial derivation	107
5.2.1. Restriction Techniques	108
5.2.2. Expansion Techniques	109
5.3. Partial derivation of the Feature Model	110
5.3.1. Restricting a Feature Model	110
5.3.2. Expanding a Feature Model	112
5.4. Partial derivation of the Architecture Model	114
5.4.1. Restricting the configuration choices of architecture Model	114



5.4.1.3. Removing an architecture element	115
5.4.2. Expanding the configuration choices of architecture Model	116
5.5. SPLs merging	116
5.5.1. Feature Models merging	117
5.5.2. Architecture models merging	117
5.6. Case study:	119
5.6.1. The e-Evaluation ASPL	119
5.6.2. The composition model	123
5.7. Conclusion	124
<b>CHAPTER 6: EVALUATION OF THE APPROACH: E-GOVERNMENT CASE STUDY</b>	<b>125</b>
6.1. Introduction	125
6.2. Background and Motivation	127
6.2.1. E-Government and SPLE	127
6.2.2. Crosscutting reuse among e-Gov SPLs	128
6.3. E-Government AMPL Engineering	129
6.3.1. E-Gov ASPLs engineering	129
6.3.2. E-Government sub-SPLs engineering	135
6.4. Discussion and evaluation	142
6.5. Conclusion	146
<b>CONCLUSION</b>	<b>150</b>
<b>REFERENCES</b>	<b>154</b>

## LIST OF ILLUSTRATIONS AND TABLES

Figure 1. 1: Example of a FM (part of the evaluation component FM from the e-Learning SPL).	25
Figure 1. 2: SPL engineering process [4].	27
Figure 1. 3: Classification of variability modeling techniques.	29
Figure 1. 4: Multiple Product Lines.	38
Figure 2.1: Sample graphical representation of components and connectors.	49
Figure 2. 2: Component Based Development process [69].	51
Figure 2. 3: Clauses in a component's textual description.	58
Figure 2. 4: Koala component model.	61
Figure 3. 1: Component-Based Product Line engineering [93].	74
Figure 3. 2: The basic IASA notations.	75
Figure 3. 3: Interfaces variability notation.	76
Figure 3. 4: Choice variability notations.	77
Figure 3. 5: Business feature diagram for e-Meeting product line.	79
Figure 3. 6: Technical feature diagram for e-Meeting product line.	80
Figure 3. 7: Reference architecture of e-Meeting product line.	82
Figure 3. 8: Variability model for "Meeting_Configuration" component.	83
Figure 4. 1: Business feature diagram for e-Administration product line	87
Figure 4. 2: Technical feature diagram for e-Administration product line	88
Figure 4. 3: Implementation feature diagram for e-Administration product line	89
Figure 4. 4: Business feature diagram for e-Education product line	90
Figure 4. 5: ASPLs life cycle within AMPL engineering [96].	96
Figure 5. 1: Restriction operations on FMs.	112
Figure 5. 2: Expansion operations on FMs.	114
Figure 5. 3: Feature models merging	118
Figure 5. 4: The evaluation ASPL FM.	119
Figure 5. 5: The reference architecture of the evaluation ASPL.	120
Figure 5. 6: The internal structure of the component 'test'.	120
Figure 5.7: Partially derived test component for e-Primary SPL.	121

Figure 5. 8: Partially derived evaluation ASPL FM for e-Math SPL.	122
Figure 5. 9: Partially derived test component for e-Math SPL.	122
Figure 5. 10: The reference architecture of e-University SPL.	123
Figure 6.1: Citizen as the central entity of e-Government.	129
Figure 6. 2: E-Government before and after applying AMPL approach.	130
Figure 6. 3: The e-APC sub- SPL FM.	136
Figure 6. 4: The e-APC sub- SPL reference architecture.	137
Figure 6. 5: Partially derived e-Meeting FM for e-APC sub-SPL.	138
Figure 6.6: Partially derived “Meeting_management_Cmp” for e-APC sub-SPL.	138
Figure 6.7: FMs merging of the e-Meeting ASPL and e-APC sub-SPL.	139
Figure 6. 8: The e-University sub- SPL FM.	140
Figure 6. 9: Partially derived e-Meeting FM for e-University sub-SPL.	141
Figure 6.10: Partially derived e-Meeting reference architecture for e-University sub-SPL.	141
Figure 6. 11: FMs merging of the e-Meeting ASPL and e-University sub-SPL.	142
Table 2. 1: CBPL approaches evaluation.	65
Table 3. 1: Mapping features to the architecture	78
Table 4. 1: Comparing e-Admin and e-Education SPLs features	91
Table 5. 1: Restriction operations of an FM	111
Table 5. 2: Expansion operations of an FM.	113
Table 6. 1: Comparing our approach to related work.	145

# INTRODUCTION

## 1. Introduction

Software engineering field is constantly evolving in the aim of developing new approaches, methods, models and tools that enable the effective management of the growing new needs. Been pervasive in the various human life sectors, these needs became continuously dynamic and complex to be handled. The effective management of these needs requires the transition to a situation where the production of quality software should be possible in a very short time and at very low costs. Nevertheless, production processes must allow the possibility to create adaptable software which can fit the individual needs. Currently, two main approaches seem to be very promising to meet jointly these objectives: the software design approach by assembling components, better known under the name of *Software Architecture* and the *Software Product Line* approach.

*Software Architecture approach*, which is an emerging discipline in software engineering, allows the production of systems in short development time by assembling certified components. The Software Architectures field deals with the complexity issue by defining a software system from an abstract view point. Breaking a problem into smaller parts makes it easier to be analyzed by moving from a complex system to separate entities and relationships between them. The architecture of a software system defines the overall structure in terms of components and interactions among them. A component, or a module, can be identified as a well defined unit that performs some function. An example of a component is a database or an object. Interactions among components occur through connectors such as a procedure call, or a protocol. In addition, properties of a component specify how connection with other components can be made via connector.

Software architecture is an abstraction of a complex system [1]. This abstraction provides a number of benefits: - it allows predicting the system properties before its actual existence in the form of a software product [2]. This ability, to verify if a future software system fulfills its stakeholders' needs without actually having to build it, represents substantial cost-saving and risk reduction. - It provides a basis for reusing elements and decisions [2][1]: software architecture patterns, strategies and decisions, can be reused across multiple systems whose stakeholders require similar quality attributes or functionality, and this help in saving design costs and reducing the design mistakes risk. - It supports early design decisions that impact a system's development, deployment, and maintenance [1]. - It facilitates communication with stakeholders [1]: Architecture gives the ability to communicate the design decisions with stakeholders before the system is implemented, when it is relatively easy to adapt it to better fulfill their needs.

Software architecture deals with the design of the overall software system structure. It firstly identifies the high level system components called the subsystems and how those components are connected to each other (connections or interfaces). The structure is defined in a way that minimizes the coupling between the different subsystems and increases the internal cohesion of each of them. Each subsystem must be: cohesive, performs a major service, contain highly connected components or classes with respect to the subsystem and relatively independent of other subsystems, and finally it can be decomposed further into smaller subsystems. These design steps must be documented in order to facilitate communication between stakeholders, capture early decisions about the high-level design, and allow reusing components between projects.

In the last decade, software architecture continues to prevail in the academic world. A very large number of research activities have been undertaken on the various software architecture aspects, such as the structural, behavioral and aspect-oriented specification of an architecture, behavior validation, interactions validation, modeling interaction ports, dynamics in software architecture, prediction of non-functional properties, hot component replacement, etc. Although the software architecture was initially based on coarse grained components, today the process of refinement can reach very fine aspects. As an example, an arithmetic operator can be seen as a component in some software architecture approaches

which base on the principle that everything must be component in a software development process. Today, a very large number of models, methods and tools have been proposed and produced.

*Software Product Lines* are emerging as a viable and important development paradigm allowing companies to realize major improvements in time to market, cost, productivity, quality, and other business drivers. Software product line engineering can also enable rapid market entry and flexible response, and provide a capability for mass customization. Software product line approach aims to systematize the reuse throughout all the software development process: from requirements engineering to the final code and test plans. The purpose is to reduce the time and cost of production and to increase the software quality by reusing elements which have been already tested and secured.

Software Product Line approach intends to adapt to the software development, the principles that we find in the automotive industry, aeronautics and electronics. Production in these industries is organized into ranges with similar parts and offering a number of options [3]. For example, in automotive industry, various car models come out the same assembling chains using the same chassis, the same engines and the same test plans. The aim is to improve productivity, decrease time to market, and reduce production, maintenance and test costs. The idea is to transpose the manufacturing principles of these industries in software development, to benefit from the before-mentioned advantages.

A Software Product Line (SPL) is a set of systems sharing a set of common features, satisfying the specific needs for particular field and developed in a controlled way from a common set of reusable elements [3]. Ultimately, different software applications are grouped into a software product line in which we differentiate [4]:

1. Commonalities: software elements that are common to all the members of the product line.
2. Variabilities: software elements that vary from a member to another member of the product line (may be common to some products but not all).

3. Product-specifics: software elements that may be part of only one product at least. Such specialties are often not required by the market per se, but are due to the concerns of individual customers.
4. Constraints existing between software elements. These constraints will be respected when these elements are assembled. They originate mainly from business rules.

During the life-cycle of the product line, a specific variability may change in type. For example, a product-specific characteristic may become variability. On the other hand, a commonality may become a variability as well.

SPL engineering seeks to systematize reuse throughout the development process. This latter is separated in two complementary phases [3] [4]: *domain engineering* and *application engineering*. Figure 1 shows the main activities conducted during an SPL engineering process.

In domain engineering the commonality and the variability for a set of envisioned product line applications are identified, documented, and produced [4]. Domain engineering or “development for reuse” involves domain analysis, domain design and domain implementation processes. The principal outputs of this process are: the identification of the product line members, the extraction of the similarity and variability between them, the construction of core assets, in addition to the production of effective means that help in using these core assets to build a new product within a product line. A core asset is a reusable artifact or resource that is used in the production of more than one product in a software product line. A core asset may be an architecture, a software component, a domain model, a requirements statement or specification, a document, a plan, a test case, a process description, or any other useful element of a software production process [5].

Application engineering or “development with reuse” aims 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 [4]. This process is repeated for each new

application, it is quite similar to traditional software development process; unless that each step is facilitated by reusing outputs of the first process.

In fact, in addition to these two major activities, there is also a coordination activity between the two processes. This activity focuses on synchronization between the two stages. If, in a natural way, decisions during the second process (application derivation) are essentially impacted by the first process (domain engineering), it is also possible that the derivation can contribute to the SPL core assets enrichment.

Reuse is essential to ensure an efficient support of customer's needs while decreasing time and cost of development. However, SPL developers must improve reuse while maintain diversity between products. This could be done by "Variability management". Variability management is a key activity that usually affects the degree to which a SPL is successful [6]. Variability refers to the ability of an artefact to be configured, customized, extended, or changed for use in a specific context [7]. This variability must be defined, represented, exploited, implemented, evolved, etc. – in one word managed – throughout software product line engineering. SPL community has spent huge amount of resources on developing various approaches to dealing with variability related challenges over the last decade. The well management and documentation of variability is a crucial factor to build successful SPLs.

## 2. Problem Statement

In this research topic we deal with two major issues closely linked in the context of this work. The first problem lies in the context of software architecture. The second one lies in product line and the field that it should cover, in our case it is the e-Government.

If we refer to the literature, the number of product lines actually implemented can be counted on fingertips. Sometimes they address very limited areas located themselves in a particular area of technology. This is the case for some SPLs in the field of telephony or the automotive industry. In other cases, highly configurable software is considered as an SPL for a very restricted domain (online meeting for example). To our knowledge, we have not yet met an SPL developed for large-scale applications domain or covering a high important scope as e-



*Government.* This may come back to the broadness and complexity of this field or even to the financial risk of a possible failure when establishing an SPL for an important domain.

In addition, when analyzing the current product lines in depth, we noticed that these product lines are based on approaches and models that do not fit actually with the basic SPLs concepts. In fact, although the SPL basic concepts are inspired by the product lines in electronics and mechanical industry, where the production activity is an assembly activity of certified components, the current SPLs are based on traditional software engineering concepts such as: modular and object design concepts. Since these models do not support the concepts of assembly and component certification, this makes the SPLs derivation hard to be fully automated.

We believe that in light of the latest advances in Software Architecture and considering the promising benefits that bring Software Architecture and components-based approaches for the production of high quality software at a reduced cost, it would be more interesting for SPLs to be based on design approaches, production approaches and models that match perfectly with their basic principles.

➤ *Hence, the first issue to be addressed is to explore the possibility of providing SPLs with core assets based fundamentally on the software architecture and to perform the evaluations about the contribution of the software architecture to product line in the e-Government context.*

The scope of a product line must be clearly limited, indicating which kinds of applications are part of the product line and which are not. According to product line researchers, a wide field could have dramatic consequences on the complexity aspect, and a very limited scope could not justify the SPL investment.

What seems, on the other hand, not being well exploited is that an SPL could be itself seen as a composition of simpler SPLs. As a result, we can think of building a product line covering a broad scope and that is composed of several elementary product lines. This perception may also be explored for a medium scope (not very large) in the reason of simplifying the development processes and getting higher quality.

In this research work, we consider the e-Government domain as the scope to be covered by the product line on which are performed the various research activities. E-Government is defined by the European Community (2004) as “the use of Information and Communication Technologies (ICT) in public administrations combined with organizational change and new skills in order to improve public services and democratic processes”. Clearly, e-Government is not only to bring the benefits of existing service on the Internet. It is not the traditional government to which we have added the Internet but a fundamental change in the way that government do business with stakeholders of its information and services [8]. E-Government provides citizens, enterprises and governments as well by exciting benefits, namely: improving the quality and availability of public services; improving information, communication, and cooperation between the different actors; reducing administrative costs; allowing citizens to a better participation in different democratic processes kinds; and so on.

On the one hand, the major factor that raises similarity between e-Government applications is the fact that they are intended mainly to citizens. Citizen represents the central point of all e-Government services. Those latter are designed to fulfill daily citizens' needs in the different life aspects (birth, education, wedding, health, employment ...). Whatever is the application's domain there is some key features related to citizens that are common to all of those applications, namely: authentication, user management, privacy, protection of personal data, collaboration... Since e-Government applications handle personal data, communicate, and transact with users they must be highly protected. Thus, security functionalities represent another common point. E-Government applications need to communicate, to share and exchange data in order to provide efficient services to citizens. Communication is also a major feature that must be considered when developing e-Government applications. Furthermore, there are other characteristics that may be included in any e-Government application, such as: e-Meeting, statistics, poll, research, and so on.

On the other hand, in the context of citizen centered services, computing becomes pervasive. The number of applications that focus on the citizen's concerns will depend on the citizen's requirements, behavior and, of course, on government services (local, regional and central). As a result, the number of

applications will increase, the lifetime of these applications may be short and new required applications must be rapidly produced.

E-Government includes several sub-fields: eServices supplied by public administrations such as: vital records, passports, identity card, voting, poll, justice, etc; in addition to other services that could be provided by private or public institutions such as: education, health, assurance, transport, retirement, services for disabled people and so on.

- *E-Government is the second major problem of this research work. It is necessary to determine whether these applications or very important parts of them could be derived from a product line, whether been simple or composite SPL.*

The objective of the work is to set up the methodological and technological bases for the establishment of a product line for the rapid production of government applications according to a component-oriented design approach.

### 3. Overview of the Proposed Solution

The work presented in this thesis presents a first step towards the foundation of an approach intended to the development of Component-Based Software product lines and the management of product lines in complex and pervasive fields like e-Government.

The approach that we propose intends the management of reuse within SPLs at two abstraction levels: between the applications included in the same SPL, and across SPLs that belong to the same field.

In order to manage reuse within SPLs, we propose “Component Based Product Line” (CBPL) approach. The approach presented in this chapter aims to reach a high level of reuse that can be obtained through the integration of two approaches: Software product lines and Component-based development. Each of these approaches promotes reuse at different granularity levels. Component-based development supplies technologies for reuse in the small, while Software product line approach intends reuse in the large. Putting them together allows us to reach large scale reuse and flexibility at the same time. Moreover, Component-based

development can overcome the lack of maturity in SPL engineering by providing efficient development technologies.

For managing reuse among SPLs we propose Aspect Multiple Product Line (AMPL) approach. AMPL approach is a new MPL management approach that aim mainly to systematize reuse across separated SPLs within an MPL. AMPL could be efficiently applied in a software field if this latter produces applications for several subfields or market segments, such as those fields are characterized by significant commonalities which could be encapsulated in specialized SPLs (ASPLs). Reuse within each MPL subfield is managed through sub-SPLs while the commonalities among them will be managed thanks to ASPLs. The AMPL engineering process is based on two main activities: *the separation of concerns* and the *partial derivation*. Separation of concerns deals with decomposition of an MPL into two SPL types: sub-SPLs and Aspect SPLs. While, partial derivation intends the integration of these two SPL types early in the development process. We note that both of separation of concerns and partial derivation are autonomous from each other, i.e. each of them can be used independently. For instance, separation of concerns can be used in a MPL environment for structuring the MPL model. It can also be adapted for decomposing a large SPL into a set of sub-SPLs and thus moving from single SPL to MPL approach. On the other side, partial derivation can be adopted for the merging of two (or more) interdependent SPLs aiming inter-reuse even if they do not belong necessarily to the same MPL.

The proposed approach is validated in the context of e-Government field. Our first perception when developing e-Gov SPLs is that reuse within each e-Gov subfield is systematized through SPLE techniques; while when inter-SPL reuse is needed no techniques are available to manage it. Consequently, systematic reuse is lost at SPLs level. We suggest, thus, benefiting from SPLE advantages not only within each MPL subfield but also across separated and interdependent MPL subfields. This way, reuse is effectively managed at two levels: between products within each SPL and between SPLs of an MPL. Our approach, avoids all of those challenges by planning for reuse from the early development stages. This planning is ensured by the introduction of ASPLs that are responsible for the production of common components through the MPL. Thus a crucial outcome of our work is the systematization of reuse between SPLs of an MPL. The ASPLs are, after that,

partially derived in order to ease their integration with their reusing SPLs. The partial derivation represents an important technique for merging separated SPLs. It helps integrating the SPLs early in the development process to avoid the distributed derivation challenges thereafter. Thus, partial derivation and early integration represent other crucial outcomes of our work.

#### 4. Dissertation Structure

The thesis is organized into two parts, the first one presents the state of the art of the studied research fields while the second one presents the proposed approach and its validation.

In the first part, chapter 1 presents the main concepts of SPL approach in order to well introduce the studied field and clarify the crucial concepts that will be addressed along this dissertation. A classification for the variability modeling techniques within SPLs is, also, proposed and discussed. In addition, we introduce the new SPLE orientation which is MPLs. For this latter, we present their challenges and state of the art.

Chapter 2 presents an overview on the basic Component-Based Development (CBD) concepts (or more generally: the software architecture) and we study their integration with SPLE. We conclude this study by specifying the main aspects that should be covered when defining a new approach that integrates SPLE and CBD approaches.

Chapter 3 presents an approach that integrates SPL engineering and component-based engineering, aiming to unify the power of these two approaches. Firstly, we present a new FM notation that is better relevant to Component Based Development. Then, we present the development process of component-based SPL (CBPL). After that, we show the extension of a component-based approach in order to allow the modeling of variability in CBPL. Finally, the proposed approach is supported by a case study.

In Chapter 4 we propose a new approach, called AMPL (Aspect MPL). We start by presenting the background of our proposal. Then we discuss the foundations of our approach and we explain the AMPL engineering process. After that, we describe a case study to validate our proposal and we discuss the obtained

results. The chapter is finished by comments on related work and comparison to ours.

In Chapter 5 we present the partial derivation transformation techniques for two SPL models: the FM and the Architecture. Then we explain how the sub-SPL model is integrated with the set of partially derived ASPLs. The presented techniques are illustrated by a case study.

In Chapter 6, we review the e-Gov domain, we reports on its main challenges and we suggest the use of the approach proposed in this thesis to solve the problems encountered in this domain.

# CHAPTER 1

## SOFTWARE PRODUCT LINES

### 1.1. Introduction

Software development processes know various problems at different stages of their life cycle. Firstly, in specific domains, it seems like the same work is done for the development of each new product: the same functionalities are developed at different places and the same changes are repeated at different places. Secondly, once the product is delivered it may not fit the customers' need. Indeed, producers want to maximize their benefits and, thus, to minimize their production's costs and time to market. In opposition, customers ask for better quality and for software tailored to their individual needs. Thirdly, complexity and size of software products are rapidly increasing due to the market's evolution. Moreover, the higher the products' diversity is, the bigger the complexity and challenges for an organization are.

The Software Product Line (SPL) approach proposes solutions to cope with those problems. Hence, setting up such an approach is not a trivial task. Organizations need to learn how to manage a product line or how to improve their way of managing it. New ways for modeling, implementing and managing software production must be introduced in order to carry out a successful product line.

In this chapter, we start by introducing the core concepts of SPL approach in section 2. Then, we present the main steps of the SPL development process in section 3. After that, we report on a review of existing variability modeling techniques in section 4. Next, we give an overview on the new SPL paradigm

which is Multiple Product Lines (MPL) in the section 5 and we discuss the approaches related to MPLs engineering. Finally, section 6 concludes the chapter.

## 1.2. Core concepts of SPL

### 1.2.1. SPL definition, advantages and challenges

SPLs are emerging as a viable and important development paradigm allowing companies to realize order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers. A SPL is “A set of software-intensive systems sharing a common, managed set of features<sup>1</sup> that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets<sup>2</sup> in a prescribed way” [5]. SPL approach aims to systematize the reuse throughout all the software development process: from requirements engineering to the final code and test plans. The purpose is to reduce the time and cost of production and to increase the software quality by reusing software elements which have been already tested and secured. These objectives are realized by putting in common the development of various SPL artefacts such as: requirement documents, design diagrams, architectures, codes (reusable components), procedures of test and maintenance, etc.

The crucial aim of introducing product line approach in software engineering is to improve reuse. SPL differs from traditional reusing approaches in two ways: the first one is that SPL approach systematizes reuse throughout all the software development process: from requirements engineering to the final code and test plans, in contrast of traditional software reuse approaches which exploit reuse only at design and coding phases [3]. The second one is that, SPL improves reuse while maintaining diversity between products. In most cases, software components intended for reuse must be adapted to meet specific customers’ requirements. If flexibility is not considered since the early development process phases, as in traditional software reuse approaches, developers will meet lot of difficulties at

---

<sup>1</sup> A feature is an end-user visible characteristic of a system [4].

<sup>2</sup> A reusable artifact or resource that is used in the production of more than one product in a SPL. A core asset may be an architecture, a software component, a domain model, a requirements statement or specification, a document, a plan, a test case, a process description, or any other useful element of a software production process [5].



components adaptation step, which may lead them to leave those components even if they will have to start from scratch. SPL simplifies the adaptation phase by involving “Variability management” activity along all the software development process.

An SPL, when it is well introduced within an organization, can bring several benefits. From a software viewpoint, SPL approach enable reusing all software artefacts kinds (including requirements models, design models, components code, production tools, test procedures and any useful output of the SPL development process) unlike traditional reuse which base on reusing only code source lines or modules. SPLs provide techniques for better variability management and thus producing a variety of products basing on the same reusable elements base. Moreover, reuse allows significant reduction of time and effort of development and maintenance. From a business viewpoint, the SPL artefacts base once constructed allows the fast production of final applications. Time to market is then decreased in addition to the reduction of production, maintenance and test costs.

However, this reduction does not come by itself, since the launch investment of such an approach is relatively large, and the return on investment is not immediate [9]. The cost of the first software developed from the SPL will be even higher than in the conventional situation and breakeven<sup>3</sup> point cannot be reached until the development of a number of software. From another side, the information amount to be collected in order to establish a new product line is often significant. New techniques should be devoted to collect and manage this information as well as important organizational changes should be done. Furthermore, particular attention should be paid on the evolution and management of change in SPLs. Managing evolution of single software is complex. Managing the evolution of a software collection is extremely complex since their consistency must be preserved. Finally, in spite of the increasing adoption of SPL approach in various software fields, engineering methodologies and techniques are still immature and research work is further required in this area.

---

<sup>3</sup> At this point, the costs are the same for developing the system separately as for developing them by product line engineering [4].

### 1.2.2. Managing Variability in SPLs

Variability, generally, refers to the ability to change or customize a system [10] [11]. In product line context, variability means the ability of a core asset to adapt to be used in different product contexts which fall within the scope of a SPL [7] [12]. Unlike conventional software development, variations in the SPL context are preplanned. A good definition of variability helps us to answer three main questions: *what does vary?* *Why does it vary?* and *how does it vary?* [4]. The answer to “what does vary?” is a variable element or a variable property of this element, also called: *variability subject*. Answering “why does it vary?” question implies specifying the reason for which an element varies. The cause of the variability may be: customer’s needs, technical reasons, legal reasons, commercial reasons, variation of other elements in the case of dependency between them, and so on. The question “how does it vary?” is answered by describing the different forms a variability subject can take, also called: *variability object*.

Those concepts must be clearly described in all SPL artifacts. Variability subject is represented by denoting an element as variable, or linking it to a Variation Point<sup>4</sup> (VP). Variability object is represented by denoting elements as Variants. However, the reason of variation is generally not represented unless when it is due to a dependency between elements in a SPL. In this case, it is, widely, represented implicitly by dependency constraints. Even don’t considered, the documentation of this question is important. For example, when the reason for which the variability existed became obsolete, the VP must be removed and this will decrease the number of VPs and so the complexity of the system (simplifying variability management eventually). On the other side, if the reason of introducing a VP was not retained, we may lose important information for variability management and SPL evolution. Another question that we can add to describe variability is: *when does it vary?* It means when the decision on a VP should be taken? The answer to this question is commonly known as the VP *binding time* [14]. The information about variability resolution or binding time is most of the time represented by an attribute related to the VP.

---

<sup>4</sup> A VP represents a delayed design decision. [16]

Variability must be identified, represented, exploited, implemented, evolved, etc. – in one word managed – throughout SPL Engineering (SPLE) [15]. Unlike traditional software development which deals only with variation over time, variation management in SPLE is multi-dimensional [16]. In addition to variation over time, SPLE handles also variation in space. This latter refers to managing differences among SPL members at any fixed point in time. When managing variability in a SPL, three main types are distinguishable [15]:

1. *Commonality*: common software elements to all members of the SPL.
2. *Variability*: software elements that vary from a member of the SPL to another one.
3. *Product-specific*: software elements that may be part of only one product of the SPL.

During the SPL life-cycle, a variability type may change. For example, a product-specific characteristic may become variability. On the other hand, a commonality may become a variability as well.

The first step in managing variability is *to identify it*. This corresponds to answering the question “what does vary?”, in other words, “what is the variability subject?” The answer leads us to denote the particular places in a software system where choices are made as to which variant to use. We refer to these places as VPs. Variability objects or *variants* are the different shapes of a variability subject. The variability points and the related variants must be identified in all kinds of development artefacts, i.e. requirements, architecture, design, code, and tests. After being identified, variability *must be represented*; approaches that allow representing variability are shown in section 4. The third step in managing variability is *to implement it*. In this step developers have to select the appropriate variability realization technique for each VP [17], it means, to choose the way in which the variability will be implemented within the software system. In fact, this task is not trivial because there are several factors that influence the choice of implementation techniques, such as: which variant or set of variants will be used? , by which software entities the variant will be implemented? , and how and when the variant will be bound to the related VP? In addition, due to the lack of guides that assist developers in selecting the most suitable variability realization

technique, ad-hoc solutions are usually proposed and used. Bosh and al [17] [10] present a taxonomy of variability realization techniques. This taxonomy presents the intent, motivation, solution, lifecycle, and consequences for each realization techniques presented. Finally, the artefacts obtained from all SPL development sub-processes *must be maintained and evolved* to fit new requirements. New variability could be added; others could be changed or even removed at all from the SPL.

### 1.2.3. Feature models

The Feature Model (FM) is the first language dedicated to the variability modeling [18]; it was first introduced in the Feature-Oriented Domain Analysis (FODA) method [19]. It has known a broad use in the field of SPLE, since it is a simple and easy to use language in comparison with other more complex modeling languages such as: Unified Modeling Language (UML) and Business Process Model and Notation (BPMN) [3].

Each product belonging to a SPL determines a particular context. Each product consists of a set of software elements represented by features [3]. Features at a particular abstraction level are successively decomposed into sub-features in the lower levels until obtaining terminal features [10]. Ideally, each feature is associated with a terminal reusable software component (component, service...) implementing the requirements determined by the corresponding feature. Since features could be added, changed or even removed from product to product, a SPL must support variability for those features. According to Klaus et al. [4] there are three types of features:

1. *Mandatory features*: represent common features between all the members of a SPL.
2. *Optional features*: are features which can be selected or left out a SPL member.
3. *Alternative features*: allows the choice of one feature out of a given set of features.

The FM is a description of the commonalities and differences between members of a SPL. It is generally described by a hierarchy of the set of a system's

features or what is called *feature tree* [9]. Figure 1.1 shows a sample feature diagram for Evaluation component extracted from the e-Learning SPL<sup>5</sup>. The figure summarizes the main notations of basic FMs. The Evaluation component must offer at least the “Online Test” feature therefore this feature is denoted as mandatory in the diagram. However, other features can be added according to the customers’ needs such as: Scoring, report card, and certification. Thus those features are represented as optional in the feature tree. From another side, only one feature from an Alternative features set could be included in a final product as it is the case for “Math qst” feature. Dependency constraints must be respected when software elements are assembled. They represent dependencies that could not be expressed only using hierarchical relationships provided by the feature tree. For instance, *requires* dependency between Report card and Scoring features means: when Report card feature is included in an e-Learning application, scoring feature must be included necessarily.

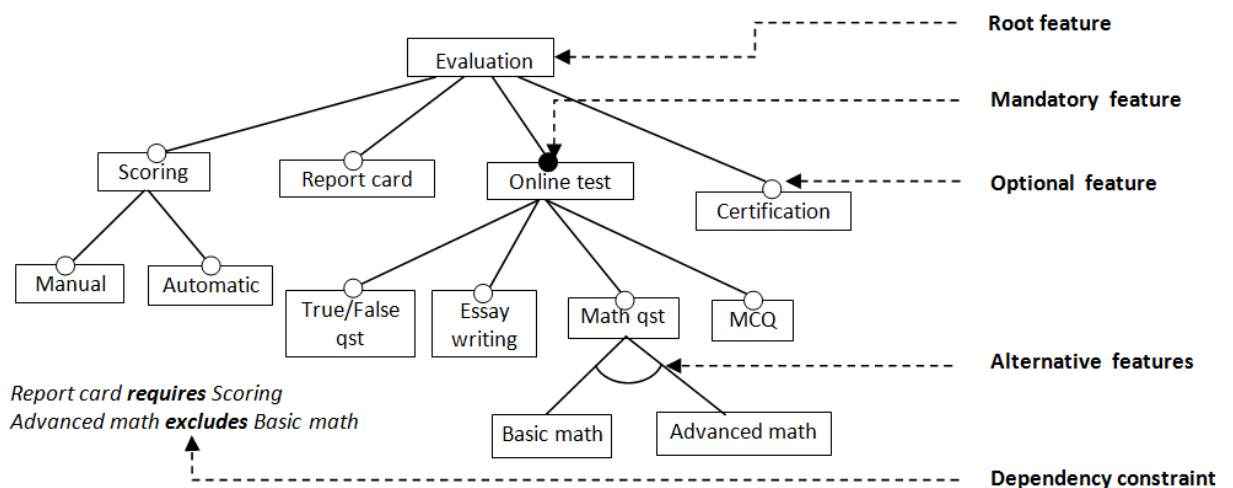


Figure 1. 1: Example of a FM (part of the evaluation component FM from the e-Learning SPL).

### 1.3. SPL engineering

SPL approach intends to adapt to the software development, the principles that we find in the automotive industry, aeronautics and electronics. Production in these industries is organized into ranges with similar parts and offering a number of options. For example, in automotive industry, various car models come out the same assembling chains using the same chassis, the same engines and the same

<sup>5</sup> The e-Learning SPL is presented in Chapter 5.

test plans. The aim is to improve productivity, decrease time to market, and reduce production, maintenance and test costs. The idea is to transpose the principles of manufacturing of these industries in software development, to benefit from the before-mentioned advantages.

So instead of developing a single application at a time, SPLE allows developing an applications-set belonging to the same domain and characterized by a set of common software elements. The purpose is to pool the development, maintenance and test activities of those common software elements in order to reduce production and maintenance costs; reduce production time and improve quality. Therefore, SPLE relies on a fundamental distinction between *development for reuse* and *development with reuse* [4] [9] as depicted by the Figure 1.2. Domain engineering or “development for reuse” is the SPLE sub-process that aims to define and realize the commonality and variability of the SPL. Application engineering or “development with reuse” aims to derive automatically or semi-automatically the final software from the reusable elements realized during domain engineering.

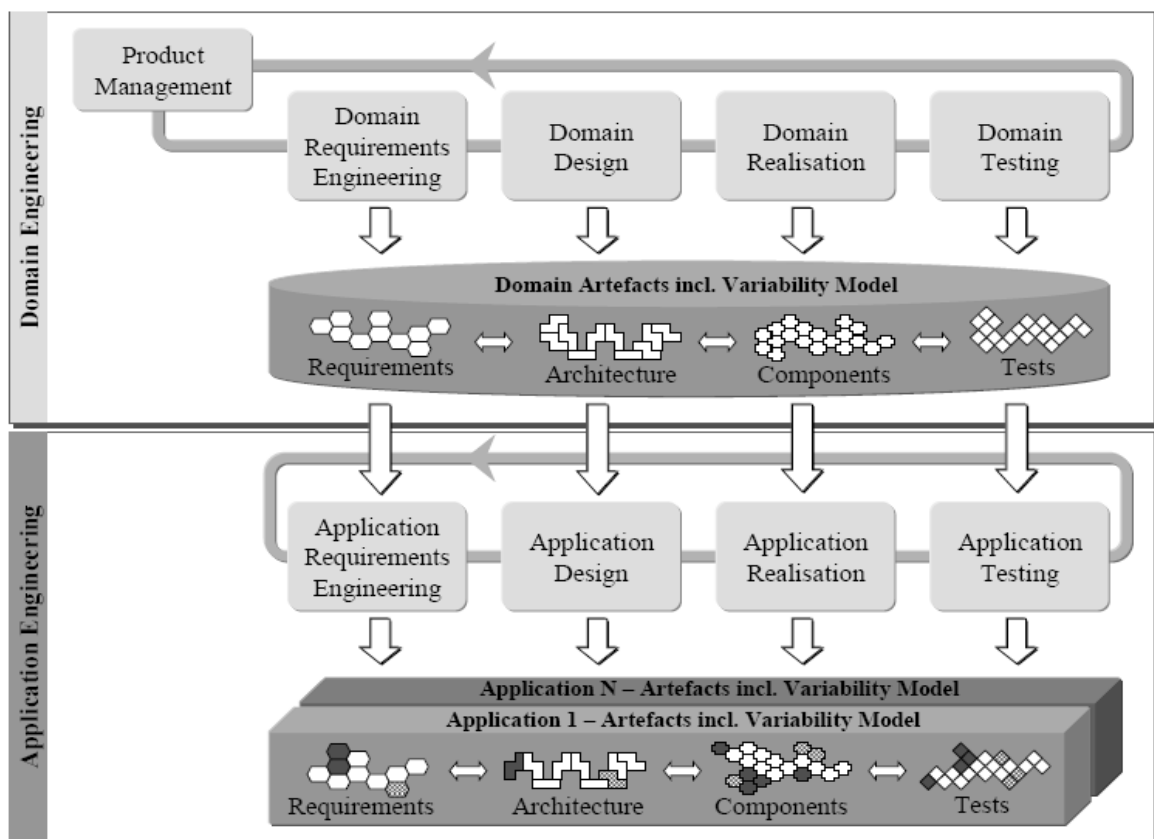


Figure 1. 2: SPL engineering process [4].

### 1.3.1. Domain engineering

Domain engineering consists in developing the reusable elements (core assets) for the SPL through domain analysis, domain design and domain implementation. The main outputs of this process are: the identification of the SPL members (scope), and the extraction of similarity and variability between them.

1.3.1.1. Product management (scoping): the crucial aim of product management is to define the scope of the SPL i.e. which products belong to the SPL and which do not. Its output is a product roadmap that determines the major common and variable features of foreseeable products. Delimiting the SPL scope is a key activity in SPLE since a too broad scope tends to increase the SPL complexity and a too narrow scope does not justify the investment made for adopting a SPL approach.

1.3.1.2. Domain requirements engineering: the main activities of this sub-process are the elicitation, documentation and management of common and variable SPL requirements. Its input consists of the product roadmap. The resulted reference requirements encompass reusable requirements specification (textual and model based) in addition to variability models.

1.3.1.3. Domain design: this sub-process takes as input the reference requirements and elaborates the reference architecture of the SPL. The reference architecture provides a common, high-level structure for all SPL applications [4].

1.3.1.4. Domain realization: during this sub-process reusable SPL assets are planned, designed, and implemented for reuse in the different SPL applications. Its inputs are reference architecture and a list of reusable software artefacts. It results in the detailed design and implementation of software artefacts.

1.3.1.5. Domain Testing: it includes activities of verification and validation of reusable software assets. Tests are performed against the SPL specification artefacts and result in reusable test artefacts.

### 1.3.2. Application engineering

Application engineering consists in developing the final products, using the core assets and the specific requirements expressed by the customers. This process is similar to traditional development process; however, each step is facilitated by reusing the outputs of the previous process (domain engineering). It intends to achieve the widest possible reuse of core assets by exploiting commonality and variability of the SPL.

1.3.2.1. Application requirements engineering: during this step, reference requirements are reused in order to allow selecting the variability and also to focus on the new software's specific needs.

1.3.2.2. Application design: designing the software architecture involves configuring the reference architecture according to the selected variability and eventually adapts it according to software specific needs.

1.3.2.3. Application realization: since reusable components are developed, this step is limited mainly to develop new components or extend the reusable components to take into account the specific needs. Reusable and application-specific components are assembled to construct a final application ready to be run.

1.3.2.4. Application Testing: as reusable components are tested during domain engineering; the main objective of this step is to ensure that interactions between reusable components and specific application components do not generate an unexpected behavior.

### 1.4 Variability modeling techniques

Variability management is a key activity that usually affects the degree to which a SPL is successful [20]. As mentioned in section 2.2, the second step in managing variability is to model it in all software artefacts. Modeling variability is a technique used to document variability and to reason about it. Its main objectives are: to make the variability explicit in the early stages of the project, and to reduce the complexity related to variability management throughout the development



process [3]. The first language dedicated for modeling variability was first introduced in the FODA method by Kang in 1990 [19]. During the last decade, numerous methods and techniques for representing SPL variability have been defined [20]. As shown in the Figure 1.3, we classify these techniques in two main categories: representing variability in separated models and representing variability in artefacts. This section gives an overview of some techniques belonging to these two categories and organizes them in more detailed classification.

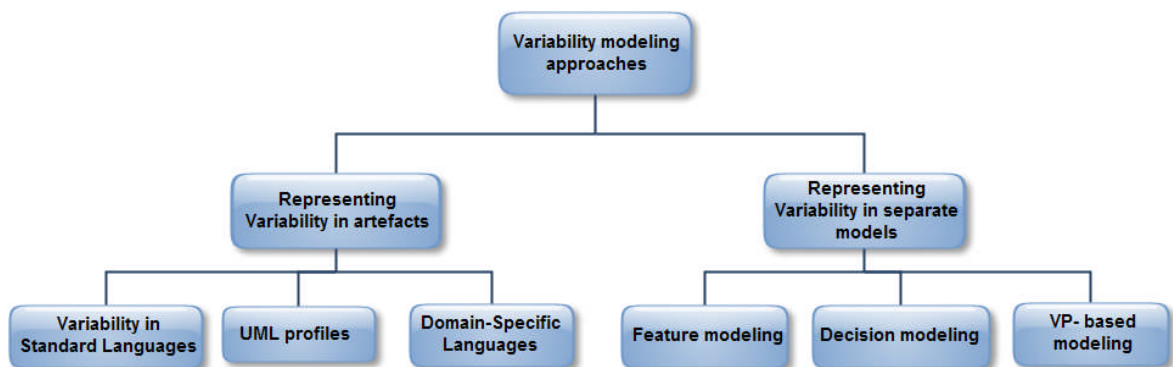


Figure 1. 3: Classification of variability modeling techniques.

#### 1.4.1. Representing variability in separated models

In separated variability models, variations are isolated from other modeling diagrams (structural and dynamic), and expressed in an explicit way using separate textual or graphical models. In this class we can distinguish three modeling kinds: *feature modeling*, *decision modeling*, and *Variation Point based modeling* (VP-based modeling). These subclasses are differentiated by several properties, mainly:

- Feature modeling is used to model variability and commonality while decision modeling and VP-based modeling focus on variability modeling. However, each of them provide derivation support;
- Mapping to artefact is an essential concept in decision modeling and VP-based modeling, while it is optional in feature modeling;
- Although decision modeling and VP-based modeling have similar properties they are differentiated by the fact that decision modeling model variability –in

general textually in terms of *decisions* while VP-based modeling model it graphically as VPs.

Next subsections define each one of these subclasses and report on some approaches that they include.

#### 1.4.1.1. Feature modeling

Feature modeling is the activity of identifying externally visible characteristics of products in a domain and organizing them into a model called FM [21]. A FM is a description of the commonalities and differences between members of a product line. It is generally described by a hierarchy of system features or what is called feature tree [22]. Feature modeling has known broad usage in SPLE due to the concept of **feature** which can effectively supports communication between various stakeholders of a SPL. Feature models represent an intuitive and natural way of representing commonalities and differences in a domain [14]. In addition to the feature tree, a FM may contain some additional information such as: descriptions of each feature, stakeholders and client programs interested in each feature, examples of systems with a given feature, constraints, and so on [23]. Furthermore, modeling the semantic content of features usually requires some additional modeling formalism, e.g. object diagrams, interaction diagrams, state diagrams, synchronization constraints, etc. Thus, FMs are usually just one out of many other kinds of models describing a piece of reusable software.

As stated previously in this chapter, feature modeling was first introduced in the FODA method [19]. Afterward, it has been widely adopted by the SPL community and a number of extensions have been proposed. According to Lianping et al [20] FODA has the largest number of approaches that based on it. The first one was Feature-Oriented Reuse Method (FORM) [24]. FODA introduces the concept of using a feature model for requirements engineering, while FORM extends it to the software design and implementation phases, and prescribes how the feature model is used to develop domain architectures and components for reuse. After that several extensions or new feature modeling approaches have been proposed. For instance, FeatuRSEB [25] is a combination between the FODA method and the Reuse-Driven Software Engineering Business (RSEB) method. RSEB uses features informally as use cases or parts of use cases. In this work; the FODA

feature diagram is changed in a tree or a network of features which are linked together by UML dependencies or refinements and the VPs are explicitly represented. This FM is used as a "roadmap" for other RSEB models, guiding the product line engineers. Feature models have been also extended by adding cardinality concept, as in the work of Riebisch et al. [26] and Czarnecki et al. [27] [28]. Cardinality-based FMs associate to each feature a *feature cardinality*. A feature cardinality is an interval of the form [m..n]. The interval denotes how many occurrences of the feature (with its entire sub-tree) can be included in a concrete configuration. Cardinality-based FMs aims to enhance the FM for representing more features variation cases.

#### 1.4.1.2. Decision modeling

Unlike feature modeling that was designed at first to model the commonalities and variations within a domain, decision modeling was initially conceived to ease the derivation of products from a SPL base. According to Czarnecki et al [29], the earliest decision modeling approach is found in the Synthesis method [30]. This latter define a decision model as *a set of decisions that are adequate to distinguish among the members of an application engineering product family and to guide adaptation of application engineering work products* [30]. Decision modeling captures variability in terms of decisions and their possible (potential) resolutions. A decision model is generally represented by a table. Rows represent decisions and columns represent properties of decisions (Identifier, Question, VP, Resolution set, Effect or constraint) [31].

KobrA approach [32] captures variability using a textual decision model related to each product line artifact. Each variability is related to at least one decision in the decision model. Each decision provides a set of possible resolutions and lists for each resolution, which diagrams must be tailored in what way to represent the specified member of the system family. Representing decisions using tabular or textual models makes decision models hard to exploit or maintain especially for large SPLs. Forster et al [31] propose a graphical notations to model several views on decision models in order to manage complexity related to the large variability.

### 1.4.1.3. VP-based modeling

Seeing existing techniques which represent variability in separated models, some techniques do not belong neither to feature modeling nor to decision modeling. They represent variability graphically in terms of VPs, variants and dependencies between them. The resulting VP-based models are, then, related to other artefacts in order to specify the variability they contain. VP-based modeling aim to ease variability management throughout all the phases of SPL development by keeping trace of variability along all the process. According to Gomaa [33], the VP-based modeling builds on Jacobson's original concept of VPs [34], such as *A VP identifies one or more locations at which the variation will occur*. This concept has been evolved after that, and constructed the base of several new approaches. Two main approaches have been proposed in this area: Orthogonal Variability Model (OVM) [4] and COVAMOF method [35].

OVM is a model that defines the variability of a SPL. It relates the variability defined to other software development models such as feature models, use case models, design models, component models, and test models [4]. The basic elements of an OVM are: VP and Variant (V). A VP is defined as *the representation of a variability subject within domain artefacts enriched by contextual information*. A VP offers a certain variant that *represents a variability object within domain artefacts*. VP and V are associated by means of *variability dependencies*. The method defines three kind of this latter: Optional, Mandatory and alternative choice. Restrictions between different VP and V are represented by *constraints dependencies*. Those relationships may link a V to V, V to VP, or VP to VP, and have two types: requires and exclude. Finally, variability defined in the OVM must be related to other artefact models by means of traceability links.

COVAMOF method [35] provides the CVV (COVAMOF Variability View) to model variability for all Product line artefacts. It defines two views of variability: VP view which represent VPs and Vs, and dependency view which represent dependencies between VPs. However, COVAMOF do not show how traceability between product line artefacts and CVV is maintained.

### 1.4.2. Representing variability in artefacts

Representing variability in separated models involves creating new models for variability in addition to the existing specification models (artefacts) used to model a system at various abstraction levels. Variability models have to be well managed and their dependencies to the artefacts must be maintained along all the development process, which is not a trivial task. Representing variability in artefacts approaches aim to simplify the task of managing variability by integrating it in the existing artefacts. The variability information is represented by standard concepts, by adding new concepts to the used language, or by considering it when conceiving a new language. So, we can distinguish three subclasses of this class: variability in standard languages, variability by enhancing languages, and domain-specific languages. Next sub-sections give an overview of each of them.

#### 1.4.2.1. Variability in standard languages

In this kind of approaches, available mechanisms in a given language are used to represent variability without extensions. Kakola et al. show an example of that [36]. They demonstrate how UML 2.0 could be used in order to express variability. They focus on three types of UML 2.0 mechanisms which support the expression of variability:

- Templates parameters, used to assign a type of variation to classes and packages.
- Plug-ins used in Component-Based Approach, such as, the variations are isolated in components which are external to the stable parts of the system (framework), with well-defined interfaces that apply to all variant components. The components are fitted into the framework through interfaces.
- Specialization-Redefinition: VPs and variants are represented by abstract classes and their subclasses.

According to them, the same concepts could be applied to other graphical or textual languages, such as Java or Specification and Description Language (SDL) [36], to support the expression of variability.

#### 1.4.2.2. Variability by enhancing languages

Actually, standard languages have not been developed to capture all variability types consistently and explicitly. So they have been enhanced to meet the needs of SPL engineering. Variability by enhancing languages has been introduced in various models kinds, mainly: UML profiles, and Architecture Description Languages (ADL).

UML profiles define UML extensions in order to express commonalities and variations in models that describe SPLs. Several UML profiles have been proposed. Matthias Clauss [37], [38] suggested a UML profile to model variability of a SPL; the resulted model is called “Generic Model”. This latter will be instantiated to get specific models for every members of the SPL. Extensions are performed using *stereotypes* and *tagged values* as follow:

- To express a VP and their variants, he applies the stereotypes «Variation-point» and «Variant» to all of: classes, components, packages, collaborations and associations.
- Uses stereotypes «requires», «mutex» and «evolution» to express respectively requires, mutual exclusion and evolution dependencies. The stereotype «evolution» is then separated to three types: « replaces », « decomposition », and « extends ».
- Uses tagged values to determine binding time, multiplicity, and to specify to which VP a V is assigned.

Matthias proposes also an extension to FM (add the feature type « external<sup>6</sup>»), however, his work focus on variability modeling of the structural aspects of SPLs.

Ziadi and al. [39] [40] [41] propose an extension to structural and dynamic views of a system, and show how the product line model will be derived to get a model for each product. They try to add more semantic to their models using, for example, the stereotype «optional» which describes that a given element may or may not be present in a product derived from the SPL. Ziadi and al. suggest also the use of two types of SPL constraints: the generic constraints that apply to all the

---

<sup>6</sup> a feature realized by the underlying platform, not by the system itself.

SPL and specific constraints that concern a specific SPL. These constraints guide the derivation<sup>7</sup> process.

Other works interest to represent variability in architectural model using UML profiles. In [42], Paula and Paulo proposed representing variability in component model based on UML extensions. Mechanisms used to define variability between components are inheritance and interfaces. In [43], Maryam and Ramtin present a UML profile to model variability in Component and Connector view (C&C) of the architecture<sup>8</sup>. Variation in components is represented by stereotypes «alt\_vp» for alternative VPs, «opt\_vp» for optional VPs in addition to «variant». The same extensions are used to represent variation in interfaces and connectors. If a connector introduce a variation it is modeled by a UML class noted by variation stereotype. Maryam and Ramtin introduce also some derivation rules. In ADLs, we can find Koalish ADL [44]. Koalish is based on the Koala ADL [45] a component model which aim to support flexible instantiation and late binding of components. Koalish extend Koala to model variability in SPLs.

#### 1.4.2.3. Domain-Specific Languages

Domain-Specific Languages (DSL) suggest that the variation within SPL should be managed with a well focused modeling language specifically tailored to the product domain in contrast to the traditional modeling languages that try to be as general as possible [36]. While general modeling languages represent domain concepts by means of classes and components, domain-specific languages express these as language constructs, so we do not have to add any annotation in order to represent variability.

However, only few works have been done in this type of approaches. We find, for example, the DSL presented in [36]. The authors show an example on a watch SPL. The concepts used for describing the variability in this SPL are the concepts

---

<sup>7</sup> A product derivation consists in generating from Product Line models the specific models of each product.

<sup>8</sup> There are two perspectives to model software architecture. Either as a single dimensional concept that is modeled precisely in an ADL, or defined as a complex entity that is represented in multiple views. C&C view of architecture could be considered as the intersection of the ADL and multiple view perspectives [46].

related to the watch, it means: Displays, Buttons, Alarms, Time units, Icons and so on. For instance, if we consider that the number of icons may vary from a watch to another, this variability point is directly presented in the modeling language using the concept “Icon”, and the instances (variants) are, for example, Timer or Stopwatch icon. Another DSL is proposed by Voelter and Visser in [46]. They suggest the use of DSLs in SPLE to fill the expressive gap between FMs and programming languages. Their choice is illustrated by real world examples. Nevertheless, they propose to combine DSLs and FMs to get better variability management and gain the benefits of both approaches.

### 1.4.3. Discussion

Instead of the important number of variability modeling approaches proposed in literature, there is lack in different aspects:

Feature modeling focus on the feature view of the system. Yet, features are not sufficient for describing all the relevant aspects of SPL and they should be supported by other modeling languages. In addition, modeling variability in a FM may lead to misinterpretations. There is no single, commonly accepted definition for the feature concept [47]. It could be interpreted as an end user visible characteristic of a system, a distinguishable characteristic of a concept (e.g., system, component, and so on) or logical unit of functional or non- functional behavior...Therefore, its interpretation is ambiguous and depend on the reasoning of the developer what may bring disagreements among SPL development teams. Moreover, the feature tree lacks a grouping mechanism that would allow arbitrary features to be assigned to some variants [4].

Decision modeling provides an important mean to take decisions on variability during derivation step. Variability views are related to all SPL artefacts, these dependencies must be maintained throughout all the development process and this task is hard to perform. VP-based modeling enhances the expressiveness capabilities of variability models what leads to clearer variability definitions and avoids misinterpretations. However, decision modeling and VP-based modeling requires the combination of conventional specification models with new notations of variability models. This implies the need for new tools in order to perform and



manage the variability models and their dependencies with other specification models. In fact, all of feature modeling, decision modeling and VP-based modeling techniques lack standardization of concepts and modeling notations which makes that various new approaches are proposed continuously.

Variability in standard languages gains from standardization and tool support. But, since there is no annotations which specifies variation we cannot distinguish between the SPL models and specific product models. Keeping track of variability throughout the development process is hard, and a lot of difficulties will be confronted in derivation phase. Variability by enhancing languages proposes solutions to the shortcomings of the standard languages. The purpose is to be able to represent variability explicitly in the different SPL artefacts and gain from tool support at the same time. Nevertheless, most of these approaches model only some aspects of the SPL.

UML profiles have known a broad utilization in SPLs. Yet, only few works consider the representation of variability in different views of the SPL model, such as the work of Ziadi et al. Even when variability is expressed explicitly, derivation rules must be well introduced to guide the developer in instantiation step, only few works take this concern into account, as in [41], [43]. Domain-specific languages can use graphical, textual, or tabular syntax, or any combination of them. Similar to programming languages, DSLs can be compiled using code generator. DSLs allow the automation of derivation activity, this latter cannot be achieved without creating the language and generators that fit with the product line.

## 1.5. Multiple Product Lines (MPLs)

### 1.5.1. MPLs meaning, benefits and issues

Single SPLs are no longer sufficient in some environments due to the emerging of reuse across several interdependent SPLs what is known as MPLs. An MPL is defined as *a set of several self-contained but still interdependent product lines that together represent a large-scale or ultra-large-scale system* [48]. The reuse and composition of multiple software product lines is also known as Nested Software Product Lines, Hierarchical Product Lines or Composite Product Lines. MPL

configurator is an assembly entity that is responsible for controlling and reusing the SPLs artifacts according to the customers' needs and producing diversified final products (see Figure 1.4).

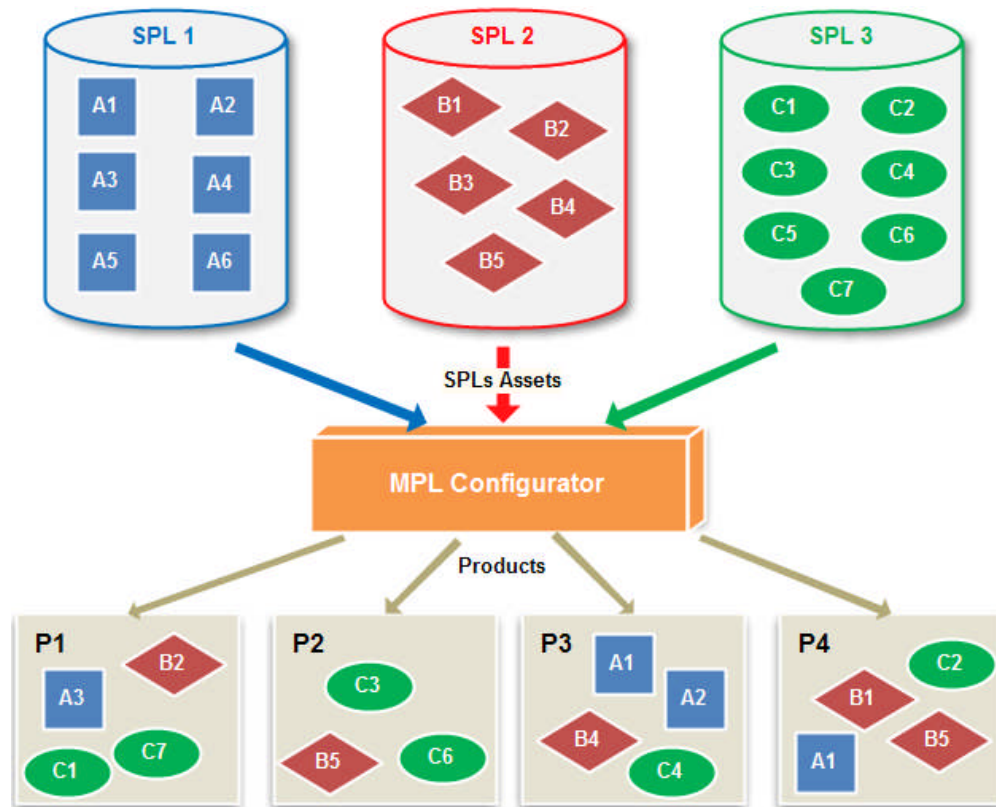


Figure 1. 4: Multiple Product Lines.

A crucial reason for introducing MPLs is the need for separating between several business purposes including different sets of commonalities and VPs. In large scale systems adopting a single SPL may generate more engineering challenges instead of resolving them. Software assets will be generic and may include several variations which make their design, implementation and maintenance hard to perform. Thus, a more focused SPL scope is needed to allow stronger constraints on variability [49]. For domains that have a creeping scope<sup>9</sup>, splitting their production activity into several SPLs allows supporting current subfields and adding new SPLs to cover new requirements when needed without having to alter the whole existing assets base. MPLs can also reduce the overall risks for the company [49]. If one SPL is modified by introducing new technologies

<sup>9</sup> A scope which is not strictly delimited, it could be extended to cover new requirements or fields as it is the case of e-Government.

or altering its infrastructure, the other SPLs included in the MPL will be safe whatever the results are. Yet, MPLs emergence has given rise to several challenges for SPLE:

Distinguishing between SPLs within the same field results in losing reuse information between them. MPLs need, then, to manage reuse across the several SPLs they include [48] in order to reach larger scale reuse. Two solutions are conceivable: adopting direct inter-SPLs reuse or developing a broad SPL covering the MPL scope. If developers choose direct reuse between separate SPLs they have to adapt components to fit the new requirements. This will draw them back to the problem of unplanned reuse. In this case, adaptations are limited and make the developers' work laborious and error prone what may push them to prefer developing components from scratch. Otherwise, Developers may choose to systematize reuse between separated SPLs by developing one SPL covering the whole MPL domain. This will result in a broad SPL covering several MPL subfields and thus several business purposes. However, numerous problems can arise from broadening the SPL scope [50], mainly: decreasing complete commonality (common components to all products) and in return increasing partial commonality (components common to a set of products), over-engineered SPL architecture and hard variability management due to the increased complexity of the SPL. Consequently, efficient methods are needed to manage inter-SPLs reuse within MPLs environments.

Current MPLs tends to compose SPLs components at derivation time. So instead of dealing with a single SPL derivation, developers must choose components to be reused from other SPLs, adapting them to the current reusing context and integrating them with the reusing application. This integration way known as distributed derivation [48] results in several problems. Reused components that are already developed using particular modeling and implementation techniques must be adapted to fit the new application requirements. Components must suit not only to one reusing product but to various products included in the various reusing SPLs. What multiplies the adaptation processes and thus delays derivation and increases cost and time of development. Moreover, choosing the right component from several competing

SPLs to reuse is by itself a problem that needs a whole decision process to be resolved. Furthermore, integration activity may require reviewing the whole reusing SPL architecture, or imply important adaptations for the reused components, making thus this process a hard and laborious task to perform.

MPLs are hard to be managed using a single model due to their size and complexity. Thus MPL model needs to be decomposed into several models that can be managed efficiently by separate teams. Techniques are then needed to decompose the MPL model into smaller units more likely to be managed easily. Yet, dependencies between SPLs models must be considered since they belong to the same field and represent together a large-scale system. Those dependencies are involved thereafter to ease the composition of MPL's SPLs when needed to get complex systems. Yet, SPLs composition approaches within MPLs are still immature.

### 1.5.2. Overview of approaches related to MPLs

Currents approaches intended for resolving MPLs challenges often base on the MPL model structuring problem, and consider the composition of SPLs instances. Here we summarize the most important approaches for three MPLs issues which interest us in our research work and that we have explained in the previous subsection:

- managing reuse across MPLs;
- structuring MPLs model;
- and the distributed derivation

#### 1.5.2.1. Managing reuse across MPLs

Schröter et al [51] [52] introduce multi-level interfaces to guaranty the correct collaboration between multiple SPLs. They distinguish between four interfaces: variability-model interfaces, syntactical product-line interfaces, behavioral product-line interfaces, and non-functional property interfaces. These interfaces aim to detach the direct dependency between SPLs and to enable modular analysis of MPLs correctness. They are defined as follow:

- Variability-model interface: is a specialization of the reused SPL's variability model.
- Syntactical interface: represents a view of an SPL's reusable code artefacts without implementation detail.
- Behavioral interface: is an agreement on the behavior of different methods.
- Non-functional interface: represents non-functional properties of an SPL that other SPLs use.

Apparently, the introduced interfaces represent views on what could be reused from each SPL within an MPL. They are defined as collaboration means between SPLs of an MPL. Nevertheless, authors do not mention how the interfaces are realized or how one SPL is reused by another one.

Van Ommering [53] [54] proposes creating product population by focusing on composition over decomposition in order to manage reuse between product families. He distinguishes between two architecture kinds: global architecture and regional architectures. The global architecture contains only the necessary elements for cooperation, and much of the architecture work is shifted to regional architectures. Global (reference) architecture is defined in terms of concepts, rules, and a global decomposition of the full functionality of the domain into subsystems. A subsystem is a large compound component that implements the functionality of a certain sub-domain. Product architecture is specialization of the reference architecture. Product families in this work do not much to SPLs which are usually intended to different business purposes but rather to products including variations. The work does not present techniques for managing reuse across separated SPLs but in fact between different products types that they consider as product families. It suggests components to be context independent in order to increase their reusability which is a technique already treated by SPLE.

Altintas and Cetin [55] propose "Software Factory Automation" method to manage reuse across distinct SPLs based on "domain specific kits" and "software asset meta model". Their strategy relies on the isolation of family design concerns in discrete building blocks, and later to compose them by means of a choreography model. Domain specific kits are responsible for the modeling and development of Domain Specific Artifacts in isolation and enable their composition

via a choreography model. This work consider a product family as a set of product lines, thus the studied SPLs have the same business purpose. The work is ambiguous in several aspects: What do represent those kits in fact? Are they common to all the SPLs of the product family? And on which criteria they must be isolated from other artefacts? Furthermore, how variation is managed? How reuse is systemized? And how those kits are integrated within reusing SPLs?

#### 1.5.2.2. Structuring MPLs model

Dhungana et al [56] propose an approach that organizes an SPL into a set of interrelated model fragments describing the variability of particular parts of the system. Model fragments help structuring the modeling space and provide support for evolution. This work is important in terms of structuring modeling space and merging models. The proposition is actually targeted to single SPLs environment, yet in MPLs environment decomposing the system into fragments that define reusable assets is not enough for managing complexity. Moreover, the work does not alter the variability management across product lines which is a crucial issue in MPLs.

Rosenmüller et al. [57] have altered the MPLs structuring problem. They propose to extend the FM with explicit modeling of SPL instances. The matter is to allow configuring a SPL using multiple instances of another SPL. In another work, Rosenmüller et al. [58] added the notion of composition model aiming to automate the configuration of MPLs. A composition model integrates multiple SPLs by describing for each SPL which instances of other SPLs it uses. The main shortcoming of this proposition is: delaying the SPLs composition until getting application level (where it is more likely to have incompatible instances derived from separate SPLs) complicates the derivation and integration tasks of reusing products with reused products. Reuse is thus limited since reused components must be adapted to fit new requirement which do not differ from conventional reuse techniques. Moreover, large scale systems are known to have complex FMs, extending FMs by modeling SPLs instances increase this complexity. Large and complex FMs need solutions for their management whereas their extension by new concepts makes this task more difficult.

Herman and Tim [59] propose to combine the FM with context variability model to model MPLs supporting several dimensions in context space. They use staged configuration to generate specialized FMs. The Context Variability model captures the commonality and variability of the context. The context is the environment in which a product resides. The Context Variability Model is combined with a conventional FM to create an *MPL-Feature model*. This work is useful for structuring MPLs models when those latter support several context dimensions. However, it would be more interesting to clarify the way these contexts are separated. This will help in splitting the MPL into several SPLs. Moreover, contexts are in continuous change and evolution, considering them at derivation time would be better than defining them in early development stages in order to take into account current requirements of the field.

#### 1.5.2.3. Distributed derivation

In conventional SPLs environments, the derivation of a final application implies usually a single user working on the derivation of a single variability model. Otherwise, MPLs environments include several sub-systems and multiple users are involved to derive the various variability models. Thus, multiple derivation processes are handled simultaneously for the various MPL sub-SPLs and this activity is known as distributed derivation. Distributed derivation is supported in the proposition of Rosenmüller et al. [57] by the extension of the FM and the composition model. SPLs instances are composed with the reusing SPL at derivation time to get a final application.

### 1.6 Conclusion

In this first chapter we have presented the main concepts of SPL approach in order to well introduce the studied field and clarify the crucial concepts that will be addressed along this dissertation. We have presented a classification for the variability modeling techniques within SPLs and we have discussed the properties of each class. Finally, we have introduced the new orientation of SPLE which is MPLs. For this latter, we have presented their challenges and state of the art.

The review of the existing approaches for MPLs engineering brings out three main aspects that must be considered when managing MPLs:

- Reuse among the SPLs of an MPL must be systematized i.e. constructed components must plan for reuse not only within a SPL but also between separated SPLs and variability should be managed efficiently within MPLs.
- Effective methods have to be developed for structuring the MPL model, starting from the various SPLs models and getting an MPL model including dependencies between the various SPLs. Dependencies have to be specified in order to simplify the MPL derivation thereafter.
- Solutions should be proposed to reduce the distributed derivation challenges or avoid them completely.



# CHAPTER 2

## SOFTWARE ARCHITECTURE AND SOFTWARE PRODUCT LINES

### 2.1. Introduction

SPL approach (chapter 1) intends to adapt to the software development, the principles that we find in the automotive, aeronautics and electronics industries. Production in these industries is organized into ranges with similar parts and offering a number of options. For example, in automotive industry, various car models come out the same assembling chains using the same chassis, the same engines and the same test plans. The aim is to improve productivity, decrease time to market, and reduce production, maintenance and test costs. The idea is to transpose the manufacturing principles of these industries in software development, to benefit from the before-mentioned advantages.

Although, the principles of SPLs are inspired by industry where production activity is based on assembling certified components, most of SPLs today are based on traditional design concepts of software engineering such as modular design and object-oriented design. These design concepts are not able to support the assembly of certified components which makes the derivation and maintenance processes difficult to perform. Consequently, SPLs remain immature because they still base on traditional design concepts of software engineering which are not adequate to the SPLE basic principles.

On another side, CBD provides the necessary means for the development of applications by assembling (existing reused) certified components. Thus, CBD matches perfectly to the SPLE principles. Furthermore, both of SPLs and

component based development promote reuse; putting them together will bring significant benefits to the software development.

Proceeding from this, in this chapter, we present an overview on the basic CBD concepts (or more generally: the software architecture) and we study their integration with SPLE. We conclude this study by specifying the main aspects that should be covered when defining a new approach that integrates SPLE and CBD approaches. Thus, we start by defining the crucial concepts of software architecture in section 2. In section 3, we focus on the IASA software architecture approach since it will support our work in the rest of this dissertation. Then, in section 4, we present a selected set of approaches integrating CBD and SPLE and we discuss their outcomes. Finally, section 6 concludes the chapter.

## 2.2. Core concepts of Software Architecture

### 2.2.1. Definition and advantages

Usually software architecture describes the structure of a software system at a high abstraction level. The software structure has always been recognized as a major concern. However, recently software architecture has emerged as an explicit discipline in the software engineering area. Nevertheless, despite the maturity of this discipline, there is no universal definition of software architecture [60]. According to [61] three classes of software architecture definitions can be distinguished. The first one defines software architecture as a high level abstraction representing the structure of a software system. The second class focuses on the concept of components and relationships between them. It defines software architecture as the structure and externally visible properties of a software system. The third one emphasizes on the fundamental concepts and constraints within which the software system is to be designed and developed. The direction of our research work falls within the second definition class. Thus, we define software architecture as *the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them* [62]. The term element is used to designate the primary building blocks of a software system (components or subsystems).

Software architecture deals with the design of the overall structure of a software system. It firstly identifies the high level components of the system called the

subsystems and how those components are connected to each other (connections or interfaces). The structure is defined in a way that minimizes the coupling between the different subsystems and increases the internal cohesion of these subsystems. Each subsystem must be: cohesive, performs a major service, contain highly connected components or classes with respect to the subsystem and relatively independent of other subsystems, and finally it can be decomposed further into smaller subsystems.

Software architecture often uses defined architectural patterns. An architectural pattern (or architectural style) *is a description of element and relation types together with a set of constraints on how they may be used* [62]. Patterns support reusing the design expertise by capturing and expressing the static and dynamic aspects of successful solutions to common design and implementation problems [63]. They help guiding the design and use of software systems and thus decreasing development effort and training cost. Each style articulates a set of subsystems (components); specifies their responsibilities and the relationships between them (connectors); and defines the guidelines for integrating components to form a system (constraints). Examples of architectural patterns would be data-centered architecture, data-flow architectures, layered architectures, object-oriented architectures, call and return architectures [64].

Software architecture can have a positive impact on several software engineering aspects [62] [65] [66], namely:

- *Understanding*: Representing the system at a high abstraction level allows a better understanding of complex and wide systems. Moreover, this common abstraction can be used as a basis for mutual understanding and communication between the system's stakeholders.
- *Reuse*: Software architecture support reuse at several aspects. Components, connectors and even frameworks can be reused in other similar contexts to their developing context. Furthermore, the architectural solutions proposed for particular problems can be applied to other systems exhibiting similar quality attribute and functional requirements.
- *Evolution*: Software architecture manifests the dimensions to a system that is expected to evolve; this makes it possible to estimate the effects and costs of the modifications. Furthermore, separating between components and

connections that allow them to interact, permit an easier change of connection mechanisms for handling several evolving concerns.

- *Analysis*: Software architecture express early design decisions about a system allowing thus new analysis opportunities namely: checking the system consistency, conformance to an architectural style, conformance to quality attributes, etc.

### 2.2.2. Components

Component represents the key element on which rely all of the design, implementation, and maintenance of component-based systems. It is the basic building block of software architecture. However, there is no consensus about its definition. Among the most cited definitions we find:

1. *An independently deliverable piece of functionality providing access to its services through interfaces* [67]
2. *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 third-party composition. (Clemens Szyperski, Component Software)* [68]
3. *A component is a non-trivial, nearly independent and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture* [64].

According to these definitions we can extract the main features of a component:

1. An independent, deliverable and replaceable composition unit of a system;
2. Provides clear functions intended for satisfying some needs;
3. Communicate with other components through its interfaces;
4. It can have features that make it reusable in a particular environment.

A component can be primitive as it can include a set of components; we talk here about *composite* components. It can provide simple functionalities as it can provide a whole complex application functionalities. We distinguish between two main component parts: the first one is the external component part called the *interface* [69]. The interface describes the set of functionalities provided or required by a component. It includes a set of interaction points usually known as

*ports*. The second part is the component implementation that describes the internal component functionalities.

The semantic definition of components includes its behavior and non-functional properties. A component behavior can be seen from two perspectives [70]: the static behavior that describes the particular “snapshots” during the system’s execution and the dynamic behavior that provides a continuous view of how a component arrives at different states throughout its execution.

The component *constraints* are the properties that must be checked when instantiating a component [70]. They define the required conditions and limits for running a component. Constraints may be defined in a separate constraint language or using the notation of the given ADL and its underlying semantic model.

A component is a software unit that explicitly describes its provided and required interfaces and its internal architecture. Components are subject to instantiation and connection with other components that meet their required interfaces or need their provided functionalities. Therefore, the key elements that must appear when documenting a component are: the required and provided interfaces, the internal architecture (for composite components), the component behavior and the components relations with other components. The Figure 2.1 shows a sample graphical component representation in a component-connector architecture view.

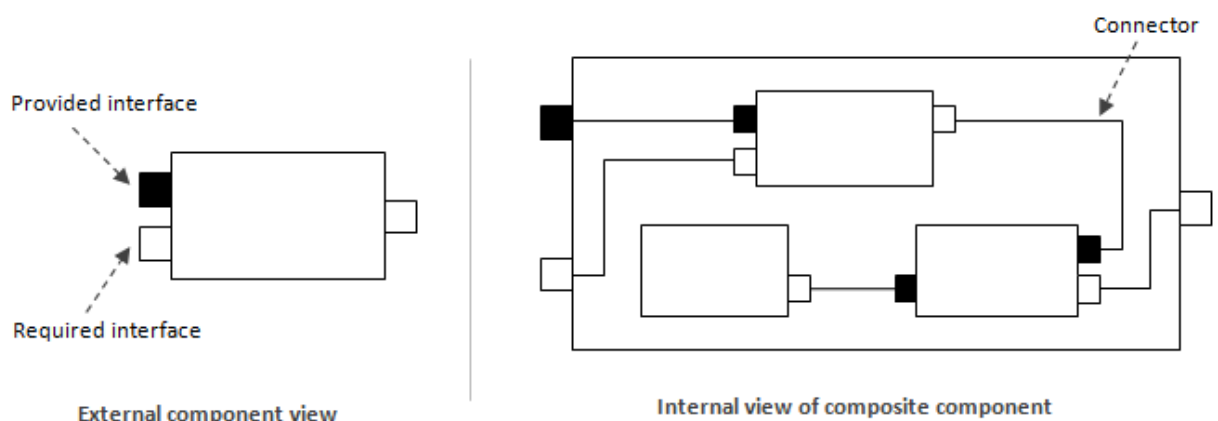


Figure 2.1: Sample graphical representation of components and connectors.

### 2.2.3. Connectors

The connectors are the glue for the system design. They represent interactions among the architecture components. Connectors mediate the communication and coordination activities among components [71]. A connector may implement simple interaction forms like: pipes; procedure call and event broadcast, as it may provide complex interactions such as a client-server protocol or a database access protocol. In the component-connector view, a simple connector is represented by a link (simple line) relating two components while a complex connector may be represented by a connection component that connects two or more components.

The main properties of a connector are: the interface and implementation. The connector interface describes the involved roles in an interaction. An interface includes a set of Interaction Points, usually called *roles*. A role (interaction point) defines the participant (ports) that can be involved in an interaction. The implementation defines the interaction protocols. Thus, the connector specification must describe both of the connector interfaces and implementation.

Finally, constraints on a connector express the conditions and limits that must be respected when running a connector i.e. the usage limits of the associated communication protocol.

### 2.2.4. Configuration

A system configuration represents the way a system is set up, or the arrangement of the elements that make up the system. It describes the overall system topology independently from the components and connectors it includes. Generally, systems are conceived in a hierarchical way, such as components and connectors may represent subsystems that have their own internal architectures.

A configuration defines the structure and behavior of a system made of components and connectors. The structural configuration of a system corresponds to a connected graph of the components and connectors building the system, while the behavioral configuration specifies the behavior by describing the links evolution between components and connectors.

### 2.2.5. Component based development process

The main objectives of CBD are: reducing cost and time for building large and complicated systems and improving the software quality by improving the component quality. The CBD process encompasses two parallel engineering activities as depicted in the Figure 2.2: Domain engineering and Component-based development [64].

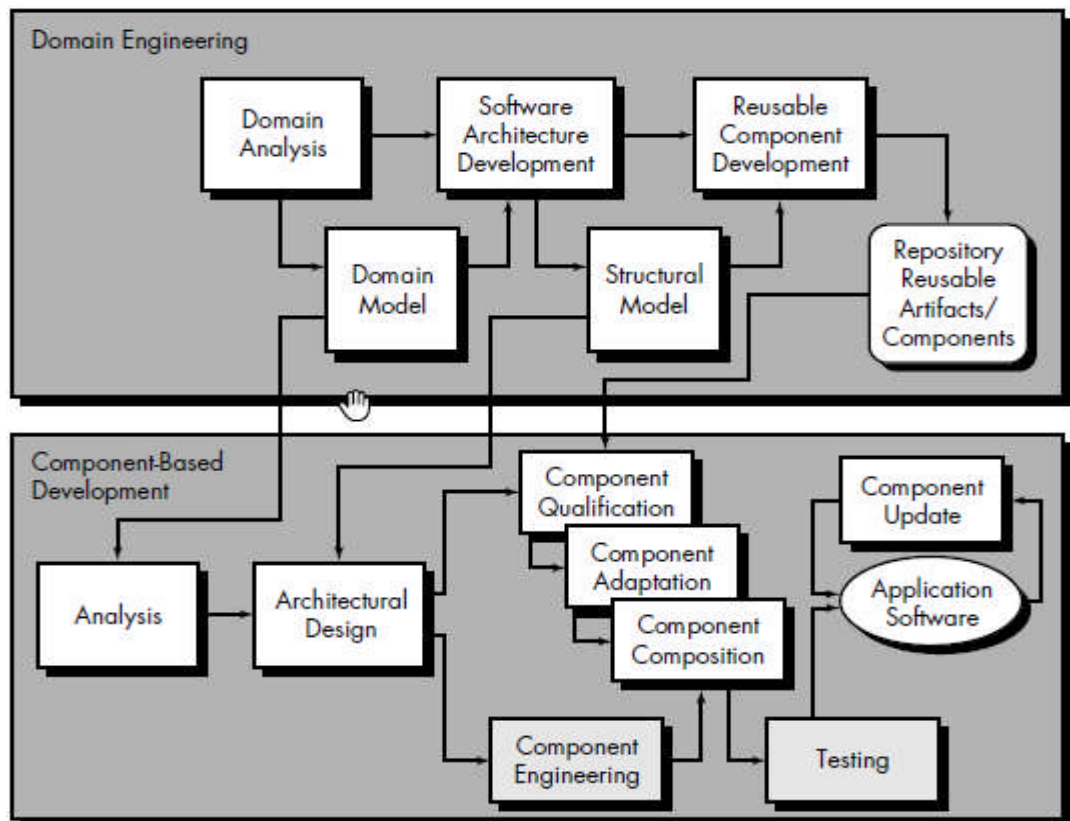


Figure 2. 2: Component Based Development process [69].

The first activity “domain engineering” aims to explore a particular application domain in order to find the possibly reusable components; the selected components are collected in reuse libraries. This sub-process includes three activities: analysis, construction, and dissemination.

The second activity “component-based development” produces applications by reusing as much as possible the existing components. It encompasses the following activities: eliciting the customers’ requirements, selecting an appropriate architectural style, selecting potential components for reuse, qualifying the components to be sure that they properly fit the architecture for the system,

adapting components if needed to properly integrate them, and integrating the components to form the application.

### 2.2.6. The ADLs

The ADL describes the system structure at an abstraction level that is the closest to the system designer intuition. An ADL is defined as *a language that provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation* [72]. Mary and David [73] present the main properties that are required for a good ADL:

- *Composition*: an ADL should be able to describe a system as a composition of independent components and connections. Composition techniques allow the combination of independent architectural elements into a large system.
- *Abstraction*: the abstraction is used for managing complexity. An ADL must have a set of abstraction concepts (components, connectors, and configuration) in order to describe the real word components and their interactions.
- *Reusability*: reusability allows developing new system by using as much as possible existing components. In an ADL it should be possible to reuse components, connectors, and architectural patterns predefined in different architectural setting.
- *Configuration*: ADLs should support the description of a system independently from its elements in addition; they should support the dynamic configuration.
- *Heterogeneity*: An ideal ADL should have the ability to inter-operate with other ADLs.

Multiple ADLs have been proposed in the academic as in the industrial environments. Among the most important ADLs we find:

*Darwin ADL*: like most ADLs, the main concept of Darwin ADL [74] is the component. A component is an instance defined by an interface that describes its provided and required services. Darwin distinguishes between two components kinds, primitive components that include the software code and composite components that represent configuration entities describing the connections between components. Darwin limits the instances number of a component during its instantiation. The main characteristic of Darwin is its capability for expressing dynamically changing process.



*Fractal ADL*: the Fractal ADL [75] is known to be an open, extensible and flexible language. It defines an abstract syntax independent of any programming language and intended to describe the architecture in terms of interfaces, links, attributes. It allows integrating new architecture description concepts to its syntax by modifying the XML descriptor that is loaded by the fractal factory. Fractal ADL aims to allow the dynamic definition, configuration and reconfiguration of component-based architecture, as well the separation of functional and non-functional concerns. A Fractal component consists of a membrane and content. The membrane includes a set of interfaces which define the required and provided services by the component and the content consists of a finite set of other components (called sub-components).

*RAPIDE*: is an event-based concurrent object-oriented language specifically designed for prototyping architectures of distributed systems. Its main goal is to check, by simulation, the validity of a given software architecture. In RAPIDE an architecture consists of a set of specifications (called interfaces) of modules, a set of connection rules that define direct communication between the interfaces, and a set of formal constraints that define legal and/or illegal patterns of communication [76].

*SEAL*: SEAL (Simple and Extensible Architecture Aspect and Action Language) is the ADL of the Integrated Approach to Software Architecture (IASA). It allows the specification and validation of the structure and behavior in software architecture. SEAL is provided by a high expression level due to the notion of *action context* [77]. It is based on the fundamental action contexts and all IASA types (components, ports, access point, connectors, exception, etc.). Its main goals are: to specify the ports behavior, to specify the interactions, to specify the global behavior of the operative part, and to define the primitive components regardless the implementation language.

### 2.3. The IASA approach

The IASA approach [70] was used to realize complex e-Government software systems, and was proved as a clear and easy specification language to design at a high level of abstraction using Aspect Oriented approach [78] [79]. IASA aims to provide the models and tools which have the ability to directly capture the

architect's mental model about a solution in the early step of a software elaboration process [80]. The IASA approach supports the Aspect Oriented Software Architecture (AOSA) specification through the distinction between two components kinds: aspect components and business components [80]. IASA allows the use of any component as an aspect component and any aspect component as a business component. Moreover, aspect components are not limited to represent technical concerns; they may be extended for other concerns such as Graphical User Interface (GUI).

In IASA, an application is organized in three spaces: the business space, the control space and the aspect space. The business space handles pure business logic which represents the solution to functional requirements of the system being designed. Components belonging to this space are called business components. The aspect space is mainly concerned with non-functional requirement of a system like tracing, security and persistence. Components belonging to the aspect space are called aspect components. The control space handles the specification of miscellaneous control operations such as the initialization and the evolution (e.g. the insertion or the removal of structural or behavioral elements) of the two other spaces. The control space is also the place where are maintained the information describing the structure and the state of the other spaces.

Next in this section we present briefly the main models of the IASA approach. The full IASA models description can be found in [70].

### 2.3.1. The IASA component model

Like most software architecture approaches, the IASA distinguishes between primitive components and composite components. Developing an application according to IASA approach consists in producing a composite component. Thus, excluding connectors, all the architecture elements are components from an operator to the whole application. The component model defines an organization for the external view that should be respected by any component in addition to an organization for the internal view, applicable only to the composite components. The external view is represented by the *envelope concept* which encompasses the ports modeling the component, while the internal view is organized into two main parts: the *Operative part* and the *Control part*.

#### 2.3.1.1. The component internal view

The internal view of a composite component is made of two vital parts: the operative part and the control part. The operative part represents the business space of the composite and it contains the components instances dealing with the functional objectives of the composite. Those instances could be static or dynamic. A static instance is defined at the specification time and cannot be removed from the operative part, while a dynamic instance can be created or deleted during the composite-instance life-cycle.

The control part represents the aspect space and it is composed of a controller and a number of aspect components. It performs the various control operations on the components of the operative part. Examples of these control operations are: managing the control flow of the various services (shutdown, sequential or parallel launch), managing exceptions, exporting the component state, and logs generation.

#### 2.3.1.2. The component external view (*envelope*)

The external view of any component consists of an envelope provided by a ports set. The main goal of the envelope concept is to allow the total insulation of the component internal view from the external world. The envelope is also used to specify the deployment map, to enable the specification of connections involving the port's structural elements and to manage the injection and deletion of the advices provided by aspect components. The ports considered in an assembly operation are the envelope ports. The ports of the internal view are connected to the envelope ports by the delegation connectors.

#### 2.3.2. The IASA connector model

The IASA connector model is largely inspired from computer network architecture. The model defines two connector categories: transport connectors and service connectors. The transport connectors are responsible for the transport of data and control flows. The transport connectors link only two points that must be compatible and could be: a complete port, a part of a port represented by one or more access points or a part of the internal structure of a complex access point. A service connector is a predefined primitive component that supports the

interconnection of more than two points. It provides complex communication services such as the coordination, distribution and load balancing services.

In IASA two connection points are connected either in a point-to-point topology or through a communication infrastructure. A basic connectivity is established between two access points by a connector called an elementary connector, while a complex connection is performed by the introduction of a communication infrastructure. Any complex connector (interconnection infrastructure) is specified by the combination of transport connectors and service connectors. The IASA approach provides a simple method to build an interconnection infrastructure inspired from computer network architecture: any interconnection infrastructure, despite its complexity, is always built by cascading Service Connectors using Transport Connectors.

### 2.3.3. The IASA access points

The main purpose of the IASA access point concept is to provide a unified way to represent component's interaction points in the specification of a system architecture using software component and/or hardware components. The access point is the smallest processed element in architecture. It is the basic element for defining a port. It exposes the required or provided resources which may be operations or data. Unlike other architecture models where an access point correspond to an entire interface, the IASA access point represents the basic concepts exchanged between two component ports. An access point is instantiated within a port. It may be linked independently to another access point which is included in the same or in a different port.

The access point is designed to support two concepts: data transfer and flow transfer. Therefore, the IASA distinguish between two access point kinds: Data Oriented Access Point (DOAP) and Action Oriented Access Point (ACTOAP). The DOAP is used to specify an explicit data transfer. It is provided with an attribute specifying the data direction (in, out, and inout). It is either a primitive DOAP or a complex DOAP made of other DOAPs. An ACTOAP represents a service which may support many distinct actions. An ACTOAP plays one of two basic roles: a server or a client.

#### 2.3.4. The IASA ports

A port is a technique for grouping related access points in the context of a common goal. The ports model the external view of a component. They reveal the required and provided resources (services and data) of a component in addition to its behavioral aspects. The behavior is described using the SEAL action language (section 3.5). A port is an autonomous entity; it could be added or removed from a component. Moreover, a port can be modified by adding or deleting an access point.

The port maintains an abstract view and a concrete view. The abstract view is represented by: the concept of access point, the actions associated with the contained access points and the behavior. The concrete view may be any model provided with a clear way leading to the implementation level (e.g. an interface based port, a UML port, an ArchJava port). The port's behavior is represented by a set of valid rules using the concept of action of the SEAL language. Each rule shows how the required or provided resource must be used. The concrete view may be any model, provided with a clear way leading to the implementation level.

#### 2.3.5. The SEAL action language

In IASA, the behavior specification (i.e. interactions and operative part behavior) is done using the action language SEAL [77]. This language relies on the notion of *Action Context* which is inspired by the UML Precise Action Semantic. Among the basic SEAL elements: the various operators, the primitive data types, the control structures and all the basic types of IASA: components, ports, access points, roles and connectors. The SEAL description is defined clearly in the *behavior* clause and *actioncontext* clause of an architecture description. This description is structured in hierarchy of clauses as depicted in the Figure 2.3. SEAL description could be also found in the internal clauses of the *port* clause and in the *connectors'* clauses within the *operativepart* and *controlpart* clauses.

Actions are the basic behavioral entities that exchange control flows and data flows through in and out data points called *ActionPins*. An action is identified by a unique name which represents its signature. Three action types are distinguishable: primitive actions (indecomposable action), abstract actions (composed of actions belonging to another context instance) and composite

actions (defined from the actions of the same context instance). A context encompasses an actions set that is valid for a particular actions domain. An action must always be defined in an action context. An action context is a namespace such as an action's name is relative to its context. This notion has been introduced in order to limit the reasoning domain of an architect and provide him with sufficient tools to express his concepts in specific situations.

```

package packageName;
import packageList ; //components and connectors packages
component componentType {
// Definition of the internal types
port { // Defining internal ports types
    }
connector{ // Defining internal connectors types
    }
component{ // Defining internal components types
    }
/// Definition of instances
ports{ // defining the component ports
    }
Operativepart{ // Describing the operative part:
    //components instances, internal components types and connections
    }
controlpart{
    //Describing the control part: components instances and connections
    //between control part components with the operative part component
    }
behavior{
    //components behavior
    //the behavior is described using SEAL which is directly transformed
    // to the programming language or it is a reference to a Java file
    }
properties{
    //specifying non-functional properties
    }
} // end of the Component description

```

Figure 2. 3: Clauses in a component's textual description.

#### 2.4. Component based product lines

Considering that SPLs are inspired by industry where production activity is based on assembling certified components, and in light of the recent progress in CBD field, literature shows that integrating these two approaches will bring significant benefits to software development. In this section, we present the main propositions in this area then we discuss their out-comings.

### 2.4.1. State of the art

Component Based Product Lines (CBPL) engineering has been introduced to overcome the lack of maturity in SPLE by unifying the strengths of two complementary approaches: SPLs and CBD. CBD supplies technologies for reuse in the small, while SPL approach intends reuse in the large. Putting them together allows reaching large scale reuse and flexibility at the same time. However, only few works have been done in this area, in this section we present briefly the main proposition intended for CBPL engineering and variability modeling in CBPLs:

#### 2.4.1.1. CBPL for Workflow Management Systems

The authors in [81] present a CBPL for Workflow Management Systems (WfMS). They propose a CBPL process for the development of WfMSs and extensions for variability representation throughout the process. The proposed CBPL process consider: - domain analysis based on the generic architecture and reference models for WfMS - design of the product line architecture and its components based on Catalysis method [82] - evaluation of the architecture with Rapide [76] language and tools. The process phases are as follow:

- *Requirements analysis*: aims to identify the similar aspects and the VPs amongst the SPL members and represent at a high level the main components and interfaces of the WfMS. The authors use the use case variability of Jacobson et al. [83] that suggests the stereotype «*extend*» to represent VPs in use cases.
- *System specification*: this stage specifies the software solution by identifying the types and the related actions. Types are represented by a class diagram called the static type model. Variability is shown in the model basing on an extension for UML. This extension use the variability stereotype «*V*» to indicate the variations in the model. This stereotype is related with the concepts of specialization and aggregation.
- *Architectural design*: From the static type mode several refinements are made to reach the components level. The components are represented by the generic packages encompassing their types and relationships. We note at this stage that no means are provided for modeling variability at components model, it is rather explained in a textual way with the components descriptions.

#### 2.4.1.2. Adaptable Components for SPLE

This work presents techniques for the implementation of large transparently adaptable components via composition and parameterization [84]. The proposed techniques (mainly *skeleton object*) can be used as implementation framework within SPLs. The *skeleton object* is a technique for implementing adaptable components basing on the *higher order function* mechanism. A higher order function is a function which accepts function parameters and a skeleton object is an object whose constructor is a skeleton (i.e., a higher order function). Since individual skeleton objects provides components that are too small, the authors propose to compose skeleton objects in order to form larger components and thus reaching coarse-grained adaptability.

This work do not present a process for CBPL engineering neither variability techniques for modeling CBPLs, but rather some useful techniques for implementing variability within CBPLs.

#### 2.4.1.3. The Koala component model

*Koala* [85] [86] is a component model based on an architectural description language *Koalish* [87] used to build a large diversity of products from a repository of components. Its aim is to manage the growing complexity and diversity of software in consumer electronics products. Various concepts are used in the koala components models [85]: interfaces, connectors, subcomponents and compound components. Koala allows the modeling of variability related to interfaces (optional interfaces) and connections between interfaces (switch). Here is a brief presentation of the Koala model [87]:

- A *component* is an encapsulated piece of software that is self-contained and configuration independent to be a reusable asset. It can communicate (provide and require functionality) with its environment only through *interfaces*.
- An *interface* is a small set of semantically related functions. It serves as the unit of binding. The triangles denote the direction of function calls (Figure 2.4). Each interface has an instance name and a type (Figure 2.4). Interfaces types are managed separately from components. An interface is represented by a dashed square if its existence is optional in the component (Figure 2.4).



- Koala distinguishes between three connectors kinds as depicted in Figure 2.4: - *straight connection* which couples every function in the tip interface to the function with the same name in the base interface – *glue module* allows to insert code between two connected interfaces - *switch* is a pseudo dynamic binding of one interface to a set of other interfaces. The switch setting is controlled through a diversity interface.

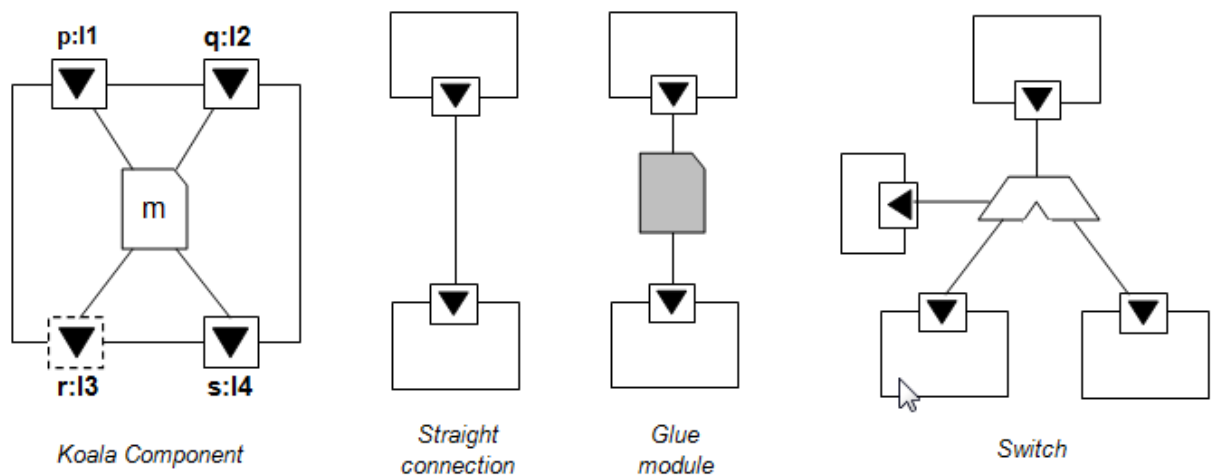


Figure 2. 4: Koala component model.

The koala approach promotes a composition rather than a decomposition process, such as components are combined in multiple ways into subsystems and subsystems in multiple ways into systems. Components and interfaces are stored in a repository that does not reflect the design hierarchy. Products are created by selecting and combining large compound components. When a large component does not satisfy the requirements, more basic components are considered.

#### 2.4.1.4. Variability Modeling Language for Architectural models

Loughran et al. [88] present a variability modeling language (VML), which supports variability representation in architectural models. The language provides mechanisms for: explicitly reference variation points in multiple architectural views and support the composition of architectural variants to variation points within classical architectural views, such as deployment, interaction, and component-connector models. The VML is comparable to the orthogonal variability modeling (OVM). OVM describes variability separately without extending architectural models with new notations. Thus, it considers variability as separate architecture

view (variability model). The VML extends the notion of OVM to capture the referencing and composition of architectural variability.

The main language elements are:

- Concerns are high-level abstractions encapsulating VPs which relate to a particular feature or any other architectural concern arising at the design stage.
- A VP has a name and variation kind (optional, alternative and parameter)
- A VP may offer a number of variants. VPs and variants should reference architectural elements within architectural views.
- A variant uses actions and expressions to invoke compositions of architectural elements.
- Actions provide the means to activate decisions which will result in architectural compositions between architectural variants and the common core elements. The key compositions mechanisms are connect, add, remove, deploy and merge.

This work entails straightly the configuration step by modeling the actions made by the system configurator. It ignores the details needed during the architecture variability modeling step. Several variability concepts are not considered in the variability modeling. The architecture variability modeling itself is not explained.

#### 2.4.1.5. Feature-oriented Solution with Aspects for CBPLs

This work seeks for deriving the SPL architecture from the FM [89]. The authors propose a solution which incorporates the aspect concept into the feature modeling by means of crosscutting features. This model is then used to map base-level and crosscutting features to base-level and crosscutting architectural elements. The approach allows thus to represent crosscutting concerns as crosscutting features and non-crosscutting concerns as base-level features by means of an aspect-feature view, to map crosscutting features to crosscutting architectural components and base-level features to base-level architectural components, and identifies and refines base-level and crosscutting component interfaces supported by use cases specification with aspects and variability. However, the main issue when modeling variability within CBPLs is not considered by this work. After mapping features to the architecture the resulted architecture

model contain base and crosscutting aspects of the system but no variability modeling mechanisms are introduced.

#### 2.4.1.6. The KobrA Approach

The basic goal of KobrA method [90] is to provide a systematic approach to the development of component-based application frameworks. It interests to the internal structure of components for which it uses UML models and decision models for modeling variability. The KobrA process consists in two main stages: framework engineering activity which aims to create and maintain a generic framework that embodies all product variants making up the SPL and application engineering activity which aims to instantiate this framework to create particular variants in the product family. A framework is the static representation of a components-set organized in a hierarchical form.

Framework engineering phase encompasses three activities: context realization, component specification, and component realization.

- Context realization activity determines the environment properties for the SPL and the framework's scope.
- Component specification activity describes by means of models the externally visible properties of a component. It comprises four models: the structural, the behavioral model and the functional models which constitute the specification models for a component, in addition to the decision model which contains about how the models change for the different applications.
- Component realization activity describes by means of models the private design of a component. Four models are included in this step: the interaction model, the structural model, the activity model and the decision model.

Application engineering uses the framework built during framework engineering to produce specific applications through two main activities: application context realization and framework instantiation. The framework is instantiated according to the contexts decisions taken by the customer. In addition to the decision models resolution, customer-specific requirements must be realized and therefore integrated into the framework.

#### 2.4.1.7. UML profile for SPL architecture

The authors present a UML profile for representing variation in the SPL architecture of middleware services [91]. The profile consists in a set of UML extension concepts. It relies on three UML extension mechanisms: constraints, tagged values and stereotypes. The UML profile includes several SPL architecture views decomposed into two categories: Conceptual views (including: conceptual structural view, conceptual behavior view, conceptual deployment view) and Concrete views (including: concrete structural view, concrete behavior view). In each view, extensions are introduced for modeling variability, for instance in the conceptual structural view all of: “mandatory”, “optional” and “alternative” stereotypes are used to describe the variability kind of subsystems.

#### 2.4.2. Discussion

In this section we will evaluate the CBPL approaches presented in the previous section. The evaluation addresses four aspects that we judge to be important when coupling CBD and SPL approaches:

1. *The CBPL development process*: as known SPLE is composed of two processes (Domain engineering and Application engineering), variability is introduced and modeled in the first process and it is resolved in the second one. The proposed CBPL approaches must define a development process that respects the SPLE methodology and benefits from flexibility and scalability offered by CBD at the same time.
2. *Variability modeling*: traditional CBD approach do not provide necessary concepts for explicitly modeling variability in systems, thus CBPL approaches should tackle this issue in a clear way.
3. *Variability implementation*: another aspect that should be considered when introducing a new CBPL approach is the definition of the underlying variability implementation techniques.
4. Finally, we consider also if the approach is *domain specific* or no so it could be of benefits for larger application environments.

The results obtained from CBPL approaches according to the before-mentioned aspects are summarized in the next table.

The CBPL development process have been defined by three of the discussed approaches: CBPL for WfMS [81], Koala approach [86] and Kobra approach [90]. Koala approach gives an overview on how systems are produced in their company and its primary focus is about components, however it does not explain a systematic method for developing the SPL core assets and deriving them after that. The CBPL for WfMS approach does not make a clear distinction between the two SPLE stages (domain and application engineering). This approach also focuses on the components development than SPLE. The Kobra process fits well the SPLE process. The two development stages are clearly described and the various views needed for system specification and variability modeling are defined and explained for each stage.

Table 2. 1: CBPL approaches evaluation.

CBPL approaches	Development process		Variability modeling		Variability implementation		Domain specific	
	Yes	No	Basic	Advanced	Yes	No	Yes	No
1. CBPL for WfMS	✓		✓			✓	✓	
2. Adaptable components		✓			✓			✓
3. Koala approach	✓			✓	✓		✓	
4. VML for SPL architecture		✓	✓			✓		✓
5. Feature aspects		✓	✓			✓		✓
6. Kobra approach	✓		✓			✓		✓
7. UML profile		✓	✓			✓	✓	

The variability modeling have been most of the time considered in a basic way by defining only few variation types (mainly: mandatory, optional and alternative),

moreover those variations were identified only for some architecture elements (mainly: components and connectors). The only work that gives more emphasis on variability modeling in components is the Koala approach. Koala provides some advanced variation mechanisms such as optional interfaces and switches. Nevertheless, several other variation kinds are not covered by the koala component model.

Variability implementation techniques assist the developer when realizing the VPs. Among the discussed approaches only the koala approach completes the CBPL process until getting the implementation step and proposes an ADL (Koalish) for that. The adaptable components approach [84] proposes a variability realization technique that could be of great benefit when implementing SPLs, yet the authors does not mention for which variation kinds this techniques can be chosen. Overall, the paper does not consider the CBPL process or the variability within architecture that are crucial concepts in our study.

Having an approach that deal with the various CBPL engineering activities is important for software engineering, however, if this approach is targeted for a specific field this will limit its use elsewhere. This is the case for the Koala approach. Koala has been proposed to deal with the issues encountered by Consumer Electronics Company. So its developers have enriched it by the necessary concepts for its specific application domain. Consequently, even if it has been enhanced by some advanced variability concepts, several other concepts related to SPLE and CBD are still needed when exploiting this approach in other software fields. We mention for instance: the lack of a clear CBPL development process, lack of support for separation of concerns, lack of several variation types (mandatory, alternative, grouped features...).

Another major issue in CBPL engineering is the products derivation. This step is most of the time ignored or tackled implicitly even if it is of great importance. When the derivation rules and steps are well defined, the exploitation of the core assets base gets its higher levels and the application engineering step is done in shorter time and with lower effort and cost. Furthermore, the implicit modeling of variability makes its derivation a hard task to perform. If variations are not clearly identified in the domain models, there derivation will encounter several challenges and won't be able for automation.

Finally, the broad use of SPLs results in what is called Multiple Product Lines (MPLs). Engineers must then provide solutions to manage variability not only within a single SPL, but also between SPLs of an MPL. This issue has been addressed by only one of the discussed approaches which is the Koala approach. Koala develops a components technology that supports the realization of freely combinable components such as a compound component is the responsible of any interaction between its sub-components. Though, Koala does not give attention to the necessity of systematizing inter-SPLs reuse.

To summarize, the main characteristics of the existing CBPL approaches are:

- lack of development methodology;
- limited variability modeling;
- implicit variability modeling;
- no provided guidance for derivation step;
- no support of inter-SPLs reuse;
- lack of advanced CBD techniques;
- slight consideration of variability implementation techniques;
- and specialization for particular application domains.

## 2.5. Conclusion and recommendations

In this chapter, we have defined in large the main concepts related to the Software Architecture discipline. We have focused on the IASA approach since it will be subject of extension when defining our approach. And we have studied the integration of CBD and SPLE through a set of CBPL approaches. The study we conducted revealed that the primary aspects that should be covered when defining a new CBPL approach are:

1. *The CBPL process* that must base on the SPLE methodology on the one side, and benefits from the CBD technologies on the other side. Both of SPLE and CBD provide powerful techniques for supporting reuse but at opposite granularity spectrum. CBD deals with reuse in the small while SPLE manage reuse in the large. Therefore, significant benefits are expected from their integration. The SPLE process lies on the distinction between two development stages: development for reuse (domain engineering) and development with reuse (SPL derivation). A set of reusable core assets is constructed during the

first stage and intended for reuse in the SPL scope. SPLE provides techniques for making the adaptations easier and the reuse systematic by managing the domain variability. CBD represents techniques for implementing variability and makes the automation of the derivation step possible by producing flexible components.

2. The proposed approach should support *variability management*. Variability must be identified for the various abstraction levels and modeled explicitly in the different modeling views. In addition, mechanisms must be defined for modeling the different variations kinds and this for each architecture element. Thus, variability must be modeled and implemented for the following levels: system architecture, composite components' internal structure, primitive components implementation, interfaces and connectors, and using the following variation types: mandatory, optional, alternative, AND, OR and XOR VPs and components groups with cardinality.
3. The proposed methodology should support the reuse systematization not only within a single SPL but also among separated SPLs included in the same field.



# CHAPTER 3

## COMPONENT BASED SPLS (CBPL)

### 3.1. Introduction

*SPLs* are emerging as a viable and important development paradigm allowing companies to realize major improvements in time to market, cost, productivity, quality, and other business drivers. Product Line approach intends to adapt to the software development, the principles that we find in the automotive, aeronautics and electronics industries. Production in these industries is organized into ranges with similar parts and offering a number of options. For example, in automotive industry, various car models come out the same assembling chains using the same chassis, the same engines and the same test plans. The aim is to improve productivity, decrease time to market, and reduce production, maintenance and test costs. The idea is to transpose the principles of manufacturing of these industries in software development, to benefit from the before-mentioned advantages.

Although, the principles of *SPLs* are inspired by industry where production activity is based on assembling certified components, most of software product lines today are based on traditional design concepts of software engineering such as modular design and object-oriented design. These design concepts are not able to support the assembly of certified components which makes the derivation and maintenance processes difficult to perform. On the other side, both of *SPLs* and *CBD* promote reuse; putting them together will bring significant benefits to software development.

*SPL* improves reuse while maintain diversity between products. In most cases, software components intended for reuse must be adapted to meet specific

requirements of customers. If flexibility is not considered since the early development process phases, developers will meet lot of difficulties in adaptation step, which lead them to leave these components even they will have to redevelop components from scratch. SPL simplify the adaptation phase by involving “Variability management” activity along all the software development process.

However, traditional component-based approaches do not support variability management. It becomes necessary to establish new approaches or enhance existing component-based approaches in order to allow an effective variability management. In this chapter, we propose an approach that integrates SPL engineering and CBD engineering, aiming to unify the power of these two approaches. Firstly, we present a new FM notation that is better relevant to CBD. Then, we present the development process of component-based SPL (CBPL). After that, we show the extension of a component-based approach in order to allow the modeling of variability in CBPL. Finally, the proposed approach is supported by a case study.

### 3.2. Composition oriented FM

One of the most common concepts of SPLE is the FM. Feature modeling was first introduced in the FODA method by Kang in 1990 [19]. Afterward, it has been widely adopted by SPL community and a number of extensions have been proposed (Chap 1- section 4). These extensions usually add increasingly new notations to the FM in order to represent the various variability types that could be included in a SPL. For instance, the extension proposed by Czarnecki [27] introduces a lot of new symbols for representing variations within FMs. Cluttering the FM by new notations makes it complex and hard to be manipulated while it is supposed to be an easy-to-understand specification model (Chap 1- section 2). On the other hand, the relationships between features are most of the time ambiguous, since the type of the links is not specified (father-son relationship, is property of, specialization/generalization, composition...). This makes the whole model ambiguous and hampers the mapping between FMs and other SPL models.

In this section, we propose a new notation for FMs that is designed to be simpler and component oriented [97]. The aim is, to simplify the FM by using more expressive notations and thus decrease the number of symbols used for modeling

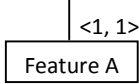
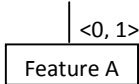

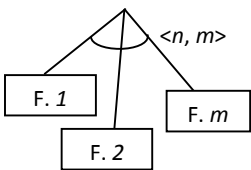
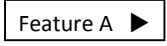

variability. In addition, our proposition will rely on the composition and specialization/generalization relationships between features of an FM in order to remove the ambiguity of relations and to simplify the transition between FM and architecture. The resulted model is a *composition oriented FM* that serves as a reference when constructing the architecture model and helps in specifying variability in the next SPLE process steps.

The links that relate the features in our FM are of two main kinds: *specialization/generalization* and *composition*. The composition relationship is represented by a simple line while specialization/generalization relationship is represented by a dashed line as depicted in the Table 1. We distinguish between four variability types: *mandatory feature*, *optional feature*, *single feature with cardinality* and *choice feature-set*. All of those types are expressed using cardinality interval  $\langle n, m \rangle$  where  $n, m \in \mathbb{N}$  and  $m \geq n$ . Cardinality means the occurrences number for a single feature and choices number that could be selected for a feature group (a sub-features set related to the same feature and that represent a choice) such as:

- Mandatory feature is a feature with cardinality:  $\langle 1, 1 \rangle$ . It can be a single feature or a grouped feature included in a choice variability.
- Optional feature is a feature with cardinality:  $\langle 0, 1 \rangle$ . It can also be a single feature or a grouped feature included in a choice variability.
- A feature that can be included in the system several times is called: solitary feature with cardinality. The number of its occurrences is denoted by a cardinality interval  $\langle n, m \rangle$  such as:  $n \geq 0$  and  $m > 1$ . In the case that  $n = 0$  then the feature is optional and if  $n > 0$  then the feature is mandatory.
- Choice relationship is used to represent a feature that is related to a variable set of sub-features restricted by cardinality. It includes all of AND, OR, XOR variability types as shown in the Table 1. For us cardinality is sufficient to represent all kinds of choices.

In addition to the variability notation we add two symbols for representing reference features and features with attributes. We use similar symbols as in [27]. Reference features allow us to split large feature models into smaller modules and refer to a sub-tree by a single feature. A feature can have an attribute type, indicating that an attribute value can be specified during configuration.

Table 3.1: Notations used in Composition oriented FM.

Notation	Meaning
—————	Composition relationship
-----	Specialization/Generalization relationship
	Feature with cardinality $\langle 1, 1 \rangle$ : Mandatory feature
	Feature with cardinality $\langle 0, 1 \rangle$ : Optional feature
	Solitary feature with cardinality $\langle n, m \rangle$
	Choice with cardinality $\langle n, m \rangle$ , such as: <ul style="list-style-type: none"> <li>- If the relation type is AND then <math>n = m</math></li> <li>- If the relation type is OR then <math>m \geq n</math></li> <li>- If the relation type is XOR (i.e. alternative) then <math>n = m = 1</math></li> </ul>
	The feature A is a reference to a feature model
	Feature A with attribute of type $T$ and value $v$

### 3.3. Component Based Product Line Engineering

SPLE relies on a fundamental distinction of development for reuse and development with reuse (Chap 1 section 3). Development for reuse or “*Domain engineering*” consists in developing core assets through the domain analysis, domain design and domain implementation processes. Development with reuse or Application engineering consists in developing the final products, using the core assets and the specific requirements expressed by the customers. Figure 1 shows the development process of CBPL that we propose [93]. The presented process is an integration of the development process of SPLs (Chap 1 section 3) and the CBD process (Chap 2 section 2).

Unlike the traditional SPL engineering, the base of core assets obtained from CBPL engineering relies mainly on software architecture (reference architecture,

refinement of components and reusable components...). While, its difference compared to the traditional CBD is the introduction of variability management activity. Since variability is handled from the early development stages, reusable components obtained from domain engineering process do not have to be adapted to the specific needs of final applications, which make the application assembly step easier and faster.

The CBPL domain engineering consists of three activities that are *Domain analysis*, *Product-line architecture design* and *Components implementation* (Figure 1). The purpose of this sub-process is to produce the reusable core assets and to provide the effective means that help in using these core assets to build new products within the SPL. The main outputs of this process are: reference requirements, reference architecture and reusable components.

### 3.3.1. CBPL domain engineering

*Domain analysis*: instead of exploring the application domain in order to find the possibly reusable components, we use the techniques provided by SPLE (variability management) to systematize the reuse of components. Thus we start by delimiting the CBPL scope, and then we identify the domain requirements and the predictable variations. The requirements are documented mainly using the composition oriented FM.

*Product-line architecture design*: in this step the common CBPL architecture is designed basing on the constructed FM in the previous step. The architecture documentation bases on the IASA extension component model that we present in the next section. The Product-line architecture serves as reference architecture for the various CBPL members. It represents an important core asset that will help in deriving final applications.

*Components implementation*: in this step components (and all reusable elements) are refined, implemented, and tested for reuse in the different CBPL applications. The resulted core assets are collected in a repository and are made available for reuse at application derivation step.

### 3.3.2. CBPL application engineering

Application engineering consists in developing the final products, using the core assets and the specific requirements expressed by customers. This sub-process

consists of three steps: *Application analysis*, *Application architecture design* and *Application assembling*. Each step is facilitated by the reuse of the outputs from the previous process. The result of application engineering is an application ready to be used.

*Application analysis*: in this first step, the variability is selected according to the application requirements i.e. the FM is derived to get a specific configuration that fit the need of the application under development. The requirement models are extended by the specific applications functionalities if those latter have not been considered during the domain engineering.

*Architecture design*: this activity is responsible for the derivation of the reference architecture according to the FM configuration obtained from the previous step to get the application architecture. The selected components for the application architecture are also derived according to the application requirements.

*Application assembling*: this activity consists in assembling the resulted components from the previous step according to the application architecture and testing the resulted product.

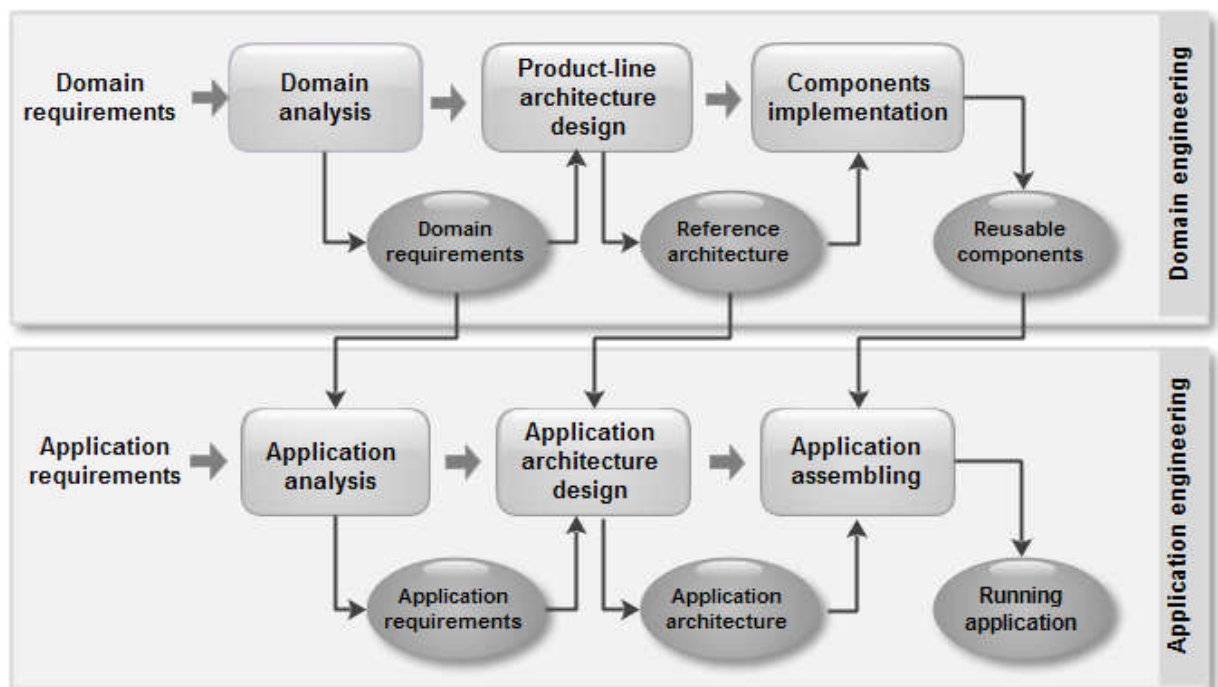


Figure 3. 1: Component-Based Product Line engineering [93].

### 3.4. Variability Modeling In CBPLs

CBPL approach is emerging as a promising and viable approach in software engineering. However, traditional component-based approaches do not support variability management. It becomes necessary to establish new approaches or enhance existing component-based approaches in order to allow an effective variability management. In this section, we propose a new approach that allows the variability specification within CBPLs [93]. We extend the architecture specification approach IASA (chapter 2- section 3) by the necessary concepts for variability modeling. In contrast to the existing work (chapter 2- section 4), our approach presents an explicit specification of the various variability types related to the different architecture concepts.

We choose the IASA approach because it allows us a clear description of components, connectors and interfaces and thus, variability could be clearly presented in each of them. Furthermore, IASA supports the aspect concept which is a crucial concept in our approach. The aspect concept provides the necessary techniques for merging variable concerns with the business ones.

#### 3.4.1. Architecture design with IASA

The design according to IASA approach uses a component-oriented process which proceeds by successive refinement. An IASA component is seen from the outside as a black-box that communicates with the external world through Ports, which define the services it can provide or require. The internal view of a primitive component is inaccessible, while the structure of a composite component is well defined, it consists of three parts: Operative Part, Aspect Part, and Control Part (Chap 2 section 3). The Figure 2 sets out the basic IASA notations.

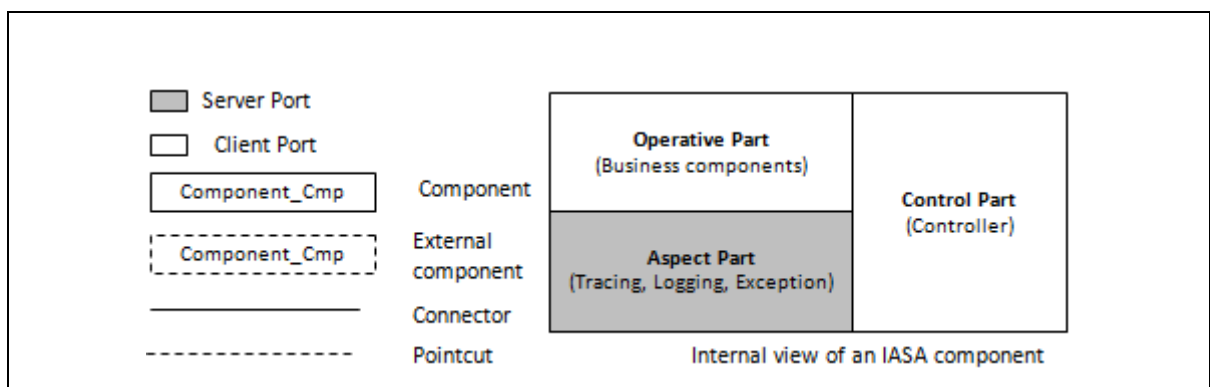


Figure 3. 2: The basic IASA notations.

### 3.4.2. Variability modeling extension for IASA

Since IASA have not been developed to capture explicitly all variability types, we propose to extend it in order to meet the SPL needs. We define three types of variability in architecture: Mandatory, Optional and choice.

#### 3.4.2.1. Mandatory and Optional elements

Mandatory architecture elements represent those elements that are common to all the SPL members. While, optional architecture elements are those elements which can be selected or not to be part of an SPL member. Each element in the architecture can be mandatory or optional according to the context in which it will be used. In order to represent explicitly these variation types, Components and Connectors are annotated by: «**Mdr**» and «**Opt**» which means respectively: *Mandatory* and *Optional*.

Considering that interfaces (ports) in the same component can be optional if they are not needed in some contexts, variability in interfaces is represented as follow:

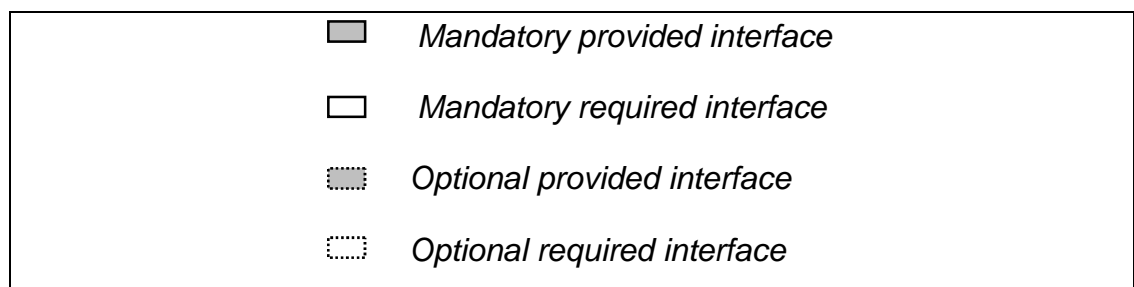


Figure 3. 3: Interfaces variability notation.

#### 3.4.2.2. Choice variations

The choice variability type represents the situations in which several choices are available at the same time. This occurs when we have several implementations for the same component or when a variable components set can be related to another component by the same connector. Thus, we distinguish between two choices kinds:



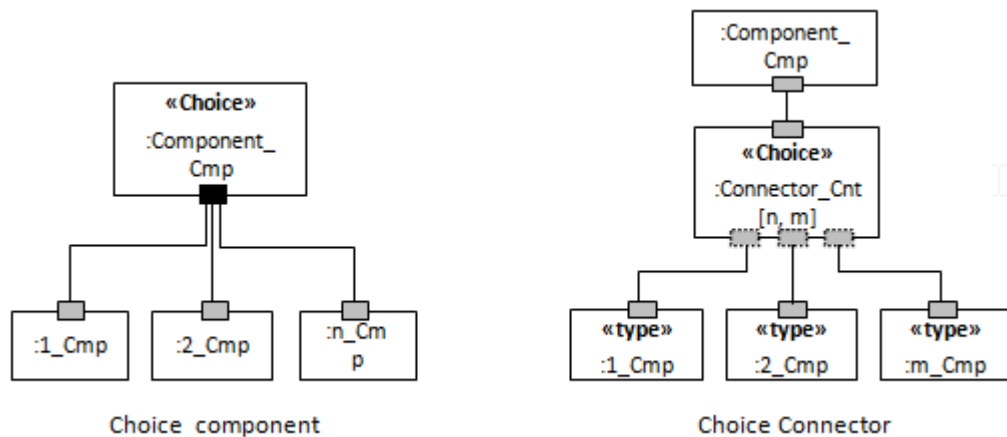


Figure 3. 4: Choice variability notations.

a. Component choice:

When there are a variety of implementations for the same component, the variable component is annotated by: **«Choice»** and related through the same interface to all its variants as shown in Figure 3. The port of the choice component that is related to the implementations choices is called choice port. It is characterized by a natural variable choicesNbr that states the implementations number related to the port. Each component implementation choice is connected to an access point from the choice port. The connectors that relate the choice component to its implementations could be optional or mandatory.

b. Choice connector:

When a component is related to a variable set of components, the relation between these components becomes a connector annotated by: **«Choice»**. The choice connector relates from one side a component and on the other side a variable components-set such as those components have a similar relationship with the former one. The ports and connectors that connect the choice connector to the different components can be mandatory or optional. The graphical specification of this component type is depicted by the Figure 3.

The number of components that can be supported by the connector is expressed by a cardinality interval  $[n, m]$ , such as:

- $n=m$  if the type of the relation is AND;
- $m \geq n$  if the type of the relation is OR;
- $n=m=1$  if the type of the relation is XOR or Alternative.

The choice connector is in fact a connection component therefore it is created in the same way as an ordinary component. Nevertheless, the choice connector is characterized by two natural variables  $n$  and  $m$  representing its cardinality interval. Such as  $n$  is the minimum components number that could be related at the same time to the choice connector, while  $m$  is the maximum components number that could be related at the same time to the choice connector.

### 3.4.3. Mapping features to the architecture

The feature modeling represents a simple way for capturing commonality and variability in the scope of a SPL. Once constructed, an FM can help widely in the variability specification for the SPL architecture. The FM we propose is component oriented in such a way that ease the passage between the two modeling steps domain analysis and product-line architecture design. Table 2 matches between the FM notations and their corresponding notations in the architecture.

Table 3. 1: Mapping features to the architecture

<b>Features relationships</b>	<b>Corresponding architecture</b>
Composition relationship	The father feature is a composite component of its sub-features.
Solitary feature with cardinality	Several implementations of the same component
Mandatory feature	Mandatory component
Optional feature	Optional component
Choice with cardinality	If it corresponds to a specialization, this results in a choice component
	If it corresponds to a composition, this results in a choice connector

The FM gives an abstract view of variability in a SPL; a more concrete variability representation is needed for the products construction. For this, representing variability in software architecture is the solution for modeling variability at the different abstraction levels (configuration, components, connectors, ports...) and to narrow much more the modeling to the code.

### 3.5. Case study: e-meeting CBPL

In this section we apply the CBPL approach explained in this chapter in a specific domain which is: E-Meeting applications. E-meeting applications exhibit an excellent solution to overcome the problems that can occur when organizing a face to face meeting namely: distance, costs and availability. Using online meeting technologies allows us to simplify the meeting organization processes, save time and displacement expenses. The meeting concept can be found in various fields. For instance: meetings for year-end deliberation, meetings of a Scientific Council, elected members meetings of an APC<sup>1</sup>, APW<sup>2</sup> or APN<sup>3</sup>, meetings of business leader, etc. E-meeting applications designed for these various meetings types share many similarities and are distinguished by some particular aspects. It becomes very important to take advantage of similarities between these applications to develop a family of e-meeting applications. This product family (SPL) once implemented is able to produce for each meeting type, the appropriate software in a very short time. In this case study we will focus on the component-based specification of the e-Meeting SPL. Next sub-sections describe the artefacts obtained from Domain analysis and Product-line architecture design steps.

#### 3.5.1. Domain analysis

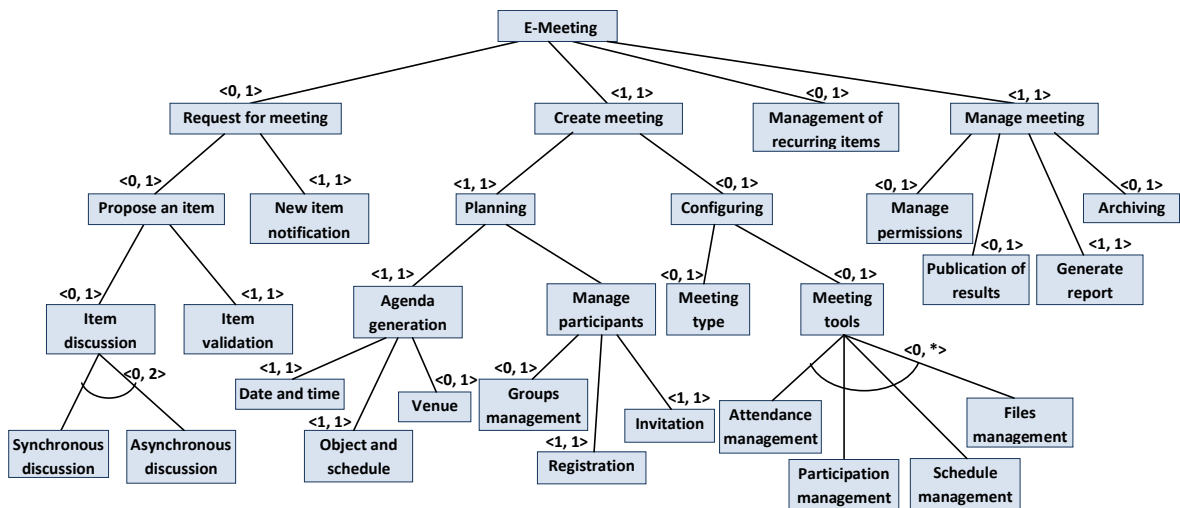


Figure 3. 5: Business feature diagram for e-Meeting product line.

<sup>1</sup> An APC is the elected assembly which governs a commune (baladiyah).

<sup>2</sup> An APW is the elected assembly which governs a wilaya (province).

<sup>3</sup> An APN is the national elected assembly.

The goal of domain analysis is to extract and document the similarities and variations between the SPL members. To document the common and variable features of our product line, we have used the FM. Figures 4 and 5 show a part of the feature model of our e-Meeting CBPL. The notation used is the one presented in section 2.

The constructed feature model is divided into two diagrams according to the features type they include: *business features and technical features*. Features in the first diagram called Business features (Figure 4), represent the business functionalities provided by the system. This diagram shows that 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 for meeting” in the FM.

“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 consists in preparing the application by fixing the meeting type (video, audio, chat...), 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.

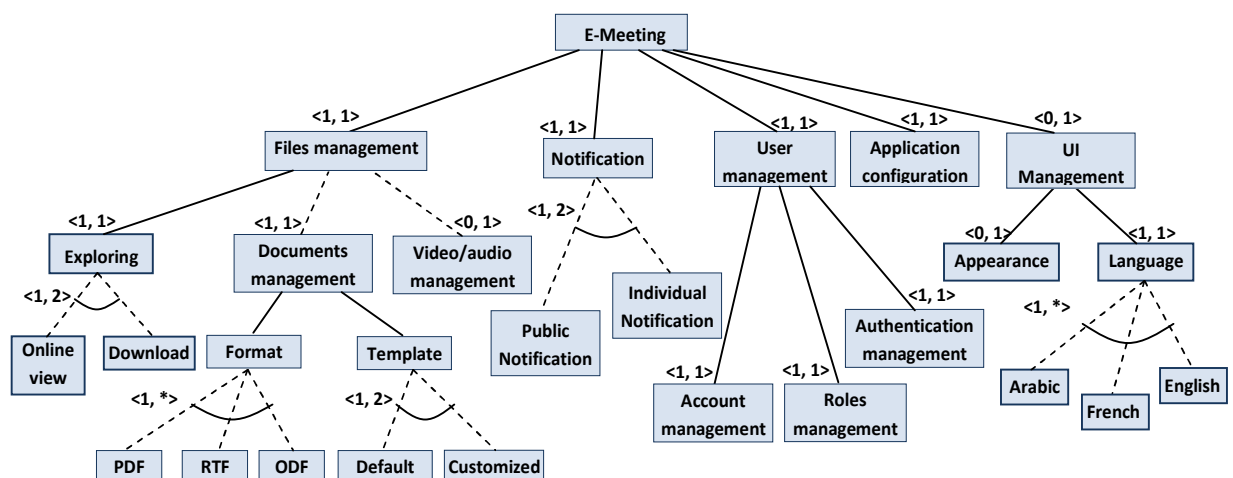


Figure 3.6: Technical feature diagram for e-Meeting product line.

The second diagram Figure 5 represents the technical features; it means features that do not reflect the business aspect of e-Meeting applications. Technical features encompass features related to the application (configuration, UI management), features related to users (Accounts management, Roles management, Authentication), and features related to files (including: text, video and audio files). Those features could be found in any e-Meeting application, but not all of them are mandatory.

### 3.5.2. Product-line architecture design

The purpose of Product-line architecture design is to establish the generic software architecture of the product line. Variability identified during domain analysis must be explicitly specified in the product-line architecture. To design the reference architecture of our composite SPL we have used the notation presented in section 4. The reference architecture of the e-Meeting SPL is reported in figure 6. The mandatory components that must be included in each member of the SPL are annotated by «**Mdr**» (Meeting\_Management, Meeting\_Planning, Roles\_Management...), while those which are optional are annotated by «**Opt**» (Items\_Management, Meeting\_Configuration...).

The mapping between FM and architecture is clear when comparing the models. Mandatory features such as: manage meeting, planning and account management (in Figures 4 and 5) correspond to mandatory components in the reference architecture (Figure 6), respectively: Meeting\_management\_Cmp, Manage\_planning\_Cmp and accounts\_management\_Cmp. Also, optional features like: Request for meeting, Configuring and UI management (in Figures 4 and 5) correspond to the following optional components in the reference architecture: Meeting\_Request\_Cmp, Meeting\_Configuration\_Cmp and UI\_management\_Cmp respectively.

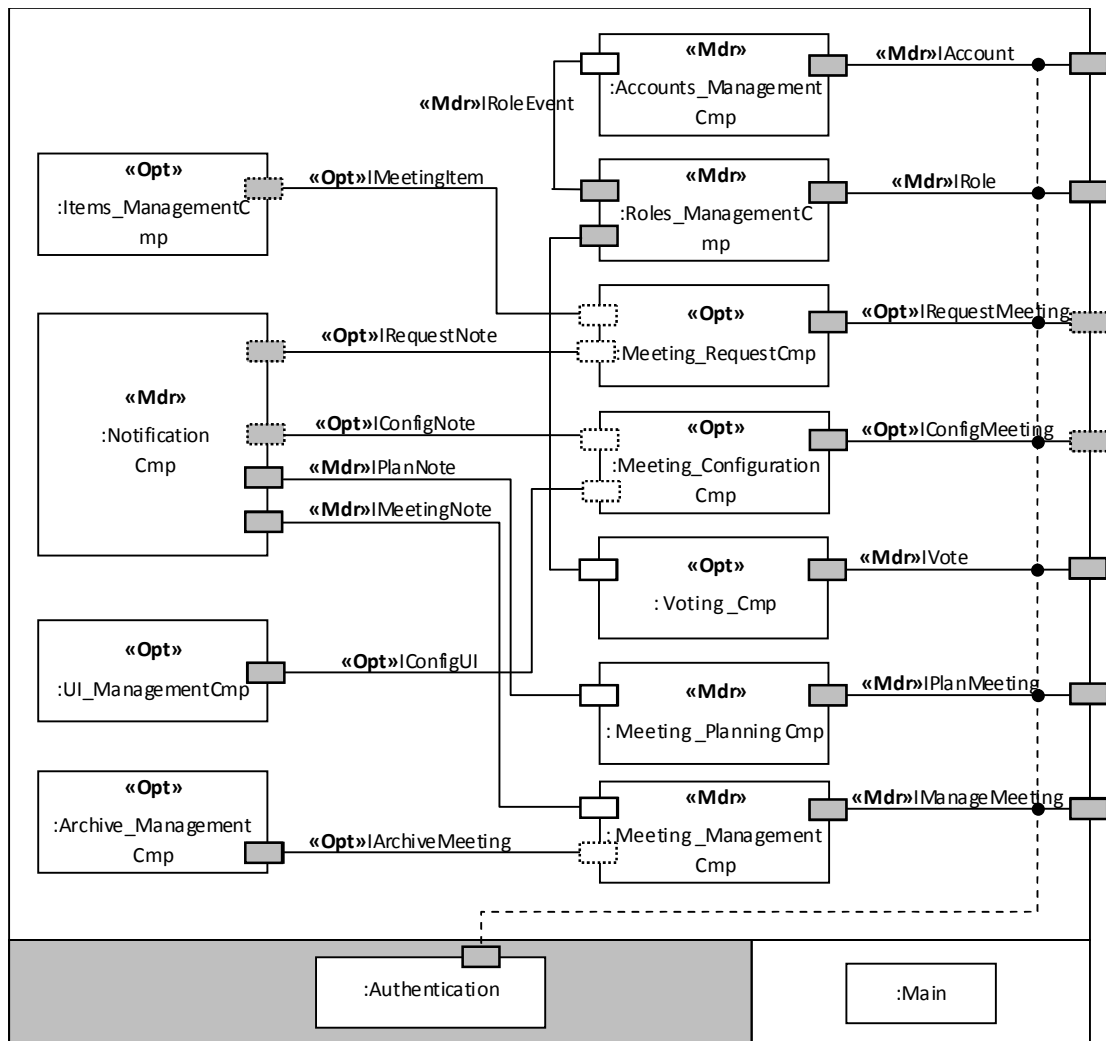


Figure 3. 7: Reference architecture of e-Meeting product line.

The component that corresponds to the “Meeting tools” feature does not appear in the reference architecture (Figure 6) because it is part of the refinement of the first level feature “Meeting configuration”. Figure 7 shows the refinement of the “Meeting\_Configuration\_Cmp”. As presented in Table 1, in the case of Choice feature, if the relation between this feature and the sub-set of feature is composition then it will become a choice connector in the architecture. Thus “Meeting configuration” maps to the choice connector “Meeting\_tools\_Cnt” (Figure 7), and each of its sub-features: “Attendance management”, “Participation management”, “Schedule manage” and “Files management” (Figure 4) correspond respectively to: Attendance\_Cmp, Participation\_Cmp, Schedule\_Cmp and FilesManage\_Cmp (Figure 7). The choice connector allows us to represent this variability type.

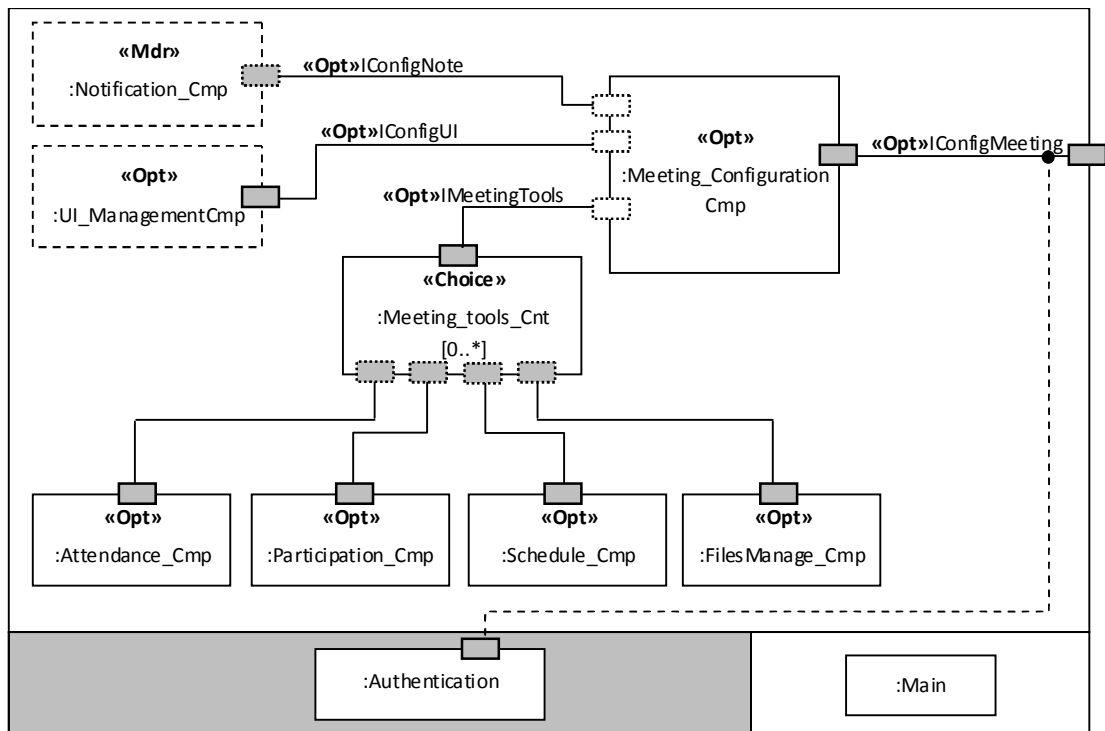


Figure 3. 8: Variability model for “Meeting\_Configuration” component.

### 3.6. Conclusion

The approach presented in this chapter aims to reach a high level of reuse that can be obtained through the integration of two approaches: SPLs and CBD. Each of these approaches promotes reuse at different granularity levels. CBD supplies technologies for reuse in the small, while SPL approach intends reuse in the large. Putting them together allows us to reach large scale reuse and flexibility at the same time. Moreover, CBD can overcome the lack of maturity in SPL engineering by providing efficient development technologies. The chapter presents also a case study for an ambitious field (e-Meeting) aiming to validate the proposed approach. The designed e-Meeting reference architecture represents a framework for all foreseeable e-Meeting applications and can be extended to cover specific requirements if needed. Since, software can be quickly derived from a CBPL by the simple composition of existing components; the activity of deriving new members in a CBPL can be greatly automated.

# CHAPTER 4

## ASPECT MULTI PRODUCT LINES (AMPL)

### 4.1. Introduction

SPL approach has been successfully applied in various fields containing a manageable variability set. However, in some large fields (like e-Government), single SPLs are no longer sufficient to manage variability due to their complexity, broadness and creeping scope. Those fields are, in fact, composed of several interdependent subfields each of them has a rather different goal. Consequently, separated SPLs have been built for each of them, resulting in a set of interdependent SPLs commonly known as MPLs (Section 5- Chapter 1). Yet, the emergence of MPLs has given rise to new challenges for SPLE.

SPLs of an MPL even developed separately, still present some commonalities since they belong to the same field. Those commonalities raise the need for reuse across an MPL. However, the reuse mechanism the most adopted within MPLs is the direct reuse of the derived components from some SPLs and their integration with the reusing SPL in order to get a final application. This reuse way is opportunistic and mostly limited since it was not planned in advance. In most cases, this reuse way requires substantial adaptations which make the developer's work laborious and error prone. So one of the crucial MPLs development challenges is systematizing reuse between the various SPLs of an MPL.

Another important challenge is the support for structuring product line models (Section 5- Chapter 1). It is infeasible to represent a MPL using a single model due



to its size and complexity. Thus, models representing the multiple SPLs within an MPL must be structured and organized in order to facilitate the whole MPL management. This capability helps coping with variability management in large and complex systems and manages the MPL scope.

Furthermore, product construction within an MPL requires the derivation of several distributed SPLs. Some SPLs may require, restrict or even exclude the inclusion of components from other SPLs. Thus, the dependencies between distributed SPLs must be considered. This process known as *distributed derivation* represents another challenge faced by MPLs.

In this chapter we propose a new approach, called AMPL (Aspect MPL), which helps dealing with the issues mentioned before. Our approach is based on two main concepts: *the separation of concerns* and the *partial derivation*. Firstly, AMPL decomposes a large scale domain into a set of subfields, such as for each subfield an SPL will be constructed. These subfields are analyzed in order to find out the common functionalities between them in addition to the specific requirements of each of them. Then, we introduce *Aspects SPLs* that are specialized SPLs in producing the defined common components. After that, the Aspects SPLs are partially derived to ease their integration with the AMPL SPLs early in the development process. The performance of this last activity allows us to avoid the distributed derivation challenges later. Finally, the resulted SPLs can be derived to produce final applications for each subfield included in the AMPL.

The chapter is structured as follows: The next section presents the background of our proposal. Section 3 discusses the foundations of our approach. Section 4 presents the AMPL engineering process. Section 5 describes a case study to validate our proposal. Section 6 discusses the obtained results. Finally, section 7 comments on related work and compares it to ours while section 8 summaries the chapter and outlines our future plans.

#### 4.2. Crosscutting Reuse within MPLs: e-Government example

Our approach has been actually inspired by our experience in developing SPLs for e-Gov field in context of the project “Towards an SPL for e-Government applications”. The project aims to set up the technological and methodological bases to the development of an e-Gov MPL. The objective of this MPL is the fast

production of software intended to the different Algerian government institutions (e-Administration, e-Justice, e-Voting, e-Meeting, e-Health, e-Education, etc.). The produced software should be compatible to ensure high level interoperability between the various government institutions. However, building a single SPL for the whole domain is infeasible due to its broadness and complexity. Therefore, a set of separated SPLs has been built and each of intends a particular e-Gov subfield, this results in an e-Government MPL. Nevertheless, those SPLs must preserve interoperability and reuse information must be kept between them in order to get faster development processes and lower costs and development effort. In this chapter we consider a sub-set from e-Gov MPL (e-Administration and e-Education) in order to illustrate the proposed approach, the complete case study is presented in Chapter 6.

Firstly, we present the specification of two e-Gov SPLs that helps showing the reuse orientation within MPLs and motivate our work. The case study will be used next to validate the presented approach.

#### 4.2.1. E-Admin SPL

The first case study addresses a very important e-Gov sub-field, which consists of online services offered to citizens by the different governmental institutions<sup>1</sup> (known as *online administration* or e-Admin): APC, Wilaya<sup>2</sup>, Daïra<sup>3</sup>, Justice, and Ministry. Considering that we can create a SPL for each kind of institutions, in this case study, we focus on services provided by APC institutions. Since these SPLs share the major part of functionalities, the core assets realized during e-APC SPL's domain engineering could be reused in the other SPLs (Wilaya, Daïra, Justice and Ministry). APC is the local *Algerian* government institution the nearest to a municipality inhabitants. Its services are involved in the daily life of citizens. Therefore, requests on these services are intensive and must be fulfilled effectively. In this section we motivate the introduction of SPL approach in this e-Government sub-field, and then we present the specification of e-APC SPL.

The e-APC system aims to enable any citizen to access through the internet to various services of an APC. Most frequently required services by the citizen from

---

<sup>1</sup> We consider her particularly services offered by Algerian e-Government to illustrate our approach basing on practical experience.

<sup>2</sup> Algeria is divided into 48 wilaya (province) headed by walis (governors). Each wilaya is further divided into Daïras, themselves divided in communes (baladiyahs).

<sup>3</sup> A Daïra is tasked to deliver passports, driving licenses and national identity cards for citizens residing in its territory.

the APC are the production of official documents like birth and wedding certificates. Inside the APC, the service delivering such official documents is called the “Civil State Service”. E-APC applications are more and more requested owing to the benefits they provide to government and citizens as well. It becomes necessary to find a solution which allows answering quickly to the large number of requests on these applications all by reducing time and cost of development and maintenance. The solution that we have adopted is the SPL approach. This latter allows us to manage variability and gain from the large scale reuse as well. Final applications are quickly developed by the simple assembling of reusable components. Quality is improved through reuse, since components are tested in many contexts, which prove their efficiency in several products kinds.

For this case, the feature model we constructed is divided into three diagrams according to the type of features it includes: *business features*, *technical features*, and *implementation features*. Features in the first diagram called business features (Figure 4.1), encompass the business services provided by e-APC application. An e-APC application may allow citizens to extract one or more kinds of vital records. Those latter might be transcribed on registers (certificate of birth, wedding, etc.) or not transcribed (family record, residence, etc). “Event declaration” functionality allows citizens to declare birth, wedding, divorce and death events online. “Documents modification” includes: “Manage mentions” such as “legal mentions” (mention of birth, wedding ...) and “modification of personal information”. An e-APC may allow also the electronic validation of provided data in addition to the download of official documents.

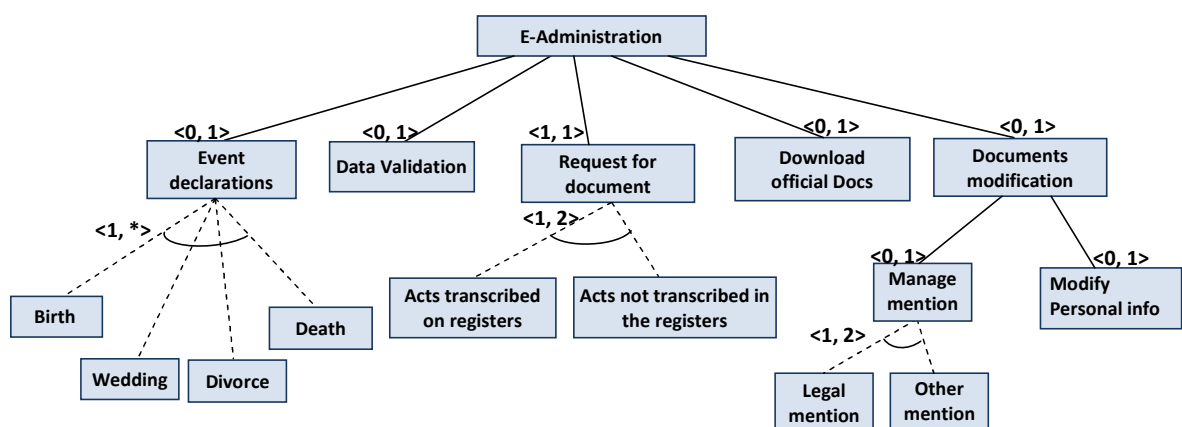


Figure 4. 1: Business feature diagram for e-Administration product line

The second diagram shown in Figure 4.2 represents the technical features; it means features that do not reflect the business aspect of e-APC applications. Technical features encompass: application configuration, users management, documents management functionalities. In addition, e-APC applications may need to communicate with other e-Government applications or between them in the context of business processes, this functionality is represented in the diagram by the feature “Communication” that could be horizontal (systems integrated across different functions) or vertical (local systems linked to higher level systems within similar functionalities)[92]. An e-APC application (and generally any e-Government application) may need additional functionalities to complete their tasks, such as: online meetings, poll and statistics. This is materialized in the feature diagram by a set of features with cardinality  $\langle 0, * \rangle$  that expose the ability for adding other choices if needed.

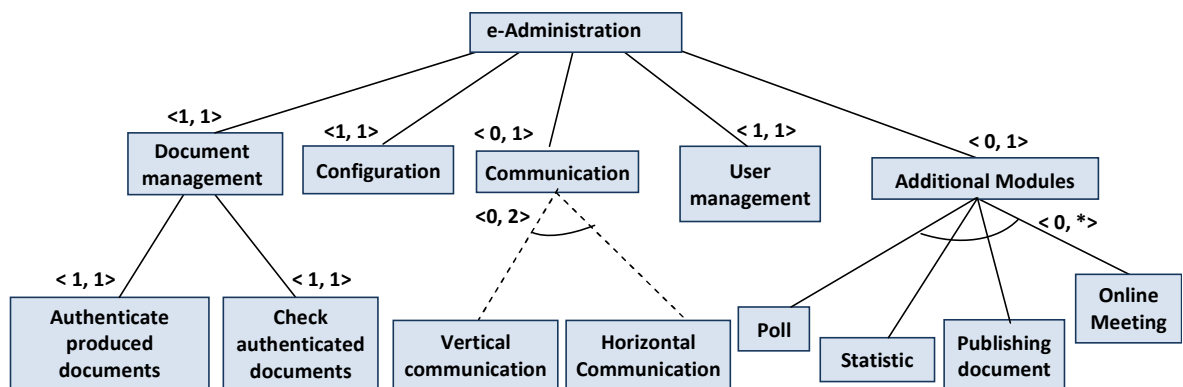


Figure 4. 2: Technical feature diagram for e-Administration product line

The third diagram Figure 4.3 represents the implementation features of the system; it means implementation details at lower and more technical levels. An e-APC application usually connects to a data base, provide a GUI, and supply authentication techniques such as: “Password”, “Digital certificates” and so on. It may also include “online viewer” for different documents kinds, and even enable to interact with citizens through different collaboration tools.

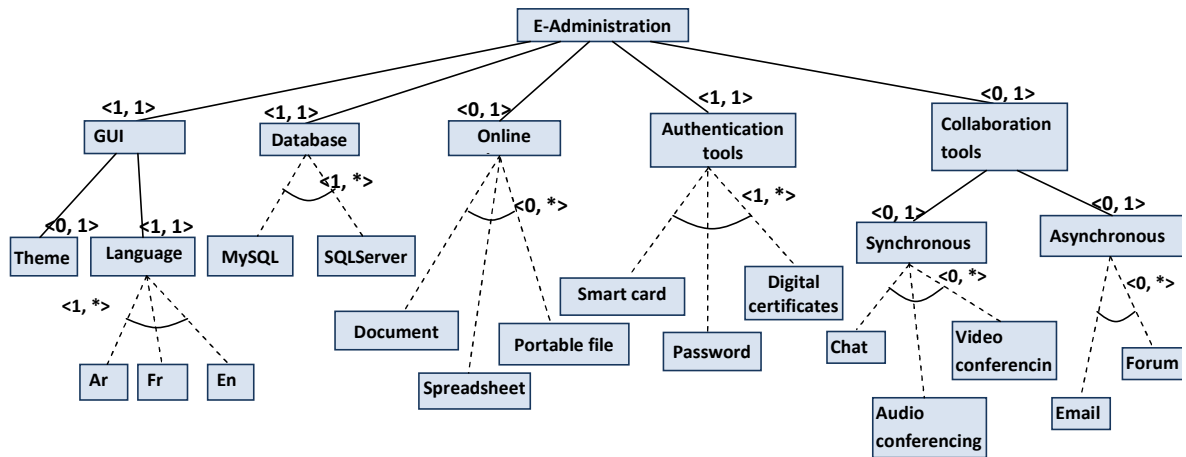


Figure 4. 3: Implementation feature diagram for e-Administration product line

#### 4.2.2. E-Education SPL

The second case study tackles another Government subfield which is *Education*. E-Education is a broad and ambitious domain which has paid a great attention recently, due to the prominent advantages it brings to education. E-Education means the use of ICT in education to improve the teaching-learning process. Instead of the several proposed e-learning solutions, SPLE present exciting benefits to this field, more detail about that can be found in our work [94]. E-Learning applications could be implemented in a variety of settings: for schools and universities to compliment or enhance classroom learning, for corporations to provide training and certification for their employees, and for organizations to provide e-learning courses to a larger learners population virtually anywhere in the world. In this case study, we are more interested in the e-Education systems supplied by Government to its citizens, which represent an important subfield of e-Gov. E-Education systems can be supplied by private or public institutions intended to primary, secondary and higher education. However, all of these applications share a set of common software elements and differ by some variable parts. The analyses we have done resulted in the feature diagrams shown in Figures 4.4, 4.2 and 4.3.

The first diagram reported in Figure 4.4 shows the business features of e-Education SPL. The main feature of an e-Education application is “Course management” that includes: the content of courses, enrollment of students in a course, and may include “Export content” functionality. An e-Education application may also comprise “Groups management”, “Download of official documents” and

“Evaluation” features. Evaluation allows users to benefit from several exercises types that could be resolved online (tests) or loaded as reports by students (work report). The score could be calculated “automatically” or “manually”. “Report cards” and “Certificates” could also be generated by the application. And finally, users could have the possibility of downloading official documents (report cards, certificates...) through the application which must allow in such case the electronic authentication of those documents.

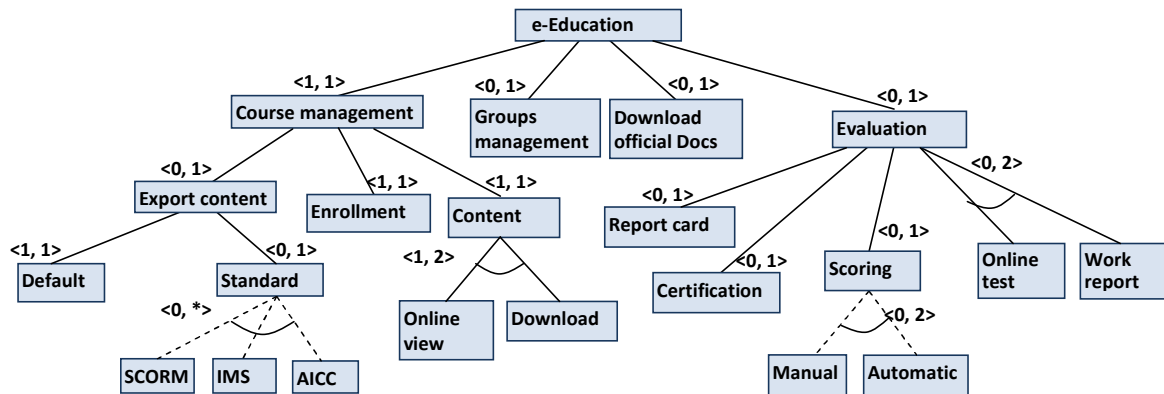


Figure 4. 4: Business feature diagram for e-Education product line

Technical features include functions related to “User management and authentication”, “Application configuration”, “Document authentication”, “Communication between applications”, and “Additional modules”. The resulted technical feature diagram is the same as the one reported in Figure 42 so we do not repeat it here.

Implementation features diagram is also similar to e-Admin implementation FM represented by Figure 4.3. E-applications usually need similar implementation tools or technologies to implement their various functionalities. Those features vary according to the capabilities and preferences of developers in addition to the influence of the current studied field.

#### 4.2.3. Comparison and results

In this subsection we review the two presented case studies, in order to analyze and compare them according to their specifications. Table 1 lists a set of features from both of e-Administration and e-Education product lines. Features are divided into three categories according to the feature diagrams they are extracted from: *business features*, *technical features*, and *implementation features*. For each category we list the first level features it includes from each SPL, such as:

1. *Business features*: represent the purpose of e-Government applications (Civil state services for e-Administration, and Course management for e-Education).
2. *Technical features*: are features that do not reflect the business aspect of applications. They comprise features related to the application (configuration, communication), features related to users (user profile management, user authentication), and features related to documents (documents authentication, monitoring and validation). Those features could be found in any application, but not all of them are mandatory.
3. *Implementation features*: are related to the implementation of the previous two kinds of features.

Table 4. 1: Comparing e-Admin and e-Education SPLs features

	<b>e-Admin SPL</b>	<b>e-Education SPL</b>
<b>Business features</b>	<ul style="list-style-type: none"> <li>- Event declarations</li> <li>- Data validation</li> <li>- Request for documents</li> <li>- Document modification</li> <li>- Download official Docs</li> </ul>	<ul style="list-style-type: none"> <li>- Course management</li> <li>- Groups management</li> <li>- Evaluation</li> <li>- Download official docs</li> </ul>
<b>Technical features</b>	<ul style="list-style-type: none"> <li>- User management</li> <li>- Document management</li> <li>- Communication</li> <li>- Configuration</li> <li>- Additional modules (Poll, Online Meeting, Statistics...)</li> </ul>	<ul style="list-style-type: none"> <li>- User management</li> <li>- Document management</li> <li>- Communication</li> <li>- Configuration</li> <li>- Additional modules (Poll, Online Meeting, Statistics...)</li> </ul>
<b>Implementation features</b>	<ul style="list-style-type: none"> <li>- GUI</li> <li>- Database</li> <li>- Online viewer</li> <li>- Collaboration tools</li> <li>- Authentication tools</li> </ul>	<ul style="list-style-type: none"> <li>- GUI</li> <li>- Database</li> <li>- Online viewer</li> <li>- Collaboration tools</li> <li>- Authentication tools</li> </ul>

From the table it is obvious that a large part of features from the two SPLs is common, and the main differences lies in business features. Technical features are the same, because they comprise technical aspects that are common to all e-Gov applications. Any e-Gov application needs to manage its users and manipulated-documents by maintaining data and authenticity. Implementation features are also identical. Choices of implementation could be available for any kind of applications. They defer according to the capabilities and preferences of developers in addition to users' requirements and the studied-field's particularities. The only kind of features that is mostly different is Business features what is

expected since each SPL has its different purpose. Nevertheless, commonality can appear in some business features such as the case of “Download official Docs” and “additional modules” features.

The large similarity amount between these two e-Gov SPLs evinces the necessity of systemizing inter-SPLs reuse. If we expand this perception to cover all the e-Gov subfields (e-Administration, e-Justice, e-Voting, e-Health, e-Education, etc.) or a large part of them, we find that applications from the other subfields need similar components for technical and implementation requirements as well as different components for their business or specific requirements. E-Gov applications usually require components to manage users and documents, tools for collaboration and inter-applications communication, additional modules such as: e-meeting components, search, statistics and so on. Promoting reuse between several SPLs targeted to the various e-Gov subfields will contribute in decreasing cost, time and effort of development.

Inter-SPLs reuse can be achieved in two ways:

1. Reusing components derived from one SPL in another SPL after adapting it to the reusing SPL requirements. This first solution requires adapting the reused components to fit each new reusing context. This reuse way is opportunistic and mostly limited since it was not planned in advance. In most cases, this reuse way requires substantial adaptations which make the developer’s work laborious and error prone.
2. Or, developing one e-Gov SPL covering the whole e-Gov domain instead of the set of separated SPLs. This solution will result in a very broad SPL covering several e-Gov subfields (public Admin, education, health, assurance, etc.). Considering that each e-Gov subfield SPL contains more than 200 features. The features number of an e-Gov SPL will reach more than a thousand features. Furthermore, not only the features are numerous but also the variations will be diversified since several business purposes are included. Complexity will significantly increase owing to the considerable variability contained in this e-Gov SPL. Furthermore, e-Gov domain is known to be a creeping domain (there borders are not well-defined and could expand by time) so challenges will be faced from the first development step (scoping).



Managing variability in such a SPL is not a trivial task and automating the derivation step will be difficult or even impossible.

In the next section, we propose an approach to overcome the before-mentioned challenges and benefit from commonalities between separated SPLs by systematizing inter-SPLs reuse.

### 4.3. Aspect Multiple Product Lines (AMPL)

The crucial goal of AMPL approach is to set up the bases for a MPLs engineering methodology. We aim by this methodology to provide MPLs engineers by the necessary means for organizing the MPL SPLs in order to simplify their management, systematizing reuse across separated SPLs and integrating interdependent SPLs.

#### 4.3.1. Separation of concerns in MPLs

Considering the e-Gov MPL that includes several subfields: e-Services supplied by public administrations such as: vital records, passports, identity card, voting, poll, justice, etc.; in addition to other services that could be provided by private or public institutions such as: education, health, assurance, transport, retirement, services for disabled people and so on. Separated SPLs can be built for each of those subfields: e-Admin, e-Voting, e-Justice, e-Health SPLs, etc. However, all of those SPLs still have in common some crucial features such as: security, Graphical User Interface (GUI), user management, communication and research. In order to well structure our MPL model, we propose to separate between features that are common to all (or a set of) MPL SPLs and those that are specific to each sub-SPL (intended for basic business sub-field functionalities). We see the common features to all (or a set of) MPL SPLs as crosscutting concerns for the MPL, since they reply to transversal needs for a set or all MPL SPLs, and we call them "MPL aspects". Thus, MPL aspects represent the common features for a set of SPLs within the same MPL. This acting way is different from the traditional separation of concerns that have been applied to SPLE [95]. In our approach, we do not encapsulate features that are scattered across several components (this could be done for each single SPL) but we operate at a higher level (MPL level)

and separate common features that are scattered across several SPLs of the MPL.

Those aspects may vary from an SPL to another one and their reuse requires adaptations to fulfill the new needs. Thus, MPL aspects can themselves be derived from dedicated SPLs. We propose to devote, for each MPL aspect, a SPL that allows systematizing its reuse throughout the MPL. We call those SPLs: Aspect SPLs (ASPLs). ASPLs serve as a base to be reused by the various MPL sub-SPLs; the development process of these latter will focus on business needs and will benefit from existing similarities between SPLs of the MPL. Thus, crosscutting reuse within the MPL is preplanned and systematized thanks to ASPLs.

Separation of concerns at MPL level helps structuring the MPL models. We distinguish between two kinds of models: MPL sub-SPLs models and ASPLs models. MPL sub-SPLs are those producing final applications for the various MPL subfields, while ASPLs produce components to be reused by the MPL sub-SPLs. The relationship between ASPLs and sub-SPLs is defined at early development stages and their integration can also be done early. Consequently, distributed derivation challenges will be avoided.

#### 4.3.2. Aspect SPLs

ASPLs [96] are a set of SPLs devoted for producing components that materialize the MPL aspects. Those aspects may include: business features (applications capabilities related to the SPL purpose), technical features (non-business capabilities as: security, user management, documents management, application configuration) or implementation features (implementation details at lower and more technical levels). Taking the case of e-Gov MPL, the key common features that we can distinguish are:

- Security: e-Gov applications must be highly protected since they handle personal data, communicate between them, and transact with users. Security is a common requirement between e-Gov MPL SPLs.
- Communication: e-Gov applications need to communicate, share and exchange data (in context of business processes) in order to provide efficient services to citizens.

- GUI: Having a unified GUI for the various e-Gov applications is an important goal that helps promoting and facilitating their usage.
- Furthermore, there are other functionalities that may be included in several e-Gov applications to complete potentially additional needs, such as: e-Meeting, statistics, poll, search, publishing documents and so on. Those additional functionalities represent also aspects for the e-Gov MPL.

Thus, in the e-Gov case, we can develop an ASPL planned for: security, e-Meeting, statistics, GUI and communication. ASPLs must be designed to fit needs of each MPL sub-SPL that is intended to reuse them. So instead of adapting components for each reusing SPL, ASPLs will take into account the variable contexts of the whole MPL.

Hence ASPLs aim, on the one hand, to systematize reuse throughout the various MPL SPLs by defining the MPL aspects, and devoting ASPLs for each of them. On the other hand, SPL development often gives more emphasis to business functionalities. Ignoring secondary (especially technical) functionalities decreases systems' performances, given that a weakness in the SPL design can cause problems throughout all its members. Improving these functionalities is one of ASPLs' advantages. Since ASPLs will be created by specialized developers and tested in different contexts, they will provide MPLs by high quality components which will participate in improving the performances of the derived applications in addition to simplifying the MPL SPLs development processes by reusing core assets derived from ASPLs.

#### 4.3.3. Partial derivation

Partial derivation<sup>4</sup> is a transformation procedure that takes as input the core assets of an SPL to be reused (ASPL) and generates a partially derived SPL ready to be integrated with its reusing SPL (MPL sub-SPL) [96] [97]. Partial derivation consists in modifying a set of Variation Points (VPs) included within the reusable SPL's core assets in order to fit the reusing SPLs' requirements. Ultimately, the partial derivation can alter a set of VPs or in some cases all the SPL VPs may be modified to meet the new needs. As we will have a full SPLs integration (not only the code is composed), all artefacts types to be composed

---

<sup>4</sup> Details about Partial derivation can be found in Chapter 5.

must be partially derived from requirements models to the architecture and implementation code. The set of partially derived artefacts will be completely derived thereafter as a part of the reusing SPL.

Partial derivation differs from the traditional derivation process by the fact that: the partial derivation process does not produce an application ready to be used (as it is the case in the ‘full’ derivation), but rather a set of partially derived artefacts intended for integration with their reusing SPL and that could be derived completely as part of it.

#### 4.4. AMPL Engineering

The AMPL engineering consists in two stages: ASPLs engineering and Sub-SPLs engineering. Figure 4.5 shows the main AMPL engineering steps [96]. We have not detailed the various SPLs development processes since they are similar to the traditional ones (Chapter 1- Section 3). So, in the Figure 4.5 and next subsections we focus on the reuse steps of ASPLs throughout AMPL engineering.

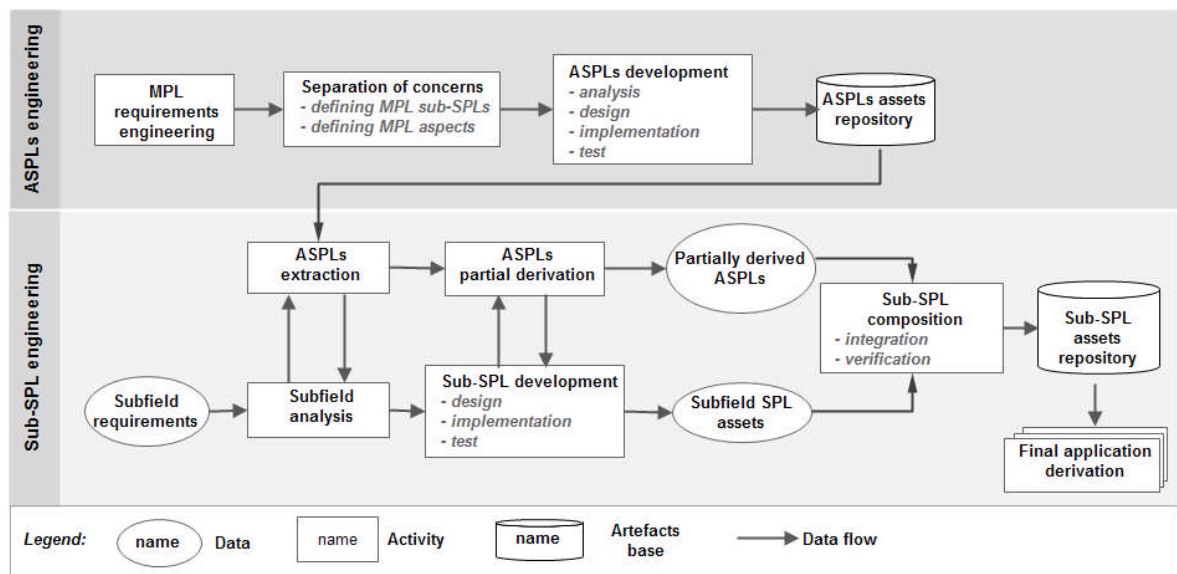


Figure 4. 5: ASPLs life cycle within AMPL engineering [96].

Before embarking the ASPLs development process, we must first define the MPL scope and separate between the MPL sub-SPLs and ASPLs. This is done through two activities: MPL requirements engineering and separation of concerns.

1. *MPL requirements engineering*: consists in defining the MPL boundaries by defining which sub-fields are included in the MPL scope, eliciting and

documenting the common and variable requirements in the various MPL subfields. This activity takes as input the MPL goals and relies on the sub-SPLs requirements if they exist, or on existing applications in the MPL field. Its output is the MPL scope and the documentation of the MPL requirements.

2. *Separation of concerns*: consist in analyzing the MPL requirements in order to separate between the MPL sub-SPLs and determine the MPL aspects (the MPL crosscutting concerns). The MPL sub-SPLs are separated according to the goal of each MPL sub-fields (for different goals we devote different SPLs). The MPL aspects are determined after that by specifying the common features to a set (or all) of MPL sub-SPLs. For each MPL aspect an ASPL is developed in the next step. The input of this activity is the MPL requirements and goals. Its output is a planning that defines the MPL sub-SPLs, the ASPLs and which sub-SPLs reuse components from which ASPLs. In addition to the documentation of the MPL requirements organized into sub-SPLs and ASPLs.

#### 4.4.1. ASPLs engineering

This first stage includes a set of ASPLs development processes, each one dedicated to produce an SPL that fulfils the requirements of several MPL subfields for a specific MPL aspect. Examples on ASPLs for e-Gov case have been demonstrated in section 3.2. ASPLs development processes are similar to the traditional SPLs development processes, unless there is no application engineering step since their aim is not to develop running applications, but to create core assets that will be reused later. Each ASPL development process takes as input one MPL aspect's requirements which are related to a set of MPL subfields that share this aspect. The aim is to fulfill this aspect's requirements for an SPLs set in order to permit crosscutting reuse among them.

The outputs of this stage consist in all the developed artefacts resulting from all ASPLs development processes, including: reference requirements, reference architectures, reusable components, and reusable tests artefacts. Those artefacts are collected in the ASPLs assets repository (Figure 4.5) that will be reused in the next AMPL engineering stage. Finally, we pay attention that the feedback obtained when developing final applications is taken into account when maintaining the ASPLs repository in order to integrate new requirements, developing them, and making them available to be reused

#### 4.4.2. Sub-SPLs Engineering

In the second stage, SPLs are developed for each MPL subfield. A sub-SPL process is similar to the conventional SPL development process, but it is facilitated by reusing ASPLs repository. This process will be performed for each MPL subfield that has been included in the AMPL scope, yet new subfields can be considered thereafter if needed. It takes as input –more than the MPL subfield requirements- the outputs of the first phase. Reusing ASPLs core assets will facilitate this process and make it faster. Developers should not worry about needs that are already developed during the first stage; they will focus only on the particular services provided by the sub-SPL. This process results in a set of core assets including the sub-SPL artefacts merged with the partially derived ASPLs artefacts. ASPLs reuse occurs in three steps (Fig. 3): extraction, partial derivation, and integration and verification.

##### 4.4.2.1. Extraction

In this step, the ASPLs repository is browsed in order to extract the relevant ASPLs to be reused by the sub-SPL under development. For example, in the e-Gov MPL, if we are about developing a SPL for e-Admin (same for an e-Education) subfield, all of security, e-Meeting, statistics, GUI and communication ASPLs are selected to be reused by those sub-SPLs, an e-health SPL may require an image processing ASPL which allows several image processing algorithms for example, while an e-Voting SPL does not require e-Meeting ASPL. The input of this activity is the subfield requirements and the ASPLs repository. The output of this step is the set of ASPLs core assets needed by the sub-SPL under development and the annotation of reuse places in the sub-SPL models. The extraction activity involves three tasks:

1. Define the ASPLs which could be of benefit to the sub-SPL according to the requirements specified during the subfield analysis and import their core assets to the SPL development site. The functionalities provided by the selected ASPLs will not be developed during the sub-SPL engineering because they are already developed during ASPLs engineering, thus, they will be reused from the imported ASPLs.

2. Determine the places in the sub-SPL where ASPLs need to be reused. We call those places “reuse-spots”. This is different from the pointcut mechanism in Aspect Oriented programming [95], since reuse-spots do not represent always aspects to the final application, they may represent business components.
3. Tag the reuse-spots by special annotations indicating that they will be replaced by the corresponding partially derived ASPLs. The annotated parts will remain black boxes until integration step. Noting that the annotations must be maintained over all the SPL artefacts. For instance in feature models, if a feature correspond to a point where reusing an ASPL is needed, this feature is annotated by «reuse spot». As illustrated in Figure 4.6-a, all of: e-meetings, communication, and statistics features represent reuse-spots for e-Admin FM. A reuse-spot may reflect a simple functionality that will be derived from an ASPL (we may reuse the e-mail option from the communication ASPL), as it may be the root of a sub-system (as it is the case for e-Meeting ASPL).

This activity is performed simultaneously with subfield analysis step in order to prepare the sub-SPL for all possible ASPLs reuse.

#### 4.4.2.2. Partial derivation

During this step, the previously selected ASPLs are prepared for integration with the sub-SPL. Partial derivation means that during this step a set of (in some cases all) VPs which are part of a chosen ASPL will be adjusted according to the reusing domain’s requirements. The input of this activity is all the ASPLs selected in the previous step. The output of partial derivation for each ASPL is another SPL partially derived that includes more or less configuration choices (according to the transformations applied) than the original ASPL. Nevertheless, the resulted SPL fulfills particularly the reusing sub-SPL’s requirements and it is ready to be merged with it.

The partial derivation of a model may result in a model that includes more or less configuration choices than the former i.e. partial derivation can expand or restrict the derived model. The choices set described by a partially derived model may be broader than the ones covered by the source model. This comes back to the fact that sub-SPLs may include some specific requirements that have not been covered by the ASPLs, so the ASPLs must be expanded to cover the new needs.

Restricting or expanding an ASPL model is decided according to the reusing SPL requirements. We distinguish between two partial derivation categories: restriction and expansion techniques. The partial derivation of a model can include transformations from both categories.

- Restricting a model means altering the model in a way that restricts the choices set covered by the resulted model. The set of transformations that can be done in this category are: - reducing a cardinality interval of a choice VP - changing a VP type from optional to mandatory - restricting an attribute by assigning a value – omitting a VP or a feature (eventually a component).
- Expanding a model means to modify this model in such a way that expand the choices covered by the resulted model. The set of transformations included in this category are: - extending a cardinality interval of a multi-choice VP – changing a VP type from mandatory to optional – adding a VP or a feature (eventually a component).

We must note that partial derivation is applied to all ASPL artefacts, not only to the FM. If a VP is altered in the FM it should be altered in the same way for all next models that contain it. Finally, partially derived ASPLs' validity must be checked against the constraints defined when developing ASPLs and those defined by the reusing sub-SPL. This will ensure the integrity of the resulted artefacts and avoid conflicts at integration step.

#### 4.4.2.3. Integration and verification:

Integration means the merging of the various partially-derived ASPLs for a specific MPL subfield with the SPL of this field. The inputs of this step are all the partially-derived ASPLs artefacts in addition to the reusing sub-SPLs artifacts. Its output is a set of complete sub-SPLs which are ready to be derived to produce final applications for the various MPL subfields. Those sub-SPLs are collected in the sub-SPLs assets repository.

During integration step the need for a composition model arises. The composition model is the description of how a complete MPL SPL is composed from a set of partially-derived ASPLs and their reusing sub-SPL i.e. it describes dependencies between the sub-SPL and the partially-derived ASPLs. In our approach, composition is performed for each sub-SPL with its reused ASPLs.



Hence, each sub-SPL requires a specific composition model to represent its dependencies with the ASPLs it is reusing. In fact, we have not to invent a new model in order to describe composition dependencies since the reference sub-SPL models (models performed during sub-SPL development as reference requirement and reference architecture) do the job. As stated before, features (and components) that will be implemented by reusing ASPLs are annotated in the sub-SPL models as reuse-spots and their relationships with the rest of the system are specified by the reference models. Consequently, the sub-SPL model is the composition model of this SPL with its reused ASPLs.

As explained in extraction step, reuse places of ASPLs are annotated in the sub-SPLs as «reuse-spots» (particular Features). In this step those black boxes will be replaced by the relevant partially derived ASPLs. For instance, features annotated as «reuse-spots» are replaced by the corresponding partially derived FMs of the reused ASPLs (the next section presents an example). Another way to integrate FMs is to consider reuse-spots features as references to the ASPLs FMs in order not to clutter the resulted model. In the case of architecture models, we must note that components implemented by ASPLs are considered as aspects for the MPL not for the sub-SPL i.e. they may represent business components. Thus, components derived from ASPLs can represent either aspect-components or plug-in components for particular sub-SPL architecture and they will be integrated accordingly.

An important point to consider during this step is models consistency. Models consistency must be kept after integration. Therefore, the resulted artifacts' integrity is checked after each integration step. This is done by checking sub-SPLs constraints with regard to the merged models.

Models integration is a challenging activity in an complex environment as e-Gov. For us, we assume that our approach is applied in an homogeneous environment, such as the same modeling languages and implementation techniques are used. Using the same modeling and implementation languages will help widely in the well performance of integration activity moreover it allows reaching final results in shorter development time and avoid long procedure of adaptation, transformation into common language, and communication between stakeholders.

#### 4.4.2.4. Application engineering

At the end of the previous stages, we obtain a set of SPLs (that construct an MPL) each of them specialized in the production of applications specialized in a particular MPL subfield and at the same time those SPLs share the common features between them through ASPLs. In short, diversity is ensured by letting the SPLs separated and the MPL crosscutting reuse is managed using the ASPLs. At this phase, the various MPL SPLs are ready to produce final applications. This last activity is responsible for deriving final applications from the sub-SPLs assets repository. During this step, variability is completely bound according to the final applications' needs. This process could be performed according to traditional methods and it result in a final application ready to be used.

#### 4.5. Discussion

In conventional SPLs environments, the derivation of a final application implies usually a single user working on the derivation of a single variability model. Otherwise, MPLs environments include several sub-systems and multiple users are involved to derive the various variability models. Thus, multiple derivation processes are handled simultaneously for the various MPL sub-SPLs and this activity is known as distributed derivation. In such a case communication is needed between the involved users in order to guarantee awareness about the decisions made in the deferent sub-SPLs [98]. If final applications should be integrated in order to produce a complete system, compatibility is needed among them, otherwise adaptation challenges will be encountered. Furthermore, competitive SPLs producing similar products may delay derivation processes of the SPL reusing their outputs. If components needed by an SPL are provided by several other MPL SPLs, choosing the right component for reuse requires a whole decision process and results in lengthening the production operation. In our approach we act differently, instead of waiting the derivation phase of an MPL and facing up the aforementioned challenges we suggest the early integration of each MPL sub-SPL with the set of SPLs it needs to reuse. SPLs intended to be reused by a particular sub-SPL, are partially derived according to the reusing SPL requirements and are integrated during its domain engineering. The matter is to

move from distributed derivation to traditional derivation since at derivation time reusable components belong already to the reusing SPL.

A key step of our contribution is partial derivation. This activity allows the early and full integration of SPLs artefacts. A major task to perform would be the automation of partial derivation. The partial derivation automation relies on two concepts: transformation rules and traceability. Transformation rules correspond to the various partial derivation techniques (see chapter 5). Those techniques can be formalized for each SPL artefact. Moreover, traceability must be kept among artefacts to automate the passage through the various abstraction levels. Thus, when mapping features to architecture, the partial derivation of a FM results in a configuration that can be used for the automatic partial derivation of the architecture model. Nevertheless, this task cannot be fully automated since there are some partial derivation activities requiring users' involvement. These activities stand mainly in the expansion partial derivation techniques. When adding a new element to the system, the developer must interfere to define the properties of the new element and its dependencies with respect to the SPL core assets.

In the presentation of our approach, we have focused on two e-Gov MPL sub-SPLs in order to illustrate the process and the new concepts. To ensure the scalability of the approach, the MPL requirements engineering must include the analysis of all the subfields expected to be part of the MPL scope. After that, ASPLs are developed taking into account the requirements of those subfields. Nevertheless, the MPL scope can be broaden by adding new sub-SPLs. When adding a new sub-SPL, the ASPLs of the MPL have to be revised for including the new requirements imposed by the new sub-SPL. The ASPLs may be also expanded by the specific requirements that could be detected during sub-SPL engineering, if those later are expected to be reused by other sub-SPLs. So the AMPL engineering process is an iterative process in such a way that keeps the MPL up-to-date continuously.

Our approach seems spending more time at MPL domain engineering. This is true because planning for reuse, analyzing MPL field, detecting MPL aspects and building the set of ASPLs requires more time than it is the case for traditional MPLs (direct development of MPL sub-SPLs). However, the aim of our approach is to avoid longer and hard decision and adaptation procedures during application

engineering phase. The matter is that the MPL base (ASPLs and sub-SPLs) once it is built, it will allow the fast production of final applications, while, in the conventional case, time is wasted for each new application derivation.

AMPL approach is a new MPL management approach that aim mainly to systematize reuse across separated SPLs within an MPL. AMPL could be efficiently applied in a software field if this latter produces applications for several subfields or market segments, such as those fields are characterized by significant commonalities which could be encapsulated in specialized SPLs (ASPLs). Reuse within each MPL subfield is managed through sub-SPLs while the commonalities among them will be managed thanks to ASPLs. Nevertheless, we note that both of separation of concerns and partial derivation are autonomous from each other, i.e. each of them can be used independently. For instance, separation of concerns can be used in a MPL environment for structuring the MPL model. It can also be adapted for decomposing a large SPL into a set of sub-SPLs and thus moving from single SPL to MPL approach. On the other side, partial derivation can be adopted for the merging of two (or more) interdependent SPLs aiming inter-reuse even if they do not belong necessarily to the same MPL.

#### 4.6. Conclusion

The emergence of MPLs in some large software fields arises several challenges. In this chapter we propose AMPL approach to tackle them and to reach a well planned MPLs development. AMPL approach bases on two main concepts: separation of concerns and partial derivation. Separation of concerns at MPL level helps systemizing reuse among SPLs by organizing the MPL models into ASPLs aiming to develop the reusable components within an MPL, and sub-SPLs targeted to produce the MPL final applications. The partial derivation is used to prepare ASPLs for integration with their reusing sub-SPLs. The early integration of MPL SPLs avoids the distributed derivations challenges encountered thereafter. In this chapter we have explained our methodology for developing MPLs, and illustrated it for the e-Gov field.

Our first perception when developing e-Gov SPLs is that reuse within each e-Gov subfield is systematized through SPLE techniques; while when inter-SPL reuse is needed no techniques are available to manage it. Consequently,

systematic reuse is lost at SPLs level. We suggest, thus, benefiting from SPLE advantages not only within each MPL subfield but also across separated and interdependent MPL subfields. This way, reuse is effectively managed at two levels: between products within each SPL and between SPLs of an MPL.

The existing works focus on resolving reuse challenges in the late development stages i.e. at derivation time. They tend to derive separated SPLs and integrate their heterogeneous instances. At this level several challenges are encountered known by the distributed derivation. This reuse way is still opportunistic because it has not been planned before, and results in long procedures of adaptation and decision. Our approach, avoids these challenges by planning for reuse from the early development stages. This planning is ensured by the introduction of ASPLs that are responsible for the production of common components through the MPL. Thus a crucial outcome of our work is the systematization of reuse between SPLs of an MPL. The ASPLs will be, after that, partially derived in order to ease their integration with their reusing SPLs. The partial derivation represents an important technique for merging separated SPLs. Moreover, it helps integrating the SPLs early in the development process to avoid the distributed derivation challenges thereafter. Thus, partial derivation and early integration represent another crucial outcome of our work.

# CHAPTER 5

## PARTIAL DERIVATION AND COMPOSITION

### 5.1. Introduction

Current MPLs tends to integrate components derived from some SPLs in other reusing SPLs within the same MPL at derivation time. This integration way (distributed derivation) results in several problems. Reused components that are already developed using particular modeling and implementation techniques must be adapted to fit the new application requirements. This became more complicated if the used modeling and implementation languages are different. Moreover, this procedure needs to be repeated for each context included in the reusing SPL. It means at each derivation time of an application that needs reusing a component from another SPL, this component must be adapted to fit the new application requirements. If the target application needs reusing several components from several other SPLs, the whole adaptation and integration process must be repeated accordingly, what result in delaying derivation and produce several adaptation and integration challenges.

The AMPL approach (presented in the previous chapter) aims, among others, to avoid the distributed derivation challenges. It switches the MPL engineering from *distributed derivation* to *traditional derivation* which is easier and less time consuming. This is done by performing the reuse process during the MPL domain engineering phase. Instead of reusing SPLs instances, we reuse the whole SPL that is needed by another SPL. Each MPL sub-SPL is merged with its reused SPLs in the early development stages resulting in a set of SPLs ready to be derived using traditional derivation techniques. In order to allow the early

integration of SPLs we have introduced a new derivation technique called: Partial derivation. In this chapter we present the partial derivation transformation techniques for two SPL models: the FM and the Architecture. **Then** we explain how the sub-SPL model is integrated with the set of partially derived ASPLs. The presented techniques are illustrated by a case study.

## 5.2. Partial derivation

Partial derivation is a transformation procedure that takes as input the core assets of an SPL to be reused (ASPL) and generates a partially derived SPL ready to be integrated with its reusing SPL (MPL sub-SPL). Partial derivation consists in modifying a set of VPs included within the reusable SPL's core assets in order to fit the reusing SPLs' requirements. Ultimately, the partial derivation can alter a set of VPs or in some cases all the SPL VPs may be modified to meet the particular needs. As we will have a full SPLs composition (not only the code is composed), all artefacts types to be composed must be partially derived from requirements models to the architecture and implementation code. The set of partially derived artefacts will be completely derived thereafter as a part of the reusing SPL.

Partial derivation is comparable to the specialization concept that was introduced by Czarnecki et al [99] [100]. They define specialization as the transformation process that takes a feature diagram and yields another feature diagram, such as the set of configurations denoted by the latter diagram is a true subset of the configurations denoted by the former diagram. Successive specialization processes result in a final configuration, this method is called *staged configuration* [99]. Specialization differs from partial derivation in two crucial ways. On the one side, the purpose of introducing specialization is to allow handling applications derivation through several configuration stages what is needed in the case of software supply chains. Final applications derivation step is then decomposed into several specialization stages each one is performed by a particular actor, whereas partial derivation aims to prepare the reusable SPLs for integration with the reusing SPLs during the domain engineering phase. On the other side, specialization is defined to be applied particularly on the feature models what is clear from its definition, while partial derivation is applied to all the artefacts

extracted from the reusable SPLs domain engineering (including requirements models, architecture and final code).

Unlike specialization, the resulting model from a partial derivation procedure does not describe necessarily a sub-set of the systems set described by the original model. In some cases, the partially derived model is extended by adding new functionalities or VPs to fulfill the particular needs of the reusing field. This is due to the fact that the resulted model will be integrated with the entire reusing SPL (with all its covered contexts) not a particular final application. Therefore, new requirements can be detected for some SPL contexts that have not been considered when developing the SPL for reuse (ASPL). This latter, should be extended by the new components needed. We can then distinguish between two partial derivation categories: *Restriction* and *Expansion* techniques. The partial derivation of a model can include transformations from both categories.

- *Restricting a model* means altering the model in a way that restricts the choices set covered by the resulted model. The set of transformations that could be done in this category are: - to restrict a choice VP - to change a VP type from optional to mandatory - to restrict an attribute by assigning a value - to omit a VP or a feature (eventually a component).
- *Expanding a model* means to modify this model in such a way that expand the choices set covered by the resulted model. The set of transformations included in this category are: - to extend a choice VP – to change a VP type from mandatory to optional – to add a VP or a feature (eventually a component).

Those techniques are applied to all the artefacts types of an SPL. The two main artefacts in our methodology are: the FM and the Architecture models. Next sections present the partial derivation transformation rules for both of them illustrated by some examples.

### 5.2.1. Restriction Techniques

#### ***Restricting a Choice VP***

A choice VP allows several configuration possibilities unlike optional and mandatory VPs that allow only two resolution possibilities. It describes the variation of a set of related elements and may limit the options by a cardinality interval. For an FM, a choice VP may correspond to: a solitary feature with



cardinality or choice with cardinality, while in the architecture it corresponds to: a choice connector or a choice component. Restricting a choice VP means reducing the configuration possibilities enabled by the VP. This can be done by removing an option or an options-set from the elements described by the VP or by reducing the related cardinality interval.

### ***Changing a Variability Type from Optional to Mandatory***

A variability type may be changed if needed by the reusing SPL. An optional functionality in the reusable SPL may become mandatory if its inclusion in applications is obligatory for particular reusing contexts. Consequently, a model element type can be changed from optional to mandatory. This results in reducing the configuration choices in the obtained model since optional type allows two configuration choices (the element can be included or not in the application), while mandatory imply only one configuration way (the element must be included in the application).

### ***Removing a Functionality***

A restriction operation may be done by removing a functionality completely from the model if it is not expected to be reused by any context included in the reusing SPL.

## 5.2.2. Expansion Techniques

***Extending a Choice VP*** It means increasing the configuration possibilities enabled by the VP. This can be done by adding an option or an options-set to the elements described by the VP or by extending the related cardinality interval.

***Changing a Variability Type from Mandatory to Optional*** A mandatory model element may take the type optional instead of mandatory if its inclusion in the reusing SPL application is not obligatory in some cases. This results in extending the configuration possibilities of the model (two possibilities instead of one).

***Adding a Functionality*** In the case of specific new requirements by the reusing SPL, the reused SPL can be extended by new functionalities. Those later must have no conflict with the model's constraints in order to preserving consistency.

### 5.3. Partial derivation of the Feature Model

The partial derivation of a FM encompasses the transformations-set explained below [97].

#### 5.3.1. Restricting a Feature Model

##### 5.3.1.1. Restricting a Choice VP

In the FMs, this means restricting *a solitary feature with cardinality* or *a choice with cardinality*:

1. We may restrict a choice with cardinality by removing a grouped feature or a set of grouped features from the choices and modifying the interval accordingly. If the features group size is  $s$  and its cardinality is  $\langle n, m \rangle$  such as  $n \leq s$ , when removing one grouped feature the new features group size will be  $s - 1$  and its new cardinality interval will be  $\langle n, \min(m, s-1) \rangle$  where  $\min(x, x')$  takes the minimum of the two natural numbers  $x$  and  $x'$ . We must note that when removing a feature all its sub-tree of features is removed from the FM. Special cases occurs when all the choices are removed or only one feature remains from the features group. In the case of removing all the grouped features, this VP disappears from the FM and the parent feature could be omitted if it is not related to other sub-features than the group.
2. Also, a choice with cardinality interval  $\langle n, m \rangle$  may be restricted by changing the interval to  $\langle n', m' \rangle$ , where  $n' \geq n$  and  $m' \leq m$ . Special cases occur when getting  $\langle 1, 1 \rangle$  or  $\langle 0, 0 \rangle$  intervals. When getting  $\langle 1, 1 \rangle$  interval and the group of features includes more than one feature, the choice type is called *Alternative*. When we restrict the choice interval to  $\langle 0, 0 \rangle$ , this implies removing the set of choices with their descendants (because no choice will be needed), and removing the parent feature of the choices set if it is not related to other sub-features than the grouped ones.
3. In the case of solitary feature with cardinality, cardinality interval could be derived in the same way as choice with cardinality interval (rule 2). Special cases occur when getting  $\langle 0, 1 \rangle$ ,  $\langle 0, 0 \rangle$  or  $\langle 1, 1 \rangle$  intervals. If we obtain  $\langle 0, 1 \rangle$  interval, the feature type is then optional. If we obtain  $\langle 1, 1 \rangle$  interval, the feature type is then mandatory. In the case of  $\langle 0, 0 \rangle$  interval, the feature is completely removed from the diagram.

### 5.3.1.2. Changing a Feature Type from Optional to Mandatory

An optional feature may be changed to mandatory type if needed, this imply the obligatory existence of this feature in final reusing applications. This operation is done by changing its cardinality from  $\langle 0, 1 \rangle$  to  $\langle 1, 1 \rangle$ .

### 5.3.1.3. Restricting an Attribute by Assigning a Value

A restriction way of an FM may be done by assigning a value to an attribute what is particular for FMs (have not been mentioned in the previous section). This operation can be done by initializing an uninitialized attribute, or changing the initial value.

### 5.3.1.4. Removing a feature

In FMs, we may omit a feature with all its descendants if it is not needed by the reusing SPL. Omitting a grouped feature or an occurrence from solitary feature with cardinality cases correspond to what is described in Section 3.1.1.

Table 5. 1: Restriction operations of an FM

<b>Restriction operation</b>	<b>Precondition</b>	<b>Effect</b>
Remove_grouped_feature( $f, g$ ) (Section 3.1.1)	$f$ is part of the features group $g$ with size $s$ and cardinality $\langle n, m \rangle$	$s := s - 1$ refine $\langle n, m \rangle$ to $\langle n, \min(m, s - 1) \rangle$ if $n == 0$ and $m == 0$ then remove( $g, M$ )
Restrict_group_cardinality( $g, M$ ) (Section 3.1.1)	$g$ is a features group in the feature model $M$ with cardinality $\langle n, m \rangle$	Refine $\langle n, m \rangle$ to $\langle n', m' \rangle$ if $n' == 0$ and $m' == 0$ then remove( $g, M$ )
Restrict_feature_cardinality( $f, M$ ) (Section 3.1.1)	$f$ is a solitary feature with cardinality $\langle n, m \rangle$	Refine $\langle n, m \rangle$ to $\langle n', m' \rangle$ if $n' == 0$ and $m' == 0$ then remove( $f, M$ )
Opt_to_Mdr( $f, M$ ) (Section 3.1.2)	$f$ is a feature with cardinality $\langle 0, 1 \rangle$	Refine $f$ cardinality to $\langle 1, 1 \rangle$
Assign_Attribute( $t, f$ ) (Section 3.1.3)	$t$ is an attribute of the feature $f$	$t = v$
Remove( $f, M$ ) (Section 3.1.4)	$f$ is a feature in the feature model $M$	Refine $f$ cardinality to $\langle 0, 0 \rangle$

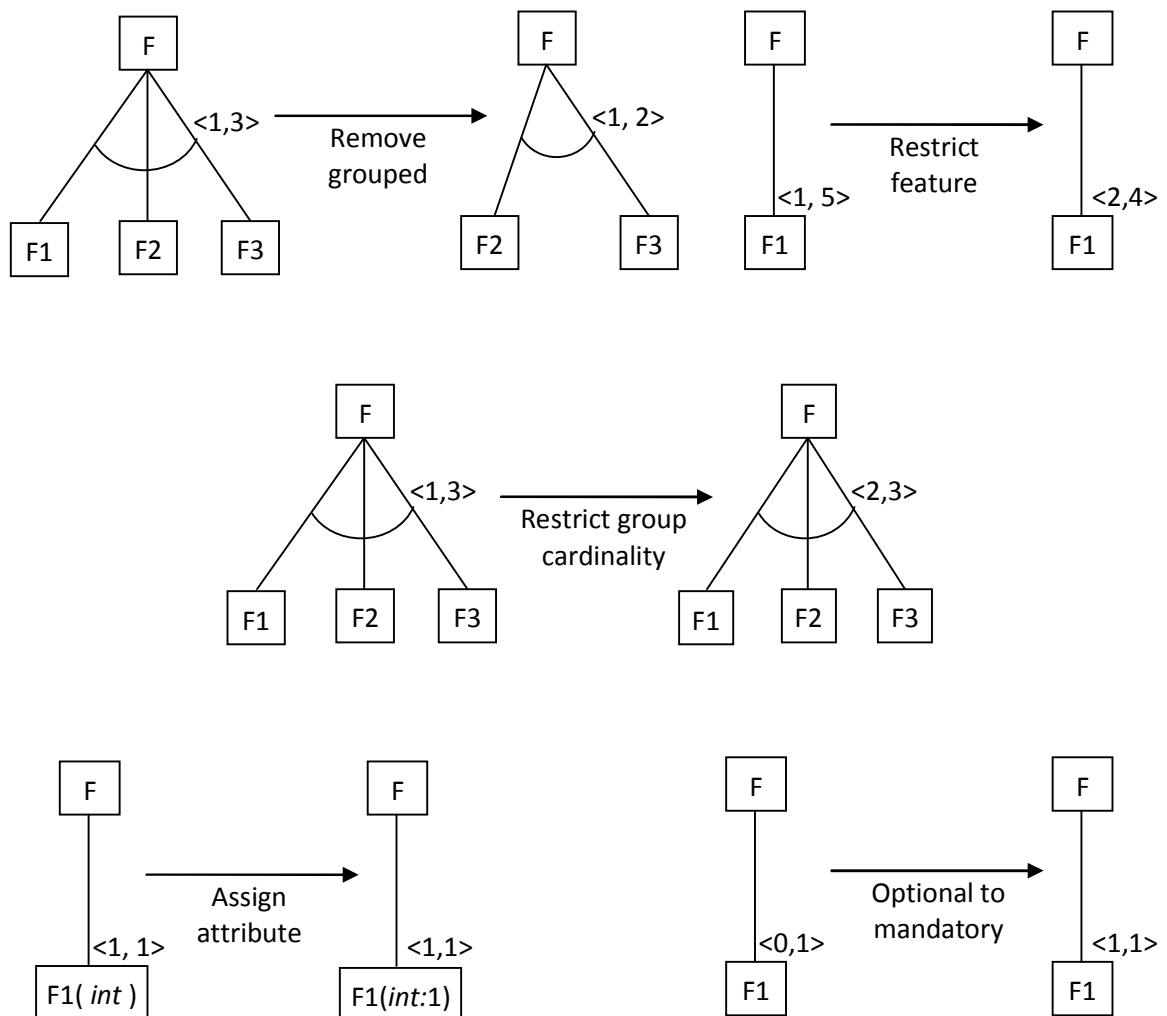


Figure 5. 1: Restriction operations on FMs.

### 5.3.2. Expanding a Feature Model

#### 5.3.2.1. Extending a Choice VP

In feature modeling, a choice with cardinality may be extended if features are added to the feature group (changing the interval is not obligatory) or its cardinality interval is extended as long as consistency is preserved. A choice with cardinality interval  $\langle n, m \rangle$  and size  $s$  may be extended to  $\langle n', m' \rangle$  where  $n' \leq n$  and  $m' \geq m$  and  $m' \leq s$ .

In the case of solitary feature with cardinality, more occurrences can be added to the feature. This is done by changing its cardinality from  $\langle n, m \rangle$  to  $\langle n', m' \rangle$  such as  $n' \leq n$  and  $m' \geq m$ . Even a feature without cardinality may become with cardinality if new occurrences are added. If it is related to sub-features, they are

eventually multiplied. Constraints related to the altered features must be reviewed and modified if needed to keep the models consistent.

### 5.3.2.2. Changing a feature Type from Mandatory to Optional

A mandatory feature may become optional feature, this is done by changing its cardinality from  $\langle 1, 1 \rangle$  to  $\langle 0, 1 \rangle$  in the FM. This transformation task decreases the probability of including the concerned feature in final reusing applications but it results in two configuration choices. Dependencies constraints must be changed or added accordingly.

### 5.3.2.3. Adding a feature

An FM can be expanded by adding a new feature (with its descendants), or a feature choice. Related constraints are changed or added accordingly. The case of adding a feature to a group, or multiplying a feature occurrences are described in Section 3.2.1.

Table 5. 2: Expansion operations of an FM.

<b>Expansion operation</b>	<b>Precondition</b>	<b>Effect</b>
Add_feature_toGroup( $f,g,M$ ) <b>(Section 3.2.1)</b>	$g$ is a features group in the feature model $M$ with size $s$ and cardinality $\langle n, m \rangle$	$s := s+1$
Extend_group_cardinality( $g,M$ ) <b>(Section 3.2.1)</b>	$g$ is a features group in the feature model $M$ with size $s$ and cardinality $\langle n, m \rangle$	If $m' \leq s$ Refine $\langle n, m \rangle$ to $\langle n', m' \rangle$
Extend_feature_cardinality( $f,M$ ) <b>(Section 3.2.1)</b>	$f$ is a solitary feature with cardinality $\langle n, m \rangle$	Refine $\langle n, m \rangle$ to $\langle n', m' \rangle$
Mdr_to_Opt( $f,M$ ) <b>(Section 3.2.2)</b>	$f$ is a feature with cardinality $\langle 1, 1 \rangle$	Refine $f$ cardinality to $\langle 0, 1 \rangle$
Add( $f,M$ ) <b>(Section 3.2.3)</b>	$f$ is a feature	

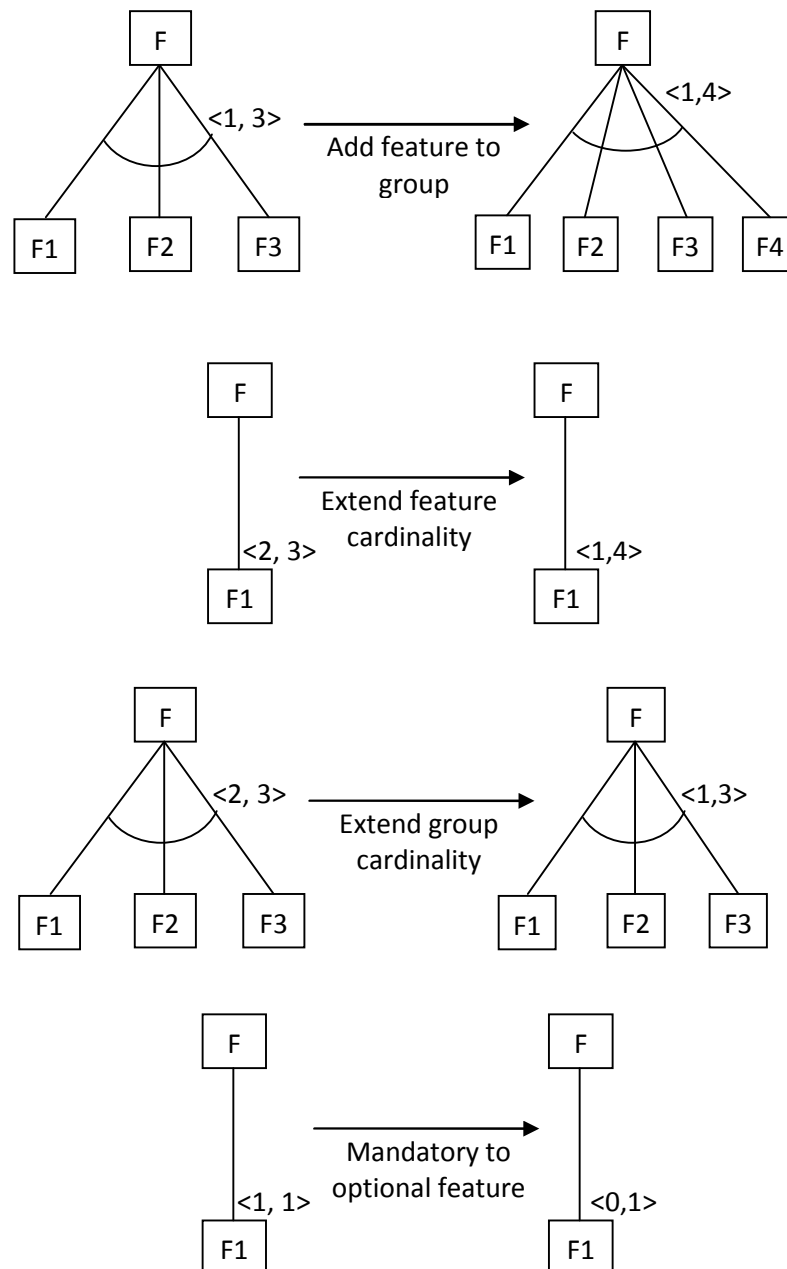


Figure 5. 2: Expansion operations on FMs.

## 5.4. Partial derivation of the Architecture Model

### 5.4.1. Restricting the configuration choices of architecture Model

#### 5.4.1.1. Restricting a Choice VP

For the architecture model we distinguish between:

1. Restricting a choice component by removing one or several implementation possibilities. A special case of this operation is when no implementation choice is left. As a result, a choice component with no implementation is completely removed from the architecture.

2. Restricting a choice connector by excluding a component or a set of components from the choices related to the connector. If the components group size is  $s$  and its cardinality is  $[n, m]$  such as  $n \leq s$ , when removing one grouped component the new components group size will be  $s - 1$  and its new cardinality interval will be  $[n, \min(m, s-1)]$  where  $\min(a, a')$  takes the minimum of the two natural numbers  $a$  and  $a'$ . Special cases occur when it remains a single component from the components choices set or when no component remains. If no component choice remains then the connector is no longer useful and it must be removed from the architecture. If the connector is related to a single component and the cardinality interval is  $[1, 1]$  the choice connector is replaced by a mandatory connector. In the case of  $[0, 1]$  interval, the relation choice connector is changed into optional connector.
3. Restricting a choice connector by reducing the choices number described by the interval. A choice connector with cardinality  $[n, m]$  may be reduced to  $[n', m']$  where  $n' \geq n$  and  $m' \leq m$ . Special cases occur when getting  $[1, 1]$  or  $[0, 0]$  intervals. If we obtain  $[1, 1]$  interval and the connector is related to more than one component, the connector type is called Alternative. If we obtain  $[0, 0]$  interval, the connector is completely omitted whatever is the number of components it is related to.

#### 5.4.1.2. Changing an architecture element variability type from Optional to Mandatory

In the architecture model, an optional component may be changed to mandatory type if its existence is obligatory in the final reusing applications. This is also valid to both of optional interfaces and connectors.

#### 5.4.1.3. Removing an architecture element

In architecture model we may: omit a component with all its interfaces and connections, omit a particular interface from a component, or omit a connection between two components. Omitting a component from a component-group and omitting a set of components cases correspond to what is described section 4.1.1.

## 5.4.2. Expanding the configuration choices of architecture Model

### 5.4.2.1. Extending a Choice VP

For the architecture model we can distinguish:

1. Adding a new implementation (or a set of implementations) to the implementations group of a choice component. A component with a single implementation may turn into a choice component if new implementations are introduced;
2. Adding a component or more to the components choice group related to a choice connector;
3. Extending the options number interval described by a choice connector. The choice connector cardinality interval  $[n, m]$  may be extended to  $[n', m']$  where  $n' \leq n$  and  $m' \geq m$  and consistency is preserved ( $n' \geq 0$  and  $m' \leq s$ ). A simple connector may change into choice connector if it must be related to more than one component.

### 5.4.2.2. Changing an architecture element Variability Type from Mandatory to Optional

A mandatory component may become optional and this results in extending the configuration possibilities of the model. This is valid also for mandatory interfaces and connectors. The IASA extension architecture style allows dealing with each variable architecture element separately which allows more complete and explicit variability representation.

### 5.4.2.3. Adding a Functionality:

An architecture model can be then, expanded by adding a new component or a component set. Related interfaces and connectors are changed or added accordingly. Moreover, new interfaces and connectors may also be added to the model if needed. The cases of adding new component implementations or extending connector cardinality are described by extending a choice VP.

## 5.5. SPLs merging

After been partially derived, the reusable SPLs (ASPLs) are merged with their reusing SPLs (see Chapter 4-Section 4.2.3). In this section we present in more detail the integration of the partially derived FMs with the FMs of the reusing SPLs



in addition to the composition of an SPL from the set of ASPLs and their reusing MPL sub-SPL.

#### 5.5.1. Feature Models merging

The reference FM of the SPL intending reuse (MPL sub-SPL) is considered as its *Composition Model*. This FM includes some features annotated as «*reuse-spots*» which means that those feature are not considered during the development of the current sub-SPL because they have been already produced by the specialized ASPLs. Therefore, at this stage the reused ASPLs' FMs are partially derived according to the reusing sub-SPL requirements and they have to be merged with its reference FM. The FMs merging may be done in two ways:

1. In the case of having a reference FM with a manageable size (not very broad or complex), we replace each feature annotated as «*reuse-spot*» by its corresponding feature tree from the partially-derived ASPL, such as the root of the feature tree represents the reuse-spot feature. Figure 5.3 depicts the steps of this merging way.
2. In the case of having a broad reference FM which risk being more complex when merged with the ASPLs FMs. We use the *reference feature* property to refer to each reused ASPL's tree. Consequently, we bind the ASPL's FM with the sub-SPL FM without real integration. Features annotated as «*reuse-spot*» change into *reference features*. Figure 5.3 illustrates this operation.

In the two merging ways, constraints of the ASPLs FMs must be checked against those of the reference FM.

#### 5.5.2. Architecture models merging

In the sub-SPL, the set of components implemented by ASPLs are represented by black boxes that will be replaced by partially-derived ASPLs thereafter. Those black boxes are annotated in the reference architecture model by «*reuse-spots*». As stated before ( Chapter 4- Section 4.2.3) the reference architecture model of each sub-SPL represents its *composition model*. During the partial derivation step, the reuse-spot components are extracted from the ASPLs according to the reusing sub-SPL requirements. Only needed interfaces and sub-components are kept. After that, at composition step, the reuse-spot components are replaced by the partially derived ASPLs components and connections are performed to link the

ASPLs components with the reusing SPL reference architecture components. The reuse-spot components may represent either *aspect components* or *business components* to the reference architecture. However, this does not influence the composition operation.

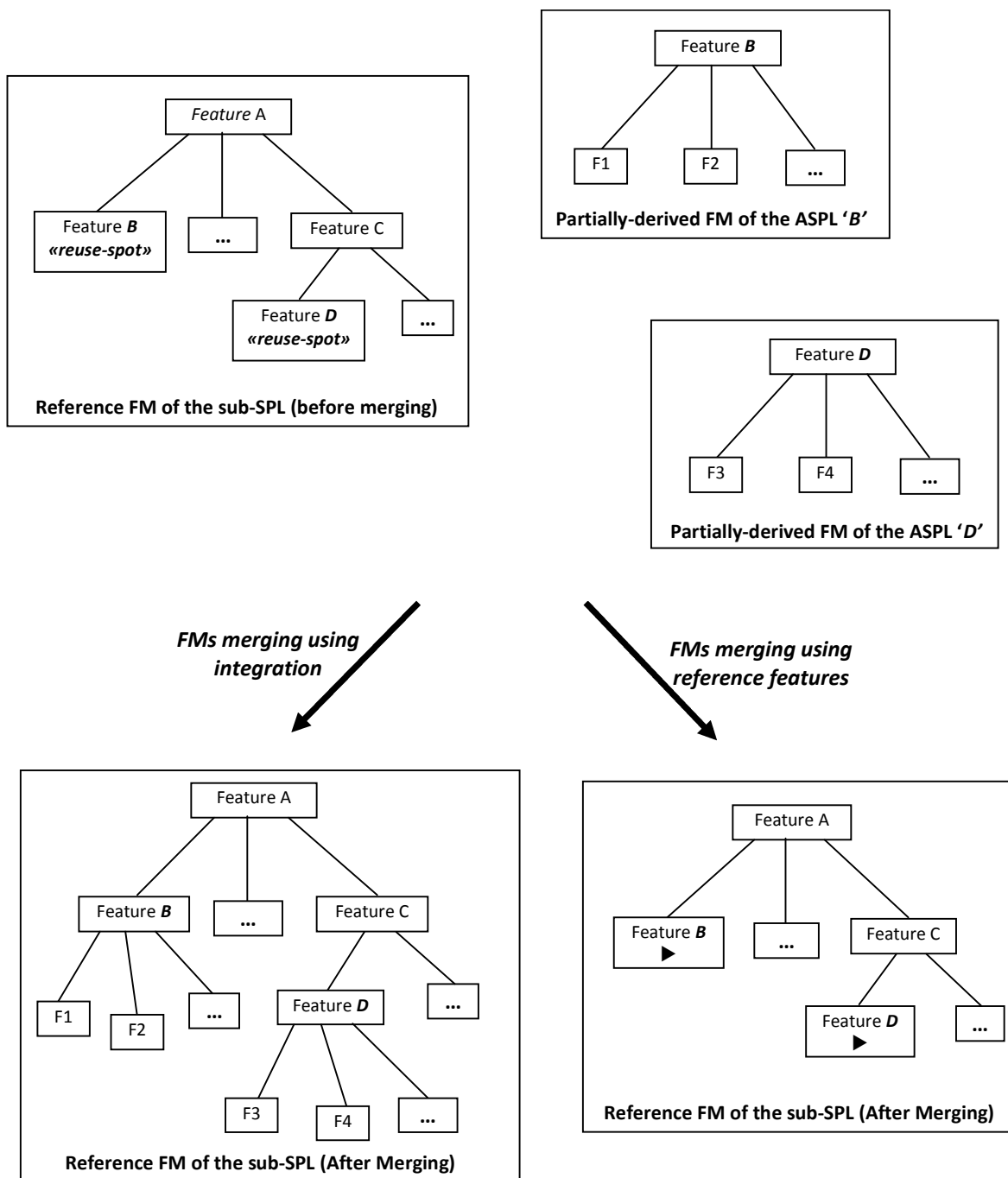


Figure 5. 3: Feature models merging

For instance, in the e-Gov MPL, security and GUI ASPLs components will take the place of aspect components in the reusing sub-SPL, while e-Meeting ASPL component will represent a business component for the sub-SPL. The integration is performed by considering each ASPL's architecture model as a refinement of its

corresponding reuse-spot component in the sub-SPL reference architecture. Constraints on architecture must also be checked in order to avoid inconsistencies.

## 5.6. Case study:

### 5.6.1. The e-Evaluation ASPL

A functionality that is usually needed in e-Learning applications is the evaluation. Evaluation aims to estimate the learner comprehension of the provided online courses. Schools and universities can use it to help students reaching better courses understanding or to perform online exams in special classes, and by the way getting a faster way to assess students' answers when using automatic scoring functionality. Enterprises and corporations may use evaluation tests to help trainees appreciating their understanding of online training.

An evaluation ASPL allows producing a variety of evaluation components intended for the various e-Learning sub-fields. It includes several questions types ranging from basic simple questions (intended for example to primary schools), to more advanced questions targeted –for example- to specialized e-Learning applications such as: languages, architecture, computer science or mathematics. Moreover, evaluation components may comprise other functionalities such as: homework, score estimation, production of report cards and certificates that vary from an institution to another. Figure 5.4 depicts the evaluation ASPL feature model.

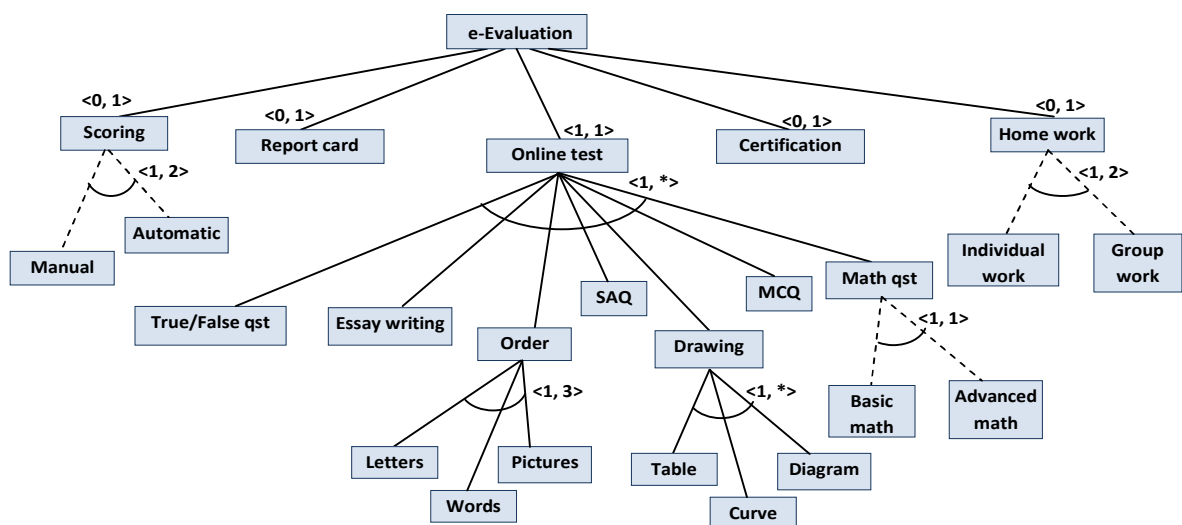


Figure 5. 4: The evaluation ASPL FM.

The reference architecture of the evaluation ASPL is presented by Figure 5.5 . The evaluation component provides at least one obligatory interface which is test interface that allows handling online tests functionality. It may provide other optional interfaces for homework, report cards and certification functionalities.

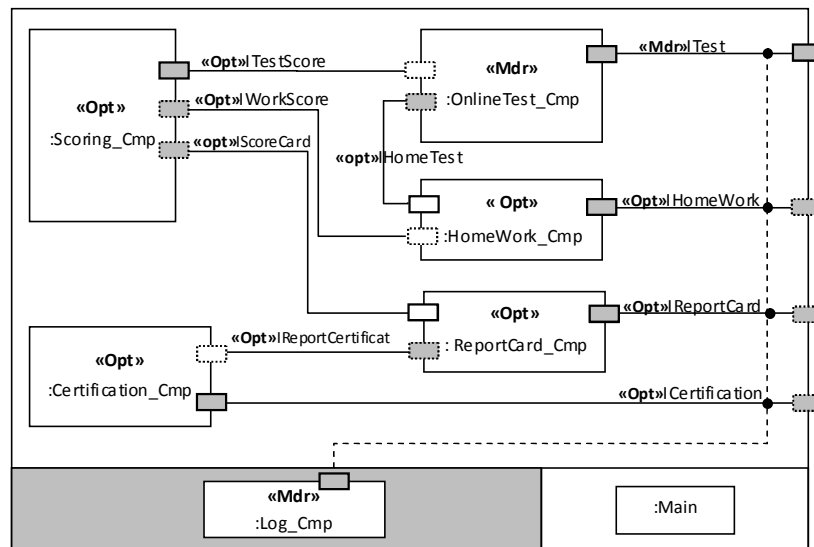


Figure 5. 5: The reference architecture of the evaluation ASPL.

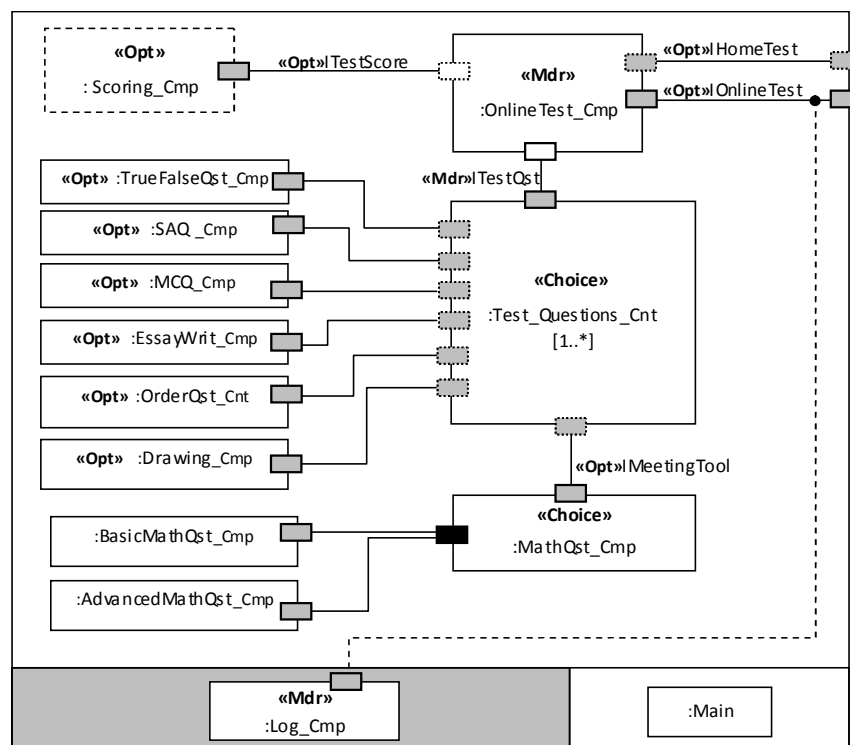


Figure 5. 6: The internal structure of the component 'test'.

Figure 5.6 shows the internal structure of the component 'test'. An OnlineTest\_Cmp instance may include one or more questions of various kinds. The question components themselves may have various implementations

according to the context as in the case of mathQst\_Cmp. The model presents a set of questions components, more questions kinds can be introduced, as we can go in more detail for each question type. For example, drawing tools may provide curves and tables tools for mathematic, modeling tools for computer science, and graphs for statistics and so on.

The partial derivation of the evaluation ASPL architecture to be reused by the e-Primary SPL results in the same reference architecture as in Figure 5.5. However, the internal structure of components is altered. For instance, the partial derivation of the test component for e-Primary SPL results in the model reported by the Figure 5.7. E-Primary applications usually need some basic questions such as: Short Answer Questions (SAQ) and Order questions, therefore the corresponding components takes mandatory type instead of optional. Only basic mathematic questions are required then the MathQst\_cmp component is replaced by BasicMathQst\_cmp component. Furthermore, new components can be added to the application such as: match the items, fill with the correct word, conjugation questions and others. Differently, if we derive partially the evaluation SPL to be reused in e-Coaching SPL the HomeWork\_Cmp could be omitted.

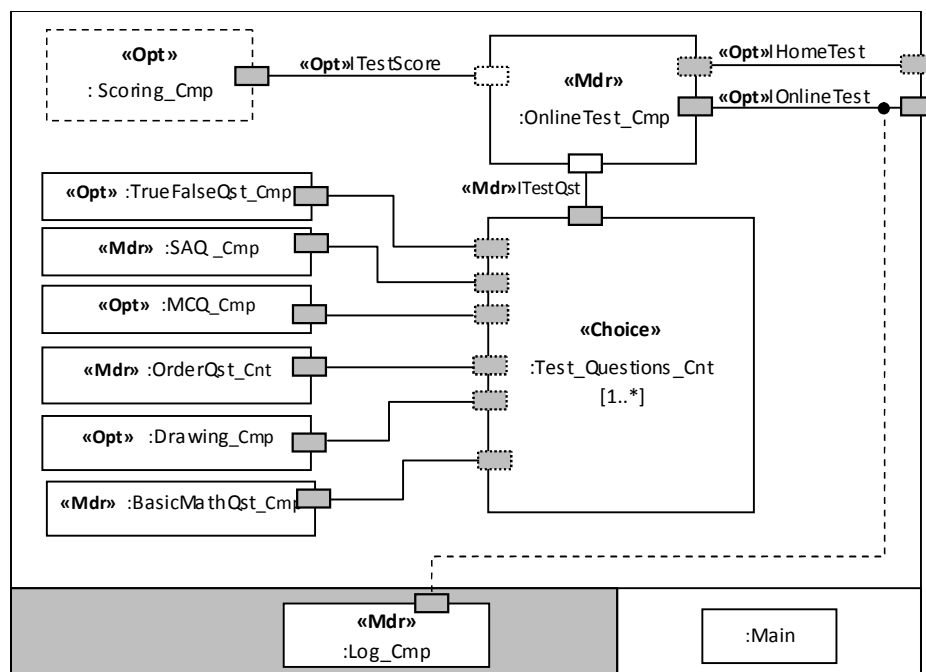


Figure 5.7: Partially derived test component for e-Primary SPL.

In the case of partially deriving the evaluation SPL for an e-Math SPL which produces specialized applications in the provision of mathematic courses, some

features should be omitted and the resulted partially derived FM is demonstrated by Figure 5.8. for the reference architecture: the resulted partially derived SPL do not need `essayWrite_cmp` and `order_cmp` components, yet it requires choosing `AdvancedMathQst_cmp` implementation for `MathQst_cmp`. The partially derived test component for e-Math SPL is shown by the Figure 5.9.

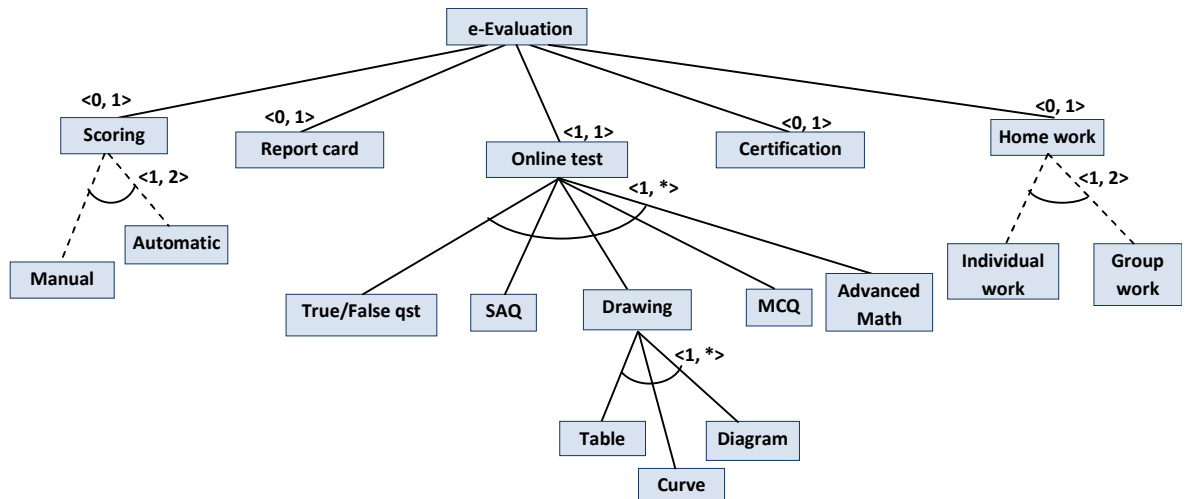


Figure 5. 8: Partially derived evaluation ASPL FM for e-Math SPL.

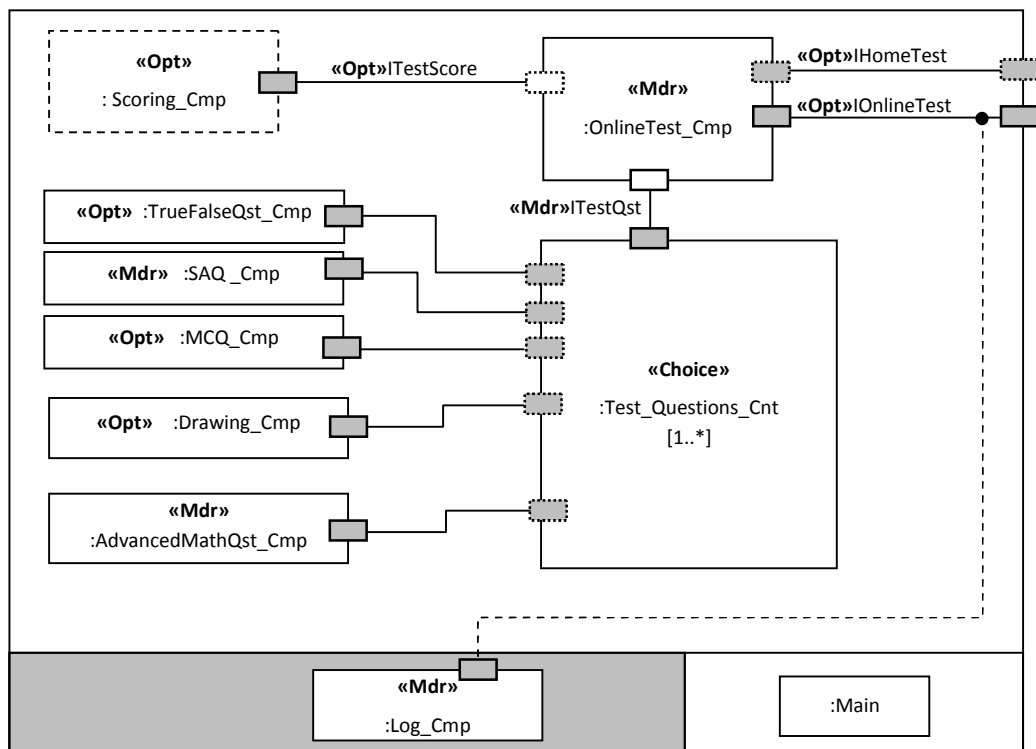


Figure 5. 9: Partially derived test component for e-Math SPL.

### 5.6.2. The composition model

We consider for this case the e-University sub-SPL from the set of e-Gov AMPL sub-SPLs. The composition model of e-University SPL with its ASPLs corresponds to its reference architecture as shown in the Figure 5.10.

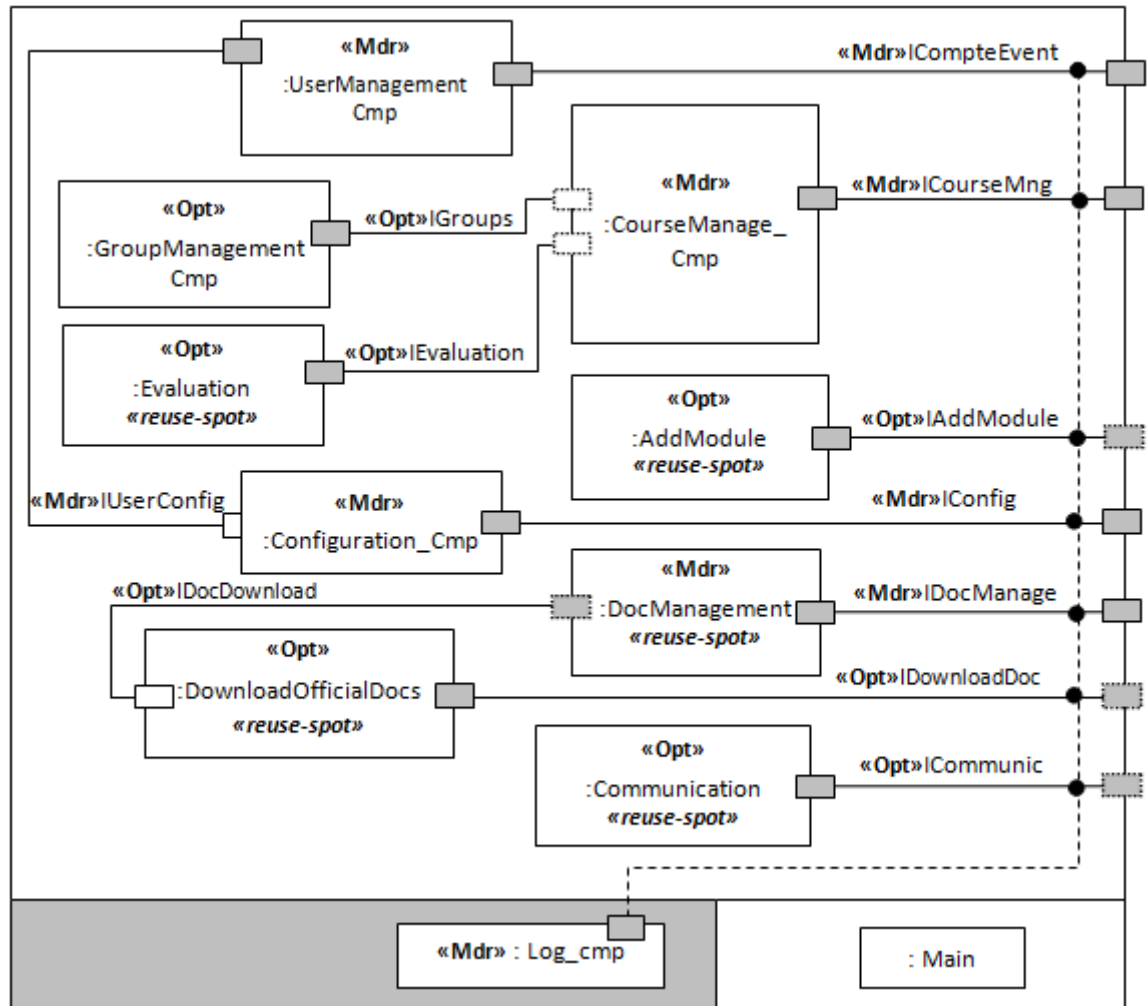


Figure 5. 10: The reference architecture of e-University SPL.

All of DownloadOfficialDocs, AddModule, Doc Management, Communication and Evaluation components are annotated by **«reuse spot»** and will be replaced by the relative partially derived ASPLs respectively Official Documents, Additional Module, Documents Management, Communication and Evaluation ASPLs. Official Documents SPL provides functionalities for documents authentication, download, archiving and so on. Additional modules encompasses SPLs producing components that do not represent the core of e-Learning applications but that can be added to those applications when needed, such as: research, poll, statistics SPLs. Communication SPL should provide e-Learning institutions by

communication components that fit their different needs such as: supporting several data formats, communication protocols, and basically to provide efficient security means. Those ASPLs are partially derived according to the reusing SPL requirements (in this case e-University SPL) and are composed with the other SPL components according to the reference architecture.

### 5.7. Conclusion

In this chapter we have presented in more detail two crucial activities of AMPL engineering: ASPLs partial derivation and integration. The partial derivation helps us to avoid delaying the SPLs composition until getting the application level, where we are more likely to have incompatible instances derived from separated SPLs. The early integration of partially derived SPLs avoids this problem, and the resulted composed SPLs will be derived as ordinary SPLs. In contrast to reusing instances, the SPLs partial derivation provides better means for reusing SPLs in a wider way. We have then illustrated these activities in the context of e-Gov AMPL.



# CHAPTER 6

## EVALUATION OF THE APPROACH: E-GOVERNMENT CASE STUDY

### 6.1. Introduction

Providing customers with efficient services is today an interest of any organization. Since Government's services are the most required by citizens, their availability and effectiveness became necessary. Government today use internet to improve its services, fulfill the requirements of their citizens, and gain in terms of time, effort and cost. Over time, citizens' requirements increase, and instead of developing one application, Government has to develop several applications for each sub-domain it includes. These applications must satisfy their users, and communicate to produce better services as well. However, developing and maintaining new software are cost; time and effort consuming. It becomes then necessary to find an efficient solution that allows the fast development of systems and overcomes the before-mentioned issues.

After the success achieved in the industry field, product line engineering has known recently a great attention in software development. SPLE aims to improve productivity and software quality, by maximizing reuse and managing variability in all software development stages. SPLE promotes large scale reuse within the scope of a particular field, which helps significantly to reduce time to market as well as cost and effort of development. Therefore, we strongly believe that adopting software product line (SPL) approach to develop e-Government applications can bring important benefits to this domain.

E-Government provides a variety of services intended to satisfy customers' requirements in several fields. Nevertheless, all of those services are characterized by several common aspects. Using SPL approach allows us to identify these aspects, and making them flexible to be reused in different contexts. In addition, applications derived from the same product line will have similar man-machine interfaces; this will allow citizens and users to be familiar with these applications, to easily switch between applications from the same product line, and encourages them to a wider use.

As e-Government is regarded as an evolutionary phenomenon [101], SPLE can affect positively the e-Government evolution in terms of decreasing time to market and promoting interoperability. Moreover, e-Government systems are delivered in a variety of versions and will be installed in several sites. Developing and installing such systems are time consuming and installed systems risk of being obsolete since technology is in continuous evolution [102]. E-Government software engineering must encompass activities that handle the extension and evolution of the systems it includes according to new requirements. However, separate evolution of applications will produce the inconsistency between them, which makes their integration difficult. The proper implementation of e-Government SPLs will allow the efficient and easier management and monitoring of all systems (its members) and as result keeping them up-to-date.

SPLE have brought several advantages to e-Gov, however if we look to the whole e-Gov domain we find that it is composed of a variety of subfields for each of them an SPL can be introduced for the fast production of applications. This result in an e-Gov MPL that includes several separated but still interdependent SPLs. This MPL must be effectively managed in order to reduce complexity and reach better production quality and time to market. In this chapter, we review the e-Gov domain, we reports on its main challenges and we suggest the use of our approach proposed previously in this thesis to solve the problems encountered in this domain.

## 6.2. Background and Motivation

### 6.2.1. E-Government and SPLE

E-Government is defined by the European Community (2004) as “*the use of (ICT) Information and Communication Technologies in public administrations combined with organizational change and new skills in order to improve public services and democratic processes*”. Clearly, e-Government is not only to bring the benefits of existing service on the Internet. It is not the traditional government to which we have added the Internet but a process of radical change in the way the state works and communicates. E-Government provides citizens, enterprises and governments as well by exciting benefits, namely: improving the quality and availability of public services; improve information, communication, and cooperation between the different actors; reduce administrative costs; allow citizens to a better participation in different kinds of democratic processes; and so on.

E-Government includes several sub-fields: eServices supplied by public administrations such as: vital records, passports, identity card, voting, poll, justice, etc; in addition to other services that could be provided by private or public institutions such as: education, health, assurance, transport, retirement, services for disabled people and so on. Applications from each sub-field are characterized by a set of common features and can be distinguished by some variable aspects. This perception leads developers to apply SPLE for each e-Government subfield to benefit from its significant advantages. Using SPL approach allows us to identify these aspects, and making them flexible to be reused in different contexts. Consequently, cost, time and effort of development are decreased thanks to the large scale reuse and services suppliers take advantage from the decreased time to market and the higher quality of software. Moreover, benefits affect not only software developers and services suppliers but also citizens as well. Applications derived from the same SPL provide similar Graphical User Interfaces (GUIs); this allows citizens and users to be familiar with these applications, to easily switch between applications from the same product line, and encourages them to a wider use. As e-Government is regarded as an evolutionary phenomenon, getting maturity in this field requires much of work and time. Hereafter, developing e-Government applications according to a SPL approach will allow faster evolution

of e-Government, since software will be produced in lower time; and performed with the goal to be flexible, and easy to maintain. At an advanced e-Government maturity level, applications need to interact in order to provide better services to citizens. SPLE can also affect positively this aspect because applications derived from the same SPL are compatible and more likely to communicate easily.

Seeking for the aforementioned benefits, SPL approach has been adopted in various e- Government subfields. Thus, Over time, e-Government has taken the form of a set of sub-SPLs each one intended to a subfield as shown in figure 6.2-a.

### 6.2.2. Crosscutting reuse among e-Gov SPLs

If we compare the features of e-Government sub SPLs (figure 2), we find that an important set of features is common to all of them. Thus, reuse can go beyond the scope of a sub SPL. A large set of core assets from one sub-SPL can be reused in other e-Government Sub-field to develop a new SPL (similarity between two different e-Government sub SPLs is illustrated in section 4). But, *from where comes this similarity?*

The major factor that raises similarity between e-Gov sub fields is the fact that they are intended mainly to citizens. Citizen represents the central point of all e-Government services. As shown in Figure 5.1, a citizen usually has a profile, can be a member of an organization and accesses services to satisfy its needs. Organizations provide services which involves citizens to fulfill a particular role. Citizen is the main consumer of e-Gov services as a person or a member of an organization. This property (customer orientation) can be seen as an opportunity and an advantage at the same time. An opportunity because it increases similarity between e-Government systems, so it allows the implementation of an e-Government MPL, and an advantage because it is considered as the major success factor for e-Government adoption [103].

E-Government services are designed to fulfill daily citizens needs in the different aspects of life (birth, education, wedding, health, employment ...). Whatever is the application's domain there is some key features related to citizens that are common to all of those applications, namely: authentication, user management, privacy, protection of personal data, collaboration...

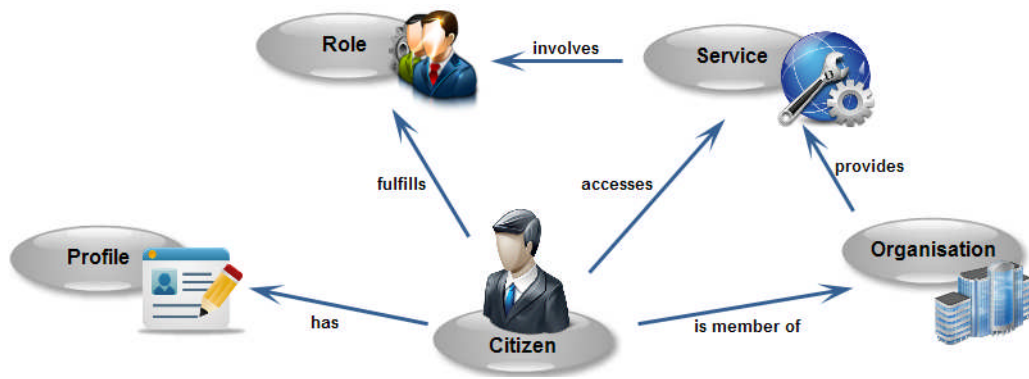


Figure 6.1: Citizen as the central entity of e-Government.

Since e-Government applications handle personal data, communicate, and transact with users they must be highly protected. Thus, security functionalities represent another common point. E-Government applications need to communicate, to share and exchange data in order to provide efficient services to citizens. Communication is also a major feature that must be considered when developing e-Government applications. Furthermore, there are other characteristics that may be included in any e-Government application, such as: e-Meeting, statistics, poll, research, and so on.

This description of the e-Gov domain match perfectly to the approach proposed in this thesis. In the next section we present the development process of an e-Gov AMPL and we discuss the results obtained compared with existing work.

### 6.3. E-Government AMPL Engineering

#### 6.3.1. E-Gov ASPLs engineering

##### 6.3.1.1. MPL requirements engineering

As stated before, e-Government MPL may include several subfields such as: e-Education, e-Justice, e-APC, e-Daira, e-Wilaya, e-Recruitment, e-Health... (See figure 2-a). For the validation of our work we have chosen two e-Gov sub-fields which are: e-Admin and e-Education. Those two subfields include in fact the following SPLs: e-Primary SPL, e-secondary SPL, e-University SPL which represent specialized SPLs of e-Education SPL, and e-APC SPL, e-Daira SPL, e-Wilaya SPL that are specialized SPLs of e-Admin SPL. For the e-Education case we consider one SPL that includes the three specialized SPLs, while for e-Admin we focus on the services provided by APCs in our models. Consequently, the

scope of our AMPL includes six SPLs: e-Primary, e-secondary, e-University, e-APC, e-Daira and e-Wilaya SPLs. More SPLs can be added to the MPL in the future since the separation between MPL sub-SPLs and ASPLs ensure the scalability of the approach (Chapter 4- Section 5).

### 6.3.1.2. Separation of concerns

The analysis of the e-Gov domain in order to determine its MPL aspects resulted in a set of commonalities (as stated in the Section 2.2.). E-Government services are designed to fulfill the daily needs of citizens in the different aspects of life (birth, education, wedding, health, employment ...). Whatever is the application's domain there is some key features related to citizens that are common to all of them, namely: authentication, user management, privacy, protection of personal data, collaboration, security, communication e-Meeting, statistics, poll, research, publishing documents and so on. The separation of concerns at MPL level results in a set of ASPLs: GUI SPL, Security SPL, e-Meeting SPL, Communication SPL, in addition to a set of sub-SPLs (in our MPL this correspond to the subfields chosen in the previous step): e-Primary, e-secondary, e-University, e-APC, e-Daira and e-Wilaya sub-SPLs. Figure 2 illustrate the organization of the e-Gov MPL before and after applying our approach.

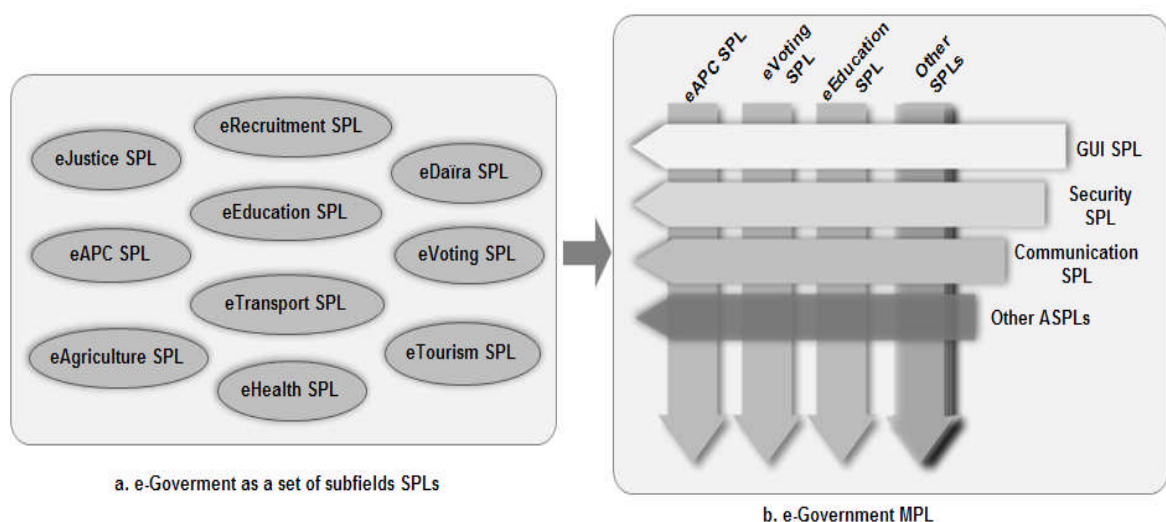


Figure 6. 2: E-Government before and after applying AMPL approach.

Some ASPLs that can be constructed in the context of e-Gov MPL are shown in this section.

### - **GUI ASPL**

GUI has an important impact on e-Government applications. Huang et al [104] assert that user-friendliness is required for e-Government to improve users' performance as well as their satisfaction with e-government. Developing an ASPL for such purpose will bring e-Government applications by similar GUIs, and thus allow citizens and users to be familiar with these applications. Users - either citizens or government agents- can easily switch between applications with GUIs coming from the same SPL, this can encourage them to wider use. Devoting SPLs for GUI development have known increased attention recently. For instance, Müller [105] propose a research plan to automate the GUIs construction for SPLs based on constraint-based techniques. In his paper he highlights the benefits of having an automatic GUI generator for software developers, and presents a survey on GUI generation for SPL.

### - **Security ASPL**

As personal data are processed and stored, and financial transactions must be performed, considering security when developing e-Government applications is a crucial issue. Citizens are afraid from using web applications that handle their personal data (such as name, date of birth, picture, ID number, and credit card details...). Using online services may expose their personal information to be misused or even destroyed. Hence, e-Government applications must be highly secured to ensure services continuity and to build citizen confidence.

Due to the complexity and the extensive nature of SPLs, developers usually focus on the business requirements of the field, and pay less attention to technical issues such as security. However, such a vital requirement must not be ignored in a critical domain as e-Government. Mellado et al have tackled this problem [106] [107] [108] and have proposed an approach to manage security requirements in SPL development. The proposed approach called Security Requirements Engineering Process for Software Product Lines (SREPPLine) aims to deal with the security requirements artefacts variability from the early stages of SPL development in a systematic way. The approach is supported by a tool (SREPPLineTool) which provides automated support to facilitate the application of security quality requirements engineering process for SPLs.

Devoting a SPL process to analyze security requirements in e-Government field and provide developers by a repository of security assets intended to be reused in several e-Government subfields, will allow the suitable and robust realization of security aspects, and thereby will increase e-Government applications' quality.

- ***Communication ASPL***

A natural progression of e-Government will be the integration of scattered systems at different levels (vertical) and different functions (horizontal) of Government services [101]. Governmental institutions should collaborate, join-up, and create chains of activities in order to improve services efficiency, enhance transparency, save time and money and reduce sources of errors. For this reason e-Government systems have to be interoperable.

Interoperability refers to the ability of two or more organizations to exchange and interpret all necessary information to collaborate. In order for organizations to be interoperable their strategies must cater for interoperation between business processes as well as ICT systems [109]. Interoperability can be seen at different levels according to the e-Government development stages. Marc N. et al [110] present three levels based on the interoperability goals: Technical interoperability maps to the goal of data exchange, Semantic interoperability maps to the goal of meaning exchange and Organizational interoperability maps to the goal of process agreement. Each abstraction level requires that the lower abstraction levels are accomplished. Janssen and al [111] discuss challenges in each level and argue that interoperability is one of the most critical issues facing government that need to access information from multiple information systems.

Adopting SPL approach to develop e-Government applications will have a positive impact on this aspect. Applications derived from the same product line are more compatible and more likely to communicate easily. Nevertheless, if systems are not planned to communicate it becomes necessary to enhance them by adding components responsible for communication. Creating a communication ASPL will resolve this issue. The communication ASPL should provide e-Government institutions by communication components which fits their different needs such as: to support several data formats, communication protocols, and basically to provide efficient security means. The appropriate implementation of a communication



product line will allow better interoperability in e-Government domain, faster achievement of advanced maturity levels (vertical and horizontal integration), and will improve public satisfaction by offering one-stop services.

- ***Statistical services ASPL***

Official statistics provide essential information for government, economy and public. Statistical services systems aim to ease the collection, study and publishing of statistical data. They can be provided by governmental agencies at all levels (local and national) and other public bodies, and they may intend all major areas of citizens' lives, such as: economic and social development, living conditions, health, education, and environment. It is obvious that those systems are characterized by several common feature (using same algorithms, providing similar models of data...) and differ by others (domains they intend, platforms they will be integrated in...). Thereby, developing an ASPL for statistical services will bring significant advantages when integrated within the e-Government MPL. For instance, a family for data-mining applications has been demonstrated in [112]. E-Learning Web miner product line [112] provides statistical services using data mining algorithms. Applications derived from this SPL can be integrated in several types of e-Learning platforms, and aim to assist instructors involved in virtual education by extracting and providing useful information which can be used to improve the learning-teaching process.

- ***E-meeting ASPL***

Meeting is a common requirement for a wide range of e-Government applications. For instance: a deliberation at the year-end, a meeting of a scientific council, a meeting of elected members of APC, APW or APN , medical meetings (for example when dealing with a special patient case), meetings of business leaders... These various meetings have many similarities and are also distinguished by particular aspects. The e-meeting ASPL once implemented is able to produce for each type of meetings the appropriate software in a very short time.

- ***Validating information ASPL***

One of the most important features that must be included in each e-Government application is validating information. Government bodies have to ensure the

delivery of valid and highly reliable information. They have no right for error due to the importance of the documents they deliver. In addition to the fact that correcting such information may require long time and administrative and legal procedures could be very complex. Establishing an infrastructure which enables to certify the information correctness before putting it on an official document will allow: firstly to win the citizens trust, and secondly contributing in the successful e-Government introduction. Validation processes in any e-Government sub-domain are characterized by several similarities:

- applied to an information with a particular type;
- go through stages of validation ranging from submission to the final validation;
- involvement of stakeholders in each stage;
- conditions of information passing (or return) from one stage to another;
- following a validation schema...

Regarding the aforementioned similarities, it will be of benefit to implement an ASPL for the development of validation components. Those components will be integrated in e-Government applications to ensure the correctness and reliability of the provided information.

#### 6.3.1.3. E-Government ASPLs engineering

This first stage includes a set of ASPLs development processes, each one dedicated to produce a SPL that fulfills the requirements of various e-Government subfields in a specific common aspect. Examples on ASPLs have been demonstrated in the previous section. ASPLs development processes are similar to the traditional SPLs development processes, unless there is no application engineering step since their aim is not to develop running applications, but rather to create core assets that will be reused later. Each ASPL development process takes as input the requirements related to the handled aspect, not only of one e-Government subfield but of a set of e-Government subfields that share this aspect. The aim is to fulfill this aspect's requirements for a set of e-Gov sub-SPLs in order to permit crosscutting reuse among them. This first phase may also take as input the feedback obtained when developing final e-Government applications to take into account new requirements, developing them, and making them available to be reused. The outputs of this stage consist in all the developed artefacts resulting

from all ASPLs processes, including: reference requirements, reference architectures, reusable components, and reusable tests artefacts. Those artefacts are collected in a repository that will be reused in the second stage of e-Government MPL engineering.

From the set of ASPLs presented in the previous section we choose the e-Meeting ASPL for presentation in this section and we use it to illustrate next development steps (partial derivation and integration). The e-Meeting SPL has been presented in section 5 from the Chapter 3 so we do not repeat its presentation in this chapter. Noting that it has been design for reuse by the following e-Gov sub-SPLs: e-Primary, e-secondary, e-University, e-APC, e-Daïra and e-Wilaya (i.e. all the SPLs included in the scope of our MPL). The FM we have obtained when developing the e-Meeting ASPL is presented in: Figure 3.5 and Figure 3.6 (chapter 3). A part of the e-Meeting ASPL architecture model is explained in: section 5-Chapter 3, and illustrated by the figures: Figure 3.7 and Figure 3.8 (chapter 3). The notations used for the models specification are explained in the chapter 3.

### 6.3.2. E-Government sub-SPLs engineering

From the scope of our MPL, we choose two sub-SPLs in order to illustrate their specification (FMs and architecture model) and integration with one of the ASPLs they reuse.

#### 6.3.2.1. The first sub-SPL: e-APC SPL

##### **a. Subfield analysis and ASPLs extraction**

The e-APC SPL delivers e-Admin applications for the various APCs institutions (Section 2.1-chapter 4). Among the e-Gov MPL ASPLs, e-APC SPL reuses all of: Download official documents, Data validation, Document management, Communication, e-Meeting, Poll, statistics, Publishing Documents. The FM of this sub-SPL is presented in the Figure 6.3. The reuse spots of the before-mentioned ASPLs are determined in the FM and their sub-features are consequently not considered during this sub-SPL subfield analysis and implementation thereafter. The reuse-spots features in the e-APC SPL are: Download official doc, Data validation, Document management, Communication, e-Meeting, Poll, statistics, Publish Docs.

### b. Sub-SPL design

Figure 6.4 shows the reference architecture of the e-APC SPL. The reuse-spots components are: AddModul\_Cmp, DownloadOfficialDoc\_Cmp, DataValidation\_Cmp, DocManagement\_Cmp, Communication\_Cmp. The refinement and implementation of those components is not performed during this sub-SPL development process since they will be provided by the corresponding ASPLs.

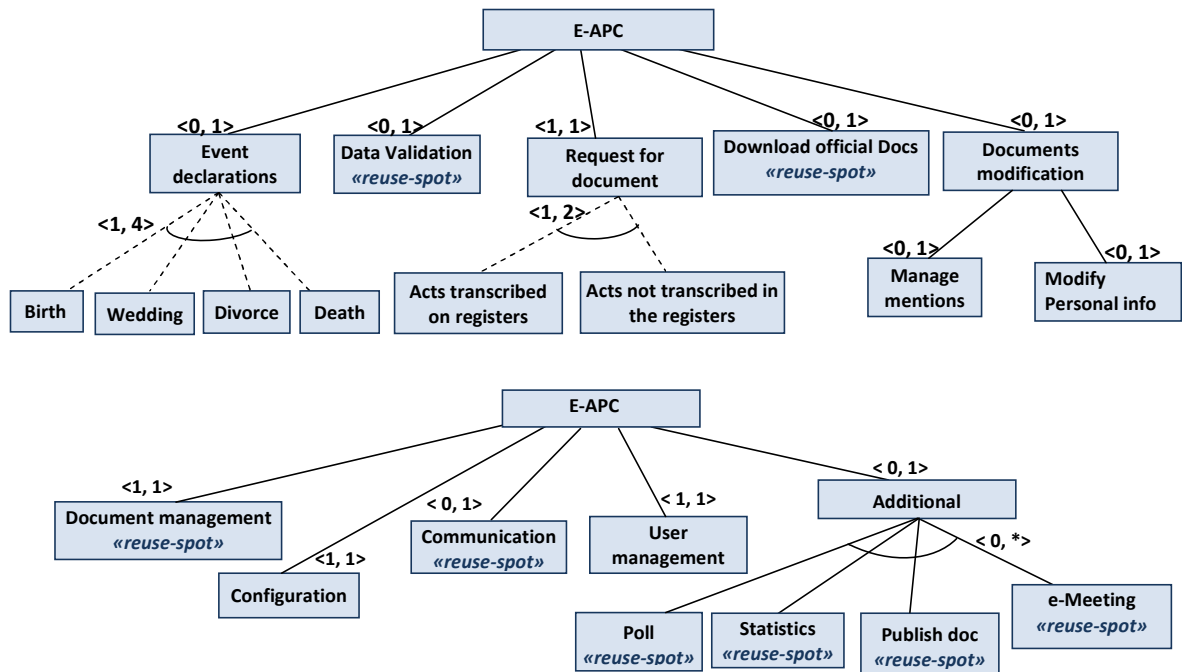


Figure 6. 3: The e-APC sub- SPL FM.

### c. ASPLs Partial derivation

For e-APC sub-SPL, e-Meeting functionality is needed to ensure the smooth running of the elected members meetings. The partial derivation of the e-Meeting ASPL FM according to e-APC sub-SPL requirements results in the diagram shown in the Figure 5. The partial derivation transformations were applied as follow:

- The e-Meetings conducted by the e-APC institutions are mainly decision making meetings, where decisions are taken according to the vote results of the elected members. Consequently, the “vote options” features become mandatory in the e-Meeting ASPL.

- The obtained results from each e-APC meeting are published for democracy reasons, thus the “publication of results” feature change from optional to mandatory type.
- We keep “Item discussion” and “management of recurring items” features because administrations usually discuss items before organizing meetings to take decisions by voting.

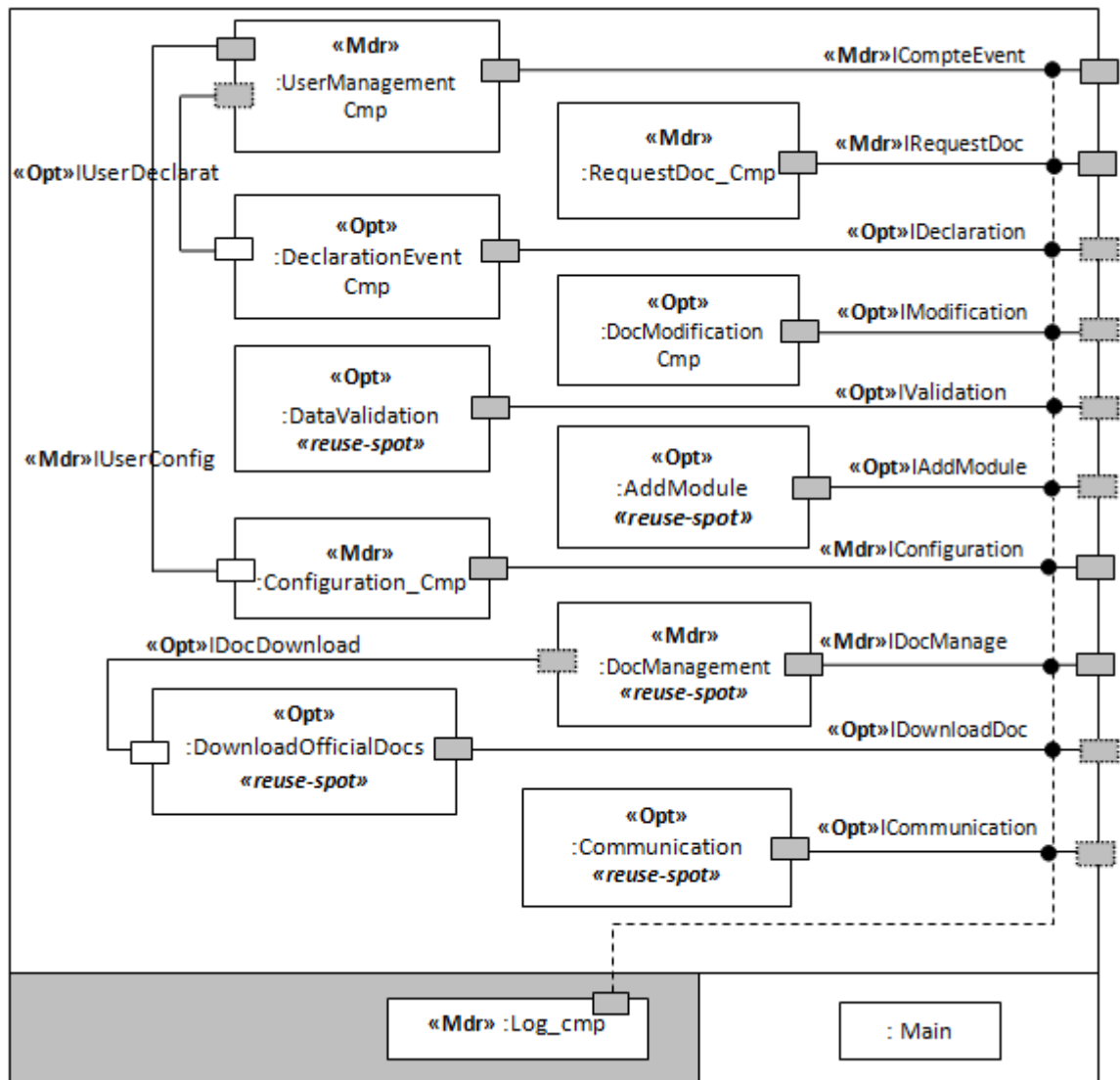


Figure 6. 4: The e-APC sub- SPL reference architecture.

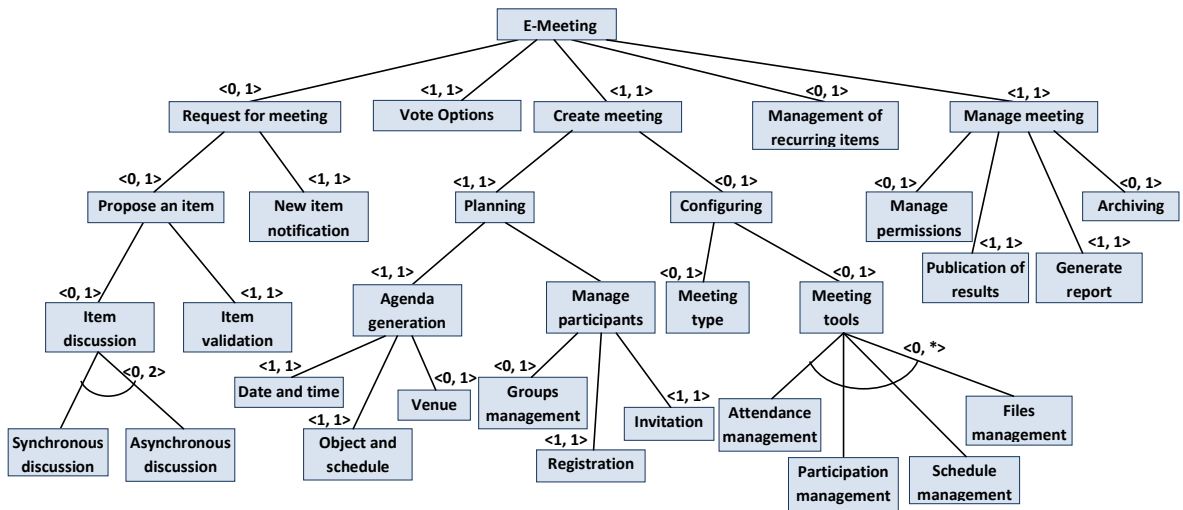


Figure 6. 5: Partially derived e-Meeting FM for e-APC sub-SPL.

The partial derivation of the e-Meeting ASPL reference architecture results in the same architecture model presented by the Figure 3.6 (chapter 3), however changes appears in the refinement of some components. For instance, Publish\_results\_Cmp is a sub-component of the composite component “Meeting\_management\_Cmp”. Publish\_results\_Cmp had the type “optional” in the e-Meeting ASPL, during the partial derivation we change its type to “mandatory” according to the partially derived FM (Figure 6.5). The resulted component refinement is depicted by the Figure 6.6. Noting that the related interfaces and connectors to this component change also to mandatory type.

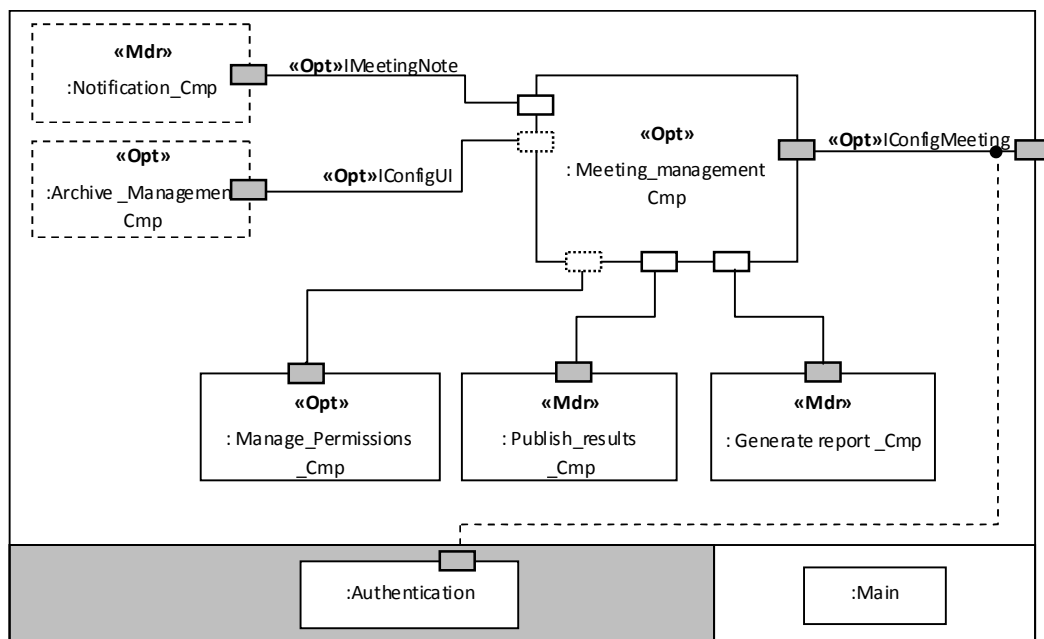


Figure 6. 6: Partially derived “Meeting\_management\_Cmp” for e-APC sub-SPL.

#### d. ASPLs integration

In the integration step, all the reuse-spots are replaced by the corresponding partially derived ASPLs. For instance, the merging of e-Meeting ASPL FM with e-Admin SPL FM is done by replacing the feature e-Meeting annotated as **«reuse-spot»** by the partially derived e-Meeting ASPL FM. This operation results in the FM depicted by Figure 6.7. Related constraints are checked in order to keep the model consistent. For example, if the e-Admin SPL needs to archive the meetings subjects and results, so the feature “Archiving” must be selected in the e-Meeting ASPL.

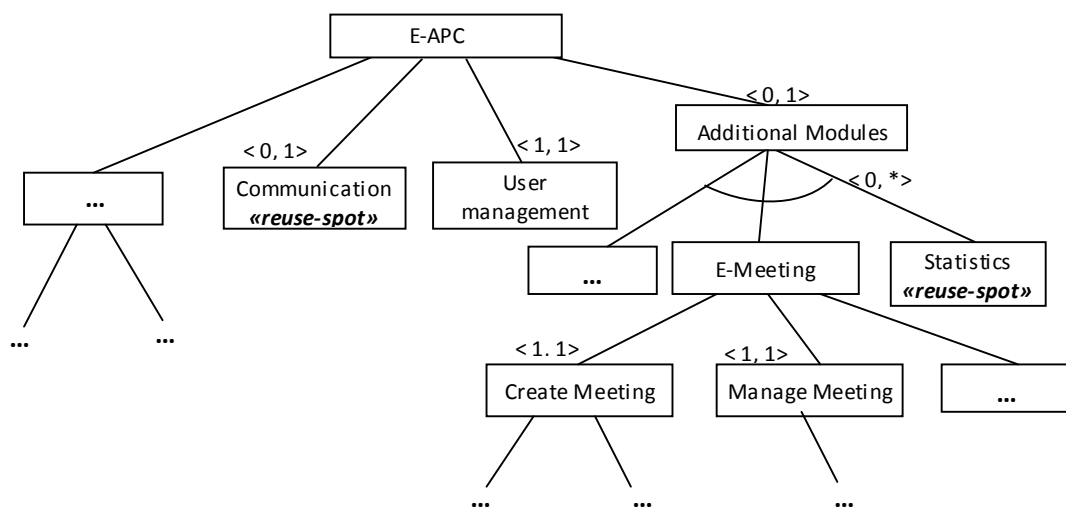


Figure 6. 7: FMs merging of the e-Meeting ASPL and e-APC sub-SPL.

Finally, the presented activities are repeated for all the needed ASPLs (e-Meeting, security, statistics, GUI and communication for the e-Admin SPL case) until getting a complete SPL ready to be derived to produce final e-Admin applications.

#### 6.3.2.2. The second sub-SPL: e- University SPL

##### a. Subfield analysis and ASPLs extraction

The e-University sub-SPL provides e-learning applications for the various university disciplines and institutions. Figure 6.8 represents the FM of this sub-SPL. Features that are considered as reuse spots are: Download official documents, Evaluation, Document management, Communication, e-Meeting, Poll, statistics, Publishing Documents. The Evaluation ASPL has been demonstrated in chapter 6 section 6.1.

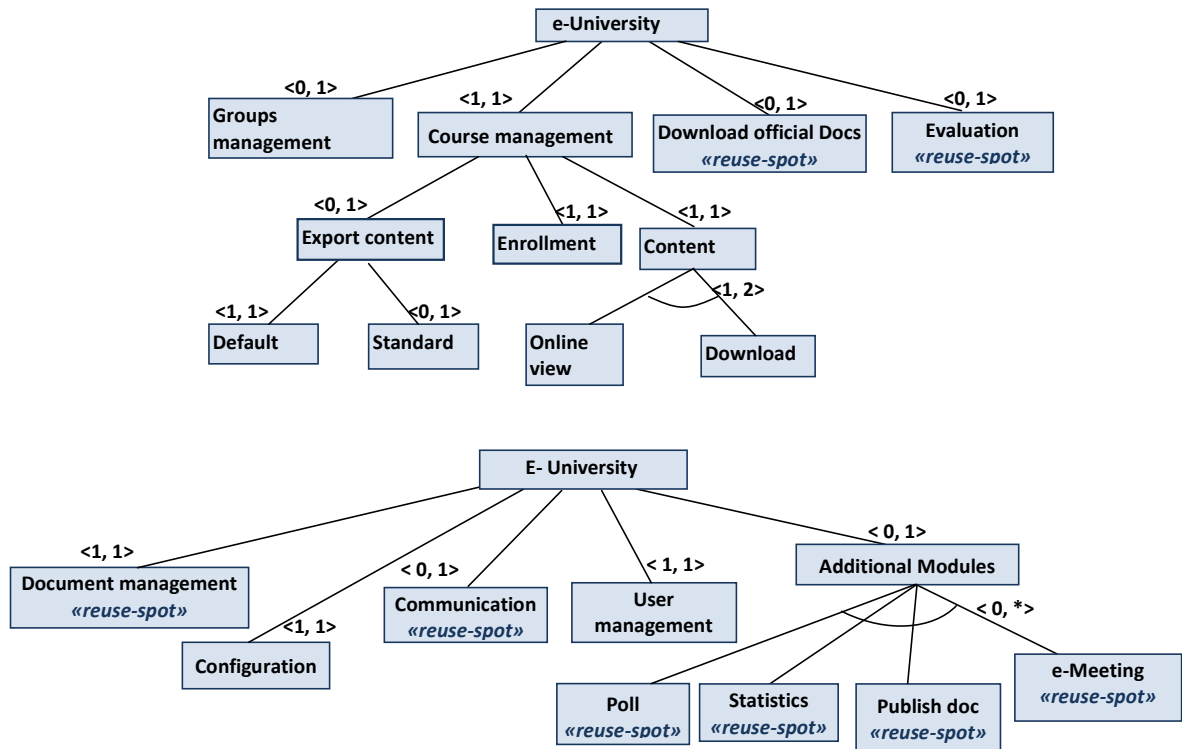


Figure 6. 8: The e-University sub- SPL FM.

### b. Sub-SPL design

The e-University sub-SPL architecture model has been reported in chapter 5 section 6.2 (Figure 5.10) as the composition model of an e-Government sub-SPL.

### c. ASPLs Partial derivation

Figure 6.9 presents the e-Meeting ASPL FM after being partially derived for integration with e-University SPL. We mean by e-Meeting for e-University applications the electronic meetings joining students to their teachers online. Thus the partial derivation decisions are taken according to the e-University requirements defined during the e-University subfield analysis. The partial derivation transformations were applied as follow:

- E-University meetings usually do not need to prepare an item to be discussed; meetings are organized according to an educational planning, therefore, the feature “Request for meeting” has been omitted in addition to all its sub-features.
- “Vote options” feature is used for decision making meetings which is not the case here. So “Vote options” feature is omitted from the e-Meeting ASPL.



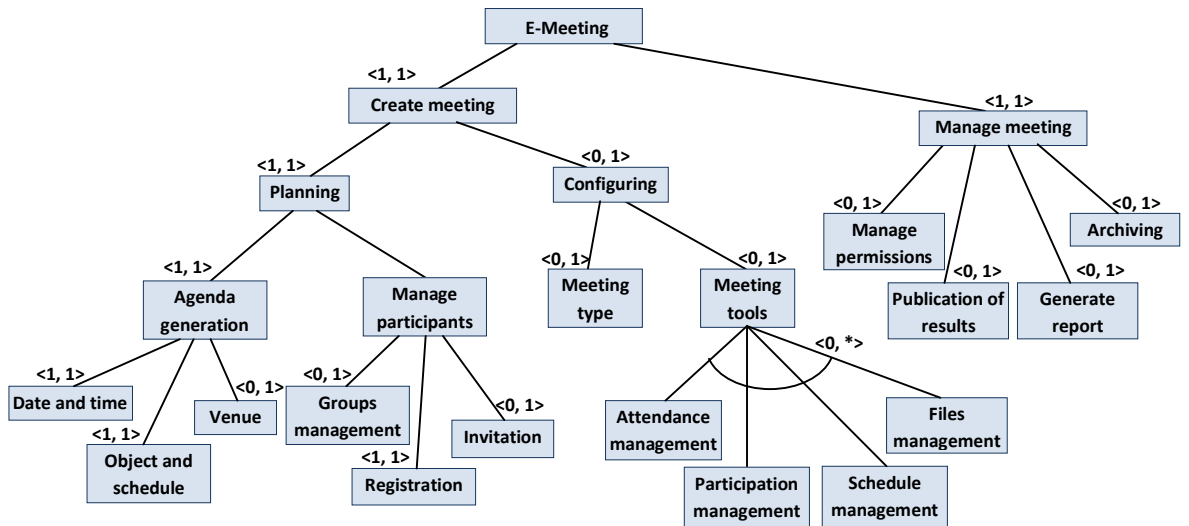


Figure 6. 9: Partially derived e-Meeting FM for e-University sub-SPL.

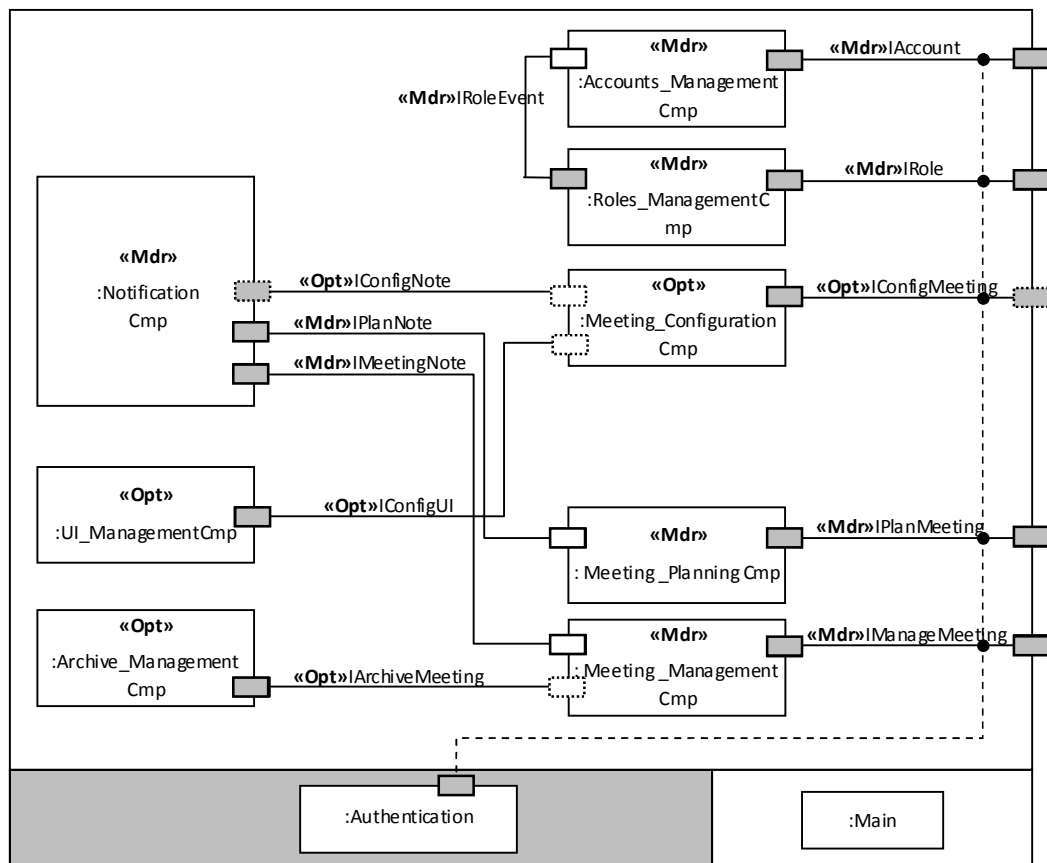


Figure 6. 10: Partially derived e-Meeting reference architecture for e-University sub-SPL.

- “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. This is also used for decision making meetings which is

not the case here. So “Management of recurring items” feature is omitted from the e-Meeting ASPL.

- Meeting reports in this case are not constantly needed; therefore, “Generate report” feature will take the type optional instead of mandatory.
- “Invitation” feature becomes also optional since students do not need to be invited each time they have an online course.

The e-Meeting ASPL architecture is partially derived according to the FM Partially derived e-Meeting FM for e-University sub-SPL. The resulted reference architecture model is reported by the Figure 6.10.

#### d. ASPLs integration

Next figure a part from the e-University sub-SPL FM after been merged with the e-Meeting ASPL FM.

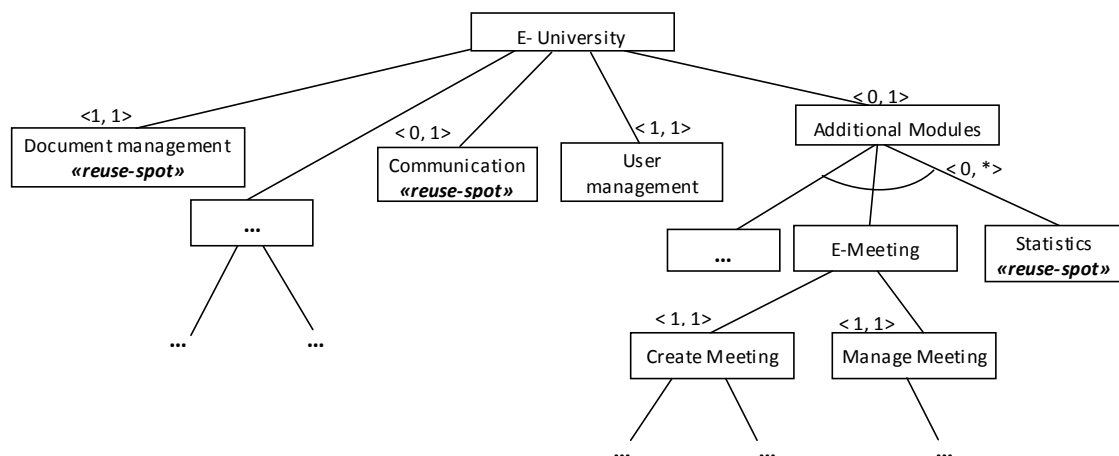


Figure 6. 11: FMs merging of the e-Meeting ASPL and e-University sub-SPL.

#### 6.4. Discussion and evaluation

At the best of our knowledge an MPL for e-Government have not been yet considered. However, SPLE have been adopted but only by few works in spite of the obvious benefits that it can bring to e-Government. In this section we show an overview of existing work in this area and we compare it to ours.

Based on an analysis of a sample set of e-Government architectures [113], Achour et al proposed an architecture model for e-Government applications [114]. The proposed architecture is composed of three layers: the Front-end services layer which represents a portal including all the governmental services, the Back-end services layer that include various workflow applications responsible for the

execution of the services offered by organizations, and the legacy systems layer which represents the various information systems already implanted within the governmental organizations. They focused after that on the backend layer, for which they proposed the adoption of SOPL (Service Oriented Product Line) approach to develop their applications. Their SPL covers some services offered by the Tunisian Ministry of the interior and local development as the demand of National Identity Card (CIN), Passport and Bulletin n°3 (B3). The reference architecture that they presented consists of business components, services and orchestrators. Variability resolution was carried out at runtime, and instantiation of the reference architecture results in workflow applications. Even SOA is a new and promising approach, challenges that it face are numerous especially when combined with SPL [115]. Furthermore, since variability is resolved at runtime the developed systems are more similar to systems developed by Process Family Engineering<sup>1</sup> (PFE) [116] than to SPLs.

Based on their experience in the development of web applications, Carroneu and al [117] built up a process of SPL to develop web systems in the e-Government domain; and create computational tool support which automates this process. The tool called "Titan Framework" base on a SPL repository to automate the SPL derivation process when creating new Web-apps in the e-Government domain. However, Titan framework generates Content Management Systems. Web-apps created using this framework are mainly e-Government portals. The proposed product line intends large scale reuse but do not consider specific e-Government features, mainly horizontal and vertical communication and highly interactive applications.

Buccella and Cechich [118] defined a methodology for creating SPLs for Geographic Information Systems (GIS) applications. Their aim was to benefit from the common set of services between GIS applications. The presented framework was illustrated by a case study that integrates geographic information in two governmental agencies. SPL approach has been used likewise to develop auxiliary eLearning applications [112]. Sanchez and al used SPL engineering to develop eLearning Web-miner product line, a family of data-mining applications

---

<sup>1</sup> The Process Family Engineering (PFE) approach explores the idea of applying SPL philosophy for managing the variability of business information systems. PFE provides only one product that evolves at runtime. [116]

aiming to assist educators involved in virtual education by extracting and providing useful information that these educators can use to improve the learning-teaching process. GIS SPL [118] and eLearning Web-miner product line [112] are SPLs intended for auxiliary e-Government applications. Web-miner SPL aims to develop data-mining applications related to e-Learning platforms, while GIS SPL allows the development of geographic Information applications. Applications derived from both SPLs can be integrated in e-Government applications to supply specific services, so they can be considered as complementary modules (that could be produced by ASPLs in our approach).

Most of the existing work relies on evolutionary approach. Authors base on their experiences in a domain and the realized single applications to extract common and variable aspects which help them to construct a product line. Their major objective is to promote reuse. However, reuse in these works is limited by the related small scope which covers a much closed set of applications or even applications that do not cover the core of e-Government functionalities.

The table below reports a summary of a comparison between our work and existing work according to four main criteria: Methodology, Scope (e-Government as a whole, a subfield of e-Government or a particular aspect related to e-Government), Reuse level (between applications in the same SPL, between separated SPLs), Reuse aspect (business aspect, technical aspects). Regarding the scope of these SPLs, it is either limited to a subfield of e-Government (online administration [114]) or to a particular aspect related to e-Government (Data-mining [112], Geographic information [118]), except one work in which the scope is not limited and may intend e-Government in general [117]. Having the ability to develop applications for several kinds of e-Government services is due to the fact that this work have based on the reuse of technical aspects that may build the base of any web application (Security, persistence, chat, Log, Skins...). In the same way, [112] and [118] have focused on features that do not represent the core of e-Government. These two SPLs can be considered in our approach as specialized SPLs that we can use to implement optional requirements in some e-Government applications. Nevertheless, authors in [114] consider business aspects in their approach. However, their scope does not exceed online administration, more specifically, a set of business processes strongly similar.

Furthermore, they were not addressed extremely important aspects in e-Government, mainly: security, communication, document authentication, data validation...

Unlike of all the existing approaches, ours consider reuse not only between application within the same SPL but also between separated SPLs as well. Reuse at this high level is ensured by the introduction of specialized SPLs. These latter are not only responsible for the development of components that are common between a wide range of e-Government applications, but also they implement key features that helps to build a successful e-Government, such as: HMI, Security, Data validation... Our approach considers in a first stage e-Government as a whole and suggests reuse across all its subfields to benefits from the common aspects between them, and to properly implement those aspects. In the second stage, for each e-Government subfield we devote a SPL process which reuses the core assets obtained during the first stage. In this way we ensure better handling of technical as well as business aspects.

Our approach pays a particular attention to an aspect that is ignored in most of existing work which is communication between e-Government applications. The need to communicate is expected in the context of e-Government, in order to improve governmental processes and thereby satisfy citizens. Considering this aspect from the early stages of development by dedicating a SPL for this aim; will facilitate the integration of systems thereafter.

As known e-Government is a broad domain, developing a SPL that covers a specific set of e-Government services does not mean the development of an e-Government SPL since the derived applications will not exceed the limited scope. To the best of our knowledge there is no other work that exploits large scale reuse between different e-Government subfields as we have presented in this thesis.

Table 6. 1: Comparing our approach to related work.

Approach	Methodology	Scope	Reuse level	Reuse aspect
<b>A Service Oriented Product Line Architecture for E-Government,</b> <i>Ines and al</i> [114]	SOPL life cycle	<b>A subfield of e-Gov</b> online admin	Between applications in a SPL	<b>Business aspects</b> “Demand for the first time”, “Demand for loss”, “Demand for modifications” for CIN, Passport and B3.

				<b>Technical aspects</b> Authentication and notification.
<b>Component-based Architecture for e-Gov Web Systems Development</b> , <i>Carromeu and al</i> [117]	PLUS <sup>2</sup> approach	<b>E-Gov</b> (not limited)	Between applications in a SPL	<b>Technical aspects</b> Security, persistence, chat, Log, Skins...
<b>Software Product Line Engineering for e-Learning Applications: A Case Study</b> , <i>Sanchez and al</i> [112]	SPL process	<b>A particular aspect related to e-Gov</b> Data-mining applications for eLearning	Between applications in a SPL	<b>Technical aspects</b> Features related to data-mining
<b>Geographic e-Services Development through Product-Line Engineering and Standardization</b> <i>Buccella and Cechich</i> [118]	SPL Process	<b>A particular aspect related to e-Gov</b> Geographic information systems in governmental agencies	Between applications in a SPL	<b>Technical aspects</b> Geographic information features
<b>E-Government MPL</b>	e-Gov MPL engineering process	<b>E-Gov</b> An e-Gov MPL composed of ASPLs and e-Gov subfields SPLs	Between applications within a SPL & between SPLs within the MPL	<b>Business aspects</b> For each e-Government subfields <b>Technical aspects</b> Between e-Government subfields (Security, authentication, Communication...)

### 6.5. Conclusion

Despite the great efforts expended by governments all around the world to implement an efficient e-Government, most of these projects fail. Researchers have identified numerous reasons for these failures basically:

- the delivered software do not meet the expected functionalities;
- late delivery of systems;
- lack of integration and interoperability;
- increasing costs;
- security and confidentiality issues;
- poor knowledge of the system and lack of suitable training;

<sup>2</sup> Product Line UML-base Software engineering.

- lack of citizens trust and satisfaction.

In this chapter we have proposed to adapt our approach (AMPL) for the development of e-Government systems. Our aim is to reach a successful e-Government by improving the development process of e-Government applications. The proposed approach takes advantages from the benefits provided by software product line engineering namely: to improve productivity and software quality, and to reduce time, cost, and effort of development. Furthermore, our approach helps significantly to tackle the aforementioned issues. SPLE promotes large scale reuse which allows the faster development of applications and decreases costs as well. Variability management is a key activity in SPLE, it allows the adoption of applications to fit the variable requirements of customers, and thus to gain their satisfaction. The rest of issues could be faced by ASPLs. Even security, communication and HMI are not the core functionalities of e-Government systems, neglecting these aspects will result in a high probability of failure. Thereby, we have suggested the dedication of a ASPL for each aspect. Security SPL will enhance safety; confidentiality and citizen's trust, communication SPL will allow easier integration of e-Government systems, while HMI SPL will allow users to be more familiar with applications without having to make a new training for each new application. Other specialized SPLs are proposed for the development of other common requirements in order to increase reuse and benefit from the great similarity between e-Government systems.

Since the proposed approach is based on SPLE it will obviously meet the challenges found in this area. Mainly the initialization of an e-Government software product line requires a considerable launching investment, however once the base of reusable elements is developed final applications can be achieved in short time. Developers must give high attention to variability management; they must use effective techniques to properly manage variability from the early development stages. It is also important to develop a software architecture model suitable to e-Government applications that facilitate the assembly (or derivation) and extension of systems. Finally, e-Government is intended mainly to citizens, thus it is crucial to consider their feedbacks in order to improve the delivered services.

## CONCLUSION

In this thesis we have presented an approach that intend the management of reuse at two abstraction levels: applications level and product lines level. The presented approach takes advantages from two software engineering disciplines: *Software Architecture* and *Software Product Lines*. It relies on the techniques and methods provided by both of them to resolve the issues encountered when developing systems for broad and complex fields such as *e-Government*.

The study that we have conducted on the SPLs approaches revealed the necessity of adapting an MPL in order to allow the efficient software production within complex fields. The review of the existing MPL engineering approaches brings out three main aspects that must be considered when managing MPLs:

- Reuse among SPLs of an MPL must be systematized i.e. constructed components must plan for reuse not only within a SPL but also between separated SPLs and variability should be managed efficiently within MPLs.
- Effective methods have to be developed for structuring the MPL model, starting from the various SPLs models and getting an MPL model including dependencies between the various SPLs. Dependencies have to be specified in order to simplify the MPL derivation thereafter.
- Solutions should be proposed to reduce the distributed derivation challenges or avoid them completely.

On the other side, we have studied the integration of CBD approach and SPL approach. The study we conducted revealed that the primary aspects that should be covered when defining a new CBPL approach are:



- The CBPL process must base on the SPLE methodology on the one side, and benefits from the CBD technologies on the other side. Both of SPLE and CBD provide powerful techniques for supporting reuse but at opposite granularity spectrum. CBD deals with reuse in the small while SPLE manage reuse in the large. Therefore, significant benefits are expected from their integration. The SPLE process lies on the distinction between two development stages: development for reuse (domain engineering) and development with reuse (SPL derivation). A set of reusable core assets is constructed during the first stage and intended for reuse in the SPL scope. SPLE provides techniques for making the adaptations easier and the reuse systematic by managing the domain variability. CBD represents techniques for implementing variability and makes the derivation step automation possible by producing flexible components.
- The proposed approach should support variability management. Variability must be identified for the various abstraction levels and modeled explicitly in the different modeling views. In addition, mechanisms must be defined for modeling the different variations kinds and this for each architecture element. Thus, variability must be modeled and implemented for the following levels: system architecture, composite components' internal structure, primitive components implementation, interfaces and connectors, and using the following variation types: mandatory, optional, alternative, AND, OR and XOR VPs and groups with cardinality.
- The proposed methodology should support the reuse systematization not only within a single SPL but also among separated SPLs included in the same field.

In order to achieve the traced objectives, we have proposed two complementary approaches: CBPL and AMPL. CBPL aims to reach a high level of reuse that can be obtained through the integration of two approaches: SPL and CBD. Each of these approaches promotes reuse at different granularity levels. CBD supplies technologies for reuse in the small, while SPL approach intends reuse in the large. Putting them together allow us to reach large scale reuse and flexibility at the same time. Moreover, CBD can overcome the lack of maturity in SPL engineering

by providing efficient technologies of development. This approach is validated by a case study in chapter 3.

AMPL approach aims to resolve some MPLs engineering challenges basing on two main concepts: separation of concerns and partial derivation. Separation of concerns at MPL level helps systemizing reuse among SPLs by organizing the MPL models into ASPLs aiming to develop the reusable components within an MPL, and sub-SPLs targeted to produce the MPL final applications. The partial derivation is used to prepare ASPLs for integration with their reusing sub-SPLs. The early integration of MPL sub-SPLs avoids the distributed derivations challenges encountered thereafter. In this paper we have explained our methodology for developing MPLs, and validated it for the e-Gov field.

The existing works focus on resolving reuse challenges in the late development stages i.e. at derivation time. They tend to derive separated SPLs and integrate their heterogeneous instances. At this level several challenges are encountered known by the distributed derivation. This reuse way is still opportunistic because it has not been planned before, and results in long procedures of adaptation and decision. Our approach, avoids all of those challenges by planning for reuse from the early development stages. This planning is ensured by the introduction of ASPLs that are responsible for the production of common components through the MPL. Thus a crucial outcome of our work is the systematization of reuse between SPLs of an MPL. The ASPLs are, after that, partially derived in order to ease their integration with their reusing SPLs. The partial derivation represents an important technique for the merging of separated SPLs. In our work, it helps integrating the SPLs early in the development process to avoid the distributed derivation challenges thereafter. Thus, partial derivation and early integration represent other crucial outcomes of our work.

Finally, the proposed approach is validated in context of the e-Government field. From another view point, the proposed approach allows to reach a successful e-Government by improving the development process of e-Government applications. It takes advantages from the benefits provided by SPL engineering namely: to improve productivity and software quality, and to reduce time, cost, and effort of development. Furthermore, our approach helps significantly to tackle the aforementioned issues. SPLE promotes large scale reuse which allows the faster

development of applications and decrease costs as well. Variability management is a key activity in SPLE; it allows adopting applications to fit the variable customers' requirements, and thus gaining their satisfaction. The rest of issues could be resolved by ASPLs. Furthermore, even if security, communication and HMI are not the core functionalities of e-Government systems, neglecting these aspects will result in a high probability of failure. Thereby, we have suggested the dedication of a ASPL for each of them. Security SPL will enhance safety; confidentiality and citizen's trust, communication SPL will allow easier integration of e-Government systems, while HMI SPL will allow users to be more familiar with applications without having to make a new training for each new application. Other specialized SPLs are proposed for the development of other common requirements in order to increase reuse and benefit from the great similarity between e-Government systems.

Since the proposed approach is based on SPLE it will obviously meet the challenges found in this area. Mainly the initialization of an e-Government SPL requires a considerable launching investment, however once the base of reusable elements is developed final applications can be achieved in short time. Developers must give great attention to variability management; they must use effective techniques to properly manage variability from the early development stages. It is also important to develop a software architecture model suitable to e-Government applications that facilitate the assembly (or derivation) and systems extension. Finally, e-Government is intended mainly to citizens, thus it is crucial to consider their feedbacks in order to improve the delivered services.

Our approach tackles some important MPLs development issues; nevertheless more research work is still needed in this area. In the future we aim to: define the partial derivation techniques for the various SPL core assets, and to formulate the proposed activities in order to allow the partial derivation process automation. It would be also important to test our approach in other MPLs environments than e-Government in order to reach further improvements.

## REFERENCES

- [1] Len, B., Clements, and P. Kazman, R., "Software Architecture In Practice", Third Edition, Boston: Addison-Wesley. ISBN 978-0-321-81573-6, (2012).
- [2] Perry, D. E., Wolf, A. L., "Foundations for the study of software architecture", ACM SIGSOFT Software Engineering Notes. 17 (4): 40. doi:10.1145/141874.141884, (1992).
- [3] Jean-Christophe, " Les lignes de produits logiciels Réutilisation et variabilité ", Publication périodique de Smals, (Juin 2009).
- [4] Klaus,P., Böckle,G., and van der Linden,F., "Software Product Line Engineering: Foundations, Principles, and Techniques", Springer,(2005).
- [5] Northrop, L.M., Clements, C.C., "A Framework for Software Product Line Practice," Version 5.0 (online), <http://www.sei.cmu.edu/>
- [6] Chen, L., Babar, M.A., Nour, A., "Variability Management in Software Product Lines: A Systematic Review," SPLC, San Francisco, California,(2009).
- [7] Bachmann, F., and Clements, P.C., "Variability in Software Product Lines," technical report CMU/SEI, ( 2005).
- [8] Yildiz, M., "E-government research: Reviewing the literature, limitations, and ways forward", Government Information Quarterly, vol. 24, no 3, (2007), p. 646-665.
- [9] Awais, R., Royer, J.C. and Rummler, A., " Aspect-oriented, model-driven software product lines: The AMPLE way", Cambridge University Press, (2011).
- [10] Van Gorp, J., Bosch, J., & Svahnberg, M., "On the Notion of Variability in Software Product Lines". In Proceedings Working IEEE/IFIP Conference on Software Architecture, IEEE, (2001, August), 45-54.
- [11] Van Grup, J., Bosch, J., Svahnberg, M., "Managing Variability in Software Product Lines". Proceedings of IEEE/IFIP Conference on Software Architecture, (2000).

- [12] Sinnema, M., & Deelstra, S., "Classifying variability modeling techniques. Information and software technology", 49(7), (2007), 717-739.
- [13] Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J. H., Pohl, K., "Variability Issues in Software Product Lines", Software Product-Family Engineering, Lecture Notes in Computer Science Volume 2290, (2002), 13-21.
- [14] Kang, K. C., Hyesun, L., "Variability Modeling", Systems and Software Variability Management. Springer, (2013), 25-42.
- [15] Van der Linden, F. J., Schmid, K., & Rommes, E., "Software product lines in action: the best industrial practice in product line engineering", Springer Science & Business Media, (2007).
- [16] Charles W. Krueger, "Variation Management for Software Production Lines", SPLC 2 Proceedings of the Second International Conference on Software Product Lines, Pages 37-48, Springer-Verlag London, UK, (2002).
- [17] Svahnberg, M., van Gorp, J. and Bosch, J., "A taxonomy of variability realization techniques", Software Practice and Experience, (2005).
- [18] Classen, A., Heymans, P., Laney, R., Nuseibeh, B., & Tun, T. T., "On the structure of problem variability: From feature diagrams to problem frames", (2007), 109-118.
- [19] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Technical Report CMU/SEI-90-TR-21, (1990).
- [20] Lianping, C., Babar, M. A. and Ali, N., "Variability management in software product lines: a systematic review." In *Proceedings of the 13th International Software Product Line Conference*, Carnegie Mellon University, (2009), 81-90.
- [21] Lee, K., Kang, K. C., & Lee, J. , "Concepts and guidelines of feature modeling for product line software engineering", In *Software Reuse: Methods, Techniques, and Tools*, Springer Berlin Heidelberg (2002), 62-77
- [22] Fey, D., Fajta, R., & Boros, A., " Feature modeling: A meta-model to enhance usability and usefulness", In *International Conference on Software Product Lines* Springer, Berlin, Heidelberg, (August 2002), 198-216.

- [23] Czarnecki, K., Eisenecker, U. W., "Generative programming: methods, tools, and applications", ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, (2000).
- [24] Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., & Huh, M., "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", *Annals of Software Engineering*, vol. 5, (1998), 143-168.
- [25] Griss, M. L., Favaro, J. Alessandro, M., "Integrating Feature Modeling with the RSEB", *Proceedings of the Fifty International Conference on Software Reuse*, Victoria, Canada, (June 1998).
- [26] Riebisch, M., Böllert, K., Streitferdt, D., & Philippow, I., "Extending feature diagrams with UML multiplicities", In *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, Vol. 23, (2002, June), 1-7.
- [27] Czarnecki, K., and Kim, Ch. H. P. "Cardinality-based feature modeling and constraints: A progress report", *International Workshop on Software Factories*, (2005).
- [28] Czarnecki, K., Helsen, S., and Eisenecker, U., "Formalizing cardinality-based feature models and their specialization", *Software process: Improvement and practice* 10.1, (2005), 7-29.
- [29] Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., & Wąsowski, A., "Cool features and tough decisions: a comparison of variability modeling approaches", In *Proceedings of the sixth international workshop on variability modeling of software-intensive systems*, ACM, (2012, January), 173-182.
- [30] Software Productivity Consortium Services Corporation, Technical Report SPC-92019-CMC. *Reuse-Driven Software Processes Guidebook*, Version 02.00.03, (1993).
- [31] Forster, T., Muthig, D. and Pech, D., "Understanding Decision Models- Visualization and Complexity Reduction of Software Variability." *VaMoS*, (2008).
- [32] Atkinson, C., Bayer, J. and Muthig, D., "Component-Based Product Line Development: The KobrA Approach", *Proceedings of the First Software Product Lines Conference SPLC1*, (2000).
- [33] Gomaa, H., & Webber, D. L., "Modeling adaptive and evolvable software product lines using the variation point model". In *System Sciences, 2004*.

*Proceedings of the 37th Annual Hawaii International Conference on IEEE*, (January 2004), page 10.

- [34] Jacobson, I., Griss, M., Jonsson, P., "Software Reuse- Architecture, Process and Organization for Business Success". ACM Press, New York, NY, (1997).
- [35] Sinnema, M., Deelstra, S., Nijhuis, J., & Bosch, J., "Covamof: A framework for modeling variability in software product families". In *International Conference on Software Product Lines*, Springer, Berlin, Heidelberg, (2004, August), 197-213.
- [36] Kakola, T. and Duenas, J. C., "Software Product Lines, Research Issues in Engineering and Management", Springer, (2006).
- [37] Clauss, M., "Generic Modeling using UML extensions for variability", In: *Workshop on Domain Specific Visual Languages at OOPSLA, USA*, (2001).
- [38] Clauss, M., "Modeling variability with UML", *GCSE 200-Young Researchers Workshop*, September 2001.
- [39] Ziadi, T., Hérouët, L., and Jézéquel, J. M., "Modélisation de Lignes de Produits en UML", In *Proceedings of Langages et Modèles à Objets (LMO03) Vannes/France*, (2003).
- [40] Ziadi, T., Hérouët, L., and Jézéquel, J. M., "Towards a UML Profile for Software Product Lines", In *Proceedings of International Workshop on Product Family Engineering (PFE-5)*, Seana / Italy, (2003).
- [41] Ziadi, T. and Jézéquel, J. M., "Product Line Engineering with the UML: Deriving Products", chapter in *Software Product Lines: Research Issues in Engineering and Management*, Springer-Verlag, (2006), 557-596.
- [42] Donegan, P. M., & Masiero, P. C., "Design Issues in a Component-based Software Product Line", In *SBCARS*, (August 2007), 3-16.
- [43] Razavian, M., Khosravi, R., "Modeling Variability in the Component and Connector View of Architecture Using UML", *6th IEEE/ACS*, (2008).
- [44] Van Ommering, R., "The Koala component model for consumer electronics software", *Philips Research Eindhoven, IEEE Computer* 33(3), (MAR 2000).
- [45] Asikainen, T., Soininen, T., & Männistö, T., "A Koala-based approach for modelling and deploying configurable software product families", In *International Workshop on Software Product-Family Engineering*, Springer, Berlin, Heidelberg, (November 2003), 225-249.

- [46] Voelter, M. Visser, E., "Product Line Engineering using Domain-Specific Languages", 15<sup>th</sup> international SPLC, (2011).
- [47] Apel, S., Batory, D., Kästner, C., & Saake, G., "A development process for feature-oriented product lines", In Feature-Oriented Software Product Lines, Springer, Berlin, Heidelberg, (2013), 17-44.
- [48] Gerald, H., Grünbacher, P. and Rabiser, R., "A systematic review and an expert survey on capabilities supporting multi product lines", Information and Software Technology 54, no. 8, (2012), 828-852.
- [49] Savolainen, J., Mannion, M., Kuusela, J., "Developing platforms for multiple software product lines", In proceeding of Software Product Line Conference, Salvador, Brazil, (2012), 220-228.
- [50] Bosch, J., "The challenges of broadening the scope of software product families", Communications of the ACM 49.12, (2006) ,41-44.
- [51] Schröter, R., "Using Multi-Level Interfaces to Improve Analyses of Multi Product Lines", Technical report, Otto-von-Guericke University Magdeburg, Germany, (2014).
- [52] Schröter, R., Siegmund, N., Thüm, T., "Towards modular analysis of multi product lines", In Proceedings of the 17th International Software Product Line Conference co-located workshops, ACM, (2013), 96-99.
- [53] Van Ommering, R., "Beyond product families: Building a product population?", International Workshop on Software Architectures for Product Families, Springer Berlin Heidelberg, (2000).
- [54] Van Ommering, R., "Building product populations with software components." Proceedings of the 24th international conference on Software engineering, ACM, (May 2002), 255-265.
- [55] Altintas, N. I., & Cetin, S., "Managing large scale reuse across multiple software product lines", In International Conference on Software Reuse . Springer, Berlin, Heidelberg, (2008, May), 166-177.
- [56] Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T., "Structuring the modeling space and supporting evolution in software product line engineering", Journal of Systems and Software 83 (7), (2010), 1108–1122.
- [57] Rosenmüller, M., Siegmund, N., Kästner, C., Syed, S. ur R., "Modeling dependent software product lines", In Proceedings of the GPCE Workshop



- on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE) (2008), 13-18.
- [58] Rosenmüller, M., and Siegmund, N., "Automating the Configuration of Multi Software Product Lines." *VaMoS* 10, (2010), 123-130.
- [59] Hartmann, H., Trew, T., "Using feature diagrams with context variability to model multiple product lines for software supply chains", In *Software Product Line Conference*, IEEE, (2008), 12-21.
- [60] Kruchten, P., Obbink, H. and Stafford, J., "The past, present, and future for software architecture", *IEEE software* 23.2, (2006), 22-30.
- [61] Fritz, S., "What is software architecture?", *SAICSIT '12 Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*. Pretoria, South Africa, (October 2012), 363-373.
- [62] Bass, L., Clements, P., & Kazman, R., "Software architecture in practice." Boston, Massachusetts Addison, (2003).
- [63] Schmidt, D. C., & Buschmann, F., "Patterns, frameworks, and middleware: their synergistic relationships", In *25th International Conference on Software Engineering*, 2003. Proceedings, IEEE, (May 2003), 694-704.
- [64] Pressman, R. S., "Software engineering: a practitioner's approach", Palgrave Macmillan, (2005).
- [65] Garlan, D., and Dewayne, E. P. , "Introduction to the special issue on software architecture", *IEEE Trans. Software Eng.* 21.4, (1995), 269-274.
- [66] GARLAN, D., "Software architecture: a roadmap". In *Proceedings of the Conference on the Future of Software Engineering*, ACM, (2000), 91-101.
- [67] Brown, Alan W., "Large-scale, component-based development", Vol. 1. Englewood Cliffs: Prentice Hall PTR, (2000).
- [68] Duncan, S., "Component software: Beyond object-oriented programming", *Software Quality Professional*, 5(4), 42, (2003).
- [69] González, R., and Torres, M. "Critical issues in component-based development", In *Proceedings of The 3rd International Conference on Computing, Communications and Control Technologies (CCCT'05)*, (2005).
- [70] Bennouar, D., "The Integrated Approach to Software Architecture", Ph.D. Thesis, Ecole Supérieure d'Informatique, Oued Smar, Algiers (in French), (2009)

- [71] Garlan, D., T. Monroe, R. and Wile, D., "Acme: Architectural description of component-based systems", *Foundations of component-based systems* 68, (2000),47-68.
- [72] Medvidovic , N. and Taylor, N R., "A classification and comparison framework for software architecture description languages", *IEEE Transactions on Software Engineering*, 26(1) :70.93,( 2000).
- [73] Mary, S., & David, G. (1996). *Software architecture: perspectives on an emerging discipline*. Prentice-Hall. "Software Architecture Perspective on an Emerging Discipline", Prentice Hall India, First edition, (2000).
- [74] Magee, J., Dulay, N., & Kramer, J., "Structuring parallel and distributed programs." *Software Engineering Journal* 8.2, (1993), 73-82.
- [75] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., & Stefani, J. B., "The fractal component model and its support in java", *Software: Practice and Experience* 36,11-12, (2006),1257-1284.
- [76] Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., & Mann, W., "Specification and analysis of system architecture using Rapide", *IEEE Transactions on Software Engineering*, 21(4), (1995), 336-354.
- [77] Saadi, A., "An Action Language for the Specification and the Validation of Software Architecture Behavior in the IASA Approach", *Magister Thesis (in French)*, LRDSI Lab, Computer Science Department, The Saad Dahlab University, Blida, Algeria, (June 2008).
- [78] Bennouar , D., Saadi, A., "The Design of an eGovernment Application Using an Aspect Oriented Software Architecture Approach", *AOSA conference*, (2009).
- [79] Bennouar, D., Henni, A., Saadi, A., "The Design of A Complex Software System Using A Software Architecture Approach", *The International Arab Conference on Information Technology*, (2008).
- [80] Bennouar , D., Khammaci, T., Henni, A., "A new approach for component's port modeling in software architecture", *In Journal of System and Software*, Elsevier, Volume 83 Issue 8, (August 2010).
- [81] Lazilha, F. R., Barroca, L., de Oliveira Junior, E. A., & de Souza Gimenes, I. M., "A component-based product line architecture for workflow management systems", *CLEI Electronic Journal*, 7, (2004).

- [82] D'Souza, D. F., A. C. Wills. "*Objects, Components and Frameworks with UML – The Catalysis Approach*", Addison Wesley Publishing Company, (1999).
- [83] Jacobson, I., Griss, M., Jonsson, P., "*Software Reuse – Architecture Process and Organization for Business Success*", New York: Addison-Wesley, (1997).
- [84] Brown, T. J., Spence, I., Kilpatrick, P., & Crookes, D., "Adaptable components for software product line engineering", In International Conference on Software Product Lines, Springer, Berlin, Heidelberg, (2002, August), 154-175.
- [85] Rob van, O., "The Koala component model for consumer electronics software", Philips Research Eindhoven, IEEE Computer 33(3), (MAR 2000).
- [86] Van Ommering, R., "Building product populations with software components", In Software Engineering, ICSE 2002, Proceedings of the 24rd International Conference on, IEEE, (May 2002), 255-265.
- [87] Asikainen, T., Soininen, T., & Männistö, T., "A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families", PFE-5, (2004).
- [88] Loughran, N., Sánchez, P., Garcia, A., and Fuentes, L., "Language support for managing variability in architectural models", In Software Composition Springer Berlin/Heidelberg, (2008), pp. 36-51.
- [89] Tizzei, L. P., Rubira, C. M., & Lee, J., "A Feature-oriented Solution with Aspects for Component-based Software Product Line Architecting", SEAA, 12, (2012), 1-10.
- [90] Atkinson, C., Bayer, J., & Muthig, D. "Component-based product line development: the KobrA approach", In Proceedings of the 1st Software Product Line Conference, (2000, August), 289-309.
- [91] Dobrica, L., & Ovaska, E. , "Applying UML Extensions in Modeling Software Product Line Architecture of a Distribution Services Platform", Model-Driven Domain Analysis and Software Development: Architectures and Functions: Architectures and Functions, (2010), 351.
- [92] Weerakkody, V., and Choudrie, J., "Exploring e-government in the UK: Challenges, issues and complexities", Journal of Information Science & Technology, 2(2), (2005).

- [93] Guendouz, A., Bennouar, D., “Component-Based Specification of Software Product Line Architecture”. In the International Conference on Advanced Aspects of Software Engineering ICAASE, (2014), 100-107.
- [94] Guendouz, A., Bennouar, D., Ramdani, A., Mazari, H., “Customer Satisfaction through E-Learning Software Product Line”, Proceedings of the 9th International Conference on Internet and Web Applications and Services, ICIW, (2014), 14-18.
- [95] Lee, K., “Variability and Aspect Orientation”, Systems and Software Variability Management, Springer Berlin Heidelberg, (2013), 293-300.
- [96] Guendouz, A. and Bennouar, D., “AMPL: aspect multiple product lines”, International Journal of Computers and Applications, DOI: 10.1080/1206212X.2020.1735761, (2020), 1-11.
- [97] Guendouz, A. and Bennouar, D., “Managing reuse across MPLs through Partial Derivation”, in proceeding of International Arab Conference on Information Technology ACIT, (2016).
- [98] Rabiser, R., Grünbacher, P., Holl, G., “Improving awareness during product derivation in multi-user multi product line environments”, Proc. Int. Conf. Automated Configuration and Tailoring of Applications, in Conjunction with 25th IEEE/ACMè Int. Conf. on Automated Software Engineering, Antwerp, Belgium, CEUR-WS, (2010), 1-5.
- [99] Czarnecki, K., Helsen, S., Eisenecker, U.: ‘Staged configuration using feature models’: ‘Software Product Lines’ (Springer, 2004), pp. 266-283
- [100] Czarnecki, K., Helsen, S., Eisenecker, U., “Staged configuration through specialization and multilevel configuration of feature models”, Software Process: Improvement and Practice, 10, 2, , (2005), 143-169
- [101] Layne, K., and Lee, J., “Developing fully functional E-government: A four stage model”, Government Information Quarterly 18, ELSEVIER, (2001), 122–136.
- [102] William G. Wood,” Government Product Lines”, Book: Software Product Lines, Springer US, (2000),183-192
- [103] Kumar, V., Mukerji, B., Butt, I., and Persaud, A., “Factors for Successful e-Government Adoption: a Conceptual Framework”, Electronic Journal of e-Government Volume 5 Issue 1, (2007), 63 – 76.

- [104] HUANG, Z. and BENYOUCEF, M., "Usability and credibility of e-government websites", *Government Information Quarterly*, vol. 31, no 4, (2014), 584-595.
- [105] Müller, J., "Generating Graphical User Interfaces for Software Product Lines: A Constraint-based Approach", In proceeding of: 15 Interuniversitäres Doktorandenseminar Wirtschaftsinformatik der Universitäten Chemnitz, Dresden, Freiberg, Halle-Wittenberg, Jena und Leipzig, (2011).
- [106] Mellado, D., Fernandez-Medina, E., & Piattini, M., "Security Requirements Variability for Software Product Lines", *The Third International Conference on Availability, Reliability and Security*, Barcelona, (March 2008), 1413-1420.
- [107] Rodríguez, J., Fernández-Medina, E., Piattini, M., & Mellado, D., "A Security Requirements Engineering Tool for Domain Engineering in Software Product Lines", *Non-Functional Properties in Service Oriented Architecture*(book), Publisher: IGI Global, Pub. Date: March 31, (2011), 73.
- [108] Mellado, D., Fernández-Medina, E., & Piattini, M., "Security Requirements Management in Software Product Line Engineering", In: *e-Business and Telecommunications*. Springer Berlin Heidelberg, (2009), 250-263.
- [109] Ralyté, J., Jeusfeld, M. A., Backlund, P., Kühn, H., & Arni-Bloch, N., "A knowledge-based approach to manage information systems interoperability", *Information Systems*, vol. 33, no 7, (2008), 754-784.
- [110] Novakouski, M., Grace, A. L., "Interoperability in the e-Government Context", technical note CMU/SEI, (January 2012).
- [111] Janssen, M., Charalabibis, Y., Kuk, G., & Cresswell, T., "E-government Interoperability, Infrastructure and Architecture: State-of-the-art and Challenges", *Journal of Theoretical and Applied Electronic Commerce Research*, vol. 6, no 1, (April 2011), I-VIII.
- [112] Sanchez ,P., Diego, G., and Marta, Z., "Software Product Line Engineering for e-Learning Applications: A Case Study," *2012 International Symposium on Computers in Education (SIIE 2012)*, Andorra, (2012), 1-6.
- [113] HELALI, R., ACHOUR, I., JILANI, L. L., "A Study of E-Government Architectures". In : *E-Technologies: Transformation in a Connected World*. Springer Berlin Heidelberg, (2011), 158-172.

- [114] Achour, I. Labed, L., Helali, R. and Ben Ghazela, H., "A Service Oriented Product Line Architecture for E-Government", The 2011 International Conference on e-Education, e-Business, Enterprise Information Systems, and e-Government, USA, (2011).
- [115] Lee, J., Kotonya, G., "Combining Service-Oriented with Product Line Engineering", IEEE Software, Volume:27 , Issue: 3 , (05 February 2010) ,35 - 41.
- [116] Bayer, J., Buhl ,W., Giese, C., Lehner, T., Ocampo, A., Puhlmann, F., Richter, E., Schnieders, A., Weiland, J., and Weske, M., "Process Family Engineering: Modeling Variant-Rich Processes", PESOA-Report No. 18, (2005).
- [117] Carromeu, C., Paiva, D. M. B., Cagnin, M. I., Rubinsztein, H. K. S., Turine, M. A. S., and Breitman, K. Component-based architecture for e-Gov web systems development", In 2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems, IEEE, (2010, March), 379-385.
- [118] Buccella, A., & Cechich, A., "Geographic e-Services Development through Product-Line Engineering and Standardization", Electronic Government and the Information Systems Perspective, Lecture Notes in Computer Science Volume 6267, (2010), 150-157.