



Towards Dynamic Software Product Lines: Unifying Design and Runtime Adaptations

Carlos Parra

► To cite this version:

Carlos Parra. Towards Dynamic Software Product Lines: Unifying Design and Runtime Adaptations. Software Engineering [cs.SE]. Université des Sciences et Technologie de Lille - Lille I, 2011. English. NNT: . tel-00583444

HAL Id: tel-00583444

<https://theses.hal.science/tel-00583444>

Submitted on 5 Apr 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Towards Dynamic Software Product Lines: Unifying Design and Runtime Adaptations

THÈSE

présentée et soutenue publiquement le 04/03/2011

pour l'obtention du

Doctorat de l'université des Sciences et Technologies de Lille
(spécialité informatique)

par

Carlos Parra

Composition du jury

Président : Patrick Heymans, *University of Namur - Belgium*

Rapporteurs : Jean-Claude Royer, *École des Mines de Nantes - France*
Philippe Lahire, *Université de Nice Sophia-Antipolis - France*

Examineurs : Svein Hallsteinsen, *SINTEF - Norway*
Anthony Cleve, *University of Namur - Belgium*
Xavier Blanc, *Université de Bordeaux 1 - France*

Directrice : Laurence Duchien, *Université de Lille I - France*

Mis en page avec la classe thloria.

"It is through science that we prove, but through intuition that we discover"

Henri Poincare

Acknowledgement

I would like to start by saying thank you to the members of my thesis committee. Jean-Claude Royer from the École de mines de Nantes and Philippe Lahire from the University of Nice. You have accepted the task of reviewing my manuscript, even though you were asked just before the Christmas period. Thank you very much, I have really appreciated your feedback. It helped me to improve my dissertation. I would also like to thank Sven Hallsteinsen for being in my committee after a long trip, and for actively boosting the interest from the research community to the topic of Dynamic Software Product Lines. In the same way I would like to thank Patrick Heymans, which was also the president of my jury, for their valuable questions and feedback. Last but not least, I would like to thank two people that worked with me during the last three years. Their ideas and work have greatly influenced this research and gave it new directions. I am really happy to have had the opportunity to work with Xavier Blanc, professor in the University of Bordeaux, and Anthony Cleve from the University of Namur and, thank you very much for your help, ideas and valuable feedback, without you this work would have not be the same.

Next, all my gratitude goes to my advisor Laurence Duchien. Working under your direction during these three years (and some months) has been an great experience. Thanks for your guidance, your advice, for being patient enough to talk to me in English during the first months of my thesis and of course for helping me to improve my French every month after. In general, thank you for always having the right word and the good idea to keep me motivated and in the good direction to get to the end of this research.

I would also like to thank all the permanent members of the ADAM team. Lionel Seinturier, Philippe Merle, Anne-Francoise Lemeur, and Romain Rouvoy, thanks for all the feedback and comments I have got from you during these last three years. Also to the former Ph.D. students, Guillaume Wagner, Jeremy Dubus, Naouel Moha and all the others that I forget now, you have been a guide and an inspiration to me. Special thanks to Carlos Noguera and Ales Plsek, former Ph.D. students and friends that welcomed me at my arrival to Lille and that helped me to get by at the beginning of my stay.

In the same way, I wish the best of lucks to the Ph.D. students that are still struggling with their own thesis, specially my spanish-speaking friends Daniel Romero, Gabriel Hermosillo, and Gabriel Tamura. And to the new generations of Ph.D.s in ADAM Russel Nzekwa, Rémi Melisson, Nicolas Haderer, Alexandre Feugas, Adel Noureddine, Clement Quinton, and Rémi Druilhe. To all of you, best of lucks and keep the good work. Also to the engineers of the team, and in particular to the two engineers that worked with me in the CAPPUCINO project, Nicolas Pessemier, and Pierre Carton. Special thanks to a former postdoc Frédéric Loïret, thanks for your "real-french" lessons, your advice, but more importantly for your friendship and for sharing with us throughout all these years. Also, I would like to thank Sébastien Mosser for his help and advice in the preparation of my Ph.D. defense. I also want to thank Corinne Davoust and Malika Debuyschere the former and current assistants of the ADAM team, who have been patient enough to deal with all my needs from travel arrangements to accommodation and visa documents.

Finally, I want to thank my family. First of all my parents, I would not be here without you. Thank you mom for supporting me all these years, for taking care of me in the distance. And to you dad, I know you would be proud of what we have achieved. Then of course, my brother and sisters: Jaime, Pilar, Mónica and Claudia; thank you all, you have always been an example to me and a great support in the hard times. Finally, to the rest of my family and to all my friends that kept in contact with me during these years, thank you very much, I hope to see you again soon.

Abstract

In the recent years, we have witnessed major advances in mobile computing. Modern devices are equipped with a variety of sensors and network interfaces that make them quite versatile. In order to take advantage of all the hardware capabilities and provide a better user experience, software has to be *context aware*, i.e. it has to monitor the events and information coming from its environment and react accordingly. At the same time, we notice that an important number of such mobile applications share several characteristics regarding its architecture, communication, storage and interfaces. This leads us to consider that context-aware systems can also benefit from the *Software Product Line* (SPL) paradigm. SPLs were defined to take advantage of commonalities through the definition of reusable artifacts, in order to automate the derivation of multiple products. Nevertheless, SPLs are limited regarding the runtime modifications implied by context awareness. This dissertation investigates on *Dynamic Software Product Lines* (DSPL). A DSPL extends a classic SPLs by providing mechanisms to adapt products at runtime to cope with dynamic changes imposed by context awareness. Our main goal is to unify design and runtime adaptations under the same definition through high-level artifacts. Such artifacts can then be used to implement DSPLs by defining the processes required to map them into concrete products at design time and at runtime.

Concretely, as the first contribution of this dissertation, we introduce both: a simple – yet complete – *variability model*, and a *composition model* that realizes variability. With the variability model we aim at defining a family of products and at identifying commonalities and variabilities along those products using *variants*. The composition model on the other side, is based on ideas from Aspect Oriented Software Development (AOSD). We use the model to divide the products in several modules called *aspect models* that are used to construct platform independent representations of variability. Each aspect model is formed by three parts: the architecture model which represents parts of a system to be added, the advice that contains a set of changes to the core application and finally, the pointcut that identifies the places where the modifications are performed. As a second contribution, we propose two processes of product derivation: *design weaving* and *runtime weaving*. Design weaving aims at building a single product. Runtime weaving aims at adapting a product being executed. Both processes use the same variability and aspect models. We thus allow developers to reuse the same artifacts used for building a software product to adapt it dynamically among various configurations. For the design weaving, we base ourselves on a model driven approach where transformations and code generation are employed to obtain source code from a set of models. For the runtime weaving, we use `FraSCaTi`, a service and component based platform with dynamic properties, to execute reconfigurations during the execution of products. We also use a context manager to process events coming from the environment and make decisions about the adaptation.

To validate our approach we define and implement a DSPL. Concretely, this research is part of the FUI CAPPUCINO project, which aims at building mobile applications for ubiquitous environments. We have implemented a DSPL for a retail case study. We successfully cover the whole cycle of design derivation and adaptation of software products. The scenario demonstrates the versatility of our approach and in particular the unification achieved through the aspect models used at design time as well as at runtime.

Résumé

Pour profiter des nombreux matériels actuellement, les logiciels s'exécutant sur des téléphones mobiles doivent devenir sensibles au contexte, c'est-à-dire, qu'ils doivent surveiller les événements provenant de leur environnement et réagir en conséquence. Nous considérons que ces logiciels peuvent bénéficier d'une approche basée sur les Lignes de Produits Logiciels (LPL). Les LPLs sont définies pour exploiter les points communs par la définition d'éléments réutilisables. Néanmoins, les LPLs ne prennent pas en compte les modifications à l'exécution des applications. Cette thèse propose une ligne de produits logiciels dynamique (LPLD) qui étend une LPL classique en fournissant des mécanismes pour adapter les produits à l'exécution. Notre objectif principal est d'unifier les adaptations à la conception et à l'exécution en utilisant des artefacts logiciels de haut niveau. Concrètement, nous introduisons un modèle de variabilité et un modèle de composition pour modulariser les produits sous forme de modèles d'aspect. Chaque modèle d'aspect a trois parties : l'architecture, les modifications, et le point de coupe. Ensuite, nous proposons deux processus de dérivation du produit : un pour la conception que vise à construire un produit, et un pour l'exécution que vise à adapter un produit. Ce travail de recherche s'est déroulé dans le cadre du projet FUI CAPPUCINO. Nous avons défini une LPLD pour une étude de cas de vente d'un hypermarché sensible au contexte. Le scénario démontre les avantages de notre approche et, en particulier, l'unification réalisée par les modèles d'aspect utilisés à la fois à la conception et à l'exécution.

Contents

List of Tables	xvii
Chapter 1 Introduction	1
1.1 Problem Statement	3
1.2 Research Goals	4
1.3 Contributions	4
1.4 Dissertation Roadmap	6
Part I State of the Art	9
Chapter 2 Background and Context	11
2.1 Software Product Lines	12
2.2 Model Driven Engineering	17
2.3 Aspect Oriented Software Development	22
2.4 Service and Component Oriented Architectures	24
2.5 Summary	29
Chapter 3 Design and Runtime Adaptation	31
3.1 DSPL and Software Adaptation	32
3.2 Design and Runtime Adaptation: Approaches and Mechanisms	33
3.3 Research Goals Revisited	44
3.4 Summary	47

Part II Contribution	49
Chapter 4 Dynamic Software Product Lines: Our Approach in a Nutshell	51
4.1 CAPucine, A Global Overview	52
4.2 Domain Engineering	53
4.3 Application Engineering	56
4.4 Summary	57
Chapter 5 Design Phase: Variability, Application and Platform	59
5.1 Introduction	59
5.2 Design Phase Challenges	60
5.3 Domain Engineering	62
5.4 Application Engineering	71
5.5 Discussion	78
5.6 Summary	80
Chapter 6 Runtime Phase: Context aggregation, verification, and dynamic reconfiguration	81
6.1 Introduction	81
6.2 Motivating Scenario and Challenges	82
6.3 Adaptation Life Cycle	84
6.4 Runtime Phase: dynamic adaptation of DSPL products	86
6.5 Discussion	96
6.6 Summary	96
Part III Validation	99
Chapter 7 Validation	101
7.1 Experimentation	102
7.2 Tool Support	115
7.3 Discussion	125
7.4 Summary	131

Part IV Conclusion	133
Chapter 8 Conclusion	135
8.1 Summary of the Dissertation	135
8.2 Perspectives	138
Bibliography	143
Appendixes	151
Appendix A Vers les Lignes de Produits Logiciels Dynamiques	153
A.1 Introduction	153
A.2 CAPucine : Vue Globale	154
A.3 Phase de Conception	156
A.4 Phase d'exécution	159
A.5 Conclusion	162

List of Figures

2.1	Costs of a product Line development.	13
2.2	Domain and application engineering processes.	14
2.3	Graphical representation of features.	16
2.4	Four-layer framework of MDA.	20
2.5	Provider, consumer, and broker.	26
2.6	SCA graphical representation.	28
4.1	A global view of CAPucine.	53
4.2	Roles in the domain engineering process.	55
5.1	Feature metamodel.	62
5.2	A sample feature diagram.	64
5.3	Architectural metamodel.	65
5.4	Aspect metamodel: simplified view.	65
5.5	Aspect metamodel: definition of the Model.	66
5.6	Aspect metamodel: definition of the Pointcut.	67
5.7	Aspect metamodel: definition of the Advice.	68
5.8	Architecture and aspect example models.	69
5.9	SCA metamodel.	70
5.10	Product derivation phases.	71
5.11	Dependency between ByLocation and Wifi.	75
5.12	Hidden dependency between Discount and SMS.	76
5.13	Woven model obtained after the composition.	78
5.14	Transformation scheme.	79

List of Figures

6.1	Double threshold pattern.	83
6.2	SCA assembly.	83
6.3	Sequence diagrams for full and low connectivity configurations.	84
6.4	Adaptation life-cycle.	85
6.5	Runtime product derivation.	87
6.6	Aspect metamodel: definition of the <code>Event</code>	88
6.7	Reconfiguration: input and output.	88
6.8	A component for managing product configuration at runtime.	89
6.9	An aspect at runtime.	92
6.10	Core architecture represented in SCA.	94
6.11	Aspect model for the runtime phase.	95
6.12	An FScript equivalent of the <code>SMS</code> advice.	95
6.13	Result of the runtime weaving.	95
7.1	e-Commerce product family.	103
7.2	XMI feature model.	105
7.3	XMI core aspect model.	107
7.4	XMI aspect model for the <code>Email</code> variant.	108
7.5	Aspect dependency.	111
7.6	Mobile client architecture.	114
7.7	<code>Model2Code</code> architecture.	116
7.8	Feature analysis processes.	117
7.9	Use of binary strings to generate product configurations.	117
7.10	Deletion of ilegal products.	118
7.11	Matching pointcuts and models for the constraint analysis.	119
7.12	Model composition processes.	119
7.13	<code>Expression</code> and <code>Modification</code> hierarchy.	120
7.14	Implementation of the <code>InstanceOf</code> expression.	120
7.15	Implementation of the <code>Composite</code> expression.	121
7.16	Model transformation processes.	122
7.17	Interface for creating elements in the target model.	122
7.18	The class <code>PropertyBuilder</code>	123
7.19	Code generation process.	123
7.20	Java code generation.	124

7.21 Extensibility of the Domain.	129
A.1 Ingénieries du domaine et de l'application.	155
A.2 Métamodèle de Features.	157
A.3 Méta-modèle d'aspects: vue simplifiée.	158
A.4 Schéma de transformation.	160
A.5 Dérivation d'un produit à l'exécution.	161

List of Figures

List of Tables

3.1	Comparison criteria for design time approaches.	35
3.2	Summary of the design time adaptation approaches.	37
3.3	Comparison criteria for runtime approaches.	38
3.4	Summary of the runtime adaptation approaches.	40
3.5	Summary of mixed adaptation approaches.	43
3.6	Synthesis of approaches for DSPL.	43
6.1	Optional variant combinations.	91
6.2	Pointcut transformation	93
7.1	Context information.	104
7.2	Summary results of the feature model analysis.	106
7.3	Summary of aspect modeling	109
7.4	Constraint analysis results.	112
7.5	Composition and code generation results.	113
7.6	Runtime adaptation summary.	115
7.7	Model2Code metrics.	125

Introduction

Contents

1.1 Problem Statement	3
1.1.1 Development Processes Differ	3
1.1.2 Lack of a unified representation	3
1.2 Research Goals	4
1.3 Contributions	4
1.3.1 Publications	5
1.4 Dissertation Roadmap	6
1.4.1 Part 1: State of the Art	6
1.4.2 Part 2: Contribution	6
1.4.3 Part 3: Validation	7
1.4.4 Part 4: Conclusion	7

Forty years after the first conference on Software Engineering [nat68] and almost twenty years after the IEEE Computer Society has standardized this discipline, Software Engineering is still struggling to produce large software systems [BJ95, FJF⁺07]. This is related to the fact that requirements keep on changing even after software has been built and deployed. To tackle such a challenge, a cornerstone element that is intrinsic to any moderns software is the notion of *adaptation*. If a system is designed and implemented to be *adapted*, then it is more likely that it can better support changes in its requirements, architecture, and even implementation. Software adaptations open the door for new kinds of systems that use smartly all the information available both at design time, to postpone and minimize the impact of developers decisions, and at run-time, to take advantage of the information available in the application environment. However, adaptations are difficult to define since they may take place at early stages of the development process, but also at runtime where there are many situations that have to be considered (e.g., limited connectivity, hardware heterogeneity, changes of user preferences, etc.). Typically, this kind of information is known as *context information*. Hence, a *context-aware* system is aware of changes in the context and is able to *adapt* to offer better user experiences [Bro96, DAS01]. A well-known example of such kind of behavior is when a system changes its behavior depending on the location [SAW94].

In this dissertation we explore the applicability of the Software Product Line (SPL) paradigm into the development of adaptable systems. SPLs aims at managing and building multiple software products from a set of previously developed and tested *assets*. An asset is understood as any software artifact that can be employed in the development of an application. In SPL engineering, commonalities and variabilities across a set of applications (i.e., product family) are identified, so that assets can be developed, and used to create new different products. We consider that the SPL engineering constitutes a suitable candidate to manage the variety of configurations that we find in context-aware software.

Two essential tasks for any successful SPL are variability management and product derivation [CN01]. Variability can be understood as the analysis of the characteristics (i.e., *features*) that make one product different from others in the same product family. A common tool in SPL are feature diagrams which are used to express the variability by defining its variants and its variation points [SHT06, SvGB05].

The product derivation defines how assets are selected according to a given feature configuration, and specifies how those assets are composed in order to build the desired product [DSB04]. While the product derivation is commonly conceived as only belonging to the development process of products, this is not always the case and it can be extended to cover the adaptation of an existing product at runtime. In fact, the idea of using SPLs to derive dynamic products has recently started to gain interest from the academic community. In [HHPS08] authors introduce SPLs for derivation of software that needs to be adapted at runtime, they refer to this kind of SPLs as *dynamic*. A Dynamic SPL (DSPL) is capable of producing systems that can be adapted at runtime in order to dynamically fit new requirements or resources changes.

This research aims at providing insights on the methodologies and tools required for the development of DSPLs for adaptive applications. We propose a unified approach that allows both: development and adaptation of software. To realize variability across the different phases of development and adaptation, and to provide a complete DSPL approach, we explore several well-known methodologies that have independently proven to bring benefits in terms of modularization, platform independence, reusability, and fast development.

In particular, we explore the Aspect Oriented Modeling (AOM). The AOM initiative introduces the Aspect Oriented Software Development (AOSD) principles in the Model Driven Engineering (MDE) development process, particularly in the composition and transformation phases [AOM, FJ09]. AOSD and MDE follow the well-known separation of concerns principle, which has been proven to provide many benefits, including reduced complexity, improved reusability, and easier evolution [TOHS99]. AOSD enables software systems to be modularized using orthogonal *aspects* that are woven at the production time [KLM⁺97]. MDE deals with levels of abstraction and considers any software artifact produced at any step of the development process as a valuable asset by itself to be reused across different systems and implementation platforms [Sch06].

We propose in this dissertation a unified approach that supports the complete software life cycle: from feature selection and initial product derivation, to runtime adaptation in response to changes of the execution environment. We concretize the notion of asset with a definition of aspect models to leverage the variability across a family of products. Derivation of products is divided in two adaptation processes: *design time adaptation* and *runtime adaptation*. The former one is in charge of creating the initial product. The latter one modifies the product once it has been created and deployed depending on the execution context. Finally our approach unifies design and runtime adaptations by representing both categories of adaptation as aspect models.

Structure of the Chapter

The reminder of this introductory chapter is organized as follows: in Section 1.1 we identify the problems that motivate this research. In Section 1.2, we present a preliminary description of the overall approach. In Section 1.3 we briefly enumerate the different contributions of this dissertation. Finally, in Section 1.4 we present a roadmap to guide the reader through the rest of this document.

1.1 Problem Statement

In spite of the increasing necessity for automating design and runtime adaptations, several problems remain open, preventing such kind of developments to be widely accepted. Here below we present the most relevant problems related to such systems.

1.1.1 Development Processes Differ

The first problem is related to the twofold product derivation. We can notice that there are two different processes for product derivation. One that covers the *design time adaptations* and one that covers *runtime adaptations*. In the literature, we can find approaches that face each process separately. For example, AOM is currently being used within more and more SPL approaches [RGF⁺06, KAAK09, LSO⁺07, PKGJ08] in order to compose assets selected from a feature diagram. However, those approaches only contribute to the design phases of the software life cycle (i.e., they are not dynamic). The system features and their corresponding assets are modeled using AOM techniques. The product derivation process is supported by automatic model compositions and transformations. As a consequence, they are used to build software systems that, once deployed, cannot be easily evolved. Some SPL approaches contribute to the runtime phase [MFB⁺08, DL06]. They are based on rules specifying the contextual changes that trigger the dynamic adaptation of a software system. Although some of those approaches make use of aspects to specify and realize dynamic adaptation, they define new mechanisms that differ from those involved in existing SPL approaches focusing on the design phase. Any automated development process for adaptive applications has to provide the means to describe adaptations at design time as well as adaptations at runtime. As a consequence, there exist no unified SPL approach that covers the complete life cycle, from design to runtime.

1.1.2 Lack of a unified representation

Even if the two processes previously identified use different technologies and occur at different moments in the life cycle of any product, they both have the same objective, i.e., to modify the product by adding and/or removing a certain group of features from the product being derived. It would be desirable to have a unified representation of these modifications, so that we can use them at design time and at runtime. Such modifications have basically the following information:

- **Why?** the **motivation** behind a software adaptation must be made explicit. This could be either a design choice (for design adaptation) or a specific runtime event.
- **When?** the **pre-condition** under which the adaptation can be realized has to be specified. This pre-condition defines all the software components that should belong to the system.

- **What and how?** the system **modifications** that realize the adaptation have to be defined. Those modifications can be applied either at the design level (e.g., class, code) or at the runtime level (e.g., links between components, values of specific fields).

1.2 Research Goals

This dissertation investigates on software engineering techniques for developing and adapting software. Our main goal is to implement dynamic software product lines. We propose a software engineering solution that introduces high-level abstractions of *assets*. Such abstractions allow us to define independent yet complementary processes for derivation and adaptation of context-aware software products. By doing it, we are able to address several issues regarding: automation, verification of correctness, code generation platform independency, and runtime adaptations. The main goals of our approach are:

- **Variability Management:** First of all, the DSPL has to provide the means to express commonalities and variabilities across the family of products. This helps to identify and build reusable assets that can be used to build new products reducing the effort and the time invested when building several products.
- **Automated Development Process:** Variability enables developers to create product configurations by selecting the features they want for their products. A second challenge for DSPLs is to use such configurations as starting points for the implementation of automated development process of adaptable software.
- **Correctness:** It is important that products are not only easier to develop, but also that their correctness rests guaranteed. When composing multiple parts to form a software product, it is possible that two or more of those parts have conflicts regarding the elements where they are going to be composed and the requirements for the composition to take place. It may happen that implicit dependencies exist between different artifacts. This may lead to composition and correctness problems. We want to define a development process that analyses such inconsistencies and prevents the incorrect products from being derived.
- **Guarantee platform independance:** It is also desirable that the development process remains platform-independent. This allows the SPL to have multiple targets and postpone the decision of a particular platform until later steps of the product derivation. Our DSPL has to be able to separate business concepts from the details of the underlying platform.
- **Continue derivation at runtime:** A fundamental issue in adaptive software development, is the management of events and context information, and its manipulation in order to modify products dynamically. With the DSPL, we want to add the support for deriving products at runtime, and at the same time preserve the same architecture defined during the design and implementation of the product.

1.3 Contributions

We propose an approach for designing and implementing Dynamic Software Product Lines. In particular, as the first contribution of this thesis, we introduce both: a simple – yet complete –

variability model, and an *aspect model* that realizes variability. With the variability model we aim at defining a family of products and at identifying commonalities and variabilities along those products using *variants*. Additionally, the variability model also allows us to define constraints between those variants. The aspect model, on the other side, is used to construct platform independent representations of variability. Each aspect is self-contained in the sense that it has the three pieces of information required for it to be integrated into any product. It defines the model which represents parts of a system to add, advices with a set of changes to the core application and finally, pointcuts that identify the places where the aspect perform the modifications. As a second contribution, we propose two independent processes of product derivation. Variability and aspect modeling allow us to define a complete development process that unifies the expression and manipulation of domain independent concerns at both design time and runtime. We use aspect models in two different processes that we call *design weaving* and *runtime weaving* respectively. Design weaving aims at building a single product. Runtime weaving aims at adapting a product being executed. We thus allow developers to reuse the same artifacts used for building a software product to adapt it dynamically among various configurations.

For the design weaving, we base ourselves on a model driven approach where transformations and code generation are employed to obtain source code from a set of models. For the runtime weaving, we use `FraSCaTi`, a service and component based platform with dynamic properties. Such a choice allows us to execute reconfigurations at runtime. We also use a context manager to process events coming from the environment and make decisions about the adaptation.

1.3.1 Publications

The results of this research have been published in international journals, conferences, book chapters and workshops as follows:

Journals

- Carlos Parra, Xavier Blanc, Anthony Cleve, and Laurence Duchien. *Unifying Design and Runtime Software Adaptation Using Aspect Models*. In Science of Computer Programming. Special edition on software evolution. *To appear*.

International Conferences

- Carlos Parra, Anthony Cleve, Xavier Blanc, and Laurence Duchien. *Feature-based Composition of Software Architectures*. In 4th European Conference on Software Architecture (ECSA 2010), Copenhagen, Denmark. August 2010.
- Carlos Parra, Xavier Blanc and Laurence Duchien. *Context Awareness for Dynamic Service-Oriented Product Lines*. In 13th International Software Product Line Conference (SPLC'09), San Francisco, USA. August 2009.

Book Chapters

- Carlos Parra, Rafael Leña, Xavier Blanc, Laurence Duchien, Nicolas Pessemier, Chantal Taconet and Zakia Kazi-Aoul. *Dynamic Software Product Lines for Context-Aware Web Services*. In *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*. Chapman and Hall/CRC, 2009.

Workshops

- Daniel Romero, Carlos Parra, Lionel Seinturier, Laurence Duchien, Rubby Casallas. *An SCA-based middleware platform for mobile devices*. In *Middleware for Web Services (MWS 2008)* at EDOC2008, Munich, Germany, 2008.
- Carlos Parra and Laurence Duchien. *Model-Driven Adaptation of Ubiquitous Applications*. Proceedings of the First International DisCoTec Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2008). Oslo, Norway, 2008.

1.4 Dissertation Roadmap

1.4.1 Part 1: State of the Art

- **Chapter 2: Context of the Research** In this chapter, we present the notions of Software Product Lines, Model Driven Engineering, Aspect Oriented Software Development and Service Oriented Platforms. The idea of the chapter is to introduce and define a common language and a base of knowledge that are used to explain our approach throughout the rest of this document.
- **Chapter 3: Design and Runtime Adaptation** This chapter presents a survey on the state of the art and the related works in the domain. We list and describe some of the most relevant works related to SPL and dynamic adaptation. At the end of the chapter we present a synthesis to enumerate the strong points and weaknesses of each one. Afterwards, we revisit the contributions of this dissertation and compare them with the surveyed works, in order to highlight the benefits of our approach.

1.4.2 Part 2: Contribution

- **Chapter 4: Dynamic SPL : Our Approach in a Nutshell** This short chapter presents a global overview of our main contribution, a framework for the development of Dynamic SPLs for adaptable software. The idea of the chapter is to introduce the whole process of product derivation. It includes two main phases, design phase, and runtime phase. We also define the roles and responsibilities across the development process.
- **Chapter 5: Design Phase : Variability, Application and Platform** In this chapter we describe in detail the processes and technologies used in the design phase of a DSPL. We start by defining the models of variability and aspects. The chapter also covers the verification of constraints and the transformations and generation of code.

- **Chapter 6: Runtime Phase : Context aggregation, verification, and dynamic reconfiguration** This is the last chapter of our contribution. It describes in detail what happens with applications derived from a DSPL at runtime. We revisit the model of aspects presented in Chapter 5 and explain how it is used to reconfigure products dynamically. We also describe briefly the middleware elements we have utilized to achieve the adaptation. Concretely, we describe FraSCAti, our runtime platform.

1.4.3 Part 3: Validation

- **Chapter 7: Experimentation and tool Support** This chapter is divided in three main parts. First we describe our case study. We start with our *top - down* approach which consist in the development of a DSPL for the examples of the Project CAPPUCINO. Then we describe the software and tool-support developed as part of this research. To conclude the chapter we present a qualitative evaluation and a discussion on the choices made for design and implementation of our DSPL framework.

1.4.4 Part 4: Conclusion

- **Chapter 8: Conclusion and Perspectives** This chapter concludes the work presented in this dissertation. We summarize the overall approach and discuss about the limitations that motivate new ideas and future directions for research in the domain.

Part I

State of the Art

Chapter 2

Background and Context

Contents

2.1 Software Product Lines	12
2.1.1 The Product Line Approach Applied to Software	12
2.1.2 Software Product Line Definition	12
2.1.3 Software Product Line Processes	13
2.1.4 Variability	15
2.1.5 Traceability	16
2.1.6 Summary of SPL and Variability	17
2.2 Model Driven Engineering	17
2.2.1 Models	18
2.2.2 Model Classification	18
2.2.3 Models, Metamodels and Metametamodels	19
2.2.4 Model Transformations	20
2.2.5 Summary of MDE	21
2.3 Aspect Oriented Software Development	22
2.3.1 Aspect Oriented Programming	22
2.3.2 Aspect Oriented Modeling	23
2.3.3 Summary of AOSD	24
2.4 Service and Component Oriented Architectures	24
2.4.1 CBSE	24
2.4.2 SOA	25
2.4.3 SCA	26
2.4.4 Summary of Service and Component Architectures	29
2.5 Summary	29

This chapter introduces the basics of several software engineering approaches used as a starting point for the overall approach presented in this dissertation. The goal of the chapter is to introduce and define a common language and a base of knowledge that will be used throughout this dissertation.

Structure of the Chapter

The chapter is organized as follows. Section 2.1 introduces the notion of Software Product Lines

(SPL). We define SPLs, discuss the domain and application engineering processes, the variability management, and the feature modeling principles. In Section 2.2 we introduce the domain of Model Driven Engineering (MDE), including the concepts of models, metamodels, and transformations. Next, in Section 2.3 we present the main principles of Aspect Oriented Software Development (AOSD), we discuss the issue of separation of concerns and the motivations behind aspects. Finally in Section 2.4 we present the Service Component Architecture (SCA) approach. We finish the chapter with a summary of the approaches presented.

2.1 Software Product Lines

The product lines in general constitute an approach that has already been used successfully in various domains, allowing stakeholders to optimize the product development process and the resources available, through the identification and reuse of common elements that are shared by several products. In avionics for example, there is the case of the american aircraft manufacturer Boeing. They have built a product line in order to produce two of their most famous aircrafts, the 757 and the 767 [Sof10]. Even if they are different aircrafts, according to Boeing, these machines share up to 60% of common parts. By means of a product line strategy, Boeing has achieved to reduce the costs at different stages of the process like: parts manufacturing, assembling, and maintenance. In the same way, other industries have taken advantages of these strategies for optimizing their resources and improving the time to market. Nokia, for example, uses product lines for building mobile phones. Hewlett Packard follows the same approach for building printers.

2.1.1 The Product Line Approach Applied to Software

With regard to software engineering, building software from a set of previously developed and tested parts represents a major advance. The benefits in terms of time, quality, and resources are considerable. Nevertheless, successful implementations of these schemas are not as abundant as in other domains. There are, however, several groups from academia and industry that work on bringing all the benefits from software product lines to the software engineering development process [MM09].

In terms of costs, as stated by [PBL05] SPLs offer benefits when producing at least a certain number of products. Figure 2.1 (taken from [PBL05]) illustrates the costs of producing one versus multiple products. The solid line sketches the costs of developing the systems independently, while the dashed line sketches the costs of developing the products using product line engineering. As it can be seen from the figure, in the case of a few systems, the price of product line engineering is relatively high, whereas it is significantly lower for larger quantities. There is a break-even point at which the two lines intersect. It indicates that the costs are the same for both approaches. As referred in [PBL05] recent empirical experiences have shown that this break-even point is located at around 3 or 4 systems in the particular case of software engineering. This is of course influenced by several factors like: expertise, domain, strategy for implementing SPL, customer base, and range of products.

2.1.2 Software Product Line Definition

As described in [Sof10], an SPL is a set of systems that share a group of manageable *features*. A feature is understood as an end-user visible characteristic of the system [KCH⁺90]. Such features

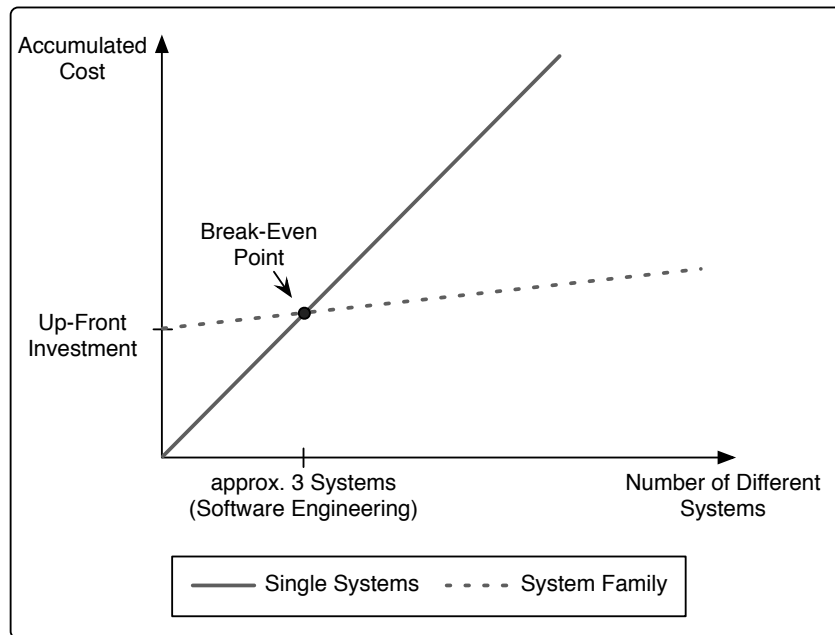


Figure 2.1: Costs of a product Line development.

satisfy the specific needs of a given market. The features are developed from a set of base *assets*. In the context of SPL, an asset is a software artifact that can be used in the development of more than one product. An asset can be a software component, a model, a planning, a document, or any other element useful in the development of system.

One of the most important parts of an SPL is the *architecture*. The architecture must consider the needs of the complete set of products in order to provide a framework for the development and reuse of new assets. These new assets have to be conceived with the required flexibility in order to satisfy the needs of the different products in the SPL. Another important part of any SPL is its *scope*. It contains the boundaries of the SPL with regard to the set of products that can be produced from base assets. The scope defines the common features (i.e. *commonalities*) and the ways in which products differ from each other (i.e. *variability*). This leads to the notion of product family. As stated in [Wit96], a product family refers to the group of software products that can be developed from a set of common assets. The products in a family generally share some of their elements in design, components, and integration rules. The size of the family depends on the capabilities of the SPL to combine the assets in a functional system that manages the concepts of business rules, architecture, platform and implementation.

2.1.3 Software Product Line Processes

Software Product Lines are usually divided in three main tasks: (1) asset development, (2) product development using the assets, and (3) line management. In the SPL community the two former processes are also known as *Domain Engineering* and *Application Engineering* respectively. In general terms domain engineering refers to the creation of assets, whereas application engineering refers to the process of using those assets in order to build individual software products.

Both the domain engineering as well as the application engineering are complementary processes and do not follow a specific order. For instance, it is possible to create assets from already developed products, in which case, assets are built from the artifacts that constitute the products. In this case, domain engineering takes place after the product itself has been built. The assets are obtained from analyzing and isolating the products so that they can be used in the development of other products. A different way of proceeding is by creating the assets from scratch. In this case domain engineering precedes application engineering. Each asset is then conceived having in mind that it is going to be used in the development of multiple software products. Figure 2.2 illustrates the domain and application engineering processes.

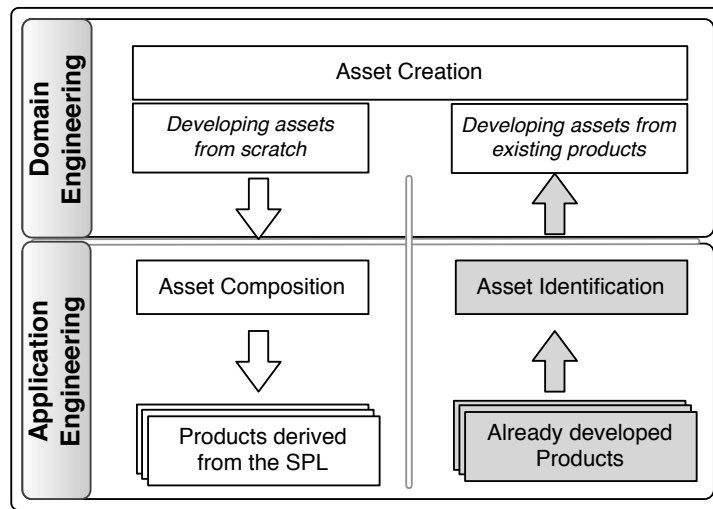


Figure 2.2: Domain and application engineering processes.

Domain Engineering

Domain engineering is the process responsible for establishing the reusable platform, and thus, for defining the commonality and variability of the product line. The platform consists of all types of software artifacts requirements, design realization, tests, etc. Traceability links between these artifacts facilitate systematic and consistent reuse [PBL05].

According to [PBL05], the main goals of the domain engineering process are:

- Define the commonality and variability of the product line;
- Define the product family, i.e., the set of applications that can be built. This represents the *scope* of the product line;
- Build the reusable artifacts that accomplish the desired variability.

Application Engineering

Application engineering refers to the process of actually combining the assets obtained during the domain engineering phase, in order to develop software product. In this process, applications

of the product line are built by reusing the artifacts and exploiting the product line variability. Application engineering aims at reusing the domain assets and take advantage of commonalities and variabilities in order to define and develop a product line application. The application engineering process is composed of activities for (i) configuring individual products inside the set of valid variation points (product configuration), and (ii) creating product line members by using the available domain assets (product derivation).

- **Product Configuration:** it refers to the selection or deselection of a valid combination of variability identified in the domain engineering process. Literature refers to this selection as *binding time* of the variability [PBL05].
- **Product Derivation:** it refers to the concrete process of building an SPL application. This process can be manual, or automated. The main input for this process is the product configuration, and the artifacts identified in the domain engineering process.

2.1.4 Variability

Variability represents the differences among set of products in an SPL. It is thanks to an adequate management of variability that multiple products can be built from a set of reusable assets. Pohl et al. [PBL05] define the notions of variability subjects and objects to describe variability. Variability subjects are the items or properties from the real world that do vary. A variability object is defined as one of the possible ways in which a variability subject varies. In software product line engineering, there are equivalent terms for variability subjects and objects: *variation points* and *variants*. A variation point is a representation of a variability subject, for example, the type of user interface that an application provides. A variant identifies a single option of a variation point. Using the same example, every single user interface that can be chosen for the application (e.g., rich, thin, web-based, mobile) is represented by a variant.

Feature Diagrams

The notion of feature diagrams was first introduced by Kang *et al.* in 1990 [KCH⁺90], as a tool in the Feature Oriented Domain Analysis (FODA). The feature diagram essentially represents a way to model variability among a set of similar products. A feature diagram, as defined by Kang et al., consist of an and/or tree of different features. Optional features are designated graphically by a small circle immediately above the feature name. Alternative features are shown as being children of the same parent feature, with an arc drawn through all of the options. The arc means that one and only one of those features must be chosen. The remaining features with no special notation are all mandatory. Figure 2.3 illustrates a generic example of a feature diagram as proposed by Kang et al. Part *a* of Figure 2.3 shows a feature diagram while part *b* shows one product configured. The selected features for the configured product are highlighted with boxes.

In addition to the feature diagrams, FODA also introduced the notion of *composition rules*. A composition rule is a constraint that establishes a relationship between features. Composition rules have two forms: (1) one feature requires the existence of another feature (because they are interdependent), and (2) one feature is mutually exclusive with another one (they cannot coexist).

A variety of interpretations and extensions to feature diagrams have appeared over the years. Several researchers and industrials have revisited the FODA ideas in order to improve

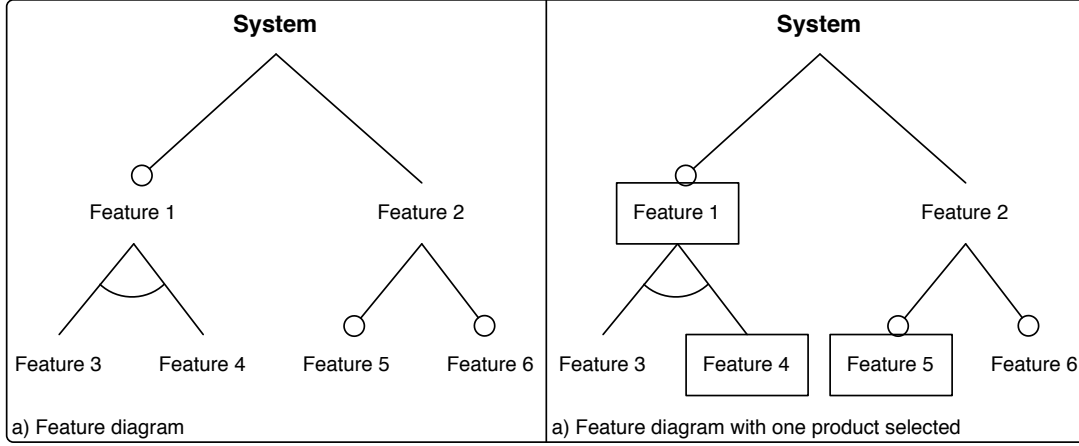


Figure 2.3: Graphical representation of features.

the expressiveness of the feature diagrams and use them as core elements in software development. For example, the notion of multiplicity for the relationships between parent and child features has been proposed, as well as different graphical representations of feature diagrams that include, among others, graphical notations for the composition rules. Consequently, currently there is no unified standard for feature modeling. However, there are efforts mainly from academia to define a unified and standardized language for feature modeling, and provide an adequate tool support for it. Among those efforts we can identify [FHMP⁺09] which proposes the Common Variability Language (CVL) for feature modeling. CVL aims at creating a generic model that can be combined with other types of models in order to define and implement variability of an SPL through transformations. Another example of such efforts is the Text-based Variability Language (TVL) [BCFH10]. TVL is a text-based modeling notation that covers most constructs of existing languages for feature modeling, including cardinality-based decomposition and feature attributes. The main objective of TVL is to provide engineers with a human-readable language supporting large-scale models through modularization mechanisms.

Other approaches for feature modeling include the approaches in [ACLF10], and [BJS09]. In [ACLF10], authors propose a domain-specific language called `FAMILIAR` which allows the definition of product families through feature models. Feature models can be analyzed to find correct configurations and the whole scope of the product family. In [BJS09], authors also propose a configuration tool called `S2T2`, which in the same way as `FAMILIAR`, allows for the definition of feature models. `S2T2` offers a graphic user interface that allows developers to configure products. The constraints of the feature model are propagated throughout the feature tree. For instance, the tool automatically selects features that are required by a given feature, when the latter is selected by the user.

2.1.5 Traceability

The last concept that we present in the context of SPL refers to traceability. Although the approach presented in this dissertation does not specifically target strategies for traceability management, we consider that several paths for future research can benefit from this topic, which is considered

of highly importance in SPL engineering. Traceability refers to the different relationships and dependencies that exist among the artifacts of different levels in a software development process. In the case of SPLs, traceability is considered a very important element to manage the complexity of variability management. Traceability aims at defining the relationships between the features at the domain level, and the assets that implement such features at the underlying levels of the product derivation. In [JZ05] authors enumerate the following difficulties linked to traceability for SPLs: (1) there is a larger number and heterogeneity of documents than in traditional software development; (2) developers need to understand the consequences of variability binding at difference phases of the development; (3) the relationships between products and the product line architecture, or between the product members themselves must be established; and (4) there is still poor general support for managing requirements and handling such relationships among different features of a product family. Furthermore, traceability can be clasified depending on the type of elements for which it creates a relationship. For example, in [AGG⁺08] the authors propose a classification of traceability. They analyze in particular the relationship between traceability and SPLs driven by models. In their classification they identify two types of traceability; *inter* and *intra*. The former one refers to the relationships between different levels of abstraction from requirements to models to implementation. The latter one refers to relationships between artifacts at the same level of abstraction: between related requirements, between models, between software components, etc.

2.1.6 Summary of SPL and Variability

We have presented the main principles of the Software Product Line engineering. We have discussed a key concept in SPLs that corresponds to variability. Identifying commonalities and variabilities among a set of software applications is one of the big challenges for any successful SPL. The languages and tools for feature modeling previously discussed focus on the problem of product configuration. They propose ways to define product families and eventually tool-supported mechanisms to define feature models and configure different products. We have also presented a brief description of traceability in its implications in software product line engineering.

In our approach, we do not specifically deal with the problem of configuration but rather with the product derivation process that follows afterwards. We focus on the mechanisms that enable the realization and composition of commonalities and variabilities for a product family. In order to define such a process, we consider that a simple, easy to understand variability model has to be defined, so that, developers can create multiple product configurations that are used as input in the SPL product derivation. This model can be later transformed towards and from the different variability languages and tools like the ones presented in this section, to take advantage of the configuration assistance and tool support offered from each particular approach.

2.2 Model Driven Engineering

The Model Driven Engineering (MDE) paradigm started back in the year 2000. That year, the Object Management Group (OMG) [Obj10] proposed an approach for the development of software called Model Driven Architecture (MDA) which was based on the idea of using models as first-class entities on the development of software. Two years later, the research community realized the importance of this approach and its transcendence outside the technical space when it was originated. They changed the name and called it Model Driven Engineering, because

they considered the previous term to be restrictive with regard to the variety of approaches that manipulate models [FEBF06]. The central activities in MDE focus on creating models of the systems [KWB03]. MDE helps software engineering practitioners in the design process by allowing them to focus on the business itself. Different points of view (models) can be defined for a system, which are expressed separately. MDE also integrates fundamental tasks like the composition of each point of view, the automation through tools for model transformations, and the code generation [FEBF06]. Furthermore, the use of models brings a higher level of flexibility in terms of: (1) implementation, since new technologies can be implemented using the models of current designs; (2) integration, using the models of a system it is possible to automatize the generation of code and the composition strategies; (3) maintenance, having a design that is understood by a machine gives developers direct access to the specification of the system and thus making maintenance tasks easier; and (4) test and simulation, since models can also be used to validate specific requirements.

2.2.1 Models

The cornerstone of any proposal in MDE are the models. One important characteristic about the whole MDE proposal, is the fact that modeling constitutes a familiar activity to human beings, in diverse domains other than computer science like physics, economics, and medicine. This is mainly because using models facilitates the understanding of real-world problems.

In computer science, a model is a description or a specification of a real-world system and its environment for a specific purpose. A model is represented frequently as a combination of drawings and text. The text can be in a *modeling* language or in *natural* language. In the same way as the system it is modeling, the model can be *static* or *dynamic*. In order for a model to be useful, it is expected that it can answer certain questions about the system it represents, in the same way as the system itself would answer.

Models, according to the way they are built, can also be *prescriptive* or *descriptive*. The former ones refer to the models that are generated with the purpose of guiding the development of the system they represent (i.e., the system represented does not exist yet). The latter ones are obtained from existing systems, and are mainly used to understand the system [Béz05].

2.2.2 Model Classification

The definition of model given previously is rather abstract and could include all kinds of models. This is why there exists a classification of models, according to the information provided about the system they represent, and the level of detail in terms of technologies used or implementation platforms.

Platform

A platform, as defined by the OMG, is a group of subsystems and technologies that provide a coherent set of functionalities through interfaces and specific patterns. An application supported by a given platform, can use such services without worrying about how those underlying services are implemented.

This brings a new property to the models. A model can be *dependent* or *independent* from the platform. A model is independent from the platform when the model does not include the

characteristics specified by the platform for the system to use the provided services. Platform independency can have a scale. In this way, a model can only describe the availability of characteristics of any platform in general, like the remote invocation of services. In a more specific way, a different model may assume the existence of a set of tools for a particular technology, like RMI or CORBA in the case of remote invocation. This means that while some models may assume the existence of an abstract characteristic, others may be linked to a particular technology [KWB03].

Platform Independent Model

The OMG defines a Platform Independent Model (PIM) as a model of a system that contains no specific information about the platform or the technology that is used to realize it. The PIM abstracts away technical details.

However, platform independence is a relative concept. For example, a model can be independent with respect to technical languages but dependent with respect to the middleware platform. Another example for PIMs are the ones created for a virtual machine. A virtual machine is defined as a set of parts and services that are defined independently from any specific platform and that are concretized in different ways for each platform. Hence, a virtual machine is a platform, so a model of a virtual machine is specific to the virtual machine, yet, that model will be independent of the different platforms for which the virtual machine can be implemented.

Platform Specific Model

Finally, a Platform Specific Model (PSM) represents a model with the system specification defined in the PIM, and that additionally includes the details of how such system makes use of the services offered by a particular underlying platform [KWB03].

2.2.3 Models, Metamodels and Metametamodels

There are two main relationships between models: *representedBy* and *conformsTo* [Béz05]. The former one applies to the relationship between a system and a model that represents it (i.e. a system is represented by a model). The latter relationship takes place at a higher level of abstraction and indicates that a model conforms to a *metamodel*. A metamodel is a language used to define the models. Hence a model is specified using the concepts defined in the metamodel. The metamodel is unique in the technical space in which it is working so that operations between models can take place as transformations, combinations, and comparisons [Béz05]. The metamodel itself is defined with a language specified in a meta-metamodel. To prevent the need of infinite layers on top, the meta-metamodel is defined using the same meta-metamodel.

Meta-Object Facility

In MDA, one of the meta-metamodels defined is the MOF (Meta Object Facility). The MOF is an OMG standard that specifies an abstract language to describe other languages [Obj06]. The purpose of the MOF is to define the essential concepts for the modeling and standardization of the design of meta-models and of their resulting models. The MOF is also known as the uppermost layer in the four-layer framework defined in MDA (see Figure 2.4).

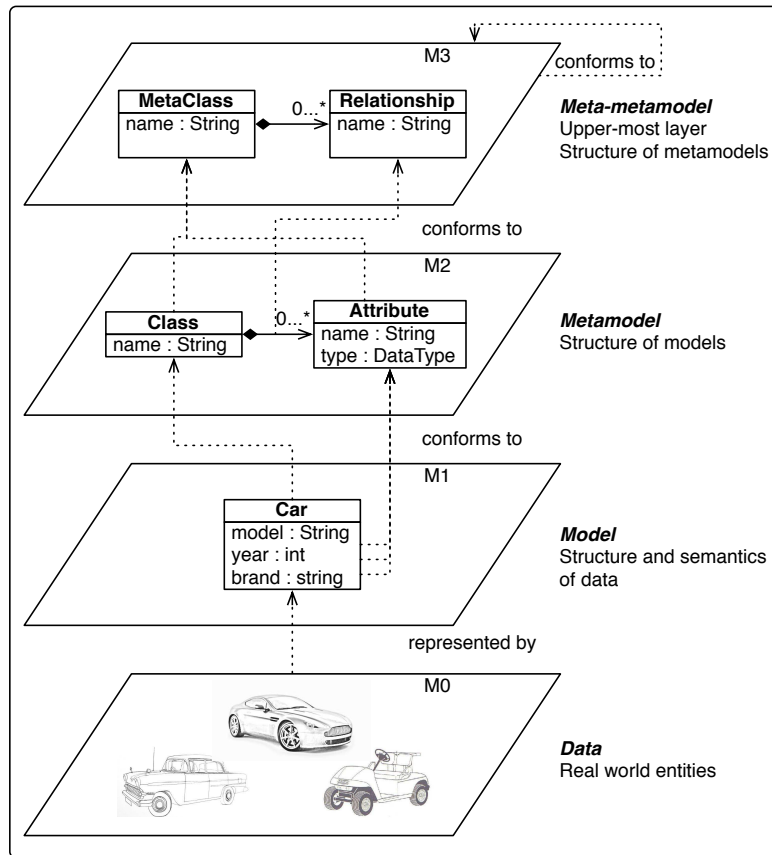


Figure 2.4: Four-layer framework of MDA.

- M0: This level corresponds to the user objects and includes information that has to be modeled and that represents the *real world*. This information is usually called *data*.
- M1: This is the level of models. It is composed of all the meta-data that describes the information from the M0 level.
- M2: this level is constituted by the meta-models. In this level, we found the descriptions (meta-metadata) that define the structure and semantics of the meta-data. A metamodel can also be seen as a language to describe several types of data.
- M3: This is the level of the meta-metamodel. It defines the structure and semantics of the meta-metadata. In other words, it is the language used to define the different metamodels (MOF).

2.2.4 Model Transformations

In general terms, a transformation is described as the process of converting a model of a system into a different model of the same system [CH03]. A transformation is usually formed by the following elements: rules, relationships between the source and the target model, and direction of the transformation (unidirectional or bidirectional). Inside a rule, there is also the notion of

scope and applicability. The strategy of application corresponds to the order in which rules are applied and the way they are organized through mechanisms of reuse, modularity, or according to the language [CH03].

Currently, two main approaches for model transformation can be identified: *model to model* and *model to text*. Each approach can have subcategories of its own. For example, for the *model to text* transformations there are approaches based on visitors, which provide mechanisms to traverse the different elements of a model and print a text sequence related to them. There are also approaches based on templates that contain the code that has to be generated and empty spaces that are filled with the information coming from the model [CH03].

In the *model to model* transformations, we find the approaches of *direct manipulation*, in which the developer of the transformation is the one who makes all the decisions concerning: the relationships between source and target elements, the order, and the strategy of application. There are also the *relational* approaches in which mathematical relationships are used to establish the type of relation between different elements of the source and target models. We can find *graphical* transformations too, where graphical elements are used to describe the relationships between source and target elements. Approaches based on *structures* are also popular, since they help developers in defining the transformations, by providing the tools and taking care of the order and strategy used to apply such transformations. Finally, there are *hybrid* approaches which are a combination of the previous approaches [CH03].

A transformation is usually formed by a set of transformation rules, which specify the way in which target elements are created in terms of the source elements. A transformation rule has two parts: left and right. The left part is in charge of accessing the source model, while the right part is in charge of making the needed operation to create or modify the target model. Both the left and right parts contain the following characteristics:

- **Variables:** variables contain the elements of the source or target model (or eventually intermediate elements).
- **Logic:** logic contains the constraints and algorithmic over the model elements. The logic can be executable or not. Non-executable logic is used to specify a relationship between the models. Executable logic can be declarative or imperative. In [Kur05], the author explains the difference between declarative and imperative model transformation languages. A language is declarative if its transformation rules specify the relationships between the source and target model elements, without involving a particular order of execution. In contrast, a language is imperative, when it specifies an explicit sequence of steps to be executed with the goal of producing a particular result. There may be a third category of transformation languages. In this category languages are considered hybrid, this means that they have a mix of declarative and imperative instructions. As mentioned in [Kur05], a transformation written in this kind of languages may have a set of declarative rules that describe the relationships between source and target models, and additionally, each rule may have fragments of imperative code, that realize additional actions to obtain the desired result.

2.2.5 Summary of MDE

In this section we have presented the main principles of Model Driven Engineering including the notion of models, metamodels and transformations. MDE has represented an evolution in software engineering, by making emphasis on the definition of models of systems. Using MDE

applications can be built in different steps, starting from an independent model that gets enriched with the particular elements of the platform and the implementation language. Furthermore, automated approaches for code generation can be implemented using the MDE principles.

However, we consider that a pure MDE approach is not enough to face challenges for the design an implementation of dynamic software product lines. For example, the configuration and correctness issues from the variability management, the composition order and correctness needed to built different products from user choices, and the reconfigurations of products at runtime, among others, are some of the challenges that require different strategies other than MDE. We consider that MDE can be useful to define a derivation process, where products are built from platform independent models. Such models can then be mapped towards platform specific models and source code. However, to face the challenges related to SPL configuration management, it is necessary to combine combine MDE with other approaches.

2.3 Aspect Oriented Software Development

Aspect Oriented Software Development (AOSD) is a set of emerging technologies that look for new ways of modularizing software systems. The origins of modularization can be traced back to Parnas in the seventies [Par72]. It refers to the separation and localization of concerns. A concern can be anything that interest developers about a system. It can be a high-level concern (i.e. user-visible), like reliability or security, or it can be a low-level concern dealing with technical issues like caching and synchronization [FECA05]. Separating such concerns helps specialists to focus on small modules of their expertise, hence improving quality and reducing the probability for failure. Separation of concerns is a fundamental part of software engineering allowing developers to divide a problem into several independent modules that deal with particular concerns in a separated way. Object Oriented Programming (OOP), for example, is a way of separating concerns, by decomposing a system into a set of objects that deal with particular functional concerns. However, OOP does not deal with what is known as *cross-cutting* concerns. A cross-cutting concern is not ideally placed into a single module (an object for example), but is rather spread across several modules [KLM⁺97]. For this reason, and to complement OOP, Aspect Oriented Programing (AOP) was defined.

2.3.1 Aspect Oriented Programing

AOP is the activity of programming with multiple cross-cutting concerns or aspects [FECA05]. It was defined in 1996 by Kickzales and his team at the Xerox PARC research center [KLM⁺97]. As mentioned before, the main objective of AOP was not to replace the OOP but rather to complement it in order to obtain applications that are clearer and better structured [PRS04]. AOP helps developers to define and implement cross-cutting concerns.

AOP specifically deals with two main problems of OOP, code dispersion (*scattering*), and code mixing (*tangling*).

- **Scattering:** it happens when similar code is spread across many program modules. Hence the same code has to be written over and over again in various modules, making it hard to maintain.

- **Tangling:** it happens when code, external to the main objective of the module, is mixed with the code of the local module in order to deal with a particular concern. This makes it harder to understand the module and causes that a change in one part of the module has an impact on other parts that should be independent.

Definitions

AOP introduces a new terminology to define aspects and describe the way they are linked to the base modules. Here we present the most important concepts that we use again in our contribution.

- **Core:** the core corresponds to the base modules of the system without any aspects. The core is the base where the concerns are added using the aspects.
- **Aspect:** an aspect is a modular unit designed to implement a concern. It may have some code (i.e., *Advice*) and the instructions on where, when and how to invoke it. To define such instructions, two essential concepts in AOP are the join points and pointcuts.
- **Join point:** a join point is a well-defined place in the structure or execution flow of a program where additional behavior can be attached. The most common elements of a join point model are method calls, though aspect languages have also defined join points for a variety of other circumstances including field definition, access, modification, exceptions, execution events and states [FECA05].
- **Pointcut:** a pointcut or pointcut designator [FECA05], describes a set of join points. A pointcut allows developers, for example, to mark a specific set of methods (chosen from all the methods in an application), where they want their aspects advice code to be woven.
- **Weaving:** finally, in AOP the weaving is known as the process of composing core functionality modules with aspects, thereby yielding a working system. Weaving can happen in different ways depending on the mechanisms used to implement it. For example, weaving can take place statically, compiling the advice together with the base code; at loading, inserting aspects when loading the code; or at runtime, linking the aspects while the base code is being executed.

2.3.2 Aspect Oriented Modeling

Aspect-oriented Modeling (AOM) aims at applying the aspect-oriented techniques to software models in order to modularize crosscutting concerns. AOM is concerned with the systematic identification, modularization, representation, and composition of concerns. This can be done at different moments of the software development process, and also using different mechanisms to model the aspects.

As stated in [FJ09], a relationship between AOSD and MDE can be established since, from a modeling point of view, the terms aspect and model can be considered synonymous. This notion of aspects goes beyond the usual meaning found in the AOP community with a broader definition. In AOM an aspect is a concern (not only crosscutting) that can be modularized.

AOM is of particular interest for the purposes of this dissertation because, as pointed out by [FJ09], a challenge in this area is, rather than achieving separation of concerns (which has been

already done in many contexts), to reduce the effort that engineers have to put when working with many inter-dependent concerns. Authors illustrate the problem in a product-line context, when an engineer wants to replace a variant of an aspect used in a system. In such a context, the engineer should be able to do this easily, quickly, and safely.

Similarly to MDE or AOP, currently there is no standard approach for using AOM. Although there are already several approaches using AOM, it is not clear how such approaches apply to different phases of software development and how they can be combined to produce a coherent aspect-oriented software development process. In the next chapter, we survey some of the most relevant works in this area, in order to better position our strategy for using aspect models.

2.3.3 Summary of AOSD

We have briefly discussed the notions and main concepts of AOSD. In our approach, we are particularly interested in AOM, since it offers the possibilities to modularize the architecture of an application using models. We consider that aspect models can be particularly useful to realize the variability of the product line. For that, a language has to be defined in order to create the aspects that are linked to the notions identified in the variability modeling. However, in the context of DSPL, a challenge that arises from the use of aspect models is the definition of a product derivation that allows aspects to: (1) represent feature selections; (2) be woven at design time to build products; and (3) be woven at runtime to adapt applications dynamically. This implies that, in order to use aspect models for product derivation, they have to be linked to the features defined for the product family, and also be used in combination with a design generation chain and a runtime platform that supports dynamic adaptations.

2.4 Service and Component Oriented Architectures

This last section focuses on the Service Component Architecture (SCA). It is relevant for the purpose of this dissertation, because as we will detail in the contribution chapters, our target platform is service and component-based. In this section we briefly discuss the origins of SCA and we summarize the essential concepts of the specification. To properly understand SCA we have to first introduce the two trends in software engineering that were at the origin of this specification: the Component Based Software Engineering (CBSE) and the Service Oriented Architecture (SOA).

2.4.1 CBSE

A different way of modularizing a program is by means of *components*. In the same way as OOP or AOP, CBSE was intended to modularize software systems [Par72], and to follow the principle of separation of concerns. CBSE aimed at creating independent entities for different modules in a software system. Szyperski *et. al* [Szy02] define a software component as unit of composition with contractually specified interfaces and explicit context dependencies only.

In general terms, a component can be seen as a black (or white) box abstracting a given behavior, with explicit input and output mechanisms called *interfaces*. It is through these interfaces that the component can communicate with other components and with its environment. Ideally, the interface is an abstraction that defines what is strictly necessary for the component to

achieve such communication. Szyperski *et al.* also defined three features that characterize any component:

- It is a unit of independent deployment: this means that a component has to be well separated from its environment and from other components, by including all of their constituents. Also, it will never be deployed partially;
- It is a unit of third party composition: this means that the component has to be self-contained, so that, no matter which third party is in charge of the composition, it can do it by using the specification of the component establishing what it provides and what it requires. In other words, a component needs to encapsulate its implementation and interact with its environment by means of well-defined interfaces;
- It has no observable state: this means that a component should not be distinguishable from (can be replaced with) copies of its own.

Several component models have been proposed from both the industry (COM [Don98] from Microsoft, the EJB specification from SUN [BMH06], CORBA [OMG04] from the OMG, etc.), as well as from academia (Fractal [BCL⁺06], SOFA [PBJ98], etc.). Some of those approaches like Fractal, have reflexivity properties. *Reflexion* is understood as the ability of a program to examine (i.e., introspection) and eventually modify (i.e., intercession) its internal structure during the execution. Reflexion is used for controlling the lifecycle of components, and enable a set of operations like starting, stopping and modifying the architecture that has been defined at design time. This is a very interesting characteristic because it enables applications to be modified at runtime. We consider reflection constitutes a useful mechanism to be used for the process of dynamic product derivation in software product lines.

2.4.2 SOA

Service Oriented Architecture (SOA) is a paradigm to develop software systems based on services that interact with each other. In [Jos07], Josuttis defines SOA as a paradigm for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners. SOA aims at facing several challenges:

- Distributed systems: this refers to the fact that in large businesses, more and more systems and companies are involved with corresponding integration and exchange.
- Different owners: concerned with the distributed nature of systems that makes it possible for different groups of engineers or even companies to manage separated systems.
- Heterogeneity: related to the differences in terms of programming languages, platforms or even paradigms among the different software systems that are part of a large enterprise application.

SOA faces those challenges by employing three main technical concepts:

- Services: the service is essential to SOA. Services provide solutions to requirements of software users. A service is an IT representation of some business functionality. [Jos07]

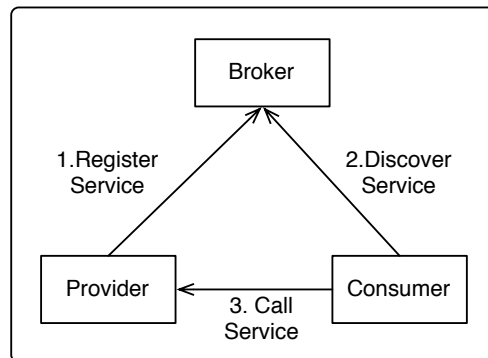


Figure 2.5: Provider, consumer, and broker.

- **Interoperability:** this refers to the ability of several systems to connect with each other and communicate successfully. It is highly desirable to achieve interoperability, specially in heterogeneous environments with a variety of different systems.
- **Loose coupling:** loose coupling refers to the amount of knowledge that one module has over another one in a system. With loose coupling, the goal is to minimize the dependencies between different modules so that, if one module of the system gets modified, it should not have an impact, or at least it should be minimal, on the rest of the modules.

It is important to notice that SOA, as a paradigm, does not apply to a specific technology. The most common application example of SOA are Web Services. Web Services are a way to realize SOA by using a specific implementation strategy.

IBM identifies three main participants in any SOA architecture regardless of the implementation [Gis01].

- **Service Provider:** corresponds to the system that implements a service (a business functionality) so that other systems can use it [Jos07].
- **Service Consumer:** also known as requestor ([Gis01]), it corresponds to the system that calls the services offered by the providers. A service provider can be also a service consumer.
- **Service Broker** is an entity that acts as a repository for software interfaces that are published by service providers. A business entity or an independent operator can represent a service broker [Gis01].

The three participants interact with each other as explained in Figure 2.5 (taken from [Jos07]). The provider registers the services it offers in the broker, then the consumer discovers such services in the broker and proceeds to call the service directly in the provider.

2.4.3 SCA

The Service Component Architecture (SCA) was born as a way to provide a technology-agnostic platform to achieve delivery, support, and management of distributed applications conforming to

the principles of SOA [Bar07]. SCA uses software components as a way to implement services. It basically brings together CBSA and SOA with a specification for developing service components that can be treated as services but that can be implemented as components. Concretely, SCA is a set of specifications describing a model for building applications and systems using SOA. It is promoted by several important software companies including BEA, IBM, IONA, Oracle, SAP, Sun and TIBCO.

The essence of SCA are components implementing business logic that offer their capabilities through service-oriented interfaces called services and that consume functions offered by other components through service-oriented interfaces, called references [Bar07].

SCA proposes a development process with two major parts: (1) implementing the components which provide services and consume other services, and (2) assembling sets of components to build business applications, through the wiring of references to services. SCA emphasizes the decoupling of service implementation and of service assembly from the details of infrastructure capabilities and from the details of the access methods used to invoke services. SCA components operate at a business level and use a minimum of middleware APIs.

The SCA specification supports service implementations written using any of several programming languages, including conventional object-oriented and procedural languages such as Java, PHP, C++, COBOL; XML-centric languages such as BPEL and XSLT; also declarative languages such as SQL and XQuery. SCA also supports a range of programming styles, including asynchronous and message-oriented styles, in addition to the synchronous call-and-return style [Bar07].

Definitions

In the SCA specification [BBB⁺07], software entities are *components* which provide interfaces (called *services*), require interfaces (called *references*), and can have *properties*. A component may be a *composite* if it has subcomponents inside. Figure 2.6 shows the graphical representation of some of the elements defined in the specification.

Following the SCA notation, components are presented as blue rounded-corner squares. Green chevrons to the left of each component represent the services that it provides, whereas purple chevrons to the right represent the references or services it requires. Properties are represented as yellow squares usually placed on top of each component.

- **Component:** a component is the basic element of business function in SCA providing services and consuming references. In [Cha07] a component is defined as an instance of an implementation that has been appropriately configured. The implementation is the code that actually provides the component's functions, such as a Java class or a BPEL process. The configuration defines how that component interacts with the outside world. A component can be implemented using different technologies, and it relies on a common set of abstractions (including services, references, properties, and bindings), to specify its interactions with the world outside itself;
- **Composite:** it represents a way to combine components into a larger structure. A composite aggregates multiple components, it can also have services and references. In other words, a composite is an assembly of components, services, references, and the wires that interconnect them [BBB⁺07];

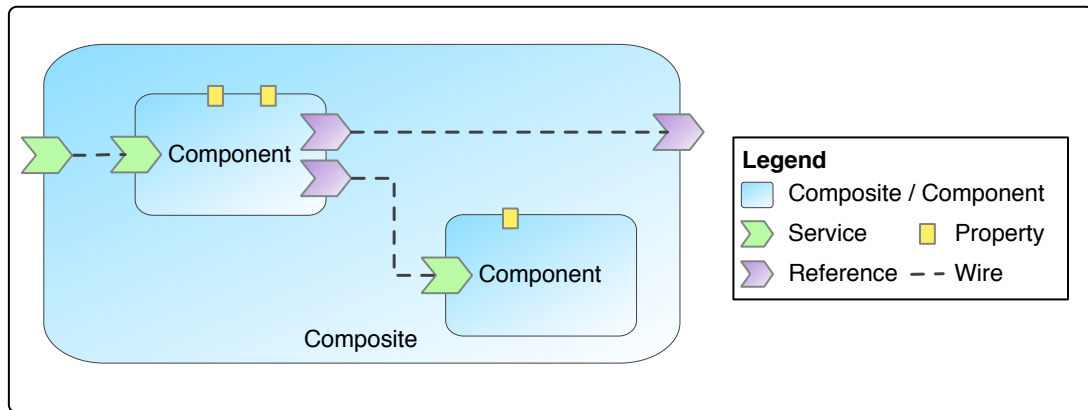


Figure 2.6: SCA graphical representation.

- **Service:** services represent the entry point to the functionality offered by a component. In essence, a service represents an addressable set of operations of an implementation that are designed to be exposed for use by other implementations or exposed publicly for use elsewhere (e.g. public web services for use by other organizations). The operations provided by a service are specified by an interface, as are the operations required by the service client (if there is one) [BBB⁺07];
- **Reference:** references represent the functionality required by one component that is provided by a different component. The SCA specification defines a reference as a service that an implementation may call during the execution of its business function. In the same way as services, references are typed using an interface;
- **Property:** properties allow for the configuration of an implementation with externally set data values. The data value is provided through a component [BBB⁺07];
- **Wire:** as shown in Figure 2.6, wires (dashed lines) connect services and references. Within a composite, valid wire sources are component references and composite services. Valid wire targets are component services and composite references [BBB⁺07].
- **Binding:** a binding specifies how the communication between a component and other elements should be done. Bindings are used by services and references. References use bindings to describe the access mechanism used to call the service to which they are wired. Services use bindings to describe the access mechanism(s) that clients should use to call the service;
- **Domain:** a domain specifies the instantiation, configuration, and connection of a set of components that are specified in one or more composite files. The domain, like a composite, also has Services and References. Domains also contain wires that connect the Components, Services and References.

2.4.4 Summary of Service and Component Architectures

In this last section, we have presented SCA and the two approaches that were at its origin: SOA and CBSE. SCA enables the development of service components that can be treated as services but that can be implemented as components. In our approach we use a platform implementing the SCA specification. We base ourselves in this platform because it offers the dynamic reconfiguration tools that we need in order to adapt products at runtime. In the second part of this dissertation, we explain how we have modeled the applications using a component and service architecture, so that they can be executed and adapted at runtime using the tools provided by the SCA platform.

2.5 Summary

In this chapter we have briefly introduced the principles and basic concepts of four main approaches in software engineering: Software Product Lines, Model Driven Engineering, Aspect Oriented Software Development, and Service Component Architecture. We have given a short explanation of each one of them.

We have also introduced the terminology used in every particular community. We consider that the approaches presented in this chapter can be combined for defining a SPL for adaptive applications. Therefore, having a common understanding of the terminology of each community is important for the comprehension of the approach presented in this dissertation.

In the next chapter, we survey different works in the literature that are closely related to our approach. We go down in specificity and review research approaches that also try to define dynamic frameworks for adaptive applications by employing some of the approaches presented in this chapter. This will allow us to make a comparison and properly position our contributions. In the second part of this dissertation, and having this comparison as a reference, we proceed with a detailed description of our approach for building dynamic software product lines.

Chapter 3

Design and Runtime Adaptation

It is important that students bring a certain ragamuffin, barefoot, irreverence to their studies; they are not here to worship what is known, but to question it.
– The Ascent of Man

Contents

3.1 DSPL and Software Adaptation	32
3.1.1 Adaptation	32
3.1.2 Static and Dynamic Adaptation	33
3.1.3 Adaptation at design time	33
3.1.4 Adaptation at runtime	33
3.2 Design and Runtime Adaptation: Approaches and Mechanisms	33
3.2.1 Design time adaptation approaches	34
3.2.2 Runtime adaptation approaches	37
3.2.3 Mixed adaptation approaches	40
3.2.4 Summary	43
3.3 Research Goals Revisited	44
3.3.1 The Need for Unification	44
3.3.2 Challenges for DSPL Revisited	45
3.4 Summary	47

Dynamic Software Product Lines (DSPL) intend to face several challenges related to the continuous changes in software at design and execution. More and more, these modifications become the rule rather than the exception. At design time, developers have to produce a family of software products instead of individual applications for solving one problem. They usually specify the software using a composition of concerns in order to obtain a complete software definition. Such a separation and composition of concerns facilitates either the definition of successive versions in the time, or different variants for different target platforms or user requirements. Moreover, software can also evolve at runtime in order to dynamically consider new requirements or context changes. This last change could be managed by self-adaptive platforms. These platforms enable software systems to add and remove some of its elements at runtime.

The term DSPL was introduced in 2008 by Hallsteinsen *et al.* [HHPS08]. In this paper authors introduce a new trend in research that aims at using the principles of traditional SPL to build products that can be adapted at runtime depending on the requirements of the users and the conditions of the environment. Because of its novelty, literature is yet scarce with regard to concrete DSPL approaches. Nevertheless, DSPL challenges can be faced using already mature approaches in software engineering, specially when dealing with software adaptation (e.g. AOM, service and component-based runtime platforms, MDE, ECA rules). This chapter aims at studying different approaches for design and runtime adaptation that can be used in the context of DSPLs. We propose a classification and a comparison of existing work. Afterwards, we refine our proposal by concretizing the research goals that fulfill the gaps current approaches present.

Structure of the Chapter

This chapter is organized as follows. In Section 3.1 we discuss the adaptation and present the two main types of adaptation considered for this survey: adaptation at design time, and adaptation at runtime. In Section 3.2 we elaborate on the criteria and classification as well as the description of the approaches surveyed. We discuss additional criteria for the comparison that, although not present in all the works surveyed, is worth mention for the relevance in the context of DSPL. At the end of the section we present a summary of the results. Next, in Section 3.3 we revisit the research goals presented in Chapter 1 to better position the contributions of this dissertation. Sections 3.3.1 and 3.3.2 elaborate on the need for a unification of adaptations and the challenges of defining and implementing such unification. Finally we conclude in Section 3.4 with a brief summary of the survey presented in the chapter.

3.1 DSPL and Software Adaptation

DSPLs focus on the development of software products that can be adapted at runtime depending on the requirements of the users and the conditions of the environment. Indeed with the increasing need of self-managed systems and the emergence of multi-scale environments, software developers need to cope with variability and adaptations. Software must be developed to be adapted and reconfigured automatically on heterogeneous platforms in accordance with the unavoidable evolution of information and communication technologies. Therefore, the adaptation is now considered as a first-class problem that must be taken into account throughout the software life-cycle [1]. In order to position our work, we start by presenting the definition of adaptation, and its implications at design time and at runtime respectively.

3.1.1 Adaptation

Software adaptation is strongly related to software evolution. Both processes deal with the modification of an application. However, as presented on [OGT⁺99, OMT08], such processes are complementary with regard to the focus and tasks that they involve. A software evolution is understood as the modifications done to a system over time. The adaptation is more related to the processes needed to modify an application including: detecting events and information that may lead to a change, planning a set of changes, and performing those changes on the application. A well-known reference of this model is the one presented by IBM [IBM06] known as MAPE for the phases it includes: Monitor, Analyze, Plan, and Execute.

The IBM model has been defined for control loops at runtime. However, software can be adapted either at the design phase or at the runtime phase. For each phase, dedicated technologies are used to specify and realize the adaptations.

3.1.2 Static and Dynamic Adaptation

Another characteristic of adaptation is the moment of time in which the business code is adapted. Literature in general refers to two types of adaptation: *static* and *dynamic*.

Static adaptation refers to the changes that are performed during development, compile or load time. During the development for instance, design languages provide adaptation mechanisms such as inheritance or composition. A slightly different approach is to adapt the application at compile time. One of the better-known examples that allow this type of adaptation is AspectJ [KHH⁺01], an aspect-orientation extension of java. With AspectJ, crosscutting features can be defined and woven with original business code at compile time. Load-time adaptation is also considered as a way of static adaptation. This kind of adaptation consists in waiting until the loading of an application to decide which components are employed. For example, as explained in [MSKC04], when an application requests the loading of a new component, decision logic might select from a list of components with different capabilities or implementation, choosing the one that most closely matches current needs.

Dynamic adaptation refers to changes that happen while the applications are being executed. This means that elements of the application such as algorithms or structures can be replaced or modified during execution without necessarily having to halt and restart the application [MSKC04]. Typically, at runtime, applications are based on platforms that support dynamic adaptation. For instance, certain CBSE platforms provide APIs to dynamically change connections between running components.

3.1.3 Adaptation at design time

In this dissertation we intend to face the challenges for the adaptations at design time. From an SPL perspective, it does not matter if the adaptation takes place at the level of models or by modifying the source code because in both cases, the adaptations are part of the derivation of a product from a user-defined configuration. For this reason we group the static adaptations techniques under the same SPL process of application engineering at design time. We consider an **adaptation at design time** as any modification performed over an application that starts and ends before the application has been deployed and its execution has actually taken place.

3.1.4 Adaptation at runtime

In a similar way as for the adaptation at design time, we group the different approaches for achieving dynamic adaptations under the same process of application engineering at runtime. We consider that independently from the approach, all share the same objective of changing the applications dynamically. Consequently, we define the notion of **adaptation at runtime** as any modification of the application that takes place during its execution.

3.2 Design and Runtime Adaptation: Approaches and Mechanisms

In this section, we survey different approaches that are related to the definition and implementation of SPLs for deriving adaptive systems. We classify the approaches based on the type of

adaptation they support. There are three main groups: (1) those who specifically deal with design time adaptations, those who specifically deal with runtime adaptations, and (3) those who try to cover both processes at the same time.

3.2.1 Design time adaptation approaches

This category includes all the approaches where the adaptation takes place before the deployment of the software artifacts that constitute the application. Usually, approaches in this category present a complete derivation process that uses variability and variability constraints for product derivation, as well as mappings and code generation processes for building the concrete artifacts that constitute the software products.

Design time criteria

Each approach in the design time category is classified regarding two main criteria: (1) the mechanisms used, and (2) the scope of the adaptation. These criteria are detailed in Table 3.1.

Arboleda *et. al.* [ARCR09] propose a Software Product Line based on Models. Their approach uses variability and constraint models in combination with AOP to derive products that integrate different concerns into a single product. All the operations to derive a product occur at design time (merging models and code generation). Regarding the scope of the adaptation, this approach emphasizes on the adaptation at the level of models and code. The adaptation modifies the models used to represent the product. Besides, since they use AOP, source code is also the target of modifications during the derivation process. In terms of mechanisms employed, the approach defines the variability of the family of products, and uses MDE to define intermediate models and AOP to compose model transformation rules.

Clarke [Cla02] discusses about composition mechanisms needed in particular in the UML metamodel to align requirements and objects. She proposes to add a specific composition relationship among elements, so that, common elements in different models (regarding the same requirements) can be identified and composed. Using this composition relationship, she discusses two ways of performing composition: *merging* and *override*. As in the previous case, the composition takes place at design time, among the different UML models. Since the result of the composition proposed are new UML models, the scope of this approach are the models. Regarding the mechanisms used, the approach uses MDE for representing the models and for defining the composition mechanisms that correspond to the same requirements.

Czarnecki and Antkiewicz [CA05] propose a mapping from feature models to application models. The idea is to allow the modeler to view directly the assets related to each feature and estimate the impact of selecting/deselecting a given variant. With a particular configuration, a template instance is obtained which represents the selection of the modeler. A template corresponds to design elements like UML diagrams. The approach focuses on design-time derivation since the results of the configuration corresponds to a UML model. Regarding the scope, the mapping of feature models to application models implies that the models and the architecture of the application are modified. Indeed, authors deal with both models and templates at the same time. While models are used mainly to represent variability, templates are used to represent design elements like UML diagrams which define the architecture of the applications being derived. The mechanisms used by the authors combine mainly MDE for modeling the applications and variability management to map such models to features.

Table 3.1: Comparison criteria for design time approaches.

Criteria	Definition
<i>Mechanisms</i>	The mechanisms used for defining and implementing the adaptation. This includes but is not limited to models and model transformations, aspect oriented modeling and model merging, and feature diagrams.
AOM, AOP	AOP and AOM are also commonly used across the different approaches as a way to achieve modularization and adaptation based on the composition of multiple modules. Aspects can be combined with feature diagrams as a way to deal with variability and constraints among several products in software product lines.
MDE	MDE is widely accepted in design adaptations. One common strategy among several approaches is the use of model transformations and code generation to automate the development of applications
Variability Management	Several approaches are based on SPL and variability management to configure and build families of similar products. Adaptation is achieved by switching across several product configurations. Typically, variability management is combined with other mechanisms like models or aspect oriented programming.
<i>Scope</i>	By scope we mean the granularity of the adaptation, it varies from fine grained granularity, as modifying methods and parameters, to coarse grained granularity, when doing architectural modifications like changing component bindings.
Model	Several approaches use models to represent applications at both design (most MDA approaches) and runtime (models at runtime). We say that the scope of the adaptation is a model if the elements that get modified because of the adaptation are models.
Architecture	For approaches where applications are based on architectural paradigms like components, services, processes (e.g. CBSE, SOA, BPEL), we evaluate if the adaptation has an impact on the structure or behavior of the elements that constitute the architecture of the application.
Code	Finally, we say that the code is the scope of the adaptation when parts of the source code (e.g. classes, methods, attributes) implementing the applications are impacted by the adaptation. For example, AOP approaches define explicit pointcuts on the source code, to extend them with added functionality.

Kienzle *et al.* [KAAK09] define aspects over UML diagrams. They use class diagrams for structure modeling, as well as sequence and state diagrams for behavior modeling. Afterwards, their approach proposes a weaving that composes such models, including dependency chains among them. Since the result of the weaving is a new model, that can be used for simulation or code generation, the approach is centered on design-time adaptation. The scope of the approach are the models that get composed thanks to the weaving of the aspects defined. The mechanisms used by Kienzle *et al.* combine AOM for defining the aspects, and MDE for the creation of class, sequence, and state diagrams.

Perrouin *et al.* [PKGJ08] propose a model-based approach at design time for product derivation in SPLs. They start with a feature model, and for each feature, there is a partial model. A merging operation takes place in order to merge the partial models of the different features selected for a particular configuration. The adaptation target corresponds to the models, since the

merging that combines different features results in a merged model. Regarding the mechanisms, Perrouin *et al.* combine variability for defining the feature model with MDE for defining the models and the merging operation.

Reddy *et al.* [RCF⁺06] present an aspect based approach for model composition. They introduce a base algorithm and different directives. The directives are used when the composition algorithm yields to incorrect results. Directives modify default composition rules, so that developers have finer-grained control on how the elements of the models are composed. Their approach focuses on aspect models and design composition. The adaptation scope in the approach are the models obtained after modifying the composition rules. The approach is mainly based on AOM and MDE for defining the elements to compose, the base composition algorithm, and the directives that modify the rules of such algorithm for the cases where there is a conflict.

Sánchez *et al.* [SLFG09] define a language for composition of assets in SPL called VLM4. This language can be used to generate model transformation rules that automate the derivation process at design time. The approach aims at creating model transformations that in the end produce as a result a model that represents the SPL configuration. Authors use variability for defining the assets to compose and MDE transformations that are generated from their own language.

Voelter and Groher [VG07] propose a complete model driven SPL where aspects are used to realize variability. AOM and AOP are both used to introduce variability at the level of models, and later at the level of generated code. The scope of this approach covers both the models when using AOM, and code when the adaptation takes place through AOP.

Wagelaar [Wag08] proposes a way to take modularization to the level of rule-based model transformations. He proposes a composition of rules so that different independent transformations can be combined and scale up to a larger model transformation. Since combining transformation rules is equivalent to modify the model that results from executing them, we consider that the scope of the approach are the models. Wagelaar focuses on MDE techniques and particularly in the combination of *model to model* transformations.

Van der Storm [ds07] proposes a formal model to bridge domain and solution models in product line engineering. His approach is based on dependency graphs that map concepts from feature diagrams to software artifacts. As with the previous approaches, the domain and solution models are used at design time during the development process of the applications. The approach by Van der Storm has also the models as its scope, since its main goal is to define a formal model which allows adaptation based on feature selection (domain problem) into the software artifacts (solution). Regarding the mechanisms, Van der Storm approach is based on variability management and SPL techniques.

Finally, Lee *et al.* [LBT09] work on product derivation by means of an aspect oriented solution to the problem of feature dependencies. Aspects are used as a way to separate feature dependencies from feature implementations. This approach attacks directly the source code of the applications being implemented, by defining aspects that are woven depending on the feature dependencies. Aspects are combined with feature diagrams as a way to deal with variability and constraints among several products in software product lines

Summary of design time approaches

As we have shown, work on adaptation at design time is prolific. Different scopes are defined as well as different mechanisms for achieving such adaptations. The results of this first group

are summarized in Table 3.5. The first column contains the reference of the work. We have two main columns for **Scope**, and **Mechanisms**. Each main column contains their respective criteria subgroups. Additionally, we have added an extra column called **Domain** to indicate, if available, the kind of domain of application (e.g., mobile computing, embedded systems, smart houses, and multimedia). We use a check mark (✓) if the approach proposes solutions or deals with the different criteria, and a dash (–) in the opposite case.

From this first group of approaches that focus on design time, we can observe that most of them include complete derivation processes by defining the variability of the products at early stages of the development. All of them can be used (at least partially) to produce families of products from different product configurations. They combine variability management with concrete techniques for software development like MDE in the case of a PIM to PSM transformation chain, AOM when modularization and composition are employed at the level of models, or AOP when aspects are woven directly to the source code.

However, due to the lack of support for dynamic adaptations, these approaches only face a subset of the challenges for DSPLs. Concretely, there is no support for adaptations at runtime. This implies that the configuration defined for each product at design time does not exist when the product is executed. We consider that a complete approach for DSPL should not only deal with the design adaptations this group of approaches support, but also with the requirements for adaptations during the execution of the applications.

Table 3.2: Summary of the design time adaptation approaches.

Reference	Scope			Mechanism				Domain			
	Model	Architecture	Code	AOM	AOP	MDE	SPL Variability	Mobile Computing	Embedded Systems	Smart House	Multimedia
[ARCR09]	–	✓	✓	✓	✓	✓	✓	✓	–	✓	–
[Cla02]	✓	–	–	–	–	✓	–	–	–	–	–
[CA05]	✓	✓	–	–	–	✓	✓	–	–	–	–
[KAAK09]	✓	–	–	✓	–	✓	–	–	–	–	–
[PKGJ08]	✓	–	–	–	–	✓	✓	–	–	–	–
[RGF ⁺ 06]	✓	–	–	✓	–	–	–	–	–	–	–
[SLFG09]	✓	–	–	–	–	✓	✓	–	–	✓	–
[VG07]	✓	–	✓	✓	✓	✓	✓	–	–	✓	–
[Wag08]	✓	–	–	–	–	✓	–	–	–	–	–
[dS07]	✓	–	–	–	–	✓	✓	–	–	–	–
[LBT09]	–	–	✓	–	✓	–	✓	–	–	–	–

3.2.2 Runtime adaptation approaches

This category includes all the approaches where the adaptation takes place during the execution of the application. Usually, approaches in this category aim at defining adaptation rules and at taking advantage of technologies that allow for runtime modifications of the applications.

Table 3.3: Comparison criteria for runtime approaches.

Criteria	Definition
<i>Mechanisms</i>	In the same way as for the design adaptations, the runtime mechanisms cover the different techniques and approaches used to achieve the adaptation.
ECA Rules	Event condition-action (ECA) rules are mechanisms employed when it is necessary to trigger a particular action in response to events. This type of mechanisms are mainly used to model an adaptation in response to changes in the execution context.
<i>Context Awareness</i>	Context awareness refers to the capability of the systems to react to changes in their environment [Bro96, DAS01]. Context information refers to all the information available in the environment when applications are being executed, and that may affect the structure or behavior of them. Examples of context information include location, temperature, hardware constraints, user preferences and personal information, time, etc. For this criteria we identify the approaches that effectively use context information as input in the decision making process particularly in the case of runtime adaptations.

Runtime reconfigurations are performed in different ways and using a variety of tools (i.e., introspection and intersection, meta object protocols, models at runtime, runtime platforms based on services and/or components).

Runtime adaptation criteria

To properly identify the different mechanisms used in this category, we have slightly modified the criteria. In addition to the elements previously identified for the design time approaches that remain valid, we have added the ECA rules in the *Mechanisms* criteria for approaches that are based on rules, and conditions. Additionally, we have added a new criterion called *Context Awareness*, that is used to classify the approaches that use context information to trigger the process of adaptation dynamically. Table 3.3 details the new criteria for runtime approaches.

In [BJC05] Batista *et al.* introduce their framework called PLASTIK. It allows both the definition of runtime components as well as their dynamic reconfiguration. It is a combination of an ADL for describing architectures, with a reflective component model. Runtime adaptation is achieved through reconfigurations that can be of two types: programmed reconfigurations, which are foreseen at design time, and ad-hoc reconfigurations, which cannot be foreseen at design and that are controlled with the help of invariants in the specification of the system. PLASTIK uses models at runtime together with ECA rules for achieving the adaptation. The component-based platform called OpenCOM and the ADL with extensions allow developers to define ECA-like conditions on which reconfiguration actions take place.

In [BSBG08], Bencomo *et al.* propose software product lines for adaptive systems. In their approach, a complete specification of the context and supported changes has to be provided thanks to a state machine. Each state then represents a particular variant of the system and transitions between states define dynamic adaptations that are triggered by events corresponding to context changes. The work of Bencomo *et al.* defines reconfiguration policies that take the form of *on-event-do-action*, where actions are changes to component configurations and events represent the notifications arriving from the environment and processed by a context engine.

David and Ledoux [DL06] present SAFRAN, an extension of the FRACTAL component model in order to modularize dynamic adaptations using aspects. The aspects represent reactive adaptation policies which trigger reconfigurations based on evolutions of the context. The adaptation is defined using FScript, a language developed to write Fractal component reconfigurations. They use WildCAT [DL05] to detect external events. WildCAT models context as a set of domains. Each domain represents a particular aspect of the context information. The information itself is modeled as pairs (name, value) inside every domain. The information changes over time, and this changes generate events, which are used by SAFRAN to trigger the adaptation process.

Pessemier *et al.* [PSDC08] introduce the Fractal Aspect Components (FAC). FAC is a model for software evolution that benefits from AOSD and CBSE. In FAC, there can be aspect components, which are regular Fractal components that embody an advice code. The adaptation takes place by adding or removing components (aspect or regular) to running applications. The runtime reconfiguration is, as in the previous case based on the support provided by the FRACTAL component model. Since FAC is based on Fractal components and use Fractal dynamic capabilities to define adaptations at the architecture level, the scope of the adaptation corresponds to the architecture of the component-based application that gets modified through FRACTAL reconfigurations.

Zhang and Cheng [ZC06] introduce a model driven process for the development of dynamic programs. Formal models are created for the behavior based on states. They separate adaptive from non-adaptive behavior in programs, making the models easier to specify.

Trinidad *et al.* [TCn07] propose a mapping from feature models onto component models. Basically, for each feature, there is one component who implements it. There is additionally a component called the *configurator* which is in charge of creating the bindings to form the desired architecture that represents a particular feature configuration. The *configurator* acts at runtime and is able to activate components linked to non-core features. The approach focuses on the relationship of features and software components. Adaptation takes place thanks to a *configurator* component that modifies the architecture of the applications components and bindings at runtime.

Finally, Dinkelaker *et al.* [DMFM10] propose a dynamic software product line using aspect models at runtime. They use aspect models to define features and feature constraints. Their approach mixes SPL principles of product derivation with the notion of dynamic variability. They distinguish static variability from dynamic variability, and for the latter one, they use dynamic AOP for the implementation. Their approach links what they call *dynamic features*, representing late variation points in an SPL, to dynamic aspects.

Summary of runtime adaptation approaches

Table 3.4 summarizes the approaches of the second category. We have added the ECA rules criterion for the mechanisms, and the context awareness. In the same way as for the previous group, a check mark (✓) indicates if the approach proposes solutions or deals with the different criteria, and a dash (–) in the opposite case.

In this second category we find approaches that offer great support for dynamic adaptations. They are usually based on platforms with reflective capabilities that they use to modify applications at runtime. Some of the approaches use context information and event rules to trigger adaptations, whenever a context occurs. However, such approaches do not offer support for design adaptations. Their starting point is usually a set of applications already developed (by

hand in most cases), and they are not interested on automating the development process before the execution. A DSPL approach can take advantage of the dynamic capabilities and runtime adaptations offered by the approaches in this category. Nevertheless, the lack of adaptations at design time, make us consider that this second category of approaches are only suitable to face a subset of the challenges for DSPLs. They have to be complemented to offer support for the initial development process that takes place before the execution.

Table 3.4: Summary of the runtime adaptation approaches.

Reference	Scope			Mechanism					Context	Domain			
	Model	Architecture	Code	AOM	AOP	MDE	SPL Variability	ECA Rules		Mobile Computing	Embedded Systems	Smart House	Multimedia
[BJC05]	–	✓	–	–	–	✓	–	✓	✓	✓	–	–	–
[BSBG08]	–	✓	–	–	–	✓	–	✓	✓	–	–	–	–
[DL06]	–	✓	–	–	✓	–	–	–	✓	–	–	–	–
[PSDC08]	–	–	–	–	–	–	–	–	–	–	–	–	–
[ZC06]	✓	–	✓	–	–	✓	–	–	–	✓	–	–	–
[DMFM10]	–	–	✓	–	✓	–	✓	–	–	–	–	–	–
[TCn07]	–	✓	–	–	–	–	✓	–	–	–	✓	–	✓

3.2.3 Mixed adaptation approaches

In this category, we analyze a third group of approaches that propose mixed solutions. Such approaches include some of the characteristics we have found separately in the two previous groups, but combined in order to provide support for adaptations to be performed at design time and at runtime.

Adaptation criteria

Since this category is a combination of the two previous categories, we use the same criteria specified for the previous groups.

In this category, we find the work by Bastida *et al.* [BNT08]. The authors introduce an approach for context-aware service composition. They propose a methodology of six processes aimed at defining an executable model composed of several services with a particular workflow, which represents a set of variants chosen for several variation points. Afterwards, the composition can take place at runtime based on ECA rules, in order to connect to new services. Regarding the mechanisms for runtime, the authors use context information defined as a dynamic property that may depend on an underlying protocol. The property is used in a predicate which is expressed in their particular ADL. This ADL associates a programmed reconfiguration action to the property. This leads to context-based reconfigurations that are triggered through a change in the dynamic property.

Apel *et al.* [ALS08] introduce the notion of Aspectual Feature Modules. They aim at combining feature oriented programming (FOP) and AOP to implement feature models when required.

Their approach uses classic feature modules for non cross-cutting concerns and AOP for special cases. Although not specified, their approach could eventually use dynamic AOP, making adaptation possible at both design and runtime. The scope of the adaptations in this approach is the source code where aspects are woven. Regarding the mechanisms, Apel *et al.* base their approach on the combination of variability to build families of similar products, and AOP for the cross-cutting concerns.

Lundesgaard *et al.* [LSO⁺07] propose an approach formed by two parts: an MDA transformation chain for building adaptive applications, and a middleware system to make decisions about adaptation based on Quality of Service (QoS) information. At runtime, the model is causally connected with the application it represents. The adaptation takes place by choosing the right configuration by modifying the application model. Then, the application *absorbs* the changes of the model. No details are given as to how this last process actually takes place. The approach uses QuAMobile, a context and QoS-aware middleware that identifies and chooses the *best* variant configuration for the current context and available resources. Such middleware works as a set of plugins that can be plugged in and out. In particular there is one plugin called *Context Manager* in charge of managing context information.

In [MBJ08], the authors present K@RT, a framework for dynamic product lines based on aspects and models. They use models at runtime for dynamic variability and deal with the combinatorial explosion. In [MFB⁺08] a close related work presents the strategy for dynamic adaptation. Models are kept at runtime as part of the application being executed, then, the target configuration is calculated for the current conditions of the executing environment. Having both models, current and target model, a difference is computed, and from this difference, a reconfiguration script is generated that takes the current configuration to the target configuration. Also from the same authors, Morin *et al.* [MBNJ09] present an evolved version of this approach when aspects are formed by advice, pointcut, and weaving directives. They use dynamic reconfigurations to modify the model of the application, the architecture of the application itself, and the code generated from the model. With respect to the scope of these approaches, the main element of the adaptation is the model. However, using the models as starting point, they also introduce strategies for source code generation. Regarding context awareness, in [MFB⁺08] authors define an adaptation model which captures all the information about the dynamic variability and adaptation of their adaptive system. Among the elements that conform to such a model, they include a context model which is a minimal representation of the environment used to define adaptation rules. In a similar way, in [MBNJ09] authors define aspect models which may include a context that they describe as a slice of the environment. No further details are given concerning the context management or the frameworks used for context aggregation.

Phung-Khac [PK10] proposes the adaptive medium approach for developing adaptive distributed applications. His approach proposes a development process where business logic is separated from the adaptation aspects of the applications. The business logic is refined in different configurations that are treated as different members of the system family. Like this, the adaptive medium approach extends the feature modeling method. On the other side, the adaptation aspects define architectural models and are in charge of adapting the business logic dynamically. In his approach, the desired adaptive application is specified at a high abstraction level and then is refined towards the implementation level. For the runtime adaptation, the approach uses FRACTAL software components. The architectural models generated by the refinement process are embedded into the adaptation control to perform the reconfigurations. Finally, since the applications are component-based, we consider that their adaptation scope corresponds to the architecture of the applications that get modified during the reconfigurations, and additionally, to the generated code that is obtained from their refinement process.

Finally, the project ECaesarJ [NnNG09] represents an approach to have design and runtime adaptations based on the programming language. ECaesarJ is an aspect oriented language that is based in its predecessor CaesarJ [AGMO06]. The language aims at facing the challenges of feature decomposition. To do so, it improves modularity of object oriented programming languages by providing extension and composition mechanisms. At the core of ECaesarJ there is the concept of *virtual classes*. A virtual class can be redefined in subclasses by adding new methods fields and inheritance relationships. This allows, for example, to define features as extensions of other features. For the composition of features, ECaesarJ supports *mixings*. It represents a form of multiple inheritance when all inherited declarations of virtual classes with the same name are composed. ECaesarJ also offers support for events and state machines. The events are used to represent explicitly behavioral abstractions. An event is composed of a source and destinations. Examples of such events include implicit join points of the source code for example method calls, attribute value changes, but can also be explicitly defined by the programmer. The state machines are supported in ECaesarJ to make it possible to organize the event handling. Because it is based on CaesarJ, the weaving of aspects takes place through several deployment process which include design time and runtime deployments. At design, the aspects are woven in a similar way as in any AspectJ-based approach. At runtime, the aspect which is defined as in java, can be instantiated at any moment. However for the weaving to actually take place the aspects deployment act as a wrapper that intercepts the calls to the business objects to enrich them with the advice code. Since the language is basically focused on java code, their mechanisms include aspect oriented techniques as well as direct code manipulation. The scope of the adaptation achievable with ECaesarJ is the business code itself that gets modularized and composed by the aspects defined with the ECaesarJ language.

Summary of mixed approaches

Table 3.5 summarizes the approaches of the third category. The criteria is the same that we have used for the runtime approaches. In the same way as before, a check mark (✓) indicates if the approach proposes solutions or deals with the different criteria, and a dash (–) in the opposite case.

This third category of approaches is the most interesting one for the development of DSPLs. The approaches in this category offer support for both design and runtime adaptations. Some of them use variability management and context information as well as models at runtime, reflective platforms, or dynamic aspects that allow them to have both source code manipulation processes for the design adaptations and dynamic reconfigurations for the runtime adaptations. Some of them also use variability management for modularizing and defining adaptations, and context information to define concrete events at runtime for adaptations. There are other approaches based on programming languages that focus on modularity and propose constructs tailored for feature decomposition.

However, for a complete DSPL, we consider that there are still two main issues missing. First of all, the approaches in this group do not offer a complete development life cycle from feature modeling and configuration to runtime adaptations. This means that design and runtime adaptations are realized through completely independent process that do not have many elements in common. Moreover, to the best of our knowledge, none of the approaches offers a unified representation of adaptations. Assets used for building applications are defined and treated in a different manner than assets used to achieve reconfigurations dynamically.

Table 3.5: Summary of mixed adaptation approaches.

Reference	Scope			Mechanism					Context	Domain			
	Model	Architecture	Code	AOM	AOP	MDE	SPL Variability	ECA Rules		Mobile Computing	Embedded Systems	Smart House	Multimedia
[BNT08]	–	✓	–	–	–	–	–	✓	✓	✓	–	–	–
[LSO ⁺ 07]	✓	–	–	✓	–	✓	–	–	✓	–	–	–	✓
[MBJ08]	✓	✓	–	✓	–	✓	–	–	✓	–	–	–	–
[MFB ⁺ 08]	✓	✓	–	✓	–	✓	–	–	✓	✓	–	–	–
[MBNJ09]	✓	✓	–	✓	–	✓	–	–	✓	–	–	✓	–
[ALS08]	–	–	✓	–	✓	–	✓	–	–	–	–	–	–
[CTPRC08]	–	✓	–	–	–	–	✓	–	–	–	–	✓	–
[PK10]	–	✓	✓	–	–	✓	✓	–	–	–	–	–	–
[NnNG09]	–	–	✓	–	✓	–	–	–	✓	–	–	–	–

3.2.4 Summary

We can now summarize the results of the approaches reviewed with regard to the challenges they face and the strengths and weaknesses of each group. In table 3.6 we summarize the findings of each of the three categories previously discussed. We have three main criteria. First, we illustrate if the category support design time adaptations, and if it uses variability management for the derivation process. Second, we illustrate if the approach support runtime adaptations, and if it uses context information for the decision making. Finally, we add a last criteria indicating if the approach offers a unified representation of design and runtime adaptations. In the next section, we further discuss the unification and revisit the challenges for a complete DSPL approach that successfully manages design time and runtime adaptations.

Table 3.6: Synthesis of approaches for DSPL.

Approach	Design Adaptation	Variability	Runtime Adaptation	Context Awareness	Unified Representation
Focus on Design and Product Derivation	Yes	Yes	No	No	No
Focus on Runtime Reconfigurations	No	No	Yes	Yes/No	No
Mixed Approaches	Yes	Yes/No	Yes	Yes/No	No

3.3 Research Goals Revisited

From the results of Table 3.6, we have concluded that the main missing issue relates to the lack of unification between adaptations at design time and adaptations at runtime. For each one, dedicated technologies are used to **specify** and **realize** the adaptations as it has been presented in all of the categories previously reviewed.

As we have pointed out in Chapter 1 both adaptation processes can be understood as the modification of the product being derived by adding and/or removing a certain group of features. It would be desirable to have a unified representation of this modification, so that it can be used at design as well as at runtime.

3.3.1 The Need for Unification

We claim that design and runtime are similar in their definition and their using process, but not in their implementation. Hence, in order to define a complete approach for DSPL, we need a unified representation of adaptations that combines design and runtime in a coherent development process. Design adaptations are often considered to be of completely different nature than runtime adaptations. Design adaptations are motivated by design goals whereas runtime adaptations are motivated by changes of the software environment. Moreover, design adaptations are considered as permanent adaptations that cannot be rolled back whereas runtime adaptations are considered as impermanent. However, whatever the technology and whatever the phase, a software adaptation is always initiated by a particular motivation and is always realized through modifications of some software artifacts. Therefore, from a specification point of view, design and runtime adaptations are not that different. We argue that a single unified language should be provided to specify both of them. Based on this language, a platform should be realized to derive the software products at design time and runtime transparently.

Having only one unified language to specify design and runtime adaptations offers several advantages. First, it formalizes similarities and differences that exist between the two kinds of adaptation. Second, it may serve as a basis to transform design adaptations into runtime ones and vice versa. Transforming design adaptations into runtime one allows one to delay the realization of some design adaptations to the runtime phase. Transforming runtime adaptation into design one prevents the realization of adaptation mechanisms that have been defined regarding specific environment state that may not arise at runtime. Third, unifying the specification of modifications done by both aspects is the first step to compute analysis such as dependency analysis between aspects.

Having a platform that derives the software products at design time and at runtime transparently offers several advantages. First, it supports the whole life cycle from the initial creation of the product (driven by feature selection) to its dynamic adaptation (driven by changes of its environment). Second, it establishes the link between the motivations (feature selection or changes of the product environment) and the adaptations of the software artifacts. Third, it can be used as a way to achieve flexibility in the tradeoff between development cycles that are fully design oriented (without any runtime adaptation), and development cycles that are fully runtime oriented (without any feature selection).

3.3.2 Challenges for DSPL Revisited

Having the unification in mind, we can now precise the goals of the approach presented in this dissertation. We investigate on software engineering techniques for developing and adapting software. Our main goal is to implement dynamic software product lines through the **unification of software adaptations** that allows developers to define and implement adaptations both at design time and at runtime.

In Chapter 1 we identified 4 main goals for our approach: variability management, automated development and correctness, platform independence, and derivation at runtime. Let us now refine those goals and group them properly according to the classification we have introduced for the reviewed works, namely: design and variability, runtime and context awareness, and unification.

Design and Variability

First of all a DSPL needs a design adaptation phase that allows developers to build products through automated processes. These processes need to take into account the variability of the product family as well as further analysis and management for different product configurations. We identify the following challenges for a design adaptation process:

- **Automated Development Process**

An SPL exploits commonalities among a set of software products in order to identify and build reusable assets that can be used to derive new products reducing the effort and time invested when building several products. A DSPL needs to automate the development process of adaptable software.

- **Variability and Correctness**

It is important that products are not only easier to develop, but also that their correctness remains guaranteed. When composing multiple parts to form a single software product, it is possible that two or more of those parts have conflicts regarding the elements where they are going to be composed and the requirements for the composition to take place. In other words, implicit dependencies may exist between different artifacts which may lead to composition problems. A design time adaptation has to exploit variability management in order to define a development process that analyses such dependencies and prevents incorrect products from being derived.

- **Guarantee Platform Independence**

It is also desirable that business concepts about the products to be derived are separated from the details of the underlying platform. The derivation process in the DSPL has to postpone as much as possible the decisions about platform and implementation. This enables developers to define multiple targets and offer support for different platforms.

Runtime and Context awareness

Second, the DSPL has to deal with runtime reconfigurations. For this process, context information has to be used to decide about the adaptation. At the same time, the reconfiguration has to respect the constraints defined during the design with the variability. We identify the following challenges that have to be faced to realize a process of adaptation at runtime:

- **Define and adaptation cycle with well-assigned responsibilities**

An equivalent process of product derivation as the one defined for the design adaptations has to be defined. It has to take as input the running product and its configuration, and has to return a new adapted version of the product. A complete adaptation loop has to be established, by differentiating different sub-process in charge of: monitoring the context information, analyzing and deciding about possible adaptations, and finally, executing the adaptation on the software product.

- **Use context information for the decision making**

A fundamental issue in adaptive software development is the management of context events, and its manipulation in order to modify products dynamically. The DSPL has to take context information into account to decide the appropriate configuration at the right moment when adaptations take place in order to offer a better experience to the final users.

- **Extend the concept of feature at runtime**

Since products in the SPL are described as a set of selected features, an important challenge to achieve dynamic product derivation is to define a way to maintain, and update, the state of a product in terms of the features it is supporting at a given moment of its execution. Furthermore, this information has to be used, in the same way as in the design time adaptations, to guarantee that the product will respect the constraints of the product family after the adaptation has taken place.

A unified representation and management of design time and runtime adaptations

Finally, to provide the unification of adaptations at design time and at runtime, the DSPL has to define a language and use an underlying platform that allows definition of adaptations independently from the moment when they take place. This would allow developers to define only once any adaptation, and use it independently at design for building a product, or at runtime for adapting an existing product.

Additional Properties for a DSPL

In addition to the challenges for the design phase, the runtime phase, and the unification of adaptations, we consider that any framework for developing DSPLs has to consider the following properties.

- **Extensibility**

Extensibility is a property of highly importance in any SPL. Since requirements are evolving constantly, it is desirable that SPLs can be extended or adapted to support the derivation of new products, different execution platforms, or new functionalities required by different stakeholders. This fact is reinforced by [PBL05] when authors define domain and application engineering processes. These two processes are usually implemented in iterative developments cycles. This practice intends to exploit the complementary nature of each process. For example, during the application engineering, it is possible to identify new requirements. Those requirements can be supported by the existing DSPL in a new iteration, by creating their corresponding assets. This allows the SPL to evolve and extend its scope over time. DSPLs are no different than traditional SPLs regarding the need for extensibility. It is important to provide the mechanisms to extend the scope of the product family and support new functionalities regardless of the derivation time.

- **Scalability**

In any SPL, one of the biggest challenges refers to the management of the combinatorial explosion of product configurations. The size of a product family can grow exponentially when features are added. Larger product families represent a challenge in terms of scalability and performance. In an approach for DSPLs, it is necessary to consider this issue because part of the management of the product family is postponed at the execution of the different products. Calculations over larger product families performed at runtime may have an impact on the adaptation and the overall performance of the products.

- **Runtime History**

When a product is adapted at runtime, it changes its configuration. If such changes include the deletion of several parts of the product, then such modifications have to remain available. Like these, products can be able to go back to a previous state before one or several adaptations have taken place. A DSPL has to take into account this kind of changes.

- **Usability**

Finally, another property for an approach in DSPLs is usability. By usability, we mean the difficulty encountered by newcomers when starting to use a new framework for DSPLs. This can be related with the changes in the development process, specially when there are automated parts that are mixed with manual parts; and also, it can be related with the use of new languages for modeling the different assets that are combined to produce the software products. We consider that a framework for DSPL has to remain usable, for the automation to have a positive impact on the effort and time invested when building individual software products, regardless of the changes on the development process introduced by the framework.

3.4 Summary

In this chapter we have surveyed several approaches in literature that are close related to the main contributions of this dissertation. We have reviewed an important number of research works that use a variety of technologies (i.e. MDE, SPL, AOSD, CBSE) in order to build software and/or adapt it at runtime. We made a classification of the approaches surveyed. This classification has been used to concretize the main objectives of this dissertation that were briefly introduced in Chapter 1.

This chapter concludes the second part of this document dedicated to the study and analysis of the background and state of the art. The next three chapters describe the contributions of this dissertation. We start with a global overview of our approach in Chapter 4, to later concentrate on the details of the processes for design time and runtime adaptations in Chapters 5 and 6, respectively.

Part II

Contribution

Chapter 4

Dynamic Software Product Lines: Our Approach in a Nutshell

Contents

4.1	CAPucine, A Global Overview	52
4.2	Domain Engineering	53
4.2.1	Roles	53
4.2.2	Models as Assets	54
4.3	Application Engineering	56
4.3.1	Design Phase: analysis, composition and generation	56
4.3.2	Runtime Phase: transformation and runtime reconfiguration	56
4.3.3	Facing the Challenges for DSPLs with CAPucine	57
4.4	Summary	57

Nowadays, applications, and especially mobile applications, need to provide means to modify their structure and their behavior dynamically. That is because current technologies and improvements in hardware enhance software developers to build applications that benefit from the information and events available at runtime. Dealing with this problem has been the focus of our approach for DSPL, which constitutes the main contribution of this dissertation. We have developed a framework for developing DSPLs called CAPucine for *Context Aware Software Product Line*. CAPucine provides a derivation process that can be performed at early steps of the development as well as during the execution of a given product. We aim at providing a framework for DSPLs that successfully faces the challenges presented in Chapter 3 by allowing products to be built at design time and adapted at runtime using the same set of assets. The purpose of this chapter is to give a global overview of the process of implementation and utilization of CAPucine.

CAPucine has been divided in several phases, and for each phase, there are different steps and associated roles. Here we present a global view of the different phases and the way they are structured to have a complete process of product derivation. To face the challenges of DSPLs, we present CAPucine using the processes previously identified for a typical product line: *Domain Engineering* and *Application Engineering*. As introduced in Chapter 2 domain engineering refers

to the definition of commonalities and variabilities of the SPL and the construction of the corresponding assets. Application engineering refers to building software products by reusing the assets identified and developed during the domain engineering.

In our case, during the domain engineering process, we create the language that allows design and runtime assets to be defined in a unified manner. Every product in the DSPL is formed as a combination of such assets. Afterwards, during the application engineering process, we differentiate two types of product derivation: one for design time adaptations, and one for runtime adaptations. Both types of derivation have the same effect: adapt an application. Both types of derivation also share elements regarding the input: a set of selected features as defined in the domain engineering processes. However, each derivation uses different technologies for performing the adaptation at design and at runtime respectively. This separation together with the unified language, allows us to face the challenges for design time adaptations, runtime adaptations, and the lack of unification, identified in Chapter 3.

Here we present a global overview of both processes. We enumerate the elements of the DSPL architecture and make emphasis on the roles, the types of assets used, and the derivation processes to follow in order to successfully derive adaptable products. Next, in the following two chapters of this dissertation we present in more detail the assets and the derivation at design time and at runtime.

Structure of the Chapter

The remainder of this Chapter is organized as follows: in Section 4.1 we present the global picture of CAPucine, covering all the phases and both architecture and product derivation lifecycle. Next, in Section 4.2 we describe the process of Domain Engineering, making special emphasis on the types of assets employed. In Section 4.3 we focus on the development cycle using assets. Finally, in Section 4.4 we present a summary of the global architecture of the SPL.

4.1 CAPucine, A Global Overview

A Dynamic SPL differs from a traditional SPL with respect to the architecture and development cycle. Such differences are reflected in terms of: (1) the different roles and responsibilities, (2) the type of assets identified and built in the domain engineering process, and (3) the derivation process. Figure 4.1 presents a global view of CAPucine. It is divided in two parts. In the top of the Figure, we find the elements for the domain engineering processes. We base ourselves in an MDE approach where the assets are represented as metamodels and models. There are three metamodels in total which represent four different domains: (1) variability (*Features*), (2) application (*Aspects*), and (3) platform (*SCA* and *Java*). This division allows one to have different roles for each domain. For example, an expert in the particular platform can be in charge of defining the platform metamodel or the transformations towards this particular platform, whereas a stakeholder can be only involved in the process of product configuration to select the features she wants for her product.

The bottom of the Figure depicts the application engineering process. Here we define two types of product derivation: one for adaptations at design time, and one for adaptations at runtime. The former one is in charge of building a product, whereas the latter one is intended to modify an existing running product. It is important to notice that such processes are triggered by completely different roles. In the first case, the product is built according to a request done by a developer. In the second case however, there is no human intervention, and the adaptation is triggered by a context event.

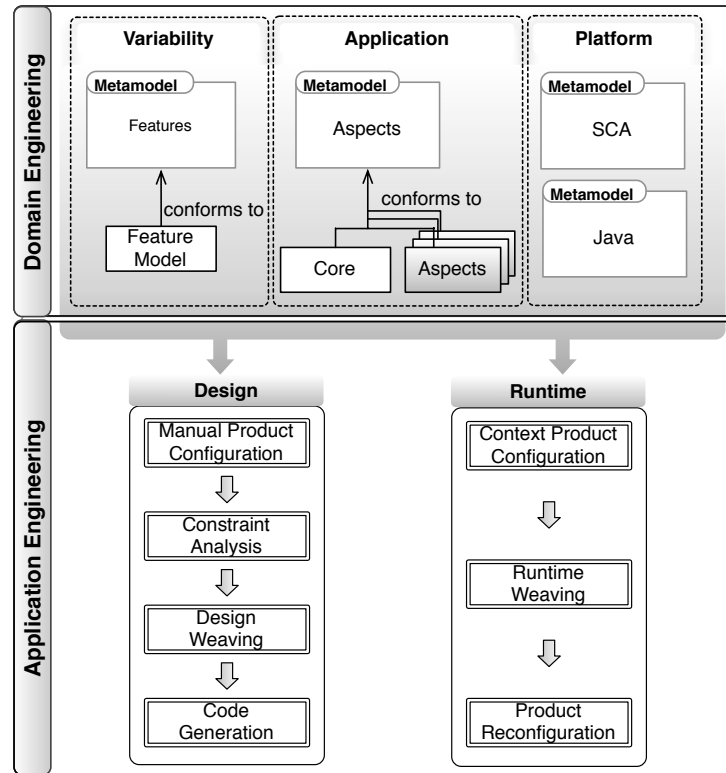


Figure 4.1: A global view of CAPucine.

4.2 Domain Engineering

In this section, we describe the whole process of domain engineering as illustrated in Figure 4.1. As we stated before, this process is concerned with the development of assets. We start by describing the different roles that interact in their creation. Afterwards we detail the assets that in this case are defined as models, metamodels, and transformations.

4.2.1 Roles

The responsibilities for developing each type of asset are shared among four different roles: application architect, platform architect, stakeholder, and asset developer.

- **Application Architect:** The architect is the expert in the particular domain business of applications that are going to be built using the product line. She is responsible for defining the application metamodel. The application metamodel defines the elements that will be used to define every product. Additionally, she also defines the variability model, which is used to define the features and scope of the product family.
- **Platform Architect:** The platform architect is the expert in the target platform where products are going to be deployed and executed. As such, she is responsible for defining the

platform metamodel. Additionally she may participate in the definition of transformations from the application model towards the platform.

- **Stakeholder:** The stakeholder is the final user of the product line. She has all the information about commonalities and variabilities of the product family. Together with the asset developer, she is in charge of reflecting this knowledge in a concrete variability model that conforms to the variability metamodel defined by the architect. Such model is the starting point of every product derivation and represents by itself the whole set of products that the product line is able to produce.
- **Asset Developer:** The asset developer has several responsibilities. First, she helps the stakeholder to define the variability model of the product line. Afterwards, she creates the aspect models that realize each feature identified in the variability model. To do so, the asset developer uses the application metamodel defined by the architect and creates a core model for commonalities and aspect models for variable features. Additionally, the asset developer also defines a series of compositions and transformations that build the product. Such transformations are created using the metamodels, which guarantees that they are generic and reusable for every model that conforms to the application metamodel.

4.2.2 Models as Assets

In addition to the roles, Figure 4.2 also illustrates two levels of assets in the domain engineering process, namely: metamodels and models. For our DSPL, we follow an MDE approach. Consequently, the assets to be created in the domain engineering process mainly correspond to metamodels, models, and transformations. We divide the creation of assets in two different categories which are: abstract assets and concrete assets. Abstract assets are needed in order to create the concrete assets that are used during product derivation to build software products. Abstract assets mainly correspond to metamodels, while concrete assets include models, model transformations and code generators. The abstract assets in CAPucine are described below.

- **Variability for feature modeling**
The first domain modeled corresponds to the variabilities and commonalities of a family of products. The objective is to provide a language that allows us to define such information. We present a feature metamodel that defines the main concepts for features and their relationships.
- **Application for core and aspect modeling**
An application is modeled as a combination of a *core* and a set of *aspects*. The core is modeled using a metamodel that includes the essential elements of a component and service based application. The aspects are intended to enrich or modify the elements of the core by adding or deleting new elements at specific points in the core structure. The core as well as the aspects use an architecture based on components that offer services and require references. The architecture model, as well as the language for defining the core and the aspect models are further explained in Chapter 5.

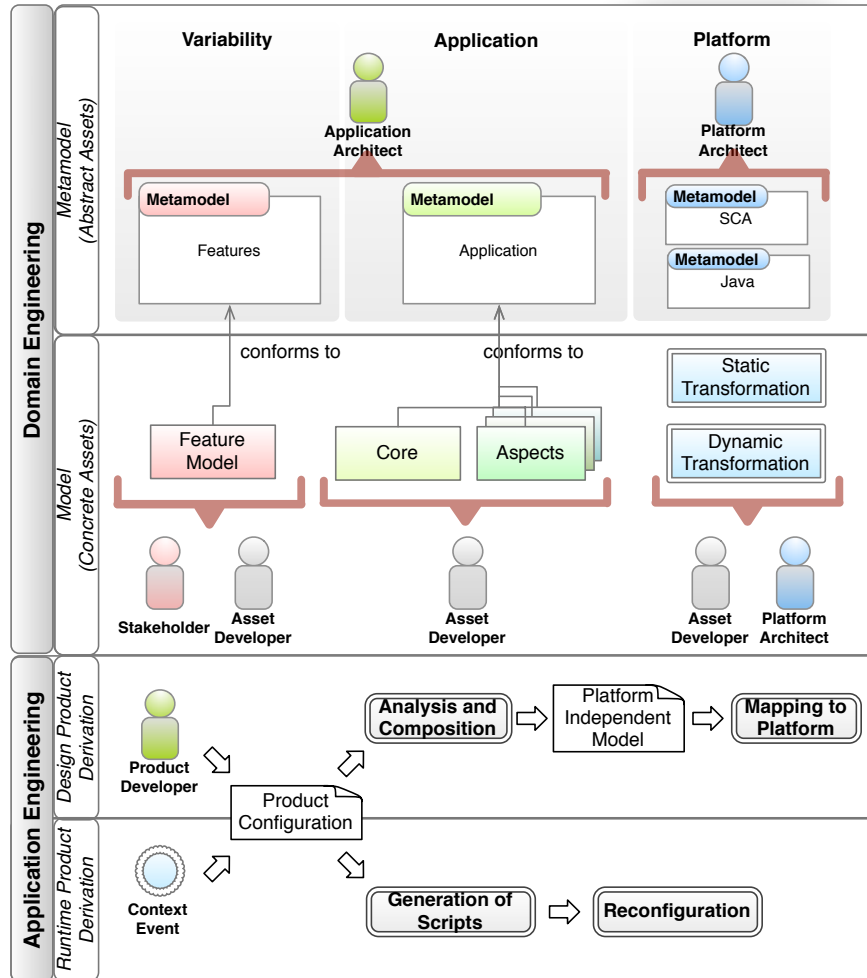


Figure 4.2: Roles in the domain engineering process.

- **Platform for SCA and Java modeling**

We use Service-Component Architecture (SCA) [Ope07b] as target platform. SCA proposes a reconciliation between the Service Oriented Architecture (SOA) and CBSE, by defining a framework for describing the composition and the implementation of services using software components. The motivation behind this choice relates to the capabilities of the platform to reconfigure the applications dynamically. In order to generate the code for the SCA runtime platform we use two metamodels: one for SCA and one for Java. The architecture of the product is defined using an SCA model, which references implementations for every component described. Such implementations are written in Java.

Concrete assets, on the other hand, refer to the models and transformations that are obtained using the metamodels to reflect the products variabilities and commonalities. The variability model together with the aspects that conform to the application metamodel are written by hand by the asset developer. SCA and Java models are automatically generated as a result of model

transformations from the application models, performed during the application engineering.

4.3 Application Engineering

The application engineering process concerns the development of individual products by selecting a subset of features and combining the assets defined during the process of domain engineering. One particular characteristic that distinguishes a dynamic SPL from a traditional one is that there are two different roles in the process of product derivation (Figure 4.2): developer and context events.

- **Product Developer:** this is the traditional role of the developer who wants to build a product by selecting a set of features. She interacts with the product line at design time and the result of her configuration is a product ready to be deployed.
- **Context Event:** this is a dynamic role which also creates a product configuration but this time at runtime. By context we mean every possible information that may affect the product behavior. Examples of such pieces of information are temperature, time, available services and resources, battery level, available memory, etc. The result of its product configuration is a reconfiguration of the product being executed.

For each particular role, there is a different process of product derivation which corresponds to the two CAPucine phases: design and runtime. Both phases share the same idea of building a product from a given configuration, but the implementation and the technologies used to implement them are different.

4.3.1 Design Phase: analysis, composition and generation

In general terms, the design phase corresponds to the building of a product out of a manual *Product Configuration*. We first perform a *constraint Analysis* step which verifies constraints at the level of features and the assets that realize them. Out of this analysis we obtain an order of composition. Next, we have the composition itself, that we call *design weaving*, which puts together the common elements of all products with the features selected for the particular product being built. Once the whole composed model reflecting the choice of the developer is built, the generation takes place. We follow an MDE approach for SPL (MD-SPL). This is reflected in the fact that our assets are models and derivation is a set of consecutive calls to *model to model*, and *model to code* transformations. At the end of the design phase, the code for the product is generated. The design phase is described in detail in Chapter 5

4.3.2 Runtime Phase: transformation and runtime reconfiguration

The runtime phase aims at modifying an application at execution. One advantage of using SPLs is that assets used for both phases are the same. Hence, by starting from the same set of models created and used in the design phase, the runtime phase generates the adequate software artifacts

that change the product dynamically in a process that we call *Runtime Weaving*. To achieve this, in this phase we rely on an adaptation cycle that includes a process of decision making, an executing platform for performing dynamic modifications, and the use of context information to trigger the process of adaptation. The runtime phase is detailed in Chapter 6.

4.3.3 Facing the Challenges for DSPLs with CAPucine

We have presented a global overview of CAPucine, our approach for developing DSPLs. Let us now revisit the challenges presented in Chapter 3 to explain how our approach is intended to face them.

Design and Variability

To face the challenges of design and variability, CAPucine defines the design phase. It is intended to automate the development of software products using as starting point a product configuration. The software products are developed using platform independent aspect models and performing a constraint analysis to guarantee that the product family constraints are respected.

Runtime and Context awareness

To face the challenges of the runtime phase and context awareness, CAPucine defines the runtime phase. It is intended to enable software products obtained in the design phase to be modified during their execution. For that, the runtime phase establishes an adaptation cycle where context information is analyzed to decide if it implies a modification of the product configuration. Such modification is then translated into a reconfiguration of the running product.

Unification

Finally, to face the challenge regarding the unification, CAPucine uses aspect models that can be used at both the design phase and the runtime phase. In the design phase, the aspect models are used in a process of model composition and transformation to obtain source code. In the runtime phase, the aspect models are transformed into reconfiguration scripts that are used to modify a running software product.

4.4 Summary

In this chapter we presented an introduction to our approach for building DSPLs called CAPucine. We emphasized on the two main processes of traditional SPLs and the differences found by extending such processes at runtime. We have presented the different roles involved in the domain and application engineering of CAPucine. We have also introduced the types of assets that we use to develop products and the two phases of product derivation. The following two chapters of this dissertation cover in detail each one of those phases.

Chapter 5

Design Phase: Variability, Application and Platform

Contents

5.1 Introduction	59
5.2 Design Phase Challenges	60
5.3 Domain Engineering	62
5.3.1 Feature Metamodel	62
5.3.2 Architectural Model	63
5.3.3 Aspect Metamodel	65
5.3.4 SCA Metamodel	69
5.3.5 JAVA Metamodel	69
5.4 Application Engineering	71
5.4.1 Constraint Analysis	71
5.4.2 Model Composition	76
5.4.3 Platform Mapping and Code Generation	78
5.5 Discussion	78
5.6 Summary	80

5.1 Introduction

Two of the most important challenges in Software Product Line Engineering concern *variability management* and *product derivation*. The former refers to how to describe, manage and implement the commonalities and variabilities existing among the members of the same family of software products. The latter deals with how to build products starting from a selection of a given set of variable features. A well-known approach to variability management is by means of Feature Diagrams (FD) introduced as part of Feature Oriented Domain Analysis (FODA) [KCH⁺90] back in 1990. An FD typically consists of (1) a hierarchy of *features*, which may be *mandatory* (commonality) or *optional* (variability), and (2) a set of *constraints* expressing inter-feature dependencies.

Nevertheless, deriving a concrete software product from an FD remains a highly complex process. It starts with the *feature configuration* step, which aims at selecting the features to include in the desired product, in strict conformance to the specified constraints. The product derivation process then necessitates the *composition* of the *software artifacts* corresponding to the selected features. This second step may be very challenging, since the impact of selecting a single feature can spread across several phases of the software development process as well as several places in the product itself. In order to enable the automated derivation of a product, it is necessary to specify the *composable elements* that reify each feature. Each of these *composable elements* must include all the information required for the composition among which (1) the locations impacted by the composition, (2) the additional elements to be composed and (3) the changes to perform in order to add support for the associated feature.

To face such challenge, we introduce the notion of aspect model as a way to realize the *composable elements*. Using aspect models enables the derivation of products by means of feature-based software composition. The definition of aspect models relies on Aspect Oriented Modeling (AOM) which consist in using the Aspect Oriented Programming (AOP) principles as part of the Model-Driven Engineering (MDE) development process [Jéz08, FJ09]. Although AOP, was initially developed to deal with crosscutting concerns in the source code, it has recently become an interesting paradigm to be used also in the early steps of software development [KLM⁺97]. MDE, on the other hand, raises the level of abstraction of the development life cycle by shifting its emphasis from code to models [Sch06]. It considers any software artifact produced at any step of the development process as valuable asset by itself to be reused across different systems and implementation platforms. In MDE, models are main software artifacts that provide the full specification of a software system. Each model describes a particular software view at a particular level of abstraction [Zam95]. This use of models follows the well-known separation of concerns principle, which has been proven to provide many benefits, including reduced complexity, improved reusability, and simpler evolution [TOHS99].

This chapter covers the complete phase of product derivation at design time. We start by introducing the variability, application and platform metamodels. We present an aspect metamodel to define high-level aspects and the weaving process at design which links the elements of the aspect at the model level. Then we present the process of product derivation which covers: product configuration, constraint analysis, aspect weaving, and platform mapping.

Structure of the Chapter

The remainder of this chapter is organized as follows. Section 6.2 discusses the need for product derivation using aspect-based composition and present the challenges for the design phase. Section 5.3 presents the domain engineering process and in particular the metamodels defined for the DSPL. Section 5.4 illustrates the application engineering process of the DSPL by describing the steps to derivate a software product. Finally, Section 5.5 revisits the challenges for the design phase and concludes the chapter.

5.2 Design Phase Challenges

In Chapter 3 we have identified three main challenges for the design phase of DSPLs. Such challenges refer to: automated development process, variability and correctness, and platform independence. To face those challenges, in the design phase of CAPucine we define a complete derivation process that starts with a feature selection, also known as product configuration, and that ends with the implemented software product. Feature diagrams are a way to specify and

manage different product configurations. However, the process of creating a product from each configuration is based on: (1) the existence of dependencies between the selected features (i.e. the features in the configuration), (2) the mapping between the selected features and software artifacts that implement them, and (3) the mappings needed to create platform specific artifacts that represent the final product. For this reason, here we present a more detailed definition of the challenges in order to explicitly deal with the problems related to correctness and variability as follows:

1. **Ensure a clear separation of concerns:** Although feature diagrams enable the clean specification of software variability as a feature hierarchy, the mapping that holds between the features and the corresponding software artifacts may prove much more difficult to define. This is especially the case in the presence of *crosscutting* features, i.e., features that are materialized at multiple places in the final product. Possibly complex interactions between features on the one hand, and between artifacts on the other hand, further complicate the definition of the composable elements.
2. **Identify inconsistencies:** When composing multiple features to form a software product, it is possible that two or more artifacts have conflicts regarding the elements where they are going to be composed and the requirements for the composition to take place. It may happen that implicit dependencies exist between artifacts that support independent features in the FD, and conversely. Such *inconsistencies* do not necessarily lead to composition problems but they have to be made explicit.
3. **Derive a suitable composition strategy:** This challenge refers to the use of information at the feature and also at the artifact level to obtain a composition strategy. For example if two features have a dependency, it is necessary to decide which feature is integrated to the product first, so that, the second feature can use the elements it needs from the first one. In other words, features have to be used to establish the order for the composition of assets.
4. **Use platform independent assets:** It is desirable that the artifacts that implement the features are platform-independent, this allows the SPL to have multiple targets and postpone the decision of a particular platform until later steps of the product derivation. Additionally, for every target platform, the SPL has to define the mappings that transform the platform-independent assets in to concrete software artifacts for the platform. This includes for example the mechanisms to obtain the source code that is compliant with the target platform.
5. **Provide an automated development process:** Independently from the techniques used for inconsistency detection and platform independence, the main goal of the design phase is to provide development process that takes advantage of commonalities and variabilities of a set of software products, to build reusable assets. Such assets can then be used to build the products in an automated way reducing the time and effort invested.

In order to face those challenges, we present a design phase that, as introduced in the previous chapter, includes the two main processes of software product lines: domain engineering and application engineering. For the domain engineering process, we present the metamodells for variability, architecture, and aspects. For the application engineering process, we define four phases that constitute the derivation process for creating individual software products: constraint analysis, model composition, model transformation, and code generation. Both domain and application engineering processes are described in the following two sections.

5.3 Domain Engineering

This section describes in detail the metamodels defined and/or used in our approach and some model examples that conform to such metamodels. There are four metamodels: Features, Aspects, SCA and Java. We focus on the former two since they are part of the contribution and were created specifically for the DSPL. For the latter two, we briefly describe them and present external references for further information.

5.3.1 Feature Metamodel

Several works on feature modeling have proposed multiple extensions to the FDs initially introduced in [KCH⁺90]. In [SHT06] Schobbens *et al.* survey different approaches to feature modeling and define an abstract syntax for feature diagrams that eliminate the ambiguity occurring in earlier proposals. They employ a mathematical notation to define the inter-feature relationships. A different approach to deal with ambiguity in FDs is by defining a metamodel like the one proposed by Pohl *et al.* [PBL05]. This metamodel presents two main concepts: *variation points* and *variants*. The metamodel presented in [PBL05] further specializes the relationships between variation points and variants, by classifying the types of relationships that may exist. They define dependencies (*optional* and *mandatory*) and constraints (*requires*, *excludes*). In our case, we have defined a feature metamodel inspired from the concepts that Pohl *et al.* have identified. We define the same concepts and relationships using the Eclipse Modeling Framework (EMF) [The10], but we change the way they are modeled, since EMF does not support the specialization or inheritance of relationships between two different meta-classes. Our feature metamodel is shown in Figure 5.1.

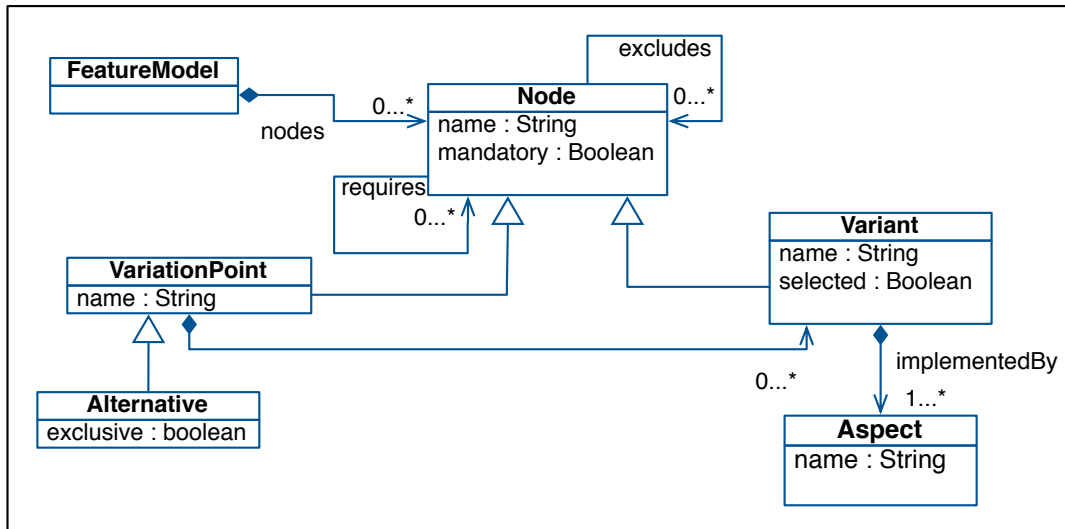


Figure 5.1: Feature metamodel.

The root of the metamodel is the meta-class `FeatureModel`. It contains a set of `Nodes`. Each node has two relationships: *requires* or *excludes*. Such relationships allow for the definition of constraints between all types of nodes. The nodes can be mandatory. We support two types of nodes: `VariationPoint`, and `Variant` that correspond to the concepts of variation point and

variant previously discussed. Additionally, a variation point can be alternative. There are two kinds of alternative variation points: exclusive or non-exclusive. The former one specifies that, from all the children variants of the alternative variation point, only one has to be selected. The latter one specifies that one or more children can be selected. Notice that in the metamodel, the constraints are defined at the level of nodes. This means that variants may require or exclude not only variants but also variation points. In the case of variation point, a *requires* means that at least one of its children has to be selected. In the case of an *excluded* variation point, none of their children can be selected. Finally, all variants have one *Aspect* that implements them. Like this, the variants are realized as the parts of the architecture that are introduced by each aspects. The weaving of such aspects results in an architecture that represents the product configuration where the variants for the aspects woven have been selected. The aspect metamodel and weaving are further explained in the Section 5.3.3

Illustrative Example

To illustrate the use of the feature metamodel, please consider the feature diagram illustrated in Figure 5.2. The diagram introduces a feature model defining a family of products with the essential functionality for an e-shopping scenario where a client connects to a server in order to find and buy items. This example is a simplified version of our case study. We use this example throughout the contribution chapters of this dissertation to illustrate the approach.

The part *a* of Figure 5.2 depicts the feature model for the e-shopping example using a traditional FD representation. Part *b* illustrates an object diagram of the equivalent model that conforms to our metamodel. From this diagram, we can identify the three types of features: (1) *mandatory* features (dark circles) which are always selected (e.g. *Notification* and *Payment*), (2) *optional* features (white circles), which can be chosen or not (e.g. *SMS*), and (3) *alternative* features (inverted arc). Exclusive alternative features are represented with a white arc (e.g. *ByDiscount*, *ByWeather*, and *ByLocation*) whereas non-exclusive alternative features are represented with a dark arc (e.g. *Wifi* and *GPS*). In addition to that, the diagram introduces one constraint indicating that location-filtered catalog needs one type of location to work.

The model corresponding to this feature diagram begins with the application element, the root of the model. It has four variation points: *Catalog*, *Notification*, *Location* and *Payment*. *Catalog* is mandatory and has an alternative exclusive variation point called *Filtered* with three different variants *ByDiscount*, *ByWeather*, and *ByLocation*. *Notification* is modeled as a variation point with two alternatives: *SMS* and *Call*. *Location* is a non-exclusive alternative variation point with two variants, *Wifi* and *GPS*. *Payment* is a variation point with two variants: *CreditCard* and *Discount*. Finally, the constraint between *ByLocation* and *Location* is represented as a property inside the requiring node, which in this case corresponds to the variant *ByLocation*.

5.3.2 Architectural Model

The target applications used in our approach are implemented to be executed in a platform based on CBSE and SOA paradigms. This choice enables products to be adapted at runtime by using the dynamic reconfiguration mechanisms available in such platform. In order to define a generic architecture to model such applications, we have created a metamodel inspired from the elements found particularly in SCA. SCA proposes a reconciliation between the Service Oriented Architecture (SOA) and Component-Based Software Engineering (CBSE), through a framework

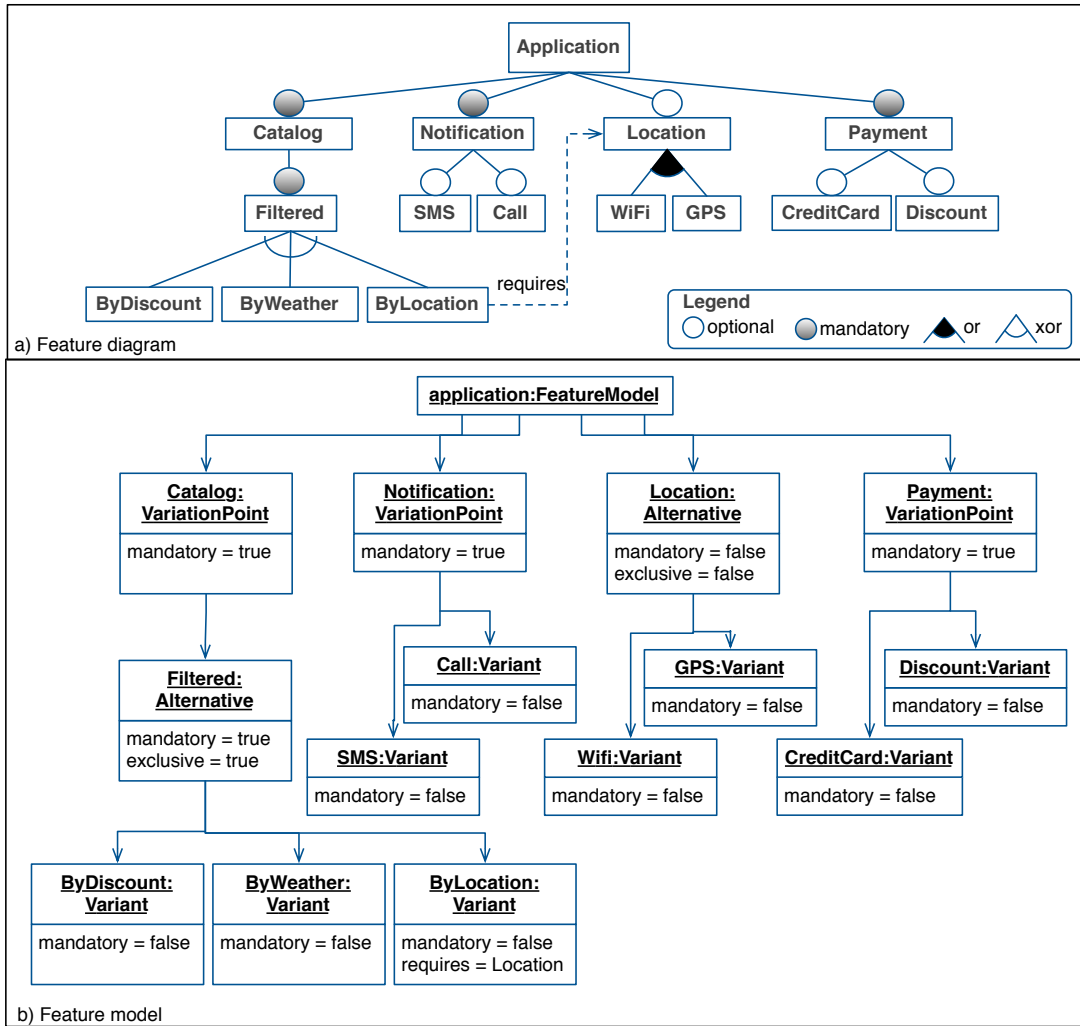


Figure 5.2: A sample feature diagram.

for describing the composition and the implementation of services using software components. Likewise, our architectural metamodel (illustrated in Figure 5.3) describes the software products in terms elements (meta-class `Element`) that provide services (meta-class `Service`) and require references (meta-class `Reference`). An element can contain other elements. This is expressed using the composite pattern of the meta-class `Container`. To fully describe a service oriented architecture, the metamodel also introduces the concepts of contracts (meta-classes `ServiceContract` and `Operation`) which are used to establish a relationship between services and references of different elements. There is the notion of `Object` to model business elements that cannot be represented as `Elements`. We can represent a set of calls between different services and references in an application using the meta-classes `Activity` and `Connection`. An activity represents a set of connections and a connection represents a link between two connection points.

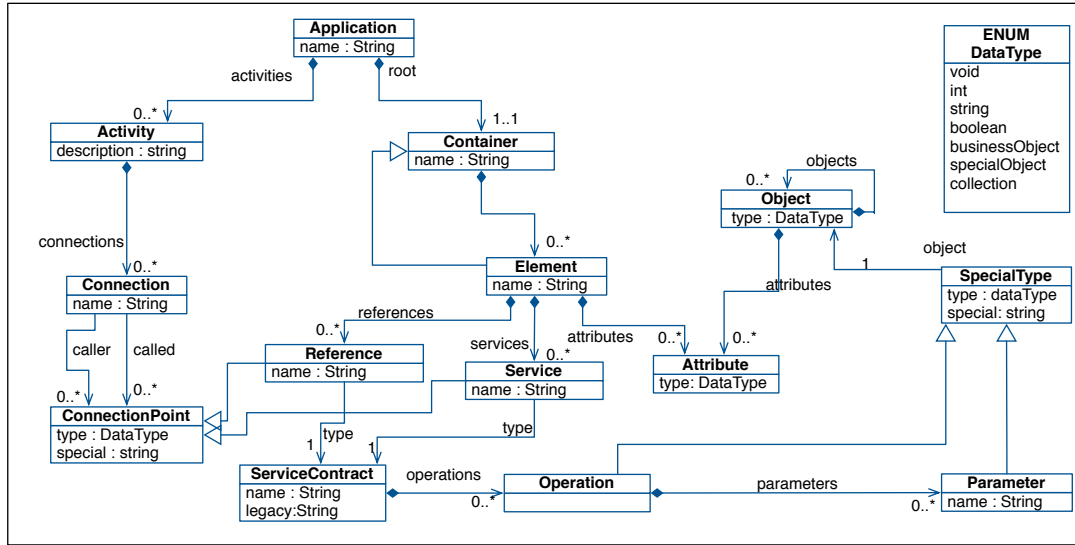


Figure 5.3: Architectural metamodel.

5.3.3 Aspect Metamodel

In our target platform, the unit of composition is the *software component* [Szy02], which is supposed to be *independent*. However, in a product family, each feature may be realized by *several* components. To fill the gap between the variability expressed in terms of features and the platform components, we have to modularize the architecture of the software products. This is the role of the aspect metamodel. It defines a language that extends the architecture metamodel in order to define both the base architecture model, that we called the *core*, and different modules that we call the aspect models. The *core* represents the commonalities of the family of products (*e.g.* mandatory features) whereas the aspect models contain different concerns, that can be woven to the base model (*e.g.* optional features). Aspect models are combined with the *core* to produce any product in our SPL.

The aspect metamodel (see Figure 5.4) is formed by four parts: the architecture of the elements to be woven (*Model*), the places where the weaving is realized (*Pointcut*), the modifications performed by the aspect (*Advice*), and optionally the moment of runtime when the aspect can be woven (*Event*). A description of each part is provided below.

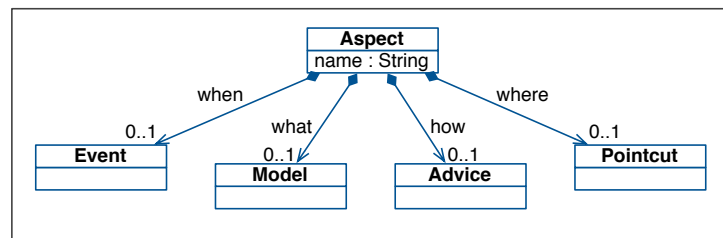


Figure 5.4: Aspect metamodel: simplified view.

Modeling the elements to be woven (*what*)

We use the architecture metamodel presented in Figure 5.3 to specify the *core* model as well as the pieces of architecture defined in every aspect. However, we have modified it to fulfill the role of Model for the aspect. Concretely, we have replaced the notion of Application for Model that belongs to the aspect. In addition to that, we have added a meta-class called ReferencedElement. This meta-class is specialized by all the meta-classes in the architecture. The main idea behind this meta-class is to provide a single entry point, so that every architectural element of the application becomes accessible from the Pointcut and Advice parts of the aspect. Figure 5.5 illustrates the modifications performed to the architecture metamodel to become the Model of the aspect.

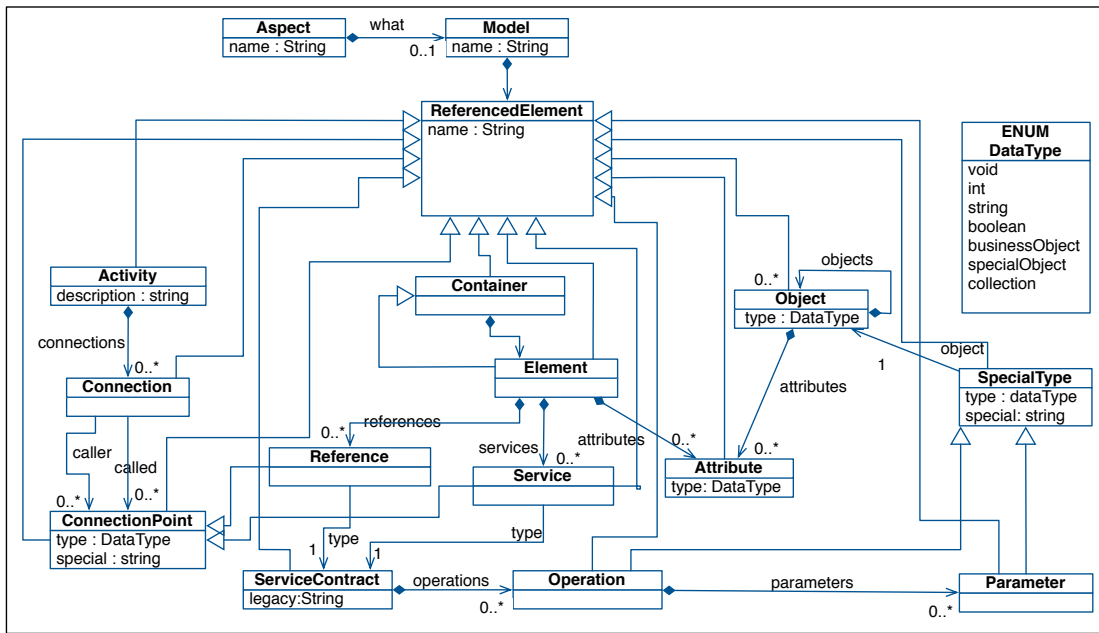


Figure 5.5: Aspect metamodel: definition of the Model.

Modeling the place (*where*)

We define the notion of Pointcut (see Figure 5.6) to define the places where an aspect can be woven. A pointcut queries the architecture of the *core* and returns all the elements that have to be present in the model in order for the weaving. The pointcut is composed of expressions (meta-class Expression) and variables (meta-class Variable). An expression can be either composite (meta-class Composite) or atomic (meta-class Atomic). A composite expression may contain nested expressions. To aggregate the results it uses an operator (meta-attribute operator) that defines the semantics of the composition (e.g., AND, OR). An atomic expression can be specialized in three different forms: InstanceOf, FindByName and Owned. InstanceOf is used to find an element using its type as a parameter. FindByName returns the elements whose name is equal to the name attribute of the expression. Finally the Owned expression looks for couples of

elements where one of the elements (parent) owns the other (child). A variable represents a place where the elements obtained as a result of the execution of an expression are stored.

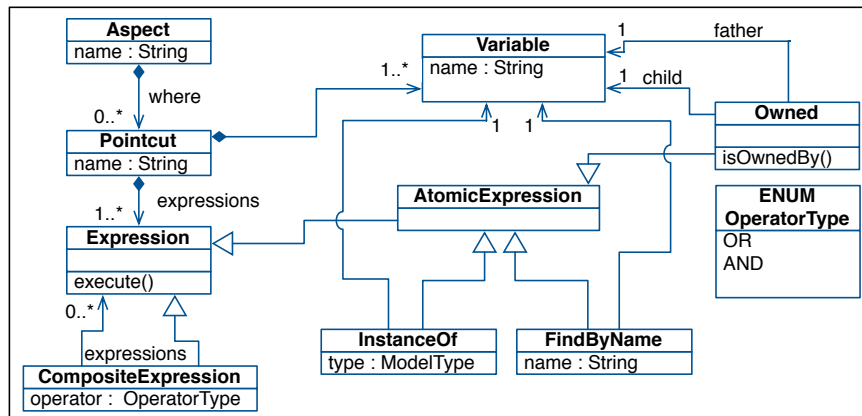


Figure 5.6: Aspect metamodel: definition of the `Pointcut`.

Modeling the modifications (*how*)

We consider the *Advice* (see Figure 5.7) to be a sequence of modifications (meta-class *Modification*). The following modifications are supported by our metamodel:

- The addition of a new model element (meta-classes `Add`). To add an element, each `Add` statement links an element of the model, represented as a `ReferencedElement`, and a `Variable` of the query, which represents the place where the element is going to be added.
- The removal of an existing model element (meta-classes `Delete`). To remove an element, each `Remove` statement references a `Variable` which represents the elements that are going to be removed.

Modeling the time (*when*)

Even if the notion of time is only relevant for the execution of an application, it is still necessary to model it. To do this, we use *context events*. By context we mean every piece of information that may affect an application. Examples of such information vary from availability of resources or services to information like temperature, location, or even hardware restrictions like memory. Consequently, a context event is a change in context information. The model part of the aspect model that is used to represent the context events is further detailed in Chapter 6.

Illustrative Example

To illustrate how architectural and aspect models are created, consider the object diagrams presented in Figure 5.8. Part *a* of the Figure depicts the core model for the e-shopping example. The core represents all the artifacts that implement the mandatory features of the diagram. In the

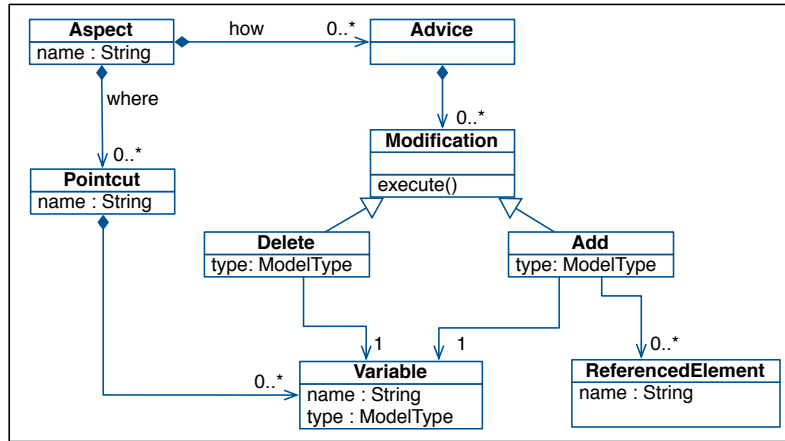


Figure 5.7: Aspect metamodel: definition of the Advice.

core, we are only interested in the architecture of the products. For this reason, the `Pointcut`, `Advice`, and `Event` parts of the aspect are empty. In this case, there is one element for every mandatory feature in the feature model. Additionally, we have created a `FrontEnd` element to have a single entry point to the functionalities offered by the application. Every element offers a number of services and requires a number of references. For example, the element `Notification` offers one service called `SendNotification` and uses two references called `SMSNotification` and `CallNotification`, which corresponds to the two variants that can be selected. For illustrative purposes, we have omitted the relationships between elements in the diagram. However, elements are connected with each other through `connections`, that link services and references. Part *b* illustrates the aspect model for the feature `SMS`. It has the four parts defined for an aspect: `model`, `pointcut`, `advice` and `event`. Since the event is only used at runtime, we leave it empty for now and focus on the other parts. The model includes an element called `SMS` with a service called `SMSNotification`. There is one pointcut that defines a composite expression and a variable named `reference`. The composite expression is formed by two atomic expressions of type `InstanceOf` and `FindByName`. The former one looks for instances of `Reference`. The latter one searches for instances named "SMSNotification". Both expressions are linked to the variable `reference`. The aspect's advice defines one modification: `addBoundElement`. This is a specialized version of the `Add` modification. It adds an element, but in addition to that, it also creates a binding with an available `Connection Point` in the core model. In this case, it adds the `SMS` element and it creates a binding to the reference found in the variable `reference`. Finally there is an event that declares a condition over a boolean observable. In other words, this aspect adds the element `SMS` to the container `Eshop`, and connects it to the corresponding reference in the `Notification` element.

In summary, the aspect metamodel combines the elements defined in the architecture (based on SCA) with the notions of pointcut, advice, and event. Like this, we are able to build a core model like the one in part (a) of Figure 5.8 that represents the commonalities of the family of products, and different aspect models that complement the core like the one presented in part (b) of the same Figure. Using this metamodel, we are able to modularize the architecture in different aspect models and link them to each one of the variants defined in the feature model. The product derivation process is then realized as the weaving of the aspect models for the selected variants with the core model, as explained in Section 5.4.

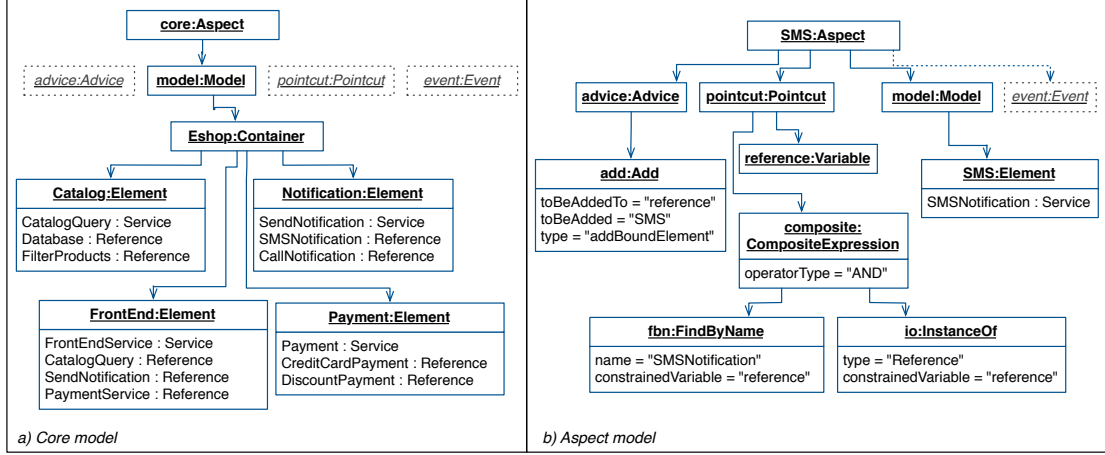


Figure 5.8: Architecture and aspect example models.

5.3.4 SCA Metamodel

For the SCA metamodel, we have used the metamodel defined by the OSOA that is part of the SCA specification. In order to integrate such metamodel as part of the assets of our SPL, we have created the corresponding resources (*ecore* files in the Eclipse Modeling Framework (EMF)) from the graphical representation available at [Ope07a]. Figure 5.9 illustrates the SCA metamodel. It presents the main concepts behind the specification. Composite, Component, Service, Reference, Property, Binding, Wire, Implementation.

5.3.5 JAVA Metamodel

With the increasing popularity of MDA approaches, several Java metamodels have been proposed ([INR10, Fal10, Bru10, PNP06]). For our SPL, we have chosen the Java metamodel proposed in the Spoon EMF Project [ND08, Bar06]. There are two main motivations for this choice. First, the metamodel is extensive enough and includes all the constructs that we need in order to build model transformations. Second, since it is part of the Spoon project, we benefit from the tools developed for this project in terms of code generation. In Spoon, the proposed Java metamodel is extensive as it considers most of the concepts of the Java language to build the abstract syntax tree of any application written in Java. It is formed by three main packages: *declaration*, *reflection* and *spoon*. Since the *reflection* and *spoon* parts deal with the imperative part of Java, and *spoon* processors to build code transformations, we only use the elements found in the *declaration* package for the *model-to-model* and *model-to-text* transformations. It contains the concepts required to represent the structure of any java application such as: *CtPackage*, *CtClass*, *CtInterface*, *CtField*, *CtParameter*, and *CtAnnotation*. For a more detailed explanation of the Spoon Java metamodel please refer to [PNP06], and [ND08].

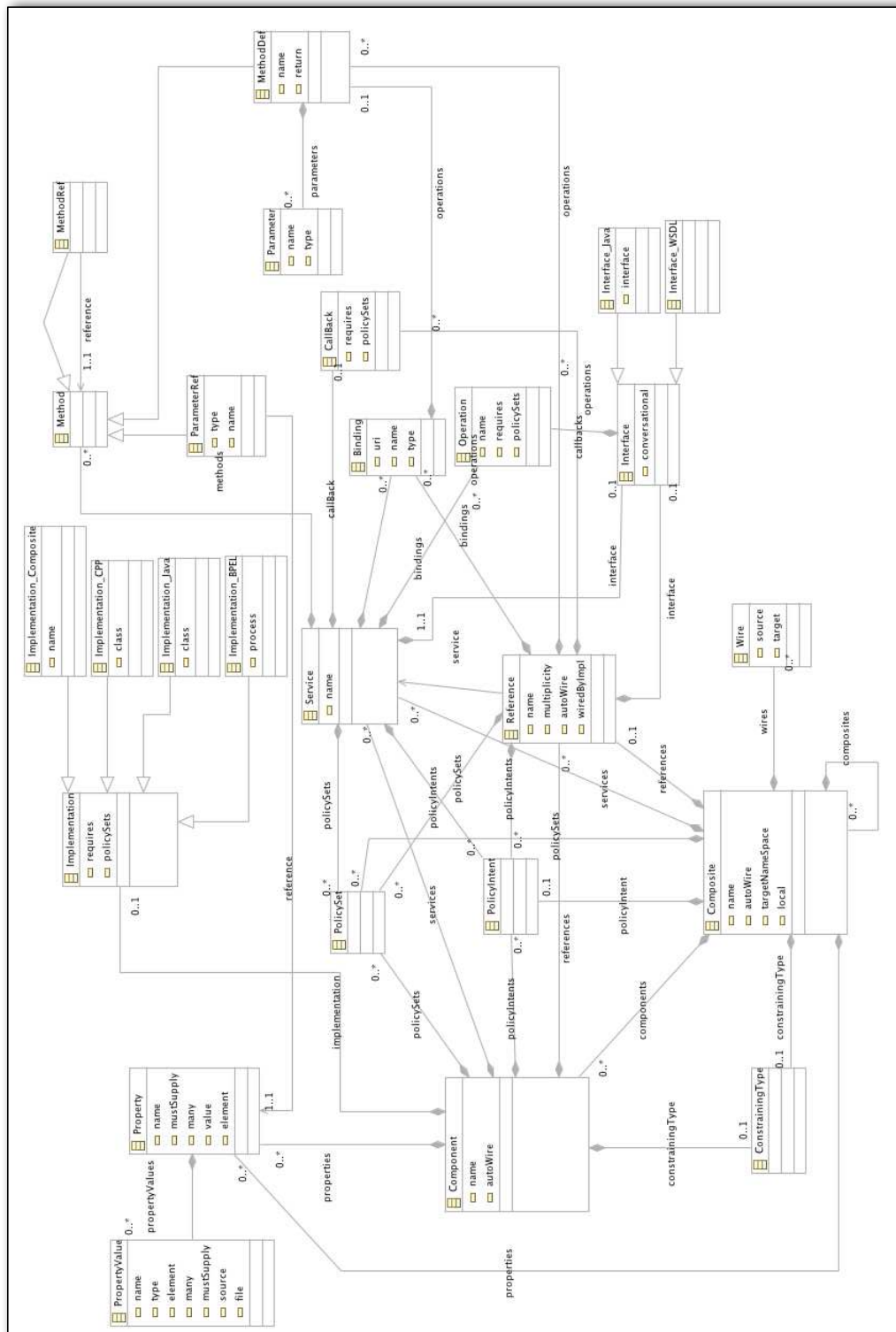


Figure 5.9: SCA metamodel.

5.4 Application Engineering

In order to obtain a software product from a product configuration, we define a product derivation process with four main phases as illustrated in Figure 5.10: (1) *constraint analysis* dealing with the analysis of constraints at both feature and aspect levels, (2) *model composition* in charge of weaving of multiple aspect models with the core to obtain a single woven model, (3) *platform transformation* to create platform specific models out of the woven model, and finally (4) *code generation* to obtain the source code of the products from the platform specific models. The product derivation goes like follows: first, the developer selects a set of features desired for his/her product, then a feature constraint analysis takes place. Out of this analysis the order of composition is obtained. This is used to perform the weaving of the different aspect models for the features selected. At the end of the composition a woven platform-independent model is obtained. The final steps of the product derivation consist in transforming this woven model into SCA and JAVA specific models and use them to generate the code of the product.

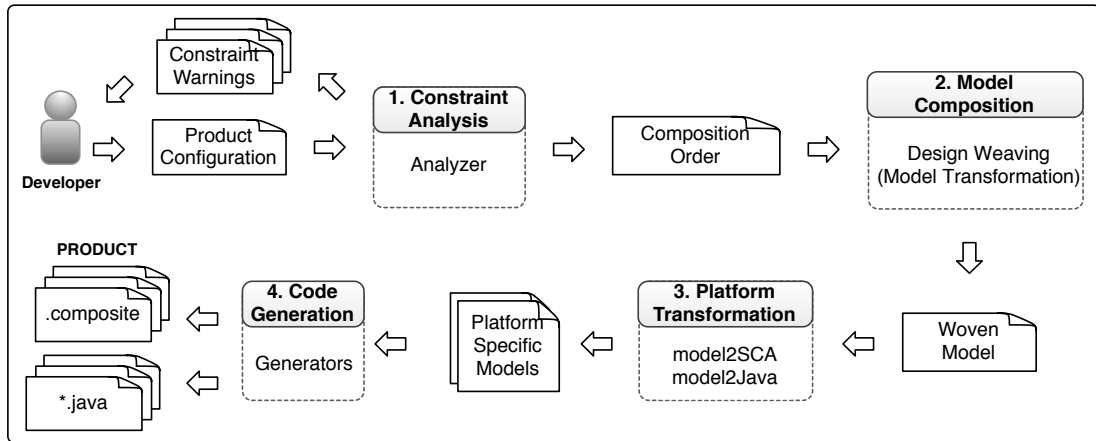


Figure 5.10: Product derivation phases.

5.4.1 Constraint Analysis

The goal of the constraint analysis is to identify and prevent the conflicts that may exist between the feature model and the aspect models. To overcome such conflicts, in our approach we propose an analysis of the inter-feature constraints of the FD and the dependencies between the corresponding aspect models.

The constraint analysis process takes place once the developer has configured a particular product. The product configuration is represented as a set of variants. Based on this selection, the constraint analysis aims at (1) checking that the constraints defined in the FD are consistent with respect to corresponding inter-aspect dependencies, (2) identifying implicit composition conflicts, and (3) deriving the most appropriate order of composition. This cross-model analysis goes in both ways: from features to aspects (*left to right*), and from aspects to features (*right to left*). Below, we specify both analyses based on the following notations:

- FD denotes the feature diagram of interest;
- $\mathcal{F} = \{F_1, F_2, \dots, F_n\}$ denotes the set of features of FD ;
- \mathcal{P} denotes the set of valid products that can be derived from FD ;
- $\mathcal{R} = \{(F_1, F_2) \in \mathcal{F} \times \mathcal{F} : F_1 \text{ requires } F_2\}$ denotes the set of *requires* constraints of FD ;
- $\mathcal{E} = \{(F_1, F_2) \in \mathcal{F} \times \mathcal{F} : F_1 \text{ excludes } F_2\}$, denotes the set of *excludes* constraints of FD ;
- A_F denotes the aspect model associated with a feature F ;
- $\mathcal{A} = \bigcup_{F \in \mathcal{F}} A_F$ denotes the set of aspect models associated with the features of FD ;
- $A.\text{Model}$ denotes the `Model` part of an aspect $A \in \mathcal{A}$;
- $A.\text{Pointcut}$ denotes the `Pointcut` of an aspect $A \in \mathcal{A}$;

Left to right analysis

The *left to right* analysis, concerns the constraints (*requires* or *excludes*) that are *explicitly* specified in the FD. Given a valid feature configuration, the analysis (1) checks that the related FD constraints actually translate as equivalent inter-aspect dependencies, (2) takes such dependencies as a basis to derive a correct weaving order, and (3) returns a warning for each FD constraint that has no “equivalent” at the aspect level.

- A “ F_1 *requires* F_2 ” constraint in the FD usually implies that the pointcut of aspect A_{F_1} references some model element(s) introduced by aspect A_{F_2} . If it is the case, A_{F_2} must be woven *before* A_{F_1} when deriving the product.
- A “ F_1 *excludes* F_2 ” constraint in the FD usually implies that the pointcuts of A_{F_1} and A_{F_2} reference common model elements.

Algorithm 1 summarizes the *left to right* analysis process, which takes as inputs (1) the feature diagram FD , (2) the associated aspect models \mathcal{A} , and (3) a valid feature configuration p . Each *requires* constraint relative to p is analyzed (lines 2–8). If the constraint translates as a Pointcut-Model dependency, the weaving order is adapted accordingly (line 6). If such a dependency is not found, a corresponding warning is returned. The analysis of *excludes* constraints (lines 9–12) is similar, except that (1) it is based on Pointcut-Pointcut dependencies and (2) it does not impact the weaving order. Indeed, the feature configuration is supposed to be valid with respect to the explicit FD constraints.

Right to left analysis

The second part of the analysis is intended to find *implicit* inter-feature constraints. Such dependencies are not specified in the FD, but hold between the corresponding aspects and, thus, may cause a conflict when realizing the composition. Similarly to the *left to right* analysis, two types of constraints are considered:

- A *requires* constraint indicates that an aspect pointcut refers to parts of the model of other aspect.

Algorithm 1 Left to right analysis

Require: A feature diagram FD , the associated aspect models \mathcal{A} , a valid feature configuration $p = \{F_1, F_2, \dots, F_k\} \in \mathcal{P}$

Ensure: A weaving order \mathcal{O} and a set of warnings \mathcal{W}

```

1:  $\mathcal{O} \leftarrow \text{toList}(p)$ 
2: for all  $(F_1, F_2) \in \mathcal{R}$  such that  $F_1 \in p$  do
3:   if  $A_{F_1}.\text{Pointcut} \cap A_{F_2}.\text{Model} = \emptyset$  then
4:      $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1 \text{ does not require } F_2 \text{ at the architectural level}\}$ 
5:   else
6:      $\mathcal{O} \leftarrow \text{switchPositionIfNeeded}(\mathcal{O}, F_2, F_1)$ 
7:   end if
8: end for
9: for all  $(F_1, F_2) \in \mathcal{E}$  such that  $F_1 \in p$  do
10:  if  $A_{F_1}.\text{Pointcut} \cap A_{F_2}.\text{Pointcut} = \emptyset$  then
11:     $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1 \text{ does not exclude } F_2 \text{ at the architectural level}\}$ 
12:  end if
13: end for

```

- An *excludes* constraint indicates that there are at least two pointcuts in distinct aspects with equivalent expressions. If such a situation occurs, then it is necessary to verify whether the corresponding advices are interfering with each other. Generally, aspects can be classified with respect to the interferences with each other in three categories: (1) *independent*, when their pointcuts and modifications do not affect other aspects, (2) *partially dependent*, when pointcuts may involve previously woven aspects but advices are independent, and (3) *totally dependent*, when pointcuts are dependent on previous aspects and advices may impact other aspects. In our case, it is the third category that may lead to composition conflicts. Consequently, aspects that exhibit such dependencies should not be woven within the same product derivation. In order to determine whether the aspects are totally dependent, one must check if the modifications introduced by one aspect have a negative impact on the other. This is similar to *critical pair analysis* [Plu93] in the domain of graph rewriting. Since there are only two types of modifications in our aspect metamodel: *add* and *delete*, the analyzer has to make sure that one aspect is not deleting an element referenced in the other aspect. If it does, the developer is warned about an implicit *excludes* constraint missing in the FD.

The *right to left* analysis is formalized in Algorithm 2. In case an implicit *requires* constraint is detected (lines 2–14), the behavior of the analyzer varies depending on whether the product configuration includes the required feature F_2 or not. If F_2 is selected, a warning is returned and the composition can be achieved according to an appropriate weaving order (line 8). If, in contrast, F_2 is not part of the configuration, then the composition is aborted (lines 10–11). Regarding the detection of implicit *excludes* constraints, the analyzer behaves in the other way around. In this case, indeed, the *presence* of excluded features F_2 in the configuration causes the composition to be aborted (lines 20–21), while their absence leads to a warning only (line 18).

Defining the composition order

The composition order is derived from the analysis in both ways. To obtain it, the analysis tool traverses the list of features in the same order as they were selected, and, whenever a feature

Algorithm 2 Right to left analysis

Require: A feature diagram FD , the associated aspect models \mathcal{A} , a valid feature configuration $p = \{F_1, F_2, \dots, F_k\} \in \mathcal{P}$, an initial weaving order \mathcal{O}

Ensure: A flag `compositionAllowed`, a possibly adapted weaving order \mathcal{O} and a set of warnings \mathcal{W}

```

1: compositionAllowed  $\leftarrow$  true
2: for all  $F_1 \in p$  do
3:   for all  $F_2 \in \mathcal{F}$  such that  $A_{F_1}.Pointcut \cap A_{F_2}.Model \neq \emptyset$  do
4:     if  $(F_1, F_2) \notin \mathcal{R}$  then
5:        $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1 \text{ implicitly requires } F_2 \text{ at the architectural level}\}$ 
6:     end if
7:     if  $F_2 \in p$  then
8:        $\mathcal{O} \leftarrow \text{switchPositionIfNeeded}(\mathcal{O}, F_2, F_1)$ 
9:     else
10:      compositionAllowed  $\leftarrow$  false
11:       $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1 \text{ implicitly requires a non-selected feature } (F_2)\}$ 
12:    end if
13:  end for
14: end for
15: for all  $F_1 \in p$  do
16:   for all  $F_2 \in \mathcal{F}$  such that  $A_{F_1}.Pointcut \cap A_{F_2}.Pointcut \neq \emptyset$  do
17:     if  $(F_1, F_2) \notin \mathcal{E} \wedge \text{totallyDependent}(A_{F_1}, A_{F_2})$  then
18:        $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1 \text{ implicitly excludes } F_2 \text{ at the architectural level}\}$ 
19:       if  $F_2 \in p$  then
20:         compositionAllowed  $\leftarrow$  false
21:          $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1 \text{ implicitly excludes a selected feature } (F_2)\}$ 
22:       end if
23:     end if
24:   end for
25: end for

```

requires (implicitly or explicitly) other feature, it is moved in the list to the position right after the feature being required. This is done in both the *left to right* algorithm (line 6) and the *right to left* algorithm (line 8). This order guarantees that the pointcuts of features requiring other features are correctly executed during the composition.

Illustrative Example

To better illustrate the constraint analysis. Lets consider the family of products defined in the feature model of Figure 5.2. First of all, we have to select a product configuration. Only the leafs of the feature model – which are represented with variants in the model – can be selected. The diagram includes 9 variants in total: *ByDiscount*, *ByWeather*, *ByLocation*, *SMS*, *Call*, *Wifi*, *GPS*, *CreditCard*, and *Discount*. Consider the following two product configurations:

- $\mathcal{P}_1 = \{\text{ByLocation}, \text{SMS}, \text{Wifi}, \text{CreditCard}\};$
- $\mathcal{P}_2 = \{\text{ByLocation}, \text{Call}, \text{GPS}, \text{Discount}\};$

For the configuration \mathcal{P}_1 the *left to right* analysis changes the order of the composition. This is due to the requires constraint that goes from the variant `ByLocation` towards any kind of location, in this case `Wifi`. Consider the pointcut of the `ByLocation` aspect and the model of the `Wifi` aspect (parts *a* and *b* of Figure 5.11). As it can be noticed, there is a match between the pointcut of the `ByLocation` aspect and the model of the `Wifi` aspect. This match corresponds to the requires constraint that has been explicitly defined at the level of features. Following the *left to right* algorithm, when the corresponding dependency is found, the order is modified so that, the aspect implementing the variant `Wifi` is woven before the aspect implementing the variant `ByLocation` (see line 6 of algorithm 1). Changing the order guarantees that the pointcut of `ByLocation` finds the service offered by the element `Wifi`. The conflict-free configuration for \mathcal{P}_1 obtained after applying the algorithm 1 is: $\{\text{Wifi}, \text{ByLocation}, \text{SMS}, \text{CreditCard}\}$.

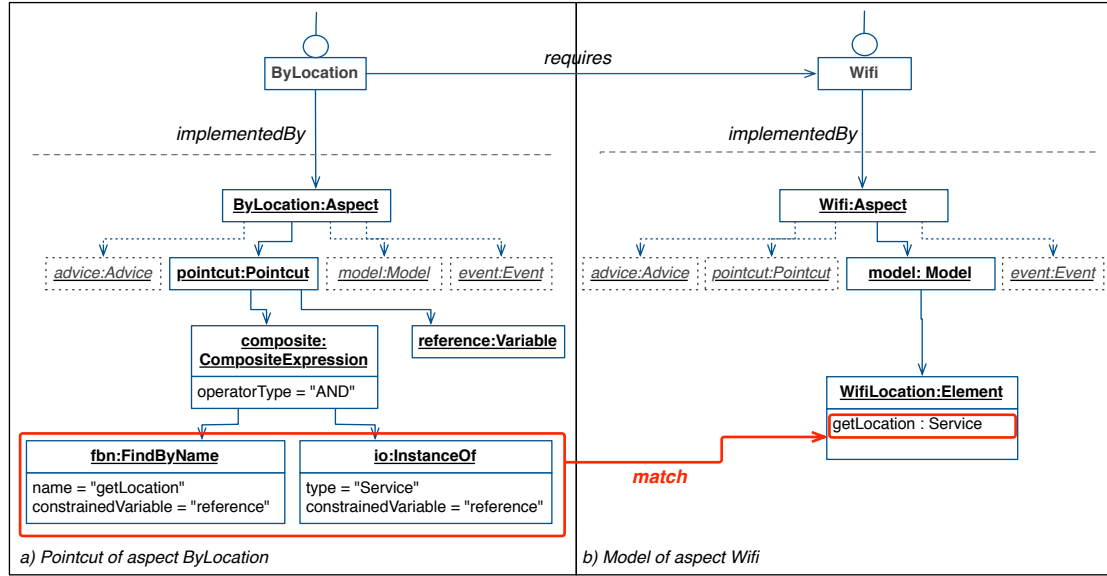


Figure 5.11: Dependency between `ByLocation` and `Wifi`.

For the configuration \mathcal{P}_2 the *right to left* analysis stops the composition. This is because there is a hidden dependency between the aspect for the variant `Discount` and the aspect for the variant `SMS`. Consider the pointcut and model parts for each aspect that are illustrated in Figure 5.12. The pointcut for the variant `Discount` needs a service that is only offered by the aspect implementing the Feature `SMS`. The dependency is found because the *right to left* analysis looks for match relationships in every aspect for the product family, and not only the ones selected for the particular configuration. Following the algorithm 2 if dependencies towards aspects that do not belong to the current configuration are found, the weaving cannot take place (see lines 9,10 , and 11). Since the variant `SMS` is not part of the configuration \mathcal{P}_2 , the weaving is stopped. This behavior repeats for any configuration that includes the variant `Discount` and not the variant `SMS`.

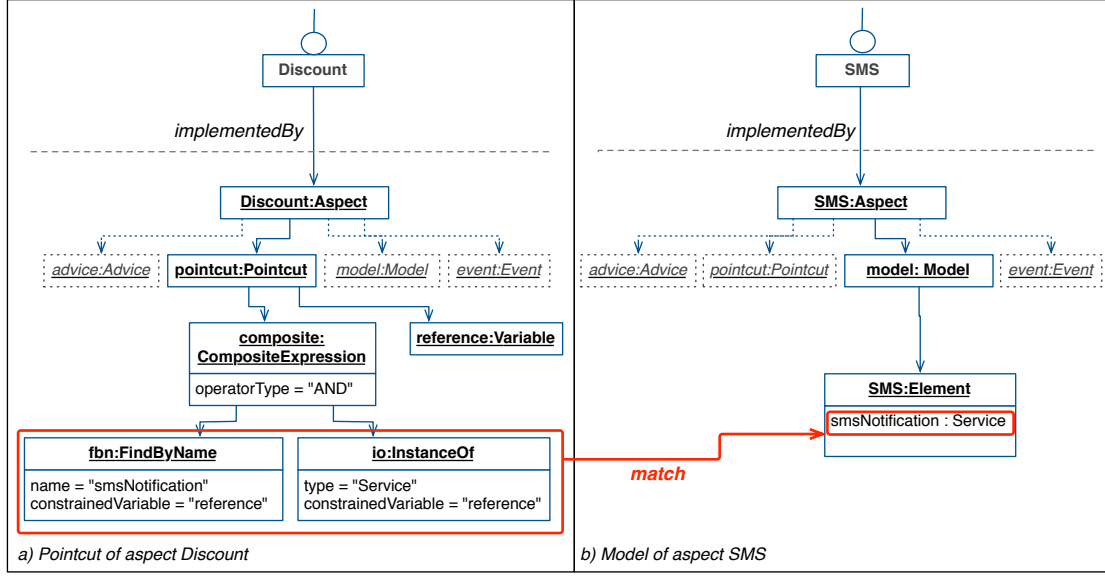


Figure 5.12: Hidden dependency between Discount and SMS.

5.4.2 Model Composition

In general terms, the model composition consists of successive calls to a single generic model transformation (weaver). This transformation takes as inputs the core model M and an aspect A to be woven, and returns a single model representing the composition of the core and the aspect. The transformation is executed as many times as aspects are to be woven in the order defined by the constraint analysis. The transformation itself relies on the aspect metamodel. It consists in iterating over the set of modifications specified in the *Advice* of A in order to execute each one of them.

The places where each modification is performed are defined by the associated *Pointcut*. The execution of this pointcut on the *core* model iterates over its *Expressions*, which can be either atomic or composite. There exist three types of atomic expressions: *FindByName*, *InstanceOf* and *Owned*. *FindByName* is a basic query that looks for every single element in the model whose name corresponds to the name introduced as a parameter. An *InstanceOf* query finds all the elements that correspond to the type introduced as a parameter. Finally, *Owned* is a query that looks for couples where one element owns the other. Each atomic expression returns the collection of *core* model elements that match the filter conditions. A composite expression is evaluated by accumulating and combining the result of each atomic expression. The way the resulting elements are combined depends on the composite operator. The *and* operator is interpreted as the *intersection* of the model elements, whereas the *or* operator translates as their *union*. The resulting elements are stored and impacted by the modifications.

At the end of the pointcut execution, all the places impacted by the aspect have been identified. Then the modifications specified by the aspect can be applied. In the case of an *Add* modification, the elements of the aspect are added to the *core* model. In the case of a *Delete* modification, the elements found in the pointcut are removed from the *core* model.

The transformation finishes when all modifications specified in the advice have been performed. At this point it is considered that the original *core* model M is now woven with the aspect A . This process is repeated for every selected. The global weaving process repeats until all the aspects implementing the variants selected in the feature configuration have been woven with the core model. The model M is no longer a *core* but rather a complete representation of an application including the right constructs for the concerns defined separately in every aspect woven.

Algorithm 3 Model Composition

Require: A weaving order \mathcal{O} , and a core aspect C

Ensure: A woven model \mathcal{C}

```

1: for all  $(F) \in \mathcal{O}$  do
2:    $A_F.Pointcut.execute()$ 
3:    $A_F.Advice.execute()$ 
4: end for
  
```

Illustrative Example

To illustrate the composition, consider the core and aspect presented in Figure 5.8. In order to weave the *core* model with the SMS aspect, the weaver executes the pointcut in the aspect SMS. This process traduces in executing its atomic expressions. The first one (`fbn`) finds all the referenced elements called "SMSNotification", in this case there is only one element with such a name. The second one (`io`), finds all the referenced elements whose type corresponds to `Reference`. Afterwards, the composite expression indicates an intersection between the two sets obtained in the atomic expressions which results in a single match. Consequently the variable called `reference` of the pointcut will have the reference `SMSNotification` of the `Notification` element. The next step in the weaver is to perform the modifications in the place indicated by the variable. The two possible modifications are described below:

- **Add:** To add an element, the meta-class `Add` has a relationship with the `ReferencedElement` being added and the `Variable` where it will be added. Nevertheless, having this relationship allows for any referenced element to be weaved at any place in the *core* model. To prevent incompatible combinations, like for example trying to add a `Container` inside an `Attribute`, we have defined a scope for these combinations. There is a set of allowed pairs of types, where the first type corresponds to referenced element being weaved and the second type corresponds to the variable where the element is going to be added. For each allowed pair, we perform the adequate operations to add the element. Allowed pairs vary from coarse grained operations (i.e. adding a new `Element` inside an existing `Container`) to finer grained ones (i.e. adding a new `Operation` inside a `Service Contract`). For the incompatible pairs, no weaving is performed.
- **Delete:** deletes the elements at the places described by the `Variable`. The deletion of an element triggers the destruction of its inner elements (for `Element` or `Container` elements).

In the aspect depicted of Figure 5.8, there is one modification. `Add` and `bind` (`addBoundElement`) the element `SMS` to the elements found by the pointcut, which in this case is a single reference. The add operation first adds `SMS` to the container `Eshop`. Then, it creates a connection between the service `SMSNotification` of `SMS` and the reference `SMSNotification` of

the element A. The resulting woven model obtained from this composition is illustrated in Figure 5.13. The classes and connections added to the `core` model are shown in dashed red lines.

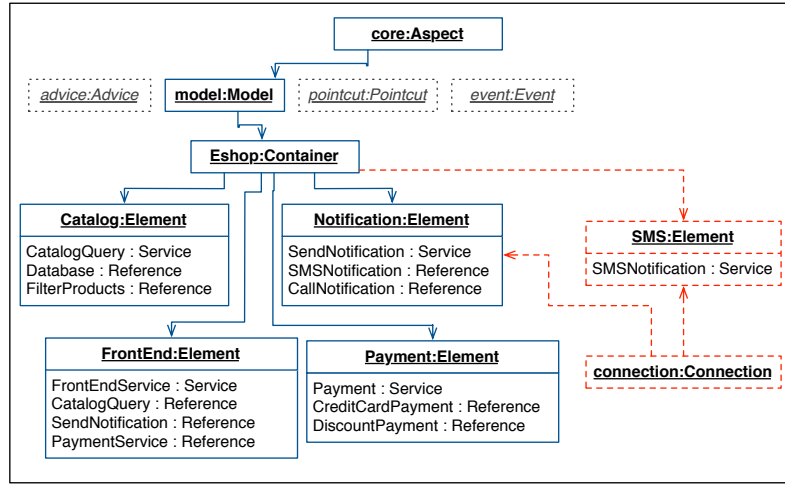


Figure 5.13: Woven model obtained after the composition.

5.4.3 Platform Mapping and Code Generation

The model shown in Figure 5.13 illustrates a woven model with the core plus the SMS variant. However, this model is still platform and technology independent. In order to generate source code from it, we follow a classic MDE approach where the model is the input to a transformation that maps its concepts into platform (SCA) and implementation (Java) specific ones. Each transformation consists of a set of rules that map the concepts of the model into the corresponding elements in Java, and SCA. For instance, an `Element` is mapped into: (1) an SCA component, (2) a Java interface defining its services, and (3) a Java class implementing the Java interface and representing the SCA component.

Figure 5.14 presents a conceptual view of how model transformations and code generation are performed. By first transforming the model concepts into Java and SCA models, we can check the consistency among these new models. This would not be possible if we had directly generated the source code from the aspect metamodel. Finally, code is generated from SCA and Java to obtain the composite descriptors and Java code using Spoon capabilities. A detailed explanation of the transformation and generation processes is presented in the tool support section of Chapter 7.

5.5 Discussion

We have presented so far the domain and application engineering processes that cover the design phase of the DSPL. Let us now revisit the challenges identified in Section 6.2 and discuss how our approach face them.

1. **Ensure a clear separation of concerns:** To face this challenge, we use the aspect metamodel to modularize the architecture of products in order to realize variability through aspect

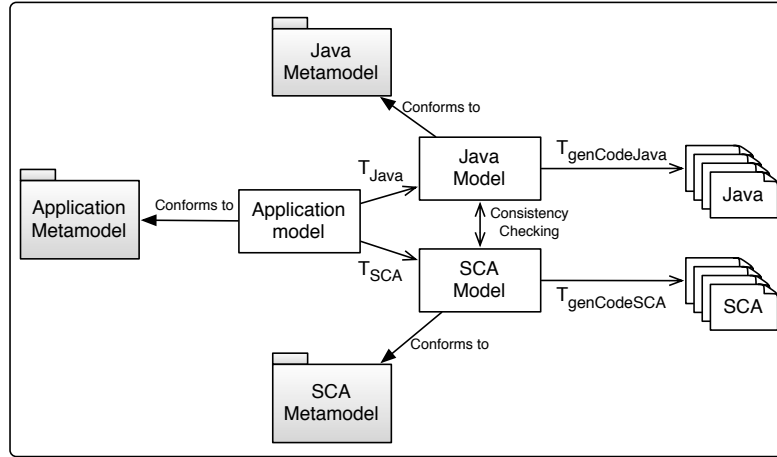


Figure 5.14: Transformation scheme.

models. The language defined for aspect models allows us to define the right constructs for every feature identified in the feature model. Consequently, we are able to define the mapping that holds between the features and the corresponding software artifacts that implement them. For features that have an impact at multiple places in the final product, the aspect model proves to be expressive enough to define the different places (i.e. pointcut) and modifications (i.e. advice) for aspects to be woven.

2. **Identify inconsistencies:** To face this challenge, we have defined the constraint analysis algorithms between features and aspect models. Such algorithms traverse both the FD and the aspect models looking for corresponding constraints that may cause incorrect products. The composition only takes place if and only if all the explicit and implicit constraints are respected.
3. **Derive a suitable composition strategy:** In addition to the inconsistency identification, the algorithms take explicit and implicit features interactions as a basis to derive a conflict-free composition strategy. This ensures that aspects models are woven to the core in the appropriate order for any product configuration.
4. **Use platform independent assets:** To face this challenge, we have provided a platform-independent language to build aspect models. We use model transformations towards a particular platform (SCA and Java) in order to complete the derivation of products and obtain source code at the end of the process. Moreover, the aspects models could eventually be transformed towards different platforms, in which case, the constraint analysis and aspect weaving processes would remain valid.
5. **Provide an automated development process:** The complete design phase of CAPucine defines a process where assests correspond to aspect models. Such models realize commonalities and variabilities of the product family and are used to build the products in an automated manner.

5.6 Summary

This chapter presented a comprehensive approach to feature-driven derivation of software products. We have illustrated an approach for automated derivation of product architectures from feature configurations combining MDE and AOM techniques. The composition process is realized through transformation-based model weaving and is guided by the explicit and implicit dependencies that exist between the selected features. The method allows to identify implicit dependencies and conflicts between features, and takes such feature interactions as a basis to derive an appropriate architecture composition strategy. Our approach relies on a clear separation of concerns provided by the underlying variability and aspect metamodels.

In the next chapter of this dissertation, we discuss the challenges of extending the derivation process of the DSPL at runtime and enable products to be modified during their execution. We present a runtime phase that adapts products dynamically using the same domain engineering assets illustrated in this chapter.

Chapter 6

Runtime Phase: Context aggregation, verification, and dynamic reconfiguration

"The only constant is change" Heraclitus of Ephesus

Contents

6.1 Introduction	81
6.2 Motivating Scenario and Challenges	82
6.2.1 Double Threshold	82
6.2.2 Challenges	83
6.3 Adaptation Life Cycle	84
6.4 Runtime Phase: dynamic adaptation of DSPL products	86
6.4.1 Context Management	86
6.4.2 Decision Making	87
6.4.3 Runtime Platform	93
6.5 Discussion	96
6.6 Summary	96

6.1 Introduction

In the previous chapter, we have introduced a process of product derivation at design time based on aspect models. Furthermore, we have illustrated a constraint analysis of implicit and explicit feature interactions and aspect dependencies. In this chapter we present an extension of such derivation process at runtime. The main objective of this process is to *adapt* the products being executed using the same assets of design weaving and ensuring that any product obtained as a result of the process remains a valid product of the SPL.

An adaptation in CAPucine can be understood as the switching from one product configuration (*current*) to a new one (*target*). The decision about the *target* configuration is obtained using the context information available at runtime. Since runtime weaving deals with applications that are being executed, the aspect models used in the previous chapter for the design phase cannot be used in the same way to adapt the running product. This implies the definition of a dynamic representation of the aspect models, and the implementation of the mechanisms allowing aspects to be woven. Among those mechanisms we can enumerate: a context manager to obtain and aggregate context information, a decision making engine that finds the adequate configuration based on context that respects the FD constraints and the aspect dependencies, and a runtime platform supporting runtime adaptations. To derive a product from the DSPL, whether at design or at runtime, we have the same input i.e., an aspect model. However, each phase relies on different processes and return different outputs. Design derivation can be seen as a refinement of a core architecture using the product configuration and its implementing aspect models. Runtime derivation, on the other side, aims at modifying an already complete and running software product based on the product configuration, but mapping the aspect model into usable reconfiguration scripts that change the products during their execution.

Structure of the Chapter

This chapter is organized as follows. In section 6.2 we introduce a motivating scenario and identify several challenges for achieving dynamic adaptation using SPL assets. Section 6.3 presents the adaptation life-cycle. Section 6.4 presents the derivation process at runtime using aspect models. In Section 6.5 we revisit the challenges for the runtime phase and discuss how the proposed adaptation phase faces them. Finally, in Section 6.6 we summarize and conclude the chapter.

6.2 Motivating Scenario and Challenges

This section introduces a motivating scenario for dynamic adaptations. It illustrates a distributed application following a double threshold pattern. We first discuss the characteristics of such example and then based on this discussion we enumerate several challenges for the process of runtime weaving in our DSPL.

6.2.1 Double Threshold

One classical example that motivates the need for dynamic adaptation refers to the adaptation of applications to the network conditions. To prevent the system to be constantly changed due to a non-stable bandwidth, adaptation can be based on a classical *double threshold* pattern. Thanks to such a pattern, the architecture of the product only varies within two different modes: *full connectivity* when bandwidth goes over the `Maximum threshold`, and *limited connectivity* when bandwidth falls below the `Minimum threshold`. (see Figure 6.1).

For example, consider the simple application depicted in Figure 6.2. It illustrates an assembly of components that represent the architecture of one product. Following the SCA notation, arrows to the left of each component represent the services that it provides and arrows to the right represent the references or services it requires. In this case there is a `GUI` component which offers a `run` service, and is bound with the `DB` component through the `getDesc` reference. The `GUI` is also bound to the `Cache` component through the `store` reference in order to retain information locally.

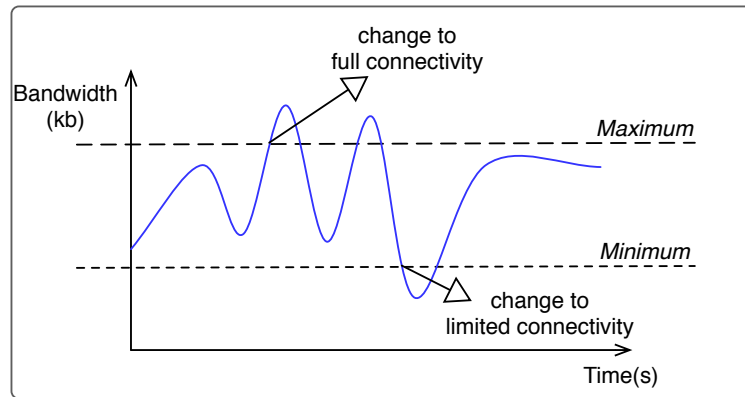


Figure 6.1: Double threshold pattern.

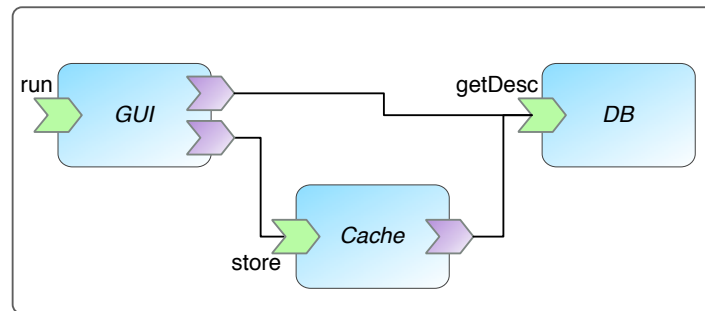


Figure 6.2: SCA assembly.

At runtime, this product can be dynamically adapted to fit changes in its environment. In this case, the *Cache* can be *selected*, depending on the bandwidth quality. On the one hand, if the bandwidth is high (*full connectivity*), then the *GUI* directly interacts with the database without using the cache. During this period of time, the cache stores locally all the movie descriptions browsed by the user, but also requests other ones that have been defined to be closely related. On the other hand, if the bandwidth is low (*limited connectivity*), then the *GUI* uses the cache, that is still connected to the database. Figure 6.3 shows the two alternative configurations that work respectively with a full connectivity and a low connectivity bandwidth.

This example shows the dynamic nature of adaptations and the importance of having a process of decision making in order to modify products. In this case, it is necessary to capture events and information in the environment like the "bandwidth". Furthermore it is necessary to analyze this information regarding thresholds and business rules in order to make the right adaptation.

6.2.2 Challenges

The example previously discussed uses information that is only available at runtime to change its internal structure and behavior. Our goal with the runtime phase, is to also cover the development of this kind of applications. We argue they can be modeled as families of products where context information (and not developers) decides about the product configuration for every particular situation. A Dynamic SPL for such kind of applications has to support a product

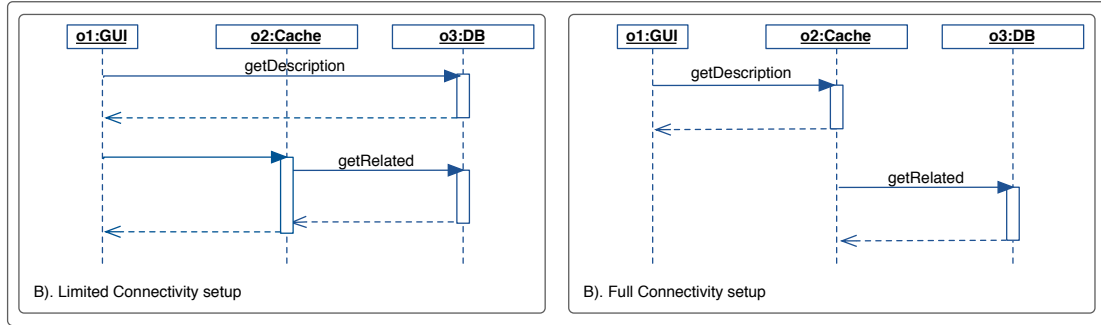


Figure 6.3: Sequence diagrams for full and low connectivity configurations.

derivation that is able to change products from one configuration (*current*) to a new configuration (*target*). To achieve such flexibility in a SPL, and take advantage of the assets developed for the design phase, we identify several challenges that have to be faced:

1. **Extend the concept of feature at runtime** Every product in the SPL is described as a set of selected features that are realized through aspect models. The first challenge to achieve dynamic product derivation is to define a way to maintain, and update, the state of a product in terms of the features it is supporting at a given moment during its execution.
2. **Find the appropriate reconfiguration using several pieces of information** A dynamic adaptation is driven by several factors like context information or the current state of the application. As a consequence, there has to be a way of receiving and analyzing events that occur in the environment, and that may have consequences in the architecture of the software products. The second challenges consist in providing the means to manage all the inputs for the decision making process that decides about the adaptations to be performed in the different products of the SPL.
3. **Avoid conflicts and maintain SPL consistency** Even if one configuration is found to fit a particular context situation, it is necessary to verify if such configuration remains legal regarding the SPL constraints (i.e. if the set of selected variants respect the FD constraints), and to define the reconfiguration. The third challenge that has to be faced is then to avoid conflictive products and specify the correct order on which the reconfiguration takes place to obtain the new configuration.
4. **Use a platform that supports dynamic adaptations** For the context-driven reconfigurations to take place, it is necessary to have a runtime platform that allows for: executing the products, managing context, and performing reconfigurations at runtime. The last challenge regards the use of a platform that provides the means to achieve such tasks.

6.3 Adaptation Life Cycle

To better understand the process of runtime weaving, we start by describing the expected adaptation life-cycle (see Figure 6.4) of any product derived from the DSPL. The adaptation life-cycle includes four main steps:

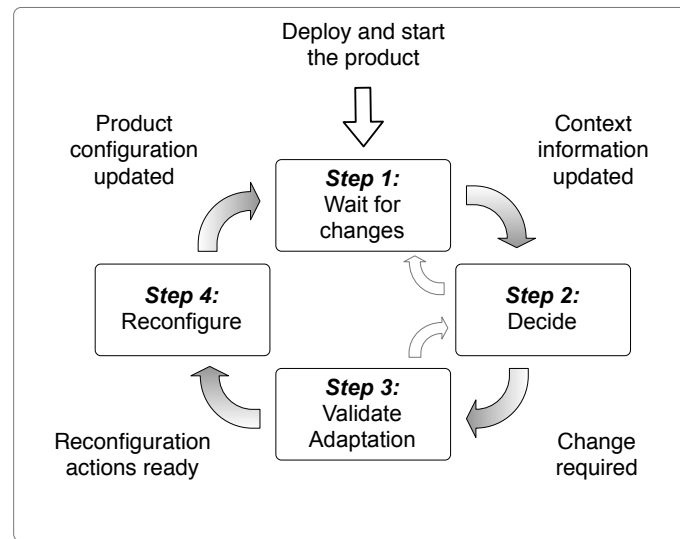


Figure 6.4: Adaptation life-cycle.

- **Step 1. Wait for changes:** The product executes as planned until the platform is notified of changes in the context state.
- **Step 2. Decide:** Next, the platform verifies if the new context information implies an adaptation of the system. If an adaptation is not needed, the cycle returns to the **step 1** to wait for further changes. If, on the other hand, the product needs to be modified, then a new configuration is created using the context information to decide which variants it has to support. Once the new configuration has been defined the cycle continues to **step 3**.
- **Step 3. Validate Adaptation:** The configuration generated in **step 2** gets validated. The objective is to avoid conflicting configurations that may lead to inconsistent products that do not belong to the scope of the product family by violating one of the constraints defined in the FD. If the configuration is correct, the cycle continues to **step 4**. On the contrary, if the new configuration is not valid, the cycle returns to **step 1**.
- **Step 4. Reconfigure:** This is the step when the reconfiguration actions are actually executed. This involves stopping the system components concerned by the modifications, removing and adding bindings between them, and restarting them with the new configuration. Finally, once the new configuration has been applied, the system returns to **step 1**, to listen for further context changes. This process repeats until the end of the product execution.

The adaptation cycle implicitly identifies several responsibilities to enable adaptation of DSPL products. First, in order to *wait for changes* (**step 1**) a Context Manager is needed to aggregate and process context information from different sources. Second, to *decide* and *validate* a new configuration (**steps 2 and 3**) it is necessary to have a Decision Making mechanism. Finally the reconfiguration itself (**step 4**) indicates the need for an Application Platform

where products can be modified at runtime following a set of reconfiguration actions. In the following sections we present in detail how these elements are integrated into our DSPL.

6.4 Runtime Phase: dynamic adaptation of DSPL products

In order to create the concrete components that implement the functionalities identified before, we define an adaptation cycle for the runtime phase as illustrated in Figure 6.5. It is formed by two main parts: **Decision Making**, and **Runtime Platform**. The adaptation cycle starts in the latter one with an event which is processed by the context manager (step 1). The context event is aggregated into a piece of context information represented as an observable. It is the value assigned to this observable that gets evaluated to decide whether or not to trigger the adaptation. The updated observable values are the input of the process of **Decision Making**. An `Adapter` defines a target configuration using as input the observables and the current configuration (step 2). Next the configuration is validated to check if it respects the DSPL constraints (step 3). If the configuration is correct, both the current configuration coming from the product, and the target configuration obtained in the previous steps are used as input to the `Script Generator` which generates a list of modifications expressed in terms of weaving or unweaving aspects (step 4). From this modifications, a reconfiguration script is obtained. Finally, this script is executed to adapt the product (step 5). The following sections describe in detail the two parts: **Decision making** and **Runtime platform**.

6.4.1 Context Management

From the (step 1) of the cycle defined in Figure 6.5, we identify the need for a mechanism of context aggregation and manipulation. In Chapter 3 we introduced the context awareness as the capability of the systems to react to changes in their environment [Bro96, DAS01]. Examples of context information include location, temperature, hardware constraints, user preferences and personal information, time, etc. To identify and react to such changes in the environment, we need to model the context information which represents the data available in the environment when applications are being executed, and that may affect their structure or behavior.

In CAPucine, we do not propose a solution for context management. However, to provide the means to gather and aggregate context information, we consider that platforms like COSMOS [RCS08] can be used for the management of context information. COSMOS, which is a component-based framework for context management, obtains the different pieces of context information from different sources like sensors, network probes, or systems, and processes it according to defined policies. These policies are described as hierarchies of context *nodes* using a dedicated composition language. Each COSMOS node in the top of the hierarchy represents a single value. This allows developers to create context policies that aggregate context information from different sources, and that present the results as single boolean values which can be evaluated to trigger a reconfiguration. In CAPucine, we start from such boolean values and model the adaptations through the definition of events inside every aspect model.

Context Event Modeling

The boolean values obtained from the context manager have to be used as input in the decision making process. To model such values in our aspect metamodel, we define a fourth part called

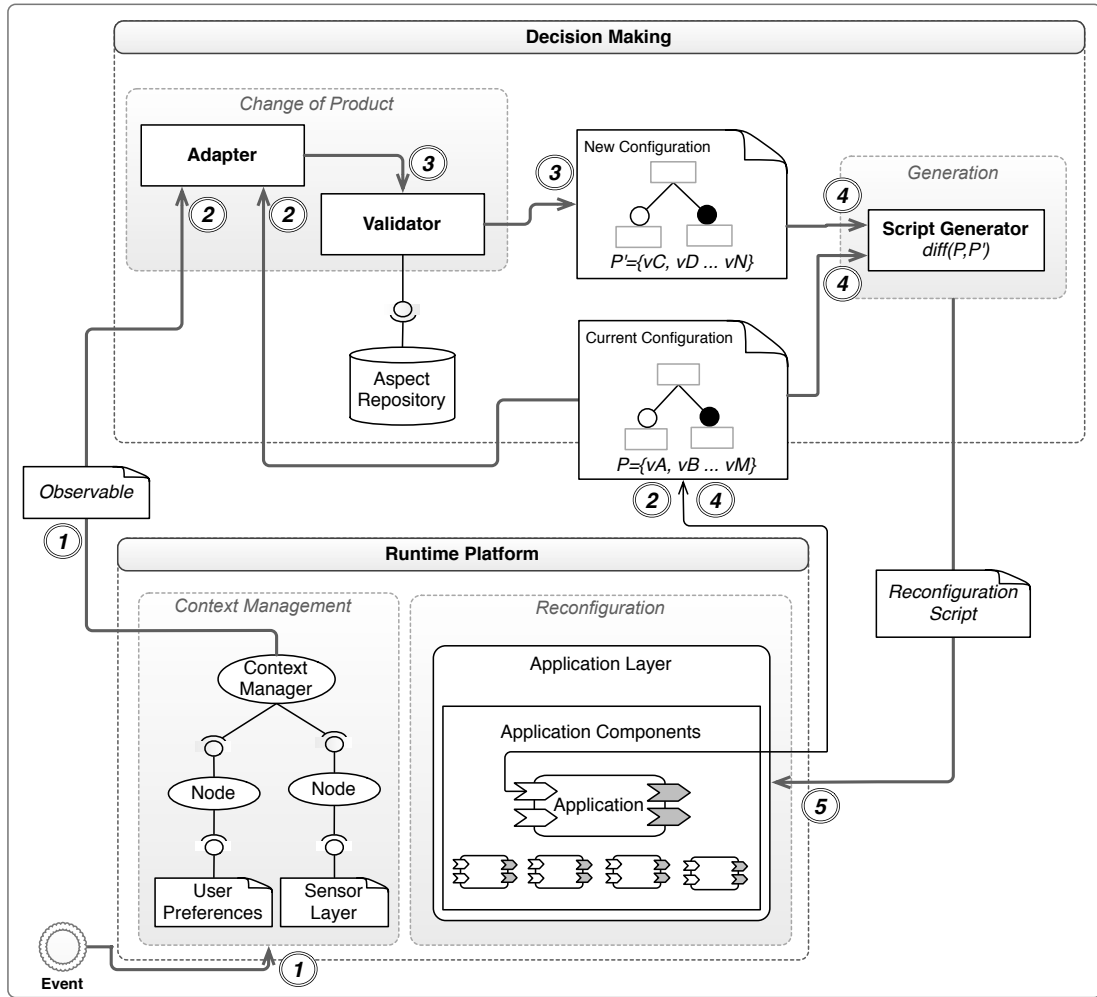
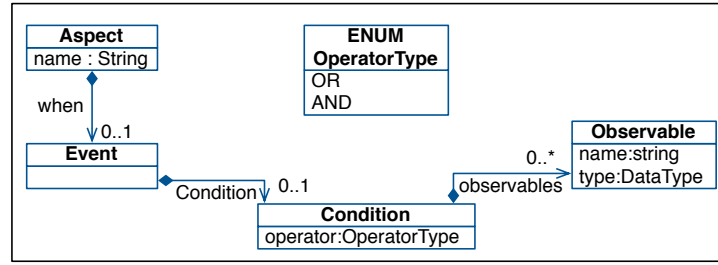


Figure 6.5: Runtime product derivation.

–when– that was already briefly introduced in Chapter 5. The *when* specifies the moment in the execution where the aspects get to be woven, and is based on the boolean values obtained from the context manager. To model this information, we use *context events*. Inside an event, we define the notion of *observable*. An observable is an abstraction of a single piece of information referring to context. It consists of a single value that can be easily evaluated to decide whether to weave an aspect dynamically or not. Figure 6.6 illustrates the structure for this part of the aspect meta-model. The *Event* meta-class defines one *Condition* that uses one or more *Observables*. The *Condition* has also an operator type (AND or OR) that is used to combine different *Observable* values. It is important to notice that only the aspect models that have events with observables in their definition can be used in the runtime phase to adapt the software products.

6.4.2 Decision Making

The decision making aims at obtaining the target configuration in terms of variants that have to be present in the product. It has two main inputs, the updated context information and the current

Figure 6.6: Aspect metamodel: definition of the `Event`.

configuration provided by each product. The output of such process is a set of reconfiguration actions written as scripts that are understood and executable in the runtime platform. Three main components constitute the decision making mechanism: *Adapter*, *Validator*, and *Script Generator*.

Adapter

The *Adapter* finds the *target* configuration expressed in terms of selected variants from the FD. As illustrated in Figure 6.7, the *Adapter* receives as input the *current* configuration for a particular product, and includes two different processes for finding, and then verifying the *target* configuration.

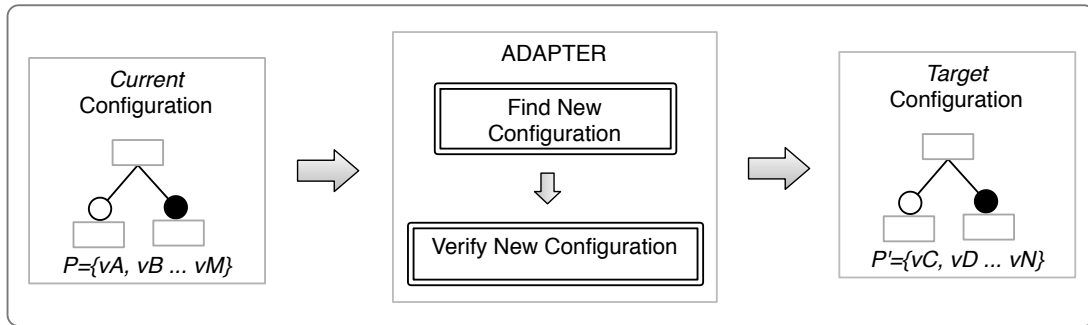


Figure 6.7: Reconfiguration: input and output.

Both the *current* and *target* configurations are expressed in terms of selected variants. However, since the notion of variant belongs into the feature model and consequently into the initial steps of the design, one of the first tasks to make the SPL dynamic is to have a dynamic representation of variants. To do this, we add an extra component to the core of every product derived from the SPL. Such component illustrated in Figure 6.8 is in charge of maintaining a representation of the current application state in terms of selected variants. It exposes one service with two operations, one for obtaining the current configuration, and one for updating the configuration. Both are used in the process of reconfiguration for, getting the current configuration and setting the new configuration every time the product gets adapted.

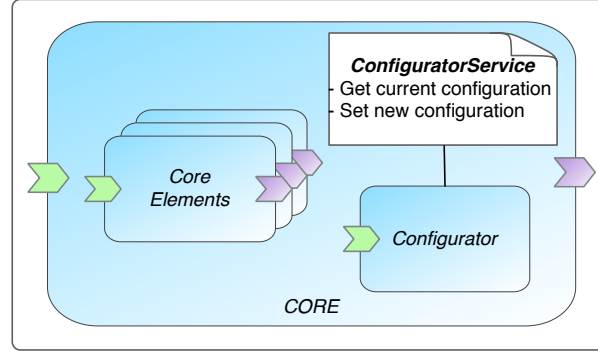


Figure 6.8: A component for managing product configuration at runtime.

In addition to variants, the `Adapter` uses the `Observable` elements of each aspect. The aspects models may include several observables indicating the moment during the execution when they have to be woven. When a context update occurs, the aspects that actually include one or multiple observables gets evaluated to decide if the variant has to be selected/deselected. This is equivalent to weaving/unweaving the aspect model.

The Algorithm 4 illustrates how the new feature configuration is obtained from context information. The algorithm requires to start, the set of updated observables, obtained from the context manager, and the current configuration obtained from the product itself. The first step consist in creating a target configuration with the same variants as in the current configuration (line 1). Afterwards, the algorithm iterates over the updated observables. For the observables whose aspect belongs to the current configuration, the algorithm verifies if the new observable value is false. In that case, the aspect has to be unwoven from the product. For the observables whose aspect does not belong to the configuration the algorithm verifies if the new observable value is true. In that case, the aspect has to be woven to the target configuration. After the variants of the aspects to weave have been selected, and the variants of the aspects to unweave have been deselected, the algorithm obtains the target configuration.

Algorithm 4 Adapter algorithm

Require: A set of updated context observables \mathcal{C}

Require: The current product configuration $P_{current} = \{F_1, F_2, \dots, F_k\}$

Ensure: A target product configuration P_{target}

```

1:  $P_{target} \leftarrow P_{current}$ 
2: for all ( $O_n \in \mathcal{C}$ ) do
3:   if ( $F_{O_n} \in P_{current}$ ) then
4:     if ( $O_n.value() = false$ ) then
5:        $P_{target}.deselect(F_{O_n})$ 
6:     end if
7:   else
8:     if ( $O_n.value() = true$ ) then
9:        $P_{target}.select(F_{O_n})$ 
10:    end if
11:  end if
12: end for

```

Validator

Since the `Adapter` decides the new configuration using only the information obtained from the updated observables, we need a process to verify if such configuration respects the feature constraints and aspect dependencies. The `Validator` guarantees that the target configuration remains a valid product from the DSPL and that it is achievable by a process of selecting and deselecting variants. However, this kind of analysis can be time-consuming for larger feature models. For this reason, in our approach, we prepare in advance the results of analyzing the feature model and the aspect dependencies at design time. This analysis results in a large table where the complete list of valid products derivable from the feature diagram are stored. Valid products are those that respect the FD constraints and that are composable. To create the table with the valid products we define the following steps:

1. **Count the number of products** First of all, from the feature model, we obtain the product family size i.e., the number of configurations that can be obtained from the model. The size depends on the number and the kind of variants found in the feature diagram: for optional variants, there are 2^n configurations, where n represents the number of optional variants. For alternative non-exclusive variants there are $(2^n - 1)$ configurations respectively, where n represents the number of alternatives for each variation point. Finally for alternative-exclusive variants there are n configurations where n represents the number of alternatives for each variation point. To illustrate this calculation, consider the feature model introduced in part *a* of Figure 5.2. This FD has in total 9 variants, classified as follows: 4 optional features (`SMS`, `Call`, `CreditCard`, and `Discount`), 2 alternative-non-exclusive variants (`Wifi` and `GPS`), and finally, 3 alternative exclusive variants (`ByDiscount`, `ByWeather`, and `ByLocation`). Applying counting formulas previously defined, we have:

$$(2^4) * (2^2 - 1) * 3 = 144$$

2. **Generate all the possible product configurations** To generate all the possible combinations we use the size obtained from each type of variants and iterate over it to generate a configuration. Every configuration represents a set of variants selected. Once we have obtained a table for each type of variant, we consolidate the three results into a single table (with 144 rows for the previous example), where every row represents a possible product configuration. For example, Table 6.1 lists all the combinations obtained from the optional variants of the feature diagram (e.g `Call`, `SMS`, `CreditCard`, and `Discount`). Each variant is shown in a column. We use a checkmark to indicate if the variant is selected, or a dash in the opposite case.
3. **Delete the configurations that do not respect the FD constraints** From the table obtained in the previous step, we eliminate the combinations that do not respect the FD constraints. For instance, the feature diagram illustrated in part *a* of Figure 5.2 includes one *requires* constraint. All the products that include the variant `ByLocation` must include one of the child variants from the alternative variation point `Location`. The products that do not respect this constraint are removed from the table. In other words, we apply the *Left to Right* analysis (see Section 5.4.1 of Chapter 5) to verify every configuration of the consolidated table.

Table 6.1: Optional variant combinations.

Configuration	Call	SMS	CreditCard	Discount
1	–	–	–	–
2	–	–	–	✓
3	–	–	✓	–
4	–	–	✓	✓
5	–	✓	–	–
6	–	✓	–	✓
7	–	✓	✓	–
8	–	✓	✓	✓
9	✓	–	–	–
10	✓	–	–	✓
11	✓	–	✓	–
12	✓	–	✓	✓
13	✓	✓	–	–
14	✓	✓	–	✓
15	✓	✓	✓	–
16	✓	✓	✓	✓

4. **Delete the configurations that have conflicts at the aspect level** Finally, to guarantee that the products in the table are actually composable we look for *hidden* constraints that are only detectable by analyzing the aspects for each variant. In other words we apply the *Right to Left* analysis (see Section 5.4.1 of Chapter 5). in order to remove the configurations that are not composable due to implicit aspect dependencies.

At the end of this process, we obtain a table with the complete list of valid products. Dynamically, using this table, the `Validator` matches the result coming from the `Adapter` with the entries in the table. If the product configuration is found in the table, the product is valid and the adaptation continues, if the product configuration is not in the table, the product is considered to be invalid and the adaptation stops.

Script Generator

The script generator has two inputs: the current feature configuration as provided by the running product, and the validated target configuration obtained by the `Adapter`, and verified by the `Validator`. Its goal is to obtain a reconfiguration script. To do it, it performs a difference between the two input configurations. The result of such operation is a list of variants that have to be deselected (i.e. aspects to unweave), and a list of variants that have to be selected (i.e. aspects to weave).

Since aspect models implement the variants and contain the information required in terms of: places affected by the reconfiguration, and actions to follow, they are used to build the weave/unweave script accordingly. A transformation takes each aspect model and generates the needed reconfiguration scripts. Each part of the aspect is transformed as described in Figure 6.9.

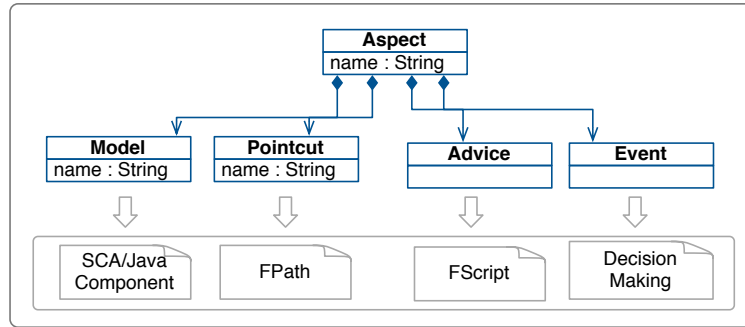


Figure 6.9: An aspect at runtime.

Model Since the model part of the aspect is expressed in the same way as both the core model and the composed model, every aspect is transformed as it was presented in Section 5.4.3 of Chapter 5. Concretely, the model is transformed into SCA and Java models and then the code is generated to obtain an SCA descriptor and Java classes implementing the components.

Advices Advices are transformed into FScript code [DLLC09], which is a scripting language dedicated to architectural reconfigurations of Fractal-based systems. A reconfiguration in our case consists of two main steps: (1) find the place, and (2) perform the modifications. The former step corresponds to FPath code. Similarly to an advice that uses pointcuts to describe the places where it performs the modifications, FScript uses FPath to find the places in the architecture where the adaptation will be applied. For each variable required by the advice, there is an FPath script (generated from the pointcut). The latter step is a translation of the `Modify` instruction into FScript code.

Pointcuts The `Pointcuts` are transformed into FPath code [DLLC09]. FPath is a query language to navigate Fractal-based Architectures. It eases the navigation of component systems and enables developers to define queries that search for elements of the architecture that match some criteria. The goal of the pointcut transformation is to map every `Expression` into the FPath script that finds the specific elements of the architecture of the application being executed. Hence, every atomic expression has an equivalence in terms of FPath. FPath also allows for multiple queries to be combined by using the `|` operator for a union and the `&` operator for an intersection. In this way, every `CompositeExpression` is translated into a union or an intersection of its sub-expressions. Table 6.2 summarizes the equivalences between the pointcut model and the FPath scripts.

Events The events are not transformed into code but rather used in the decision making mechanism to obtain the list of updated observables and its corresponding values, as it was presented in Section 6.4.2.

Table 6.2: Pointcut transformation

Model Pointcut	FPath Equivalent	Meaning
<i>FindByName(name)</i>	<i>\$root/descendant-or-self::*[name().== 'name']</i>	-All the elements which name is equal to "name".
<i>Type</i>		
- <i>Service or Reference</i>	<i>\$root/descendant-or-self::*/interface::*</i>	-All the interfaces.
- <i>Attribute</i>	<i>\$root/descendant-or-self::*/attribute::*</i>	-All the attributes.
- <i>Element</i>	<i>\$root/descendant-or-self::*</i>	-All the components.
<i>Owned</i>	<i>\$root/descendant-or-self::*/child::*;</i>	-All the components owned by another component.
<i>CompositeExpression</i>		
- <i>Operator= OR</i>	<i>(exp1 exp2) or union()</i>	-Union of two expressions.
- <i>Operator= AND</i>	<i>(exp1 & exp2) or intersection()</i>	-Intersection of two expressions.

6.4.3 Runtime Platform

The runtime platform is the second element that interacts with software products in the adaptation cycle illustrated in Figure 6.5. The platform is formed by two main parts: the context manager, which we have discussed previously; and a runtime executing platform, which supports the execution and dynamic adaptation of software products. For the latter one, we use FraSCAti a platform based on the principles of CBSE and SOA.

Application

Svahnberg *et. al.*, [SvGB05] define that variability realization techniques are used to integrate assets while building the final products [SvGB05]. Moreover, authors clearly identify *Component-Based Software Engineering* (CBSE) as one of the variability realization techniques that can be used at runtime. In CAPucine we use the FraSCAti platform [SMF⁺09]. FraSCAti is a Fractal-based SCA implementation. SCA establishes that components are the basic building blocks. Each component requires and provides services. SCA supports several service description languages like WSDL and Java interfaces, several programming languages such as Java, C++, and BPEL, several communication protocols between applications such as SOAP, CORBA, Java RMI, and JMS. Fractal [BCL⁺06], on the other side, is a hierarchical and reflective component model intended to implement, deploy, and manage complex software systems. Fractal offers several features like composite components (components containing subcomponents), sharing (multiple enclosing components for the same subcomponent), introspection, and re-configuration. A Fractal component can expose elements of its internal structure and offer introspection and intercession capabilities. Several controllers have been defined in the Fractal specification like the binding controller that allows the dynamic binding and unbinding of component interfaces, and the life-cycle controller that allows to perform operations like stop and start the execution of a component. In FraSCAti, Java-based SCA components are simultaneously both SCA-compliant and Fractal-compliant. The main benefit of this particular property is that all the components can be dynamically reconfigured at runtime.

Illustrative Example

To illustrate the use of FraSCAti, consider the product family introduced with the feature diagram of Figure 5.2. In particular, consider the core model defined for this example and the aspect model that implements the feature SMS. For the purposes of the example, we postpone the weaving of the aspect model. Let us assume that no aspects has been woven during the design phase. Consequently, in order to obtain a product, the core model is transformed into platform specific models and code. The resulting architecture of the core in SCA is illustrated Figure 6.10 with a component diagram. It shows SCA components for each of the elements identified in the architecture shown in Figure 5.8 (FrontEnd, Catalog, Notification, and Payment) with its corresponding services and references. Additionally, the core includes the `Configurator` component that, as it was explained in Section 6.4.2, is used to obtain and update the current configuration of the product in terms of selected variants.

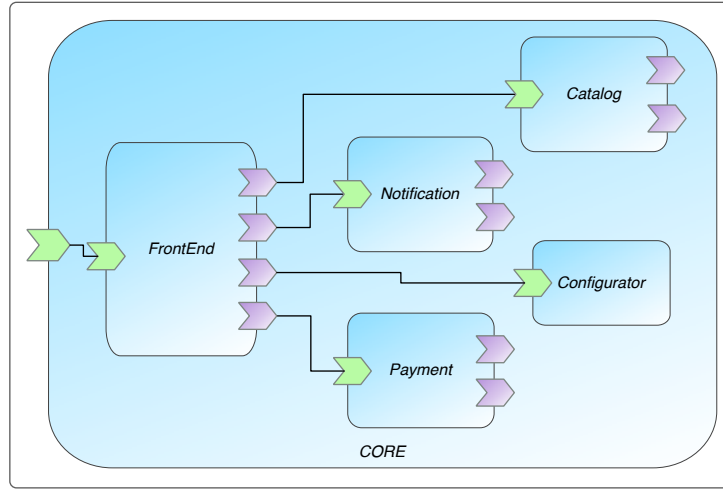


Figure 6.10: Core architecture represented in SCA.

To illustrate how the adaptation works, we proceed with the weaving of the aspect for the same variant SMS. First of all, the aspect has to be completed. In other words, we have to add the `Event` part that is missing from the aspect model used during the design phase. If the aspect does not have an event, there would not be any observables associated and therefore, it would not be used as part of the decision making to adapt a product at runtime. Consider the new version of the aspect illustrated in Figure 6.13 for the runtime phase. What differentiates this aspect from the design one, is its fourth part: the `Event`. It defines one condition over an observable called `online`. This means that dynamically, whenever the system receives a notification that it has lost the internet connection, the aspect must be woven.

Afterwards we perform the mapping presented in Figure 6.9. The aspect gets transformed into different snippets of code for: (1) the part of the architecture it contains (in Java and SCA), and (2) the modifications it performs over the core (in FScript and FPath). The snippet of code in Figure 6.12 depicts the script obtained from the `Advice` and `Pointcut` parts of the aspect implementing the variant SMS of the feature diagram.

As it can be seen from the code, in this case, the modification consist in adding a new element `SMSNotification` and bind it to an existing component `Notification`. The first part of the snippet (lines 5 through 8) looks for the places where the modification takes place. This part represents the equivalent of the pointcut expression that looks for a reference in the `Notification`

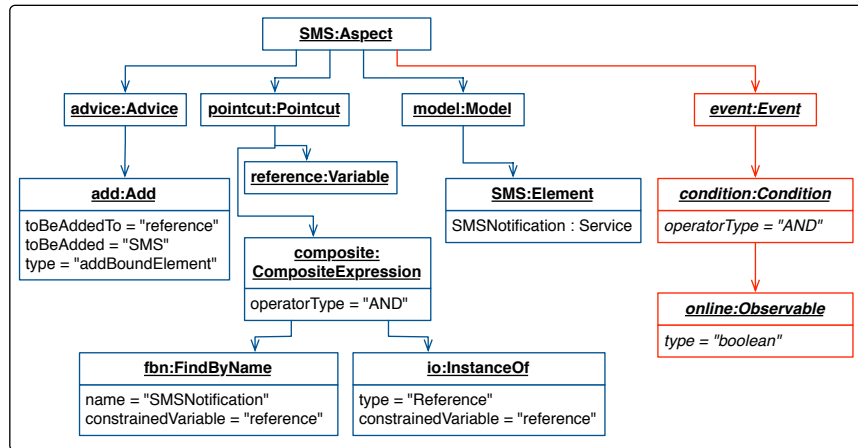


Figure 6.11: Aspect model for the runtime phase.

```

1 action addElement()
2 {
3   —Step 1: Find the place (FPath)
4   notifier=$root/descendant-or-self::*[name(.)== 'Notification'];
5   refNotifier=$root/descendant-or-self::* / interface::*[name(.)== 'sendSMSNotification'];
6   sms=$root/descendant-or-self::*[name(.)== 'SMSNotification'];
7
8   —Step 2: Perform the adaptation (FScript)
9   stop($notifier); —stops the notifier
10  bind($refNotifier, $sms/interface::smsNotification); —creates the binding
11  start($notifier); —starts the notifier component now bound to the SMS
12  start($sms); —starts the SMS component
13 }

```

Figure 6.12: An FScript equivalent of the SMS advice.

element. The second part performs the `addBoundElement` which creates a binding between the reference of the `Notification` element and the service offered by the `SMSNotification` element that is being added.

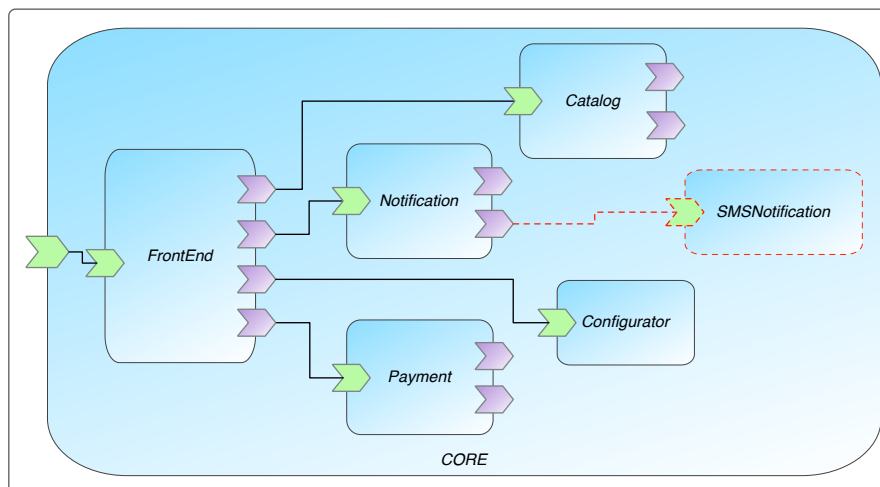


Figure 6.13: Result of the runtime weaving.

6.5 Discussion

We have presented the adaptation cycle and all its components used to adapt products at runtime in the DSPL. Let us now revisit the challenges identified at the beginning of the chapter and discuss how our approach faces each one of them.

1. **Extend the concept of feature at runtime** To face this challenge, we have added a component to every single product derived from the DSPL that is in charge of keeping their own configuration in terms of selected variants. The component is used to obtain the current configuration and to update it with the target configuration after an adaptation has been performed.
2. **Find the appropriate reconfiguration using several pieces of information** To face this challenge, we have implemented a decision making mechanism that obtain a target configuration using the context events that are queried and processed through the observables. The result is transformed into a set runtime modifications which are the equivalent of the processes of weaving or unweaving the aspect models defined in the design phase.
3. **Avoid conflicts and maintain SPL consistency** To avoid conflicts we have included a verification process as part of the decision making mechanism that guarantees that, from all the target configurations obtained from the context updates, only valid configurations (respecting FD constraints and aspect dependencies) are used to generate the reconfiguration scripts.
4. **Use a platform that supports dynamic adaptations** To overcome this challenge, we have based our DSPL on two platforms on which our DSPL is based: FraSCaTi and COSMOS. The former one fulfills the needs for an executing platform with dynamic reconfiguration properties, whereas the latter one deals with context management, which is used for the decision making mechanism, to chose the *target* configuration of the software products being adapted.

6.6 Summary

In this chapter we have presented the runtime phase of the DSPL. We have introduced an adaptation cycle with two different parts for the adaptation of software products: **Decision Making** and **Runtime Platform**. We have stated that, in order to maintain the DSPL product family, the **Decision Making** has to reason about adaptations in terms of variants, and that only valid configurations can lead to an adaptation of a product. We have also presented the adaptation itself as the change of a product from one *current* configuration, to a new *target* one. The aspect models of our DSPL are mapped into reconfigurations scripts in order to adapt the products at runtime. The **Runtime Platform** allows us to adapt the products and manage context information. We have based the reconfiguration on FraSCaTi that, due to its reconfiguration properties, enables products to switch configurations. Finally, we have used COSMOS to aggregate context information and be able to reason about context updates in terms of boolean observables.

This chapter concludes the contribution of this dissertation, which started in Chapter 4 with a global picture of the Dynamic Software Product . We have explained in detail and with simple examples the two Phases defined for a DSPL: Design Phase (Chapter 5) and Runtime Phase

(Chapter 6). The next part of this dissertation is dedicated to discuss the advantages and limitations of the DSPL proposed, as well as to give more details on the experimentation and tool support developed for the research work.

Part III

Validation

Chapter 7

Validation

“The difference between theory and practice is that in theory, there is no difference between theory and practice. Anonymous”

Contents

7.1 Experimentation	102
7.1.1 The eStore Family with CAPucine	102
7.2 Tool Support	115
7.2.1 Model2Code Architecture	115
7.2.2 Feature Analysis	115
7.2.3 Model Composition	118
7.2.4 Model Transformation	121
7.2.5 Code Generators	122
7.3 Discussion	125
7.3.1 The Unification Challenge	125
7.3.2 Qualitative Analysis	128
7.4 Summary	131

In this chapter, we present the experimentation results of using CAPucine based on a retail case study, which is an extended version of the example product family introduced in chapters 5 and 6. We particularly emphasize on the complete derivation processes including the design and runtime phases as well as the different processes that are part of each phase. We also present the tools built around CAPucine. This includes a set of analysis tools and aspect weavers, as well as model transformations and code generators called `Model2Code`. We present all the phases involved in `Model2Code` that enable developers to build products starting with a feature selection and obtain at the end the source code and adaptation scripts. Finally we discuss the results of the experimentation with a qualitative analysis and a discussion on the motivations behind our choices, as well as advantages and limitations of our approach.

Structure of the Chapter

The remainder of this chapter is organized as follows. In Section 7.1 we describe the `CAPPUCINO` project and our experimentation with the case study for the mobile *e-Commerce* product family.

Next, in Section 7.2 we describe in detail, all the tools implemented for the development of applications using the DSPL. In Section 7.3 we present a discussion on the advantages and limitations of our approach. Finally, Section 7.4 summarizes and concludes the chapter.

7.1 Experimentation

The scenario presented in this section has been prepared as part of the CAPPUCINO project. CAPPUCINO is a collaboration project between two research laboratories, two commerce companies, and one IT company in the northern region of France. The main objective of the CAPPUCINO project is to build and adapt ubiquitous applications in open environments for the commerce industry. As part of the project, we have collaborated with our research and industrial partners in order to define a common scenario for mobile and adaptable applications. We have used this scenario as the target of the experimentation of CAPucine.

To highlight the challenges of dynamic software product lines, we use a mobile commerce scenario that introduces some of the new opportunities brought by mobile and context-aware systems, such as the availability to use context-aware information. The motivation behind such a case study is to define a family of adaptive applications in which final users interact with the system in order to perform typical tasks of an electronic commerce scenario like browsing and selecting different items from an online catalog. There are two special characteristics that make this particular scenario an appropriate case study for a DSPL: (1) it is oriented to mobile devices, and (2) it exploits context information to enrich the user experience.

We cover the two derivation processes defined in CAPucine. Starting from a product family, we build and validate the different assets that are used in the development and adaptation of software products. For the implementation of mobile applications we use Google Android as mobile platform and FraSCAti for the execution and reconfiguration. In general terms, the scenario presented in this chapter is a richer version of the examples presented in chapters 5 and 6. It shows a typical interaction between a final user and one of the applications obtained from our target family as described in the following paragraph:

Alice uses a software system in her mobile phone to search and buy different items from an online catalog. In particular, the system can use external information to offer customized services. For example, when subscribing to the reward point program, Alice automatically receives special offers and prices. She can also register important events in her calendar like her best friends birthdays. Using this information, customized notifications with a focused product offer may be pushed to her mobile phone. Alice can also find gift ideas by using special services that use information like weather or location, to retrieve a list of items that match the season or that are close in the surrounding shops. If she does not have Internet access, she can still browse an offline version of the catalog. The system can also send notifications via SMS or email

7.1.1 The eStore Family with CAPucine

In this case, Alice –the final user– uses its mobile application to search and buy items from an online catalog. We start by identifying a set of features representing all the different product configurations that constitute the product family. The first step is then to create a feature model to represent such features.

Feature Model and Product Family

We start by defining the feature diagram and corresponding feature model that represents the product family. The essential functionalities identified from the scenario constitute a family of mobile applications that includes among others: catalogs of items, search of items, local and remote queries, calendar, authentication, historic of items bought, and order management. Figure 7.1 depicts the feature diagram for this example.

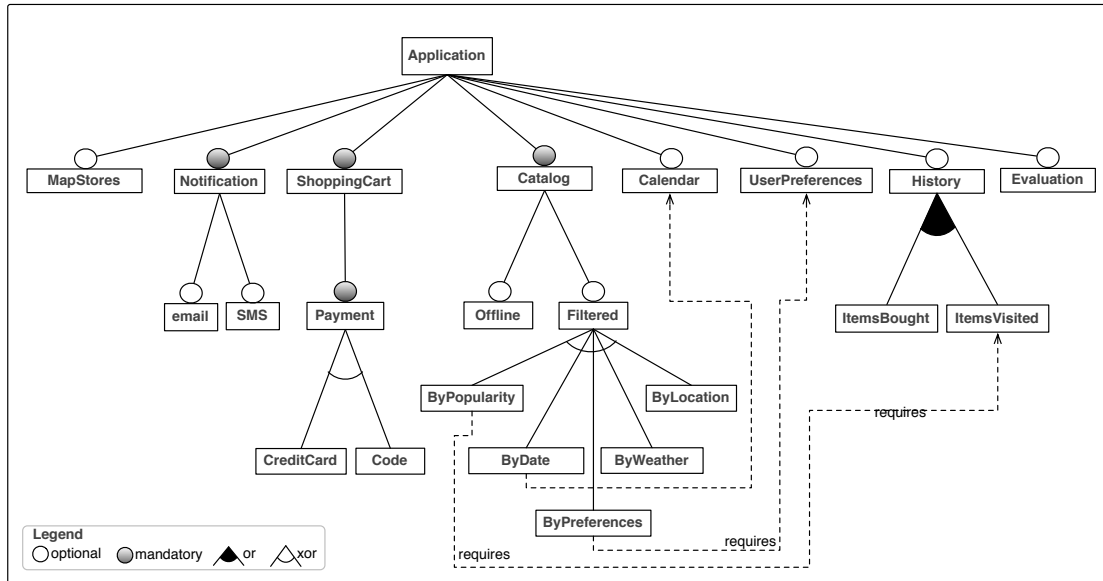


Figure 7.1: e-Commerce product family.

The feature diagram starts with the *Application* feature, the root of the diagram. It has eight different features: *MapStores*, *Notification*, *ShoppingCart*, *Catalog*, *Calendar*, *UserPreferences*, *History*, and *Evaluation*. *MapStores* is optional, it represents the functionality of visualizing the different stores in a map. The diagram includes a *Notification* feature which represents a way to communicate with the final user. The notification can optionally take place via eMail or sending an SMS. Next, there is the *ShoppingCart* feature. It has one child mandatory feature *Payment*. The payment can take place in two different ways: using a *CreditCard*, or by entering a buying code (*Code*). The diagram also includes a mandatory feature called the *Catalog*. It contains one optional feature *Offline* for working with cached catalog information locally. In addition to that, the *Catalog* feature has also a *Filtered* alternative, which is used to reduce the size of the catalog using different criteria like: popularity of the items being sold *ByPopularity*, special dates (*ByDate*), user preferences (*ByPreferences*), weather (*ByWeather*), or geographical location (*ByLocation*). The application may also have access to a *Calendar* for keeping track of special dates, and *UserPreferences* for customized services. What is more, the application can maintain a history of items that have been previously visited and bought (*ItemsVisited* and *ItemsBought*). This can be used to propose close-related items to the user based on their consumption habits. Finally, the application may allow the final user to give an evaluation about previously bought items (*Evaluation*). Additionally, the feature diagram introduces constraints between different variants in the feature model. Concretely, there are three *requires* constraints linking the following couples of variants:

ByDate and Calendar, ByPreferences and UserPreferences, and finally ByPopularity and ItemsVisited.

Context Information

In addition to the features identified below, we define the context information to which we want the products in the family to react during their execution. Table 7.1 summarizes the different elements of context to monitor, and the variants that are impacted by the modification of the context information.

Table 7.1: Context information.

Concept	Definition	Variant
<i>Internet Connection</i>	Indicates if the application has access to the Internet	Affects the variants that require an internet connection to work. In this case the variants affected are Email, and ByWeather
<i>Location</i>	Indicates the current geographical location of the user.	Affects the variants that use location information for customizing specialized services. In this case the variants ByLocation, ByWeather, and MapStore, are affected.
<i>Weather</i>	Provides the weather forecast for a specific location.	Affects the variants that use weather information. In this case the variant ByWeather is affected.
<i>UserPreferences</i>	Provide information about the user preferences.	Affects the variants that use the user information. In this case the variants ByPreferences and UserPreferences are affected.

Modeling the product Family using CAPucine

Using the metamodel of Figure 5.1, we proceed with the creation of an EMF representation of such diagram. The snippet of code in Figure 7.2 illustrates the different tags and elements of an EMF representation of such diagram. As it can be noticed from the code, the *requires* relationships are defined as attributes in the source variants and specify a link towards the required elements. Also, notice that the tags for the variants may include a boolean attribute called *selected* (lines 16, 24, and 39). This means that the same XMI file is used to represent a family, and one particular configuration. In practice, we need as many files as product configurations there exist.

To begin the derivation, we perform an analysis of this feature model in order to calculate the total amount of legal products which are defined as the configurations that respect the constraints of the feature diagram. In total there are 16 variants that can be selected (leaves in the diagram). The variants are classified as follows: 7 exclusive alternative variants (ByPopularity, ByDate, ByPreferences, ByWeather, ByLocation, CreditCard, and Code); 2 non-exclusive alternative variants (ItemsBought, and ItemsVisited); and 7 optional variants (Offline, MapStores Calendar, eMail, SMS, UserPreferences, and Evaluation).

Combining the three different lists to obtain the consolidated product family we obtain a total of 3840 possible configurations. This number comes from multiplying the lengths of each

```

1<?xml version="1.0" encoding="ASCII"?>
2<features:FeatureModel xmi:version="2.1"
3xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
4xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5xmlns:features="platform:/resource/Adapter/resources/metamodel/features.ecore">
6  <nodes xsi:type="features:Variant" name="MapStores" mandatory="false"
7    implementingAspect="amMaptStores"/>
8  <nodes xsi:type="features:Alternative" name="Notification" mandatory="true" exclusive="true">
9    <variant name="Email" mandatory="false" implementingAspect="amEmail"/>
10   <variant name="SMS" mandatory="false" implementingAspect="amSms"/>
11 </nodes>
12 <nodes xsi:type="features:VariationPoint" name="ShoppingCart" mandatory="true">
13   <variationPoint xsi:type="features:Alternative" name="Payment" mandatory="true"
14     exclusive="true">
15     <variant name="CreditCard" mandatory="false" implementingAspect="amCreditCard"/>
16     <variant name="Code" mandatory="false" implementingAspect="amCode" selected="true"/>
17   </variationPoint>
18 </nodes>
19 <nodes xsi:type="features:VariationPoint" name="Catalog" mandatory="false">
20   <variant name="Offline" implementingAspect="amOffline"/>
21   <variationPoint xsi:type="features:Alternative" name="Filtered" mandatory="false"
22     exclusive="true">
23     <variant name="ByPopularity" mandatory="false" requires="//@nodes.6/@variant.1"
24       implementingAspect="amPopularityFilter" selected="true"/>
25     <variant name="ByDate" mandatory="false" requires="//@nodes.4"
26       implementingAspect="amCalendarFilter"/>
27     <variant name="ByPreferences" mandatory="false" requires="//@nodes.5"
28       implementingAspect="amPreferencesFilter"/>
29     <variant name="ByWeather" mandatory="false" implementingAspect="amWeatherFilter"/>
30     <variant name="ByLocation" mandatory="false" implementingAspect="amLocationFilter"/>
31   </variationPoint>
32 </nodes>
33 <nodes xsi:type="features:Variant" name="Calendar" mandatory="false"
34   implementingAspect="amCalendar"/>
35 <nodes xsi:type="features:Variant" name="UserPreferences" mandatory="false"
36   implementingAspect="amUserPreferences"/>
37 <nodes xsi:type="features:Alternative" name="History" mandatory="true" exclusive="false">
38   <variant name="ItemsBought" mandatory="false" implementingAspect="amItemsBought"
39     selected="true"/>
40   <variant name="ItemsVisited" mandatory="false" implementingAspect="amItemsVisited"/>
41 </nodes>
42 <nodes xsi:type="features:Variant" name="Evaluation" mandatory="false"
43   implementingAspect="amEvaluation"/>
44</features:FeatureModel>

```

Figure 7.2: XMI feature model.

particular list: 128 for the optional variants, 10 for the alternative exclusive variants, and 3 for the alternative non-exclusive variants. After deleting the configurations that do not respect the constraints, like for example all the ones that include the feature `ByPreferences` and not the feature `UserPreferences`, we obtain a total of 2817 product configurations left. This is the number of legal configurations obtained from the model introduced in Figure 7.1. Table 7.2 summarizes the results of this first analysis.

Aspect Modeling

Aspect models are used to create a high-level representation of variants. For the scenario, we have built a core model that represents the mandatory features of the feature model, and one aspect model for each variant in the feature model regardless of their type (optional, alternative exclusive, or alternative non-exclusive).

Since mandatory features are always present in every product, they are modeled all together as the core model. The core model of our example is illustrated in the snippet of code of Figure

Table 7.2: Summary results of the feature model analysis.

Concept	Value
Variants	16
Number of optional variants	7
Combinations obtained from optional variants	128
Number of alternative exclusive variants	7
Combinations obtained from alternative exclusive variants	10
Number of alternative non-exclusive variants	2
Combinations obtained from alternative non-exclusive variants	3
Consolidated Size	3840
Invalid configurations (requires and excludes violations)	1024
<i>Final size of the product family</i>	2816

7.3. The model starts with a `Container` element (line 8), which groups all the other elements in the architecture. There are four additional elements: `Catalog`, `Notification`, `Payment`, and `FrontEnd`. Besides the `Container` and the `FrontEnd`, which are created as part of the architecture, note that each element is intended to realize one of the mandatory features found in the feature model. Additionally, as we discussed in Chapter 6 we need an extra element called the `Configurator` (lines 31-33), which is in charge of keeping the configuration of the product at runtime. The core model also includes business objects and contracts that specify the relationships between services and references in the elements. To simplify the code, we have omitted those elements from the snippet of code presented in the figure.

After the core model, we created an aspect model for each variant in the feature model. There are in total 16 aspects which realize the different variants of the feature model. As an illustration, consider the aspect model presented in the snippet of code of Figure 7.4. It contains the four parts of an aspect model, advice, pointcut, and event. The model includes the elements needed to have a notification via e-mail when the client makes an order to the store. The advice indicates how those elements have to be added. In this case, we define an `addBoundElement` meaning that the elements of the model are added to the core, and then a connection is created between the new element and the one found by the pointcut. The pointcut indicates a couple of queries that look the reference in the component that is in charge of sending the notification. Finally, the event indicates that, at runtime, this aspect is woven when there is an internet connection.

We summarize the different characteristics of the aspects models created for every particular feature in the Table 7.3.

```

1<?xml version="1.0" encoding="ASCII"?>
2<model.java.capucine:Aspect xmi:version="2.1"
3xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
4xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5xmlns:model.java.capucine="platform:/resource/Adapter/resources/metamodel/capucine.ecore"
6name="core">
7  <what>
8    <referencedElement xsi:type="model.java.capucine:Container" name="eshop">
9      <services name="FrontEndService" type="//@what/@referencedElement.25"/>
10     <elements name="Catalog">
11       <services name="CatalogQuery" type="//@what/@referencedElement.10"/>
12       <references name="RemoteCatalog" service="//@what/@referencedElement.11"/>
13       <references name="LocalCatalog" service="//@what/@referencedElement.17"/>
14       <references name="FilterProducts" service="//@what/@referencedElement.12"/>
15     </elements>
16     <elements name="Notification">
17       <services name="SendNotification" type="//@what/@referencedElement.9"/>
18       <references name="SMSNotification" service="//@what/@referencedElement.13"/>
19       <references name="EmailNotification" service="//@what/@referencedElement.14"/>
20     </elements>
21     <elements name="Payment">
22       <services name="PaymentService" type="//@what/@referencedElement.8"/>
23       <references name="CreditCardPayment" service="//@what/@referencedElement.15"/>
24       <references name="DiscountPayment" service="//@what/@referencedElement.16"/>
25     </elements>
26     <elements name="ShoppingCart">
27       <services name="Authentication" type="//@what/@referencedElement.18"/>
28       <services name="Payment" type="//@what/@referencedElement.26"/>
29       <references name="PaymentService" service="//@what/@referencedElement.8"/>
30     </elements>
31     <elements name="Configurator">
32       <services name="ConfiguratorService" type="//@what/@referencedElement.24"/>
33     </elements>
34     <elements name="FrontEnd">
35       <services name="FrontEndService" type="//@what/@referencedElement.25"/>
36       <references name="FECatalog" service="//@what/@referencedElement.10"/>
37       <references name="FENotification" service="//@what/@referencedElement.9"/>
38       <references name="FEShoppingCart" service="//@what/@referencedElement.26"/>
39       <references name="FEConfigurator" service="//@what/@referencedElement.24"/>
40       <references name="FEUserPreferences" service="//@what/@referencedElement.19"/>
41       <references name="FEHistory" service="//@what/@referencedElement.20"/>
42       <references name="FEEvaluation" service="//@what/@referencedElement.21"/>
43       <references name="FEMapStores" service="//@what/@referencedElement.22"/>
44       <references name="FECalendar" service="//@what/@referencedElement.23"/>
45     </elements>
46     <activities name="MyActivity">
47       <connections name="ConnectionOne"
48         caller="//@what/@referencedElement.0/@elements.0/@references.1"/>
49       <connections name="promoteContainer"
50         caller="//@what/@referencedElement.0/@services.0"
51         called="//@what/@referencedElement.0/@elements.5/@services.0"/>
52       <connections name="payment"
53         caller="//@what/@referencedElement.0/@elements.3/@references.0"
54         called="//@what/@referencedElement.0/@elements.2/@services.0"/>
55     </activities>
56   </referencedElement>
57   <!--Business Objects-->
58   <!--Contracts-->
59 </what>
60</model.java.capucine:Aspect>

```

Figure 7.3: XMI core aspect model.


```

1<?xml version="1.0" encoding="ASCII"?>
2<model.java.capucine:Aspect xmi:version="2.1"
3xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
4xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5xmlns:model.java.capucine="platform:/resource/Adapter/resources/metamodel/capucine.ecore"
6name="amEmail">
7  <how>
8    <modifications xsi:type="model.java.capucine:Add" type="addBoundElement"
9      toBeAddedTo="//@where.0/@rootExpression/@variable"
10     toBeAdded="//@what/@referencedElement.0"/>
11  </how>
12  <where name="notifier">
13    <rootExpression xsi:type="model.java.capucine:CompositeExpression">
14      <variable name="reference"/>
15      <expressions xsi:type="model.java.capucine:InstanceOf" name="type" type="Reference"
16        constrainedVariable="//@where.0/@rootExpression/@variable"/>
17      <expressions xsi:type="model.java.capucine:FindByName" name="EmailNotification"
18        constrainedVariable="//@where.0/@rootExpression/@variable"/>
19    </rootExpression>
20  </where>
21  <what>
22    <referencedElement xsi:type="model.java.capucine:Element" name="Email">
23      <services name="EmailNotification" type="//@what/@referencedElement.1"/>
24    </referencedElement>
25    <referencedElement xsi:type="model.java.capucine:ServiceContract"
26      name="NotificationEmailContract">
27      <operators special="" name="notifyEmail"/>
28    </referencedElement>
29  </what>
30  <when>
31    <condition>
32      <observable name="online" type="boolean"/>
33    </condition>
34  </when>
35</model.java.capucine:Aspect>

```

Figure 7.4: XMI aspect model for the Email variant.

Table 7.3: Summary of aspect modeling

Aspect	Variant	Model Elements	Modifications	Pointcuts	Runtime	Observables
core	Mandatory	124	–	–	–	–
amOffline	Offline	16	1	1	NO	0
amPopularityFilter	ByPopularity	12	1	1	NO	0
amCalendarFilter	ByDate	27	2	2	NO	0
amPreferencesFilter	ByPreferences	25	2	2	YES	1
amWeatherFilter	ByWeather	16	1	1	YES	2
amLocationFilter	ByLocation	15	1	1	YES	1
amMapStores	MapStores	15	1	1	YES	1
amCreditCard	CreditCard	12	1	1	NO	0
amCode	Code	21	2	2	NO	0
amCalendar	Calendar	17	1	1	NO	0
amEmail	eMail	15	1	1	YES	1
amSms	SMS	12	1	1	NO	0
amUserPreferences	UserPreferences	16	1	1	YES	1
amItemsBought	ItemsBought	13	1	1	NO	0
amItemsVisited	ItemsVisited	13	1	1	NO	0
amEvaluation	Evaluation	17	1	1	NO	0
<i>Total</i>		386	19	19	–	7

Constraint Analysis

At this point we have both the left side (Feature Configuration), and the right side (Aspect Models). It is possible now to perform the constraint analysis presented in Chapter 5. We have chosen 15 random legal product configurations out of the 2816 that can be obtained from the feature model to show the different warnings and outputs that can be obtained from running the algorithms. The results of the constraint analysis are summarized in Table 7.4. For each product we present its configuration using checkmarks for the variants selected. Next we present the results of the *left to right* (l2r) and *right to left* (r2l) algorithms, the order of the composition (Result), and the execution time (Time) in milliseconds.

As it can be seen from the results, the analysis for each product takes slightly short times. Nevertheless, the more variants there exist, the more aspects to verify for each product with consequences in performance, but such an overload is related to the nature of the product family itself. Additionally, since this process is executed during the design phase, time and performance are less critical than correctness and conflict-free composition.

Regarding the results of the *left to right* analysis, we notice that the left to right algorithm modifies the order of composition of the products 6,7,8, and 15. That is because the variants `ByDate` and `ByPreferences` are placed in the selection before the variants they require, in this case `Calendar` and `UserPreferences`. This also means that the analysis has found the corresponding dependencies at the level of aspects for the *requires* relationships. On the other hand, for the products 1,4, and 11, the analysis prints a warning *RR(7,15)* (for *Redundant Requires*) between the variants `ByPopularity` and `ItemsVisited`. This means that the *requires* relationship between those variants does not have a corresponding dependency at the level of aspects, and hence, it might be considered for removal from the feature diagram. However, as previously stated, even if there is no corresponding dependencies, the constraint does not necessarily represent an error, it may come for example from a business rule and is not related with the implementing aspect models.

On the other side, the right to left analysis shows a warning for the products: 3, 4, 7, 10, 11, 13, and 14. The message is presented in the table as *HR(5,3)* (for *Hidden Requires*) between the aspects implementing the variants `Code` and `SMS`. This means that the aspect for the variant `Code` has a dependency with the aspect for the variant `SMS`. To understand this dependency, consider the snippet of code presented in Figure 7.5.

The first part (lines 2-11) corresponds to the pointcut defined in the aspect implementing the variant `Code`. The pointcut looks for a reference called `SMSNotification`. The only place to find such a reference is in the model part of the aspect implementing the `SMS` variant (lines 15-24 in the same Figure). As a result, products 3,10,11,13,14 are not allowed for composition. In the case of products 4 and 7, the analysis indicates the same dependency, however, the composition is allowed because those configurations include the variant `SMS`. Additionally, the order does not need to be changed since the variant `SMS` is already placed before the variant `Code`. After the analysis to the whole product family, a total of 704 configurations are removed because of the dependency problem found in the *right 2 left* analysis.

```
1<!--Pointcut of the variant Code -->
2 <where name="BoundTo">
3   <rootExpression xsi:type="model.java.capucine:CompositeExpression">
4     <variable name="notification"/>
5     <expressions xsi:type="model.java.capucine:FindByName" name="SMSNotification"
6       constrainedVariable="//@where.1/@rootExpression/@variable"/>
7     <expressions xsi:type="model.java.capucine:InstanceOf" name="" type="Service"
8       constrainedVariable="//@where.1/@rootExpression/@variable"/>
9   </rootExpression>
10 </where>
11 <what>
12
13
14
15<!--Model of the variant SMS-->
16 <what>
17   <referencedElement xsi:type="model.java.capucine:Element" name="SMS">
18     <services name="SMSNotification" type="//@what/@referencedElement.1"/>
19   </referencedElement>
20   <referencedElement xsi:type="model.java.capucine:ServiceContract"
21     name="NotificationSMSContract">
22     <operators name="notifySMS"/>
23   </referencedElement>
24 </what>
```

Figure 7.5: Aspect dependency.

Table 7.4: Constraint analysis results.

Product	MapStores	E-mail	SMS	CreditCard	Code	Offline	ByPopularity	ByDate	ByPreferences	ByWeather	ByLocation	Calendar	UserPreferences	ItemsBought	ItemsVisited	Evaluation	L2R	R2L	Result	Time
1	✓	✓	✓	✓	-	-	✓	-	-	-	-	-	-	✓	✓	-	RR(7,15)	-	{1,2,3,4,7,14}	52ms
2	-	-	-	✓	-	✓	-	-	-	✓	-	-	-	-	✓	✓	-	-	{4,6,10,15,16}	42ms
3	✓	✓	-	-	✓	-	-	-	-	-	✓	-	-	✓	✓	-	-	HR(5,3)	Not allowed	48ms
4	✓	-	✓	-	✓	✓	✓	-	-	-	-	-	-	✓	✓	✓	RR(7,15)	HR(5,3)	{ 1,3,5,6,7,14,15,16 }	61ms
5	-	-	✓	✓	-	-	-	-	-	✓	-	✓	-	-	✓	✓	-	-	{3,4,10,12,15,16}	53ms
6	✓	-	-	✓	-	-	-	-	✓	-	-	✓	✓	✓	✓	-	Order	-	{1,4,13,9,12,14,15}	54ms
7	-	-	✓	-	✓	✓	-	-	✓	-	-	-	✓	-	✓	-	Order	HR(5,3)	{ 3,5,6,13,9,15}	58ms
8	✓	-	-	✓	-	-	-	✓	-	-	-	✓	-	✓	-	✓	Order	-	{1,4,12,8,14,16}	47ms
9	-	-	✓	✓	-	✓	-	-	-	✓	-	✓	-	✓	✓	-	-	-	{3,4,6,10,12,14,15 }	66ms
10	✓	-	-	-	✓	-	-	✓	-	-	-	✓	-	✓	-	-	-	HR(5,3)	Not allowed	44ms
11	✓	✓	-	-	✓	-	✓	-	-	-	-	-	-	-	✓	-	RR(7,15)	HR(5,3)	Not allowed	40ms
12	-	-	✓	✓	-	-	-	-	-	✓	-	-	-	✓	✓	-	-	-	{ 3,4,10,14,,15 }	43ms
13	-	✓	-	-	✓	-	-	-	-	-	✓	-	-	✓	-	-	-	HR(5,3)	Not allowed	40ms
14	✓	✓	-	-	✓	✓	-	-	✓	-	-	-	✓	✓	✓	✓	-	HR(5,3)	Not allowed	75ms
15	✓	-	-	✓	-	-	-	✓	-	-	-	✓	-	✓	✓	-	Order	-	{ 1,4,12,8,14,15 }	48ms

Table 7.5: Composition and code generation results.

Product Product	Configuration	LoC Woven	Time	LoC Java	Time	LoC SCA	Time	Total LoC Java & SCA
1	{ 1,2,3,4,7,14 }	196	26ms	944	1592ms	192	231ms	1136
2	{ 4,6,10,15,16 }	191	21ms	979	1773ms	174	223ms	1153
3	{1,3,5,6,7,14,15,16 }	205	29ms	1006	1632ms	204	226ms	1210
4	{ 3,4,10,12,15,16 }	194	25ms	1006	1634ms	183	228ms	1189
5	{ 1,4,13,9,12,14,15 }	204	26ms	1047	1716ms	204	225ms	1251
6	{ 3,5,6,13,9,15 }	203	25ms	1047	1763ms	200	222ms	1247
7	{1,4,12,8,14,16 }	197	25ms	1047	1704ms	187	229ms	1234
8	{ 3,4,6,10,12,14,15 }	203	28ms	1030	1897ms	200	228ms	1230
9	{ 3,4,10,14,15 }	195	21ms	1047	1737ms	182	235ms	1229
10	{ 1,4,12,8,14,15 }	200	25ms	902	2260ms	195	291ms	1097

Transformation and Generation of Code

From the results of the constraint analysis, and using the order obtained for the allowed configurations, we obtain the woven models. We can proceed with the generation of code. As it was presented in Chapter 5 we use two *model to model* transformations towards Java and SCA. Using the models obtained in those transformations we perform *model to text* transformations to obtain the source code. Table 7.5 summarizes the results obtained from executing the transformations and generators on the 10 products whose composition was allowed in the constraint analysis. For each product, we show its configuration in terms of variants selected, the time to execute the transformation towards Java and SCA, and the size of the product obtained measured in terms of the number of lines of code.

Mobile Development

After the code of the products has been generated, we proceed with the execution and runtime adaptation. In particular we are interested by the context information and events that may trigger adaptation processes, which in CAPucine are translated into switching the configuration of the product being executed.

To execute the products obtained at design time, and given that CAPucine does not include any elements to automate the development of the Graphic User Interfaces (GUI), we need an interface between the final user and the functionality of the products. In order to test the products, we have developed by hand a mobile application using the Android platform.

We have chosen Android for two main reasons. First of all, its Java API allows us to access information about context that we use for testing the runtime adaptations. Second, although not yet implemented, it is feasible and foreseen to build a mobile version of *FraSCAti*. This means that in the future, SCA applications will possibly be executed in mobile phones. For the time being, a prototype has been developed which allows an android Activity to start an SCA composite as an Android service. However, the SCA application cannot be reconfigured, since the current mobile version of *FraSCAti* does not include the elements that enable reflection and dynamic reconfigurations. For this reason, rather than deploying the products inside the mobile device,

we have developed a GUI that acts as a client of our product which is executed in a desktop computer. The Android application interacts with both the application and the adapter as shown in Figure 7.6.

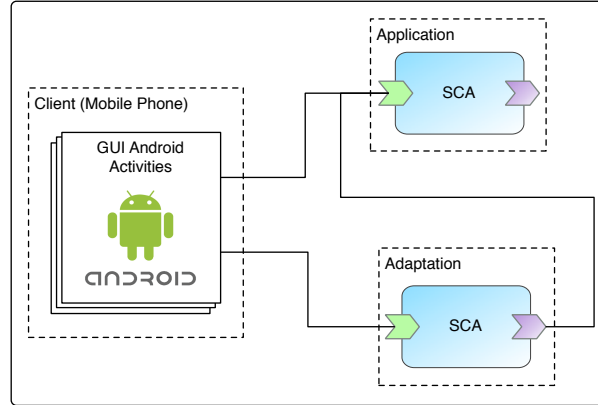


Figure 7.6: Mobile client architecture.

In this case, the GUI communicates with the adapter in order to provide the context information that is only accessible via the mobile device. The GUI also is bound and consumes the services provided by the application. In addition to that, the adapter communicates with the application in order to access the `Configurator` service whenever there is a need for an adaptation.

Context awareness and Aspect models

As we explained in Chapter 6, the reconfiguration scripts are obtained from the aspect models that include events, and hence, that can be woven dynamically. In our example, 6 aspect models include events as indicated in Table 7.3. More specifically, those aspects correspond to the variants: `ByPreferences`, `ByWeather`, `ByLocation`, `MapStores`, `Email`, and `UserPreferences`. Table 7.6 summarizes the results for the generation of reconfiguration scripts for these aspect models. For each aspect we present: the observables defined as part of its `Event`, the lines of code generated from their reconfiguration, and the execution of time for the generation. Notice that the number of lines of code only correspond to the reconfiguration operations, like the ones presented in the snippet of code in Figure 6.12. It does not include the Java or SCA lines of code for the implementation of the elements (in the `Model` part of the aspect) that are being added.

Summary of the experimentation

The experimentation presented in this section illustrates the development process of SPLs with CAPucine. We have presented the results of the model-based derivation process introduced in Chapter 5 and the adaptations at runtime introduced in Chapter 6 using the product family for the electronic store as a case study. We have illustrated in detail the results obtained from both

Table 7.6: Runtime adaptation summary.

Aspect Model	Context Observables	LoC Script	Time
ByPreferences	preferences	18	153ms
ByWeather	location,online	15	152ms
ByLocation	location	14	156ms
MapStores	location	12	148ms
Email	online	12	145ms
UserPreferences	preferences	14	147ms

the constraint analysis and the MDA process of transformation and code generation using the 16 variants and their corresponding aspect models, as well as the runtime adaptations presented in Chapter 6. To realize such experimentation, we have used the set of tools built around CAPucine. The next section presents the architecture and implementation details of such tools.

7.2 Tool Support

Automation within SPLs in general, and DSPLs in particular, aims at reducing the load of work on developers by identifying repetitive tasks that do not need human intervention and can be automatized. Usually, in software engineering these tools are in charge of generating source code of functionalities that are common and can be factorized from different applications. Such tools allow developers to focus on the business requirements and to spend less time solving common problems. CAPucine proposes different tools that can be used at different phases of the development process, from the analysis and requirements, and going all the way down to the execution and adaptation of products at runtime. In order to have a better understanding of the tools, in this section, we describe in detail the set of tools implemented around CAPucine that we call *Model2Code*.

7.2.1 Model2Code Architecture

Model2Code is divided in several parts covering the different phases of the application engineering process in CAPucine. Figure 7.7 illustrates such parts and how they correspond with separated phases of the CAPucine architecture. There are four parts in total as follows: (1) **Feature Analysis**, that covers the algorithms for the analysis of the feature and aspect models; (2) **Model Composition**, that covers the process of weaving of the aspect models; (3) **Model Transformation**, which includes a set of model transformations towards the specific platform; and finally (4) **Code Generators** to obtain the source code of the products. On top of these parts, we have implemented a simple GUI to access all these functionalities. However, several parts of *Model2Code* can be used programmatically or manually.

7.2.2 Feature Analysis

The first part of *Model2Code* includes a sets of algorithms for the analysis of the feature diagrams and aspect models. Figure 7.8 depicts the two main processes included in the feature

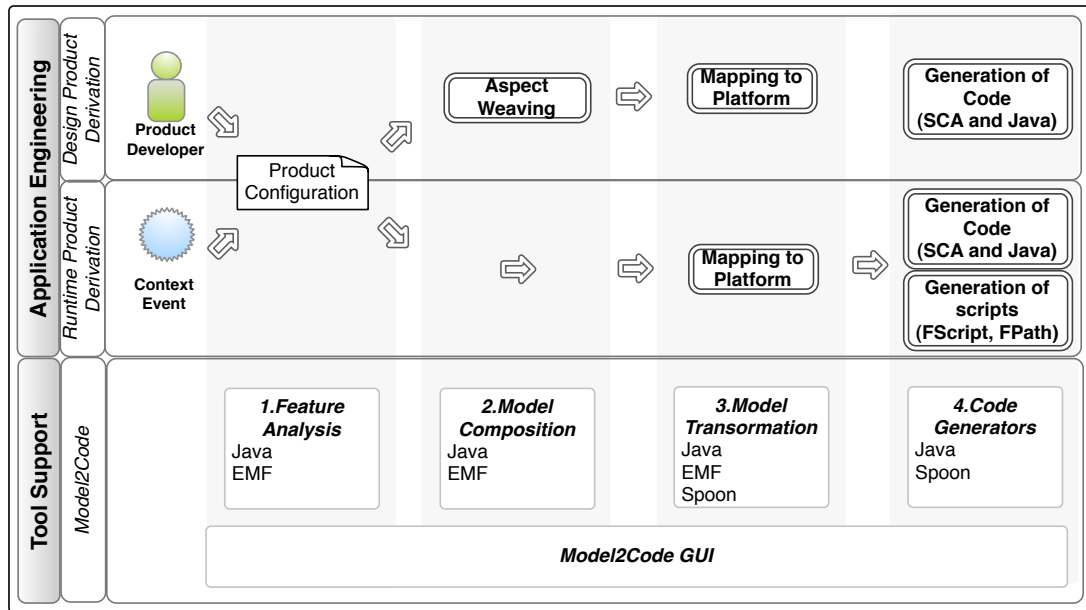


Figure 7.7: Model2Code architecture.

analysis. The first part of the analysis uses as input the feature model and obtains a list with all the legal configurations achievable for that model. The second part corresponds to the constraint analysis between features and aspect models. It uses as input the feature model, all the aspect models implementing the different variants of the features and a product configuration, and produces as a result a refined set of configurations (excluding the products with aspect dependency problems), the order of weaving, and several warnings for the developers. The list of legal configurations is used to verify at runtime if the adaptations respect the constraints of the feature model. The weaving order is used as input in the model composition part to weave the core model with the different aspects belonging to the given product configuration. Both parts of the analysis are implemented in Java and use the EMF API to create object representations of the feature and aspect model elements.

Legal configurations

For the first analysis, we generate a table with all the legal combinations from a feature model. To do that, we create different tables for the three types of variants available in the feature model: optional, non-exclusive alternative, and exclusive alternative. The size of each table is obtained by applying the formulas introduced in Section 6.4.2 of Chapter 6. Afterwards with the help of a binary string, we iterate over the size of each table and we create all the different combinations. The snippet of code of Figure 7.9 illustrates the cycle and the use of such binary string to create all the combinations for the optional variants.

Once the three tables have been created, we consolidate all the results into a single table, and eliminate the products that do not respect the *requires* or *excludes* constraints from the feature model. To do this, we basically traverse the feature model looking for couples of features that are linked through *requires* or *excludes* relationships. With the couples identified, we verify if there

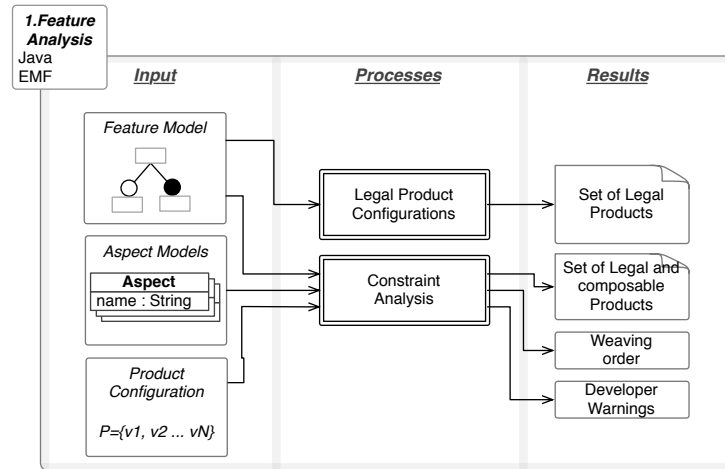


Figure 7.8: Feature analysis processes.

```

1 //Use a binary number to find all the possibilities
2 int maxSize;
3 String binary, reverse;
4 for(int i = 0; i <= iterator ; i++)
5 {
6     int maxSize = Integer.toString(Integer.toBinaryString(iterator)).length();
7     String binary = Integer.toBinaryString(i);
8     String reverse;
9     if(binary.length() < maxSize)
10    {
11        reverse = new StringBuffer(binary).reverse().toString();
12        while(reverse.length() < maxSize)
13        {
14            reverse = reverse.concat("0");
15        }
16        binary = new StringBuffer(reverse).reverse().toString();
17    }
18    for(int j=0;j<binary.length(); j++)
19    {
20        if(binary.charAt(j) == '1')
21        {
22            tableNormal[i][j]=this.normalVariants.get(j);
23        }
24    }
25 }

```

Figure 7.9: Use of binary strings to generate product configurations.

are products in the consolidated table that contain one feature of the couple and not the other one for the *requires* case, or both features at the same time in the *excludes* case. If such products are found, they are removed from the consolidated results. This operation is illustrated in the snippet of code of Figure 7.10. There is one verification for each constraint (lines 4 and 13). If an inconsistency is found in any case, the product configuration is removed from the consolidated collection.

```

1 //requires
2 Variant required, requiring;
3 //...find couples
4 if (configuration.contains(requiring) && !configuration.contains(required))
5 {
6     consolidated.remove(configuration);
7 }
8 //...
9
10 //excludes
11 Variant excluded, excluding;
12 //...find couples
13 if (configuration.contains(excluding) && configuration.contains(excluded))
14 {
15     consolidated.remove(configuration);
16 }
17 //...

```

Figure 7.10: Deletion of ilegal products.

Constraint Analysis

This process refers to the constraint analysis explained in Chapter 5. In a similar way as for the legal configurations, we use as input the EMF feature model and additionally, we also use the aspect models for each variant and a particular product configuration. The algorithms for the *left 2 right* and *right 2 left* analysis have been implemented in Java.

In the implementation of the algorithms we use `match()` methods to verify if the aspects have the corresponding relationships from the ones defined in the feature diagram. This is what we defined in the algorithm 1 as:

$$A_{F_1}.Pointcut \cap A_{F_2}.Model = \emptyset.$$

In the actual implementation, we compare sets of elements from the collections of objects obtained when instantiating an object representing the XMI aspect model file. For example, the method presented in Figure 7.11 illustrates the verification of the pointcut of one aspect with the model of a second aspect. As it can be noticed from the code, the pointcut of the first aspect is executed in the model of the second aspect (line 10). This action creates a collection with the elements that satisfy the pointcut expressions. If such collection is not empty, it means that the match exists.

7.2.3 Model Composition

The model composition (see Figure 7.12) refers to the iterative process of weaving a core model with one or several aspect models. The inputs for this process are: the core model, the aspect models to weave, and strategy (order) of composition or order obtained from the constraint analysis.

For the implementation of the weaving process, we rely on the Java code obtained from the EMF model. EMF generates the classes and interfaces needed to represent every meta class defined in the Aspect Metamodel as Java objects and collections. However, in order to add the composition logic required for the weaving process, we extend the generated code by adding `execute()` methods inside the `Expression` and `Modification` meta-classes. For the expressions, we use this method to write the specific algorithms in every class inherited from `expression`

```

1 private boolean matchPointcutModel(String sourcest, String targetst)
2 {
3     Aspect source = getAspectByName(sourcest);
4     Aspect target = getAspectByName(targetst);
5
6     for (Pointcut pointcut : source.getWhere())
7     {
8         if (pointcut != null)
9         {
10             pointcut.getRootExpression().execute();
11             if (pointcut.getRootExpression().getVariable().getContents().size() > 0)
12                 return true;
13         }
14     }
15     return false;
16 }

```

Figure 7.11: Matching pointcuts and models for the constraint analysis.

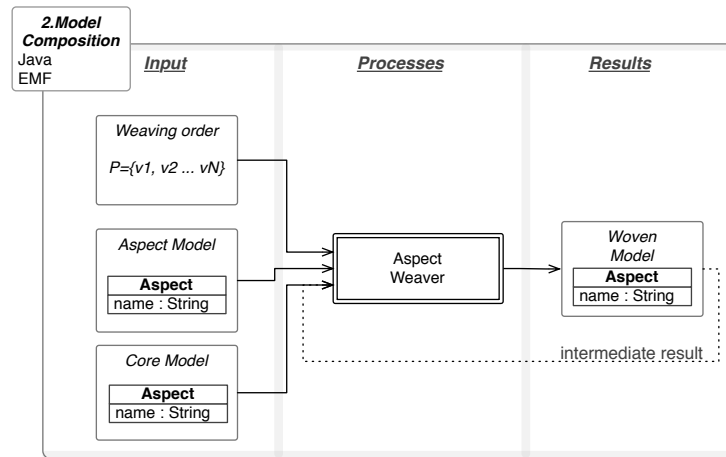


Figure 7.12: Model composition processes.

in the pointcuts: `CompositeExpression`, `FindByName`, `InstanceOf`, and `Owned`. Notice that `Atomic` expression is abstract, only concrete instances of it are allowed. Likewise, for the modifications, we define such method to insert the code in the classes that inherit from the `Modification` in the advices: `Add` and `Delete`. Figure 7.13 shows the hierarchy of the classes for both the `Expression` (part a) and the `Modification` (part b).

Pointcut execution

The first part of the weaving executes every expression defined in the pointcut of the model. Since every class in the lower level of the `Expression` hierarchy overrides the method `execute()`, the pointcut execution consist in iterating over the expressions and calling their `execute()` method. Every atomic expression (`FindByName`, `InstanceOf`, and `Owned`) introduces the code that filters the elements for its own case. For example, consider the code presented in Figure 7.14. It presents the implementation of the `InstanceOf`. It basically traverses the set of elements in the model and retains only the ones whose type corresponds to the type specified in the model.

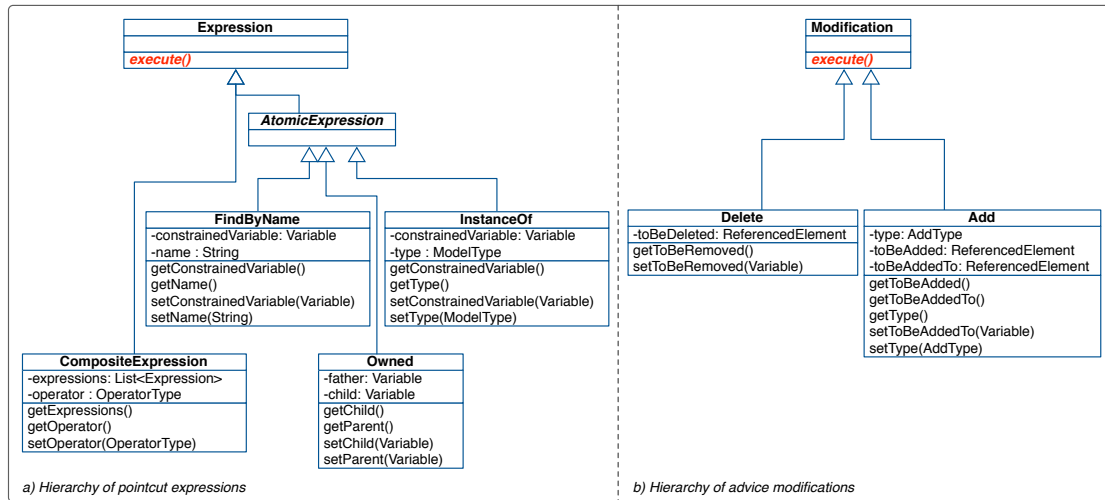


Figure 7.13: Expression and Modification hierarchy.

Notice that, thanks to the EMF generation, the type defined in the model is available as a private attribute `type` of the class `InstanceOf` (see line 12).

```

1      @Override
2      public EList<ReferencedElement> execute(Aspect core)
3      {
4          EList<ReferencedElement> list = new BasicEList<ReferencedElement>();
5          for(Iterator<EObject> it = core.eResource().getAllContents(); it.hasNext();)
6          {
7              EObject eo = it.next();
8              String className= eo.getClass().getCanonicalName();
9              String simpleName = className.substring(className.lastIndexOf(".") + 1,
10                                                         className.length());
11              simpleName=simpleName.replace("Impl", "");
12              if(simpleName.equals(this.getType().getName()))
13              {
14                  list.add((ReferencedElement)eo);
15              }
16          }
17          return list;
18      }

```

Figure 7.14: Implementation of the `InstanceOf` expression.

The results of all atomic expressions are consolidated in the `CompositeExpression` execution. It performs the intersection or union for each case, of the results obtained from the atomic expressions. Figure 7.15 shows the implementation of this method. It firsts calls the `execute` methods of all the expressions it contains. Afterwards, with the results obtained from the atomic expressions, it performs an intersection or union of such elements, depending on the type selected in the model.

```

1 @Override
2 public EList<ReferencedElement> execute(Aspect core)
3 {
4     ArrayList<HashSet<ReferencedElement>> all = new ArrayList<HashSet<ReferencedElement>>();
5     HashSet<ReferencedElement> intersection = new HashSet<ReferencedElement>();
6     int counter=0;
7     for (Expression exp : this.getExpressions())
8     {
9         all.add(new HashSet<ReferencedElement>(exp.execute(core)));
10    }
11    if(this.operator.getValue() == OperatorType.AND_VALUE)
12    {
13        for (HashSet<ReferencedElement> hashSet : all)
14        {
15            if(counter == 0)
16            {
17                intersection.addAll(hashSet);
18                counter++;
19                continue;
20            }
21            intersection.retainAll(hashSet);
22            counter++;
23        }
24    }
25    else if(this.operator.getValue() == OperatorType.OR_VALUE)
26    {
27        for (HashSet<ReferencedElement> hashSet : all)
28        {
29            intersection.addAll(hashSet);
30            counter++;
31        }
32    }
33    CapucineFactory factory = new CapucineFactoryImpl();
34    if(this.getVariable() == null)
35        this.setVariable(factory.createVariable());
36    this.getVariable().getContents().clear();
37    this.getVariable().getContents().addAll(intersection);
38    return (new BasicEList<ReferencedElement>(intersection));
39 }

```

Figure 7.15: Implementation of the Composite expression.

Advice execution

Likewise, the second part of the weaving consist in iterating over every modification defined in the advice and executing it. Since the results of the pointcuts may contain any element in the model, the add algorithm has to consider every possible combination of types in the form (*place, element to add*). There are as many combinations as allowed pairs we define. For the rest of the cases, no weaving is performed. The deletion is much simpler than the addition, the `execute` method only has to find the elements and delete them from their containing collection.

7.2.4 Model Transformation

The model transformation process (see 7.16) takes the woven model obtained from the composition phase and maps it into new platform-dependent models for SCA and Java. Hence, the only input for this process is the woven model obtained before. The output of this process are two new models that represent the woven model in terms of SCA and Java elements. Additionally, there is a verification between the new models to assure that the concepts defined in the SCA model have an equivalent implementation in the Java model. We have chosen to implement the transformations in this order because both, the SCA and Java models require only the architecture described

in the woven model. The verification between these results does not add any additional elements to the models already created. It just verifies that the elements of the architecture are reflected in both languages, and that the naming strategy is consistent for the code generated to be consistent.

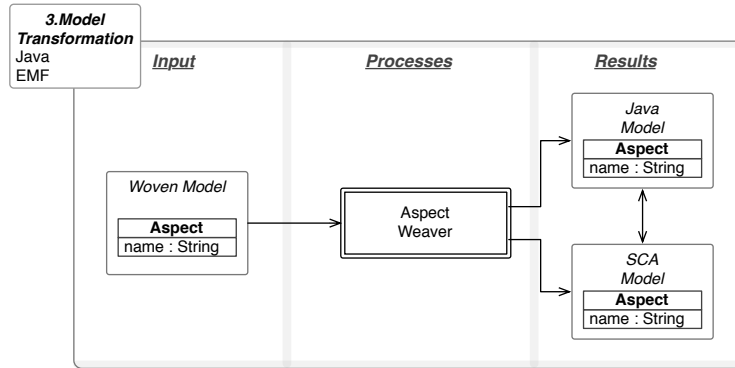


Figure 7.16: Model transformation processes.

The transformations in `Model2Code` are based on a classic strategy that traverses the source model. In this case, we only use the information found in the `Model` part of our aspect meta-model. The `Model` is hierarchical and defines one and only one starting point or root element that in this case is called the `Model` which contains several referenced elements. Starting from such root, the model is traversed with the help of iterators for every specialization of `ReferencedElement` found in the model. To improve the readability of the transformations and simplify further development, we have grouped all the methods used to traverse the source model in two different classes: `ToSCATransformer` for the transformations towards SCA, and `ToJavaTransformer` for the transformations towards Java. In both cases, for each element found in the `Model` part of the aspect model (i.e., `Element`, `Reference`, `Service`, `Contract`) there is an associated `ElementBuilder` class that implements the interface shown in Figure 7.17.

```
1 public interface ElementBuilder<X extends Object, Y extends EObject>
2 {
3     X build(Y element);
4 }
```

Figure 7.17: Interface for creating elements in the target model.

All the transformation code is based on the principle given by this interface. This means that for each element of the source model (the object `X`), there are one element in the target model (the object `Y`). There are as many `ElementBuilder` classes as many elements to create in the target model. For example, Figure 7.18 illustrates the `PropertyBuilder` class that creates a `Property` in SCA using the information of an `Attribute` in the aspect metamodel.

7.2.5 Code Generators

The code generators include three main parts: generation for java, generation for SCA, and generation for FScript and FPath. Figure 7.19 shows the three different generators with their corresponding input and outputs. For the generation, we use the `PrintWriter` interface from Java.

```

1 public class PropertyBuilder implements ElementBuilder<Property, Attribute>
2 {
3
4     public Property build(Attribute attribute)
5     {
6         String attributeName = attribute.getName();
7         Property property = new ScaFactoryImpl().createProperty();
8         property.setName(attributeName);
9         property.setElement(attribute.getType().getName());
10        return property;
11    }
12 }

```

Figure 7.18: The class PropertyBuilder.

Consequently, there are as many writers as elements to be generated in the three different target languages.

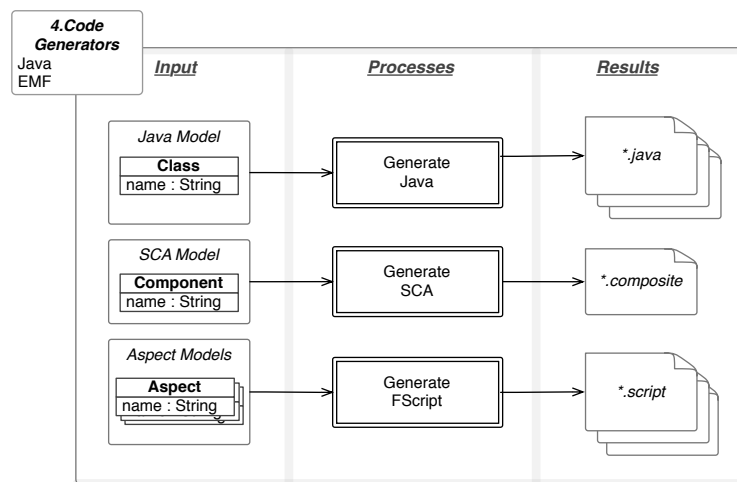


Figure 7.19: Code generation process.

Java Generation

For the Java generation, thanks to the use of the metamodel defined for the Spoon EMF project [Bar06], the generation of source code is direct. It consists in calling the Spoon EMF tools providing the XMI file. The tools implemented by this project include a `PrettyPrinter` mechanism that creates the Java classes associated with the elements of the model, and writes their contents with the proper syntax and indentation (see Figure 7.20). Additionally, we use Spoon templates to generate the code of certain methods that are common to every implementation like for instance the *getters* and *setters* for the attributes of classes.

SCA Generation

For SCA we also use `Writer` classes for each element in the SCA metamodel. The current implementation defines the writers defined for the following SCA types: `Component`, `Composite`,

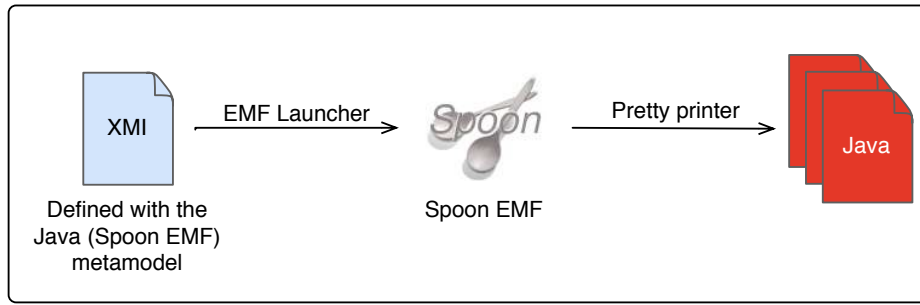


Figure 7.20: Java code generation.

Service, Reference, Property, and Wire. The result is the composition of the tags generated by each particular writer, that are put together under a same `Composite` element as defined in model used as input.

FScript

The generation of scripts for the runtime adaptation takes a different input. Unlike the case of SCA and Java, the generation of scripts takes as input the aspect model and directly creates the FScript code without an intermediate model transformation towards an FScript metamodel. Hence, we have defined a Java class called `FScriptWriter` that groups all the logic of the FScript generation. The current implementation supports the aspect advices for `Services`, `References` and `Connections`. With this subset of elements we are able to create adaptations that add components at runtime and create the bindings between those components and the core that is being executed.

Model2Code metrics

To conclude the description of the tool support, we present several metrics for the different parts of `Model2Code`. Table 7.7 summarizes these metrics. It includes the four parts previously introduced (i.e., Feature analysis, Model Composition, Model Transformation, and Code Generators), and a fifth part with other functionalities, like the GUI and the tests. For each part, the table enumerates the number of packages, classes (and interfaces), and finally the total amount of lines of code developed. It is important to notice that in the case of the design weaver, an important part of the code is automatically generated by the EMF tools. We have included the code because we modify it at multiple parts to add the behavior required to execute the composition of the aspect models. Notice also that the numbers shown in Table 7.7 do not include elements like: the code for the Java and SCA metamodels, the different EMF files (metamodels and models), and the configuration files (ANT scripts) that are used to manage the tool.

Table 7.7: Model2Code metrics.

Part	Packages	Classes	LoC
1. Feature Analysis			
Constraint Analysis	6	50	624
Legal Products	10	303	1201
Feature Metamodel	3	18	1026
2. Model Composition			
Design Weaver (with Aspect Metamodel)	4	105	6762
3. Model Transformation			
Architecture to Java	2	15	797
Architecture to SCA	2	10	350
4. Code Generators			
Generation of Java code	3	12	408
Generation of SCA code	3	10	392
Generation of FScript code	2	5	298
Others			
CAPucine GUI	8	21	2783
Configuration	4	22	1148
Tests	8	20	456
Total	55	591	16245

7.3 Discussion

The experimentation has helped us to identify the benefits as well as the weaknesses of CAPucine for the overall management of variability and context awareness in different setups, and in particular within DSPLs. In this section, we present discussion on the approach where we revisit the unification challenge identified in Chapter 3. Afterwards, we present a qualitative evaluation of CAPucine based on the criteria for a DSPL framework that was also introduced in Chapter 3.

7.3.1 The Unification Challenge

In Chapter 3, we have identified the unification of design and runtime adaptations as the most important challenge to face through a framework for DSPLs. To face this challenge we have made several decisions regarding the main assets used by CAPucine and the strategy for product derivation at design time as well as at runtime. Here we present a justification to such choices.

- **On the difference between design and runtime adaptations**

Applications can be adapted either during the design or the runtime phase. Design adaptations are motivated by design decisions, whereas runtime adaptations are motivated by changes in the executing environment. While a motivation of a design adaptation cannot be modeled (it is a choice), the motivation of a runtime adaptation can be modeled as a condition on the state of the environment. Moreover, design and runtime adaptations are different also because they are realized by means of different mechanisms that use different technological platforms.

- **On the reason for unifying design and runtime adaptations**

Although design and runtime adaptations have different *motivations*, and are *performed* using different mechanisms, they can be *specified* thanks to a unified language. Our aspect metamodel provides such a unified language. The main advantage is the fact that design and runtime adaptations are modeled thanks to aspects that share three principal descriptions: the *what*, the *where* and the *how*. In fact, the only key difference comes from the fact that aspects that can be woven at runtime must have a description of the event: the *when*. As a consequence, aspects can be easily reused. Moreover, one can think about weaving aspects during the design phase, although they have been originally defined to be woven at runtime. Changing a runtime aspect into a design one is quite easy as it only consists in removing its event description. Vice versa, one can think about weaving aspects during the runtime phase, although they have been originally defined to be woven at design. Changing a design aspect into a runtime one is much more complex as it needs to add an event description.

In this point it is important to notice that, since aspects are used for both types of adaptations, our approach introduces a strong dependency between the design process and the runtime adaptation. In fact, this dependency is what makes the unification of adaptations possible and allows the products to be derived dynamically. However, it also implies that when creating the aspect models, one has to be aware of their twofold nature. Ideally, to help designers to define new assets, the tools used for creating the aspect models should provide enough information about the models being developed, so that, developers get immediate feedback of the correctness and possibilities of each model. This would not only ease the task of asset development but it would also guarantee that what is defined at design remains valid for an aspect used either at design for building the initial product or at runtime when it gets transformed into reconfiguration scripts and used for a dynamic adaptation. For the time being, our tools perform a simple validation that takes place before the weaving, which in most cases happens after the model has been created and hence makes it difficult to identify and fix the problems. In the perspectives presented in Chapter 8 we revisit this issue and indicate possible ways to improve the tool support in the short term.

- **On the necessity of weaving aspects at design time**

One question that may naturally arise is: why do we need design weaving if we can perform any adaptations at runtime? Design weaving is important for three main reasons: automation, performance, and platform independence. Regarding automation, if there is no design weaving, the initial application has to be built manually. That is because runtime weaving just modifies an existing application. Regarding performance, it should be noted that all aspects that can be woven during the runtime phase need to define an event description. Regarding our example, it clearly appears that some aspects do not have any event that motivates their weaving. Those aspects are definitively intrinsic design aspects. Without design weaving, we could weave them at the beginning of the execution, by defining a *fake* runtime event. This may potentially affect the performance of the application, while design weaving has no impact on performance. We firmly believe that intrinsic design aspects have to be woven as early as possible, and that runtime weaving only has to operate on aspects that depend on runtime events. Regarding platform independence, obtaining a woven model through design weaving is only an intermediate step in the derivation of an executable product. The woven model obtained in our approach completely differs from the woven code produced by an AOP technology. In the AOP world, the woven code produced is language and platform-specific, and is not supposed to be further modified. In

our approach, the woven model is a generic artifact that belongs to a high level of abstraction and constitutes a valuable input of a subsequent generation process, where it is still possible to make design decisions like execution platform and implementation languages.

- **On the use of aspect models for variability modeling**

In our approach the use of aspect models provides a clear separation of concerns. It separates the core of the application from optional and possibly crosscutting functionalities. In our case study, we have defined a set of variants expressed in a feature model. Each variant (or feature), being crosscutting or not, is represented through an aspect model. If a feature is not crosscutting, then the corresponding aspect pointcut is simple, as it only captures a single element of the base architecture. In contrast, if the feature is crosscutting, then the aspect pointcut and the aspect advice become more complex since they have to deal with multiple elements and different modifications. In summary, our approach allows aspects to be defined with multiple expressions and multiple modifications. Furthermore, the weavers at design time and at runtime are able to deal with these aspects. We consider that, regardless of the crosscutting nature of the variants, the proposed aspect metamodel and the two weaving processes provide the required flexibility for variants to be realized as aspect models and derivation to be defined as the weaving of such aspects.

- **On the use of MDA and AOM**

In general terms, we consider that the use of a well-defined metamodel enhances the integration of aspects within complementary model-driven development strategies. This permits: (1) to define independent business models that are transformed into platform-specific models depending on the needs of a particular application, and (2) to have a unified approach in which software products and related adaptations can be modeled at the same time and derived from the same type of artifacts (e.g., aspect models). Furthermore, each aspect model is self-contained, and can be woven by a generic weaver that resolves the pointcuts, and then executes all modifications defined in the advice. Finally, as we have shown in the experimentation, CAPucine aspect models are used as a way to realize features and obtain assets for both the initial and iterative phases. In such a context, our approach provides support for product derivation at design time, based on the selection of optional features, in order to build an initial product. It also supports dynamic product reconfiguration at runtime, translating the iterative configuration changes in reaction to context changes.

- **On the applicability of the approach**

The CAPucine framework for the development of DSPLs that we have presented has been validated regarding the examples of the CAPPUCINO project when we model and derive applications based on SCA. We consider that the same approach is applicable to other contexts that share the same characteristics as our example. Such characteristics refer basically to the architectural decisions we have made in order to automate the development process. This means that families of applications built for a component based and service oriented platform and that need to be modified at runtime are well suited to fit in the CAPucine framework. In that trend, for example, we consider the applicability of our approach to define a family of applications for the FraSCAti platform. Since FraSCAti is a platform already developed, it constitutes an interesting example, when the approach can be used in a *bottom - up* strategy, starting from the implemented assets and going upwards to obtain the high level representations and obtaining the variability model and the different configurations. In addition to that, FraSCAti has been built as a highly modular platform that can be customized with different plugins with respect to the user needs.

From a general perspective, in order to target other types of families the framework needs to be extended to support the derivation of different assets and products. In the next section we discuss the issues related to the extensibility both at the domain engineering as well as at the application engineering level.

7.3.2 Qualitative Analysis

This section presents a qualitative evaluation of CAPucine in terms of the properties for a DSPL framework introduced in Chapter 3. We identified 4 main properties: extensibility, scalability, runtime history, and usability.

Extensibility

In our approach, we give support to a given extent for extensibility. Since our approach is based in MDE principles, models are used as first-class entities for the development and adaptation of the different products, the extensibility is achieved by adding new metamodels, transformations, code generators and models. We distinguish two types of extensibility in our approach that are derived from the two processes of SPLs: extensibility of the domain, and extensibility of the application.

- **Extensibility of the domain**

This type of extensibility refers to the modification of the core assets defined during the process of domain engineering. This type of extensibility represents a major challenge since it aims at modifying the core assets that constitute the architecture of the product line. An extension of this type may be for example the addition of new target platforms (other than SCA or Java), the definition of a different composition model (i.e., change the model part of the aspect metamodel).

To successfully realize this kind of extensibility and support these new core assets, developers are required to add the metamodels for the new platforms (in the same way we have provided metamodels for SCA and Java), and also a set of transformations that takes every aspect model and map its concepts into coherent elements that are particular to the new platform. Additionally, for the derivation process to be complete, the code generators for the new models have to be also provided. Although this type of extensions requires an important effort from developers, there are several ways in which we can ease the task. For example, the transformations and code generators already developed for our DSPL can be used as templates to build the new transformations towards different platforms. Consider the diagram presented in Figure 7.21. It describes the elements that have to be provided in order to extend the scope of the transformations in the Software Product Line. The current implementation of the DSPL supports Java and SCA. For a new target platform, the elements to provide (presented with dashed lines) are: (1) the metamodel of the new target domain, (2) a model transformation from the aspect model, (3) a code generation for the particular platform of implementation language.

Additionally, if the dynamic properties of the product line are important, then the extensions to new platforms have to guarantee, that these new platforms do provide the mechanisms for modifying products at runtime, in the same way as our target platform *FraSCaTi* does. Without this support, the runtime phase is not complete.

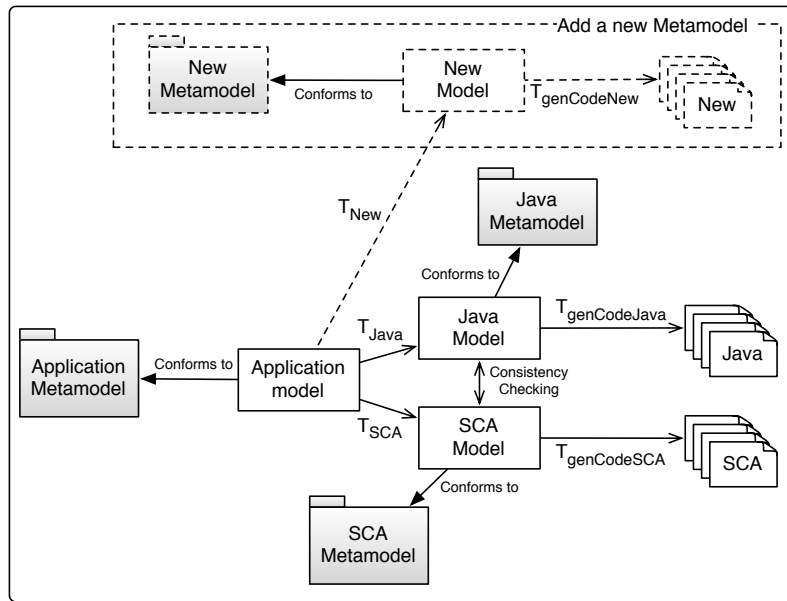


Figure 7.21: Extensibility of the Domain.

- **Extensibility of the application**

The second type of extensibility refers to the addition of new functionalities for a particular product family. This type of extensibility is easier to handle than the previous case since it only affects the assets developed for every particular product family and not the whole architecture of the product line. An extension of this type may be for example the addition of new features, the development of new aspect models to realize each feature and in general, every modification that only affects the elements developed during the application engineering phase.

We have built the architecture of CAPucine based on the variability and aspect metamodels, which are independent of any particular technology. This allows developers to create product families for different application domains. To create a new product family or enrich an existing one, developers are required to create or modify the feature model to add the new supported features, and use the metamodels provided to create aspect models for each optional or alternative feature in the feature model. If new mandatory features are also added, it is necessary to modify the core aspect by adding the architectural elements that support them.

Since model transformations for design weaving as well as code generators have been developed for any model that conforms to the metamodel, no further modification is needed to derive the products for the new product family. For the runtime weaving, aspects are required to specify an event. As described in Chapter 6, the runtime weaving only takes place for aspect models that specify the moment during the execution when they have to be woven. If aspect models do not include the event part, they are considered as design-only, and runtime adaptation scripts will not be generated from them.

Scalability

In any SPL, one of the biggest challenges is to deal with the combinatorial explosion of product configurations. We have illustrated in 6 that the size of a product family grows exponentially with every optional feature added to the diagram. As it was the case for the example used in our experimentation, starting from a rather small feature diagram with 16 variants, we have obtained more than two thousand product configurations.

Larger product families represent a challenge in terms of scalability and performance. To face it, we try to anticipate as much as possible the different configurations that are valid and that can be reached by the weavers at design time. We have defined analysis that calculate the set of valid product configurations from a given feature model. Furthermore, we use the aspect models in our constraint analysis to reduce the number of configurations to eliminate those that cannot be composed. The results are stored in tables that are accessed at runtime to guarantee that no matter the adaptation required, the products will always respect the feature constraints and the aspect dependencies.

However, if the size of the product family grows bigger, the time for the adaptation cycle, and in particular the time for the `Validator` to verify if the *target* configuration is correct will be impacted by the manipulation of a large number of valid configurations. In addition to that, there are other elements that impact the performance. For instance the reconfiguration of products is by itself a time and resource consuming operation. If it is combined with other processes, it may result in performances that are not satisfying for mobile environments as the one presented in our experimentation.

Runtime History

A remaining problem in adaptable systems refers to the capability of the system to go back to a previous state before one or several adaptations have taken place. In our approach this traduces in defining a process of *unweaving* for the aspect models that might get woven due to context events. As we have described in Chapter 6 this is only possible for the aspects that do not include any `Delete` modifications. If a weaving performs a `Delete` modification, the elements that got deleted are not stored anywhere and recovering those elements would not be possible. This kind of adaptations is not supported yet. We consider that all the features selected at design time, should remain as part of the products and should not disappear, regardless of the different context events that may occur. Consequently, for the dynamic reconfigurations we only support aspects that extend the products functionality by adding new elements. Deletion is reserved for design time adaptations. Another concern at runtime refers to the volatility of context information itself. Since we confine the definition of context-information to the design of an aspect model, new context observables, that were not initially specified by an aspect model, cannot be taken into account. CAPucine does not support the on-the-fly introduction of new observables and their corresponding adaptation rules.

Usability

In our approach, we propose two languages for variability and architecture and composition languages as well as a specific derivation process inspired by the MDA approaches to transform PIMs into PSMs. Since we have already discussed the tool support offered by `Model2Code` for the derivation process, here we discuss the usability of CAPucine regarding the languages used for the variability, and architecture.

- **Variability**

With regard to variability, we have defined the feature metamodel (see Chapter 5). This metamodel is small and easy to understand. We have focused on modeling the essential concepts for features including variation points, variants, and the different types of interactions among them. We have demonstrated in our case study and experimentation in Section 7.1, how the metamodel is used to define the variability model.

However, the simplicity of the feature metamodel also implies that it lacks the expressiveness required to define elements found in other approaches like: multiplicities (when OR and XOR alternatives are not accurate enough), mixed exclusive and non exclusive variants from the single variation point (this property is specified into the variation point and not in every variant), and finally selection and deselection of variation points (only variants can be selected from a feature model). We are aware of the lack of such properties in our language. Nevertheless, our goal was to create a language expressive enough to allow developers to create product configurations, used as input in the derivation of products. Even if we lack the expressiveness of other approaches, we consider that the feature metamodel is expressive enough to be used in the overall DSPL approach.

- **Architecture**

The second language that we have introduced is represented with the aspect metamodel (see Chapter 5). This metamodel plays an essential role in the application engineering process of CAPucine since it enables developers to define only once, artifacts that are used at both design and runtime. The metamodel uses the terminology from AOSD in order to represent the modularization of the software products and allows the realization of variability through aspect models. One of the advantages of such metamodel is that, we provide a generic weaver for design and runtime. This means that, regardless of the business being modeled, as long as it conforms to our metamodel, the weavers and transformation tools provided can be used with no extra effort.

However, an aspect contains several pieces of information (architecture, modifications, places, events). Developers need to concretize several pieces of information in a single model. Currently our approach offers no extra support for the development of aspect models. They have to be created as any other model in EMF, which for large products, may result in a complex and error-prone process.

7.4 Summary

In this chapter we have presented a validation for CAPucine. We started with the experimentation we followed to describe a sample Product Family with CAPucine using a *top – down* approach. We started from scratch and define the domain and application engineering processes for building a family of products. The scenario used has been defined for the purposes of a collaborative work in the context of the CAPPUCINO project. In the initial phase, products can be configured with the help of a feature diagram. In a second step, we add dynamic properties to the systems and extend the scope of the product family using several use cases based on context information. This enables the products to switch among several configurations at runtime, in response to changes in the environment. We have also presented in detail the tools built around the DSPL. This includes a set of transformations and code generators called `Model2Code`, the implementation of the algorithms and their application along the product derivation process. The last part of this chapter presents a discussion and justification for the choices made in CAPucine

and a qualitative evaluation of CAPucine. For the discussion, we revisit the challenges identified in Chapter 3 and, in particular, we elaborate on how the design and runtime derivation processes presented in Chapters 5 and 6 combined enable the unification of software adaptations. For the evaluation, we have discussed how our approach stands regarding properties like: extensibility, scalability, and usability.

This chapter concludes the third part of this document dedicated to the validation of our approach. In the next chapter we summarize the main contributions of this dissertation, present the conclusions of the research work, and define a set of perspectives for future work.

Part IV

Conclusion

Conclusion

“It’s more fun to arrive a conclusion than to justify it.” Malcolm Forbes

Contents

8.1 Summary of the Dissertation	135
8.1.1 Contributions	137
8.2 Perspectives	138
8.2.1 Scope and short-term perspectives	138
8.2.2 Long-term perspectives	139

8.1 Summary of the Dissertation

The software industry presents new challenges related with the user expectations and the speed at which applications have to be developed. In addition to that, requirements keep on changing, making dynamic adaptations more the rule rather than the exception. As a consequence, this has brought the need for automated development processes that help developers involved in the projects to accomplish their goals with the quality expected and respecting the deadlines. The strategies proposed by the SPL community are ambitious and promise to ease such processes by defining reuse strategies that are usually implemented through generative and compositional schemes. However, successful implementations of such strategies are still scarce and do not extend properly to software systems that have to be adapted constantly, after their implementation.

This research has tried to bring new insights on this domain, and in particular, in the challenges of developing adaptable applications using SPL principles, and the possible ways to face those challenges.

We have defined a simple –yet complete– language for defining variability and enabling the process of product derivation. We have also defined an architecture metamodel to have a high-level representation of SOA and CBSE applications. In addition to that, we have defined an AOSD-like metamodel that includes the notions of pointcuts, and advices. With this metamodel we are able to modularize the architecture of the applications in two parts: the first part that holds

the mandatory elements for any product, and the second part that holds a set of independent aspect models representing optional elements.

We have also introduced the variability management for different product families. This is perhaps, one of the most interesting and challenging topics for the construction of SPLs that actually generate applications matching the user expectations. It has been demonstrated that using a specification of commonalities and variabilities for a set of software products, it is possible to involve the users in a customization process, where they can be able to decide about the elements constituting their software product by selecting/deselecting among the different features available. The decisions are reflected with product configurations that are used as a strategy for the composition which results in complete PIMs for every particular configuration. The weaving is based on a model composition that creates a single woven model from two different parts: (1) the elements implementing the commonalities for every product (i.e. the *core*), and (2) the elements implementing the variability from the particular configurations (i.e. *aspects*). Thanks to the weaving process, it is possible to build different products that share a certain set of features, but that are differentiated by a set of variabilities selected specifically to satisfy every particular user. To complement the aspect weaving, we have defined metamodels for the target platform and implementation language. Using metamodels for each particular domain provides a separation of concerns that allow developers to focus on the business of the applications being modeled and postpone decisions about a particular platform and implementation technology. We have proposed a separation in different domains for: the architecture, the platform, and the implementation. We have also proposed a set of model transformations to obtain the source code from the application and platform domains. By using the transformations, models are enriched in every stage of the derivation process to produce the platform specific models that represent a software product. In this way, source code is automatically generated.

Furthermore, we have defined a second process of product derivation that extends the classic one to the execution time. Starting from a product configuration obtained from analyzing the context information, products can be adapted at runtime. We have enriched the aspect metamodel with a context event to be aware of the information that triggers the dynamic adaptations. Products can then be adapted to match the specific executing conditions using aspect models that specify the context events. An adaptation in the DSPL is realized as the switch from a given product configuration (*current*) to a new one (*target*). The switch itself is expressed with a set of weaving/unweaving operations for the aspects affected by the context update. To realize the runtime derivation process, we have based the adaptation on a set of realization techniques for context-aware management, such as the FraSCAti, a platform capable of performing dynamic reconfigurations.

Finally, thanks to the aspect metamodel, our approach unifies the product derivation processes at design time and at runtime. The aspect metamodel allows developers to define at the same time a base architecture and different modules that can be integrated to this architecture. The metamodel enables aspect models to have a twofold weaving. On the one hand, they can be used to represent user selections and build customized products. On the other hand, they can take into account the information coming from the environment, and use it as a way of binding the variability. In this case, the same aspect model can be transformed into reconfiguration scripts that adapt the product by switching its configuration. Consequently, developers only have to define aspect models once and use them at design time as a way to build a product or at runtime to adapt the product for a particular context situation.

8.1.1 Contributions

1. Unification of design and runtime adaptations

The main contribution of this dissertation refers to the unification of design and runtime adaptations. It is achieved through the definition of a single metamodel that enables SCA-based architectures to be modularized, and through two different phases for product derivation, one at design to build a product and one at runtime to adapt a product. The design phase is implemented using model transformations and code generation, at runtime the platform `FraSCAti` supports the dynamic adaptations.

The main benefits of the proposed approach are (1) a clear separation of concerns, (2) a unified definition of design and runtime adaptations, (3) an explicit link between software adaptations and their motivations, and (4) a supporting platform that allows adaptations to be executed in different moments of the software life cycle, from the initial product configuration to its successive dynamic reconfigurations. As a result, our tool-supported approach allows software adaptation processes to reach high levels of reusability and flexibility.

2. Variability Definition

The variability metamodel allows one to define the functionality for a set of applications through abstractions of their main features. In our case, we have defined a variability model that includes the notions of variant, variation point, alternatives, and constraints between variants and variation points. The metamodel is simple and easy to use, and through our experimentation we have demonstrated that it is expressive enough to model different kinds of relationships within a product family.

3. Conflict-free composition strategy for design time and runtime

We have introduced a composition of models that is realized through the weaving of a core and different aspect models. The weaving process is guided by the explicit and implicit dependencies that exist between the selected features. Our proposal relies on a clear separation of concerns enabled by the underlying variability and aspect metamodels. Our method allows to identify implicit dependencies and conflicts between features, and takes such feature interactions as a basis to derive an appropriate architecture composition strategy. The conflict-free strategy is part of the derivation processes at design and at runtime, which enables the derivation and adaptation of SCA-based software products.

4. Generative and automated development

CAPucine presents a comprehensive approach to feature-driven composition of software architectures. It allows the automated derivation of product architectures from feature configurations, by combining MDE and AOM techniques. The composition process is realized through transformation-based model weaving. Afterwards, code generation is based on templates from which common methods are generated for the different products of the family. Templates include a certain amount of empty fields that are filled with the information that comes from the models. The use of templates guarantees that the code is correct and well formed. Besides, its integration with the models implies a high degree of reuse throughout the different products.

5. Automated tool-supported derivation of products

Product derivation has been automatized through the set of tools called `Model2Code`. This tools offer support for the feature configuration, the analysis of constraints and the code generation. At the same time, they provide the mechanisms to invoke every process of

the transformation chain separately, in the cases where only a part of the functionality is required; or following the complete derivation process strategy, where starting from simple feature selections we obtain software products as well as the required reconfiguration scripts for a dynamic adaptation.

8.2 Perspectives

Even if the results obtained from this research are consequent, there are still several areas that have to be further developed. In the following paragraphs we discuss some of these areas as well as some future works that could lead to further improvements in the DSPLs.

8.2.1 Scope and short-term perspectives

The work realized in the context of this dissertation is not complete. Several areas are still open for improvement so that we can cover the whole strategy for DSPL that we want to promote. Here below we present some of the works that would further improve CAPucine in the short term:

- **Verification of Models and Metamodels**

As we briefly discussed it in Chapter 7, model creation is still a complex and error-prone task. It would be desirable to have a methodology and a set of tools to help architects and developers to create this type of assets. If the model presents errors, the derivation process is affected. So far, the verification is done programmatically when the models are loaded for the processes of analysis and transformation. This usually occurs very late in time. Ideally models should be verified right after their creation so that, different types of checks can be applied and developers get immediate feedback of the issues encountered. An improved tool for creating the models would help developers to find mistakes and missing parts of the models right after their creation. In this direction, an interesting option may be to use a set of tools for defining domain specific languages (DSL). Certain tools used for defining this type of languages like `xText` enable developers to add a set of verifications to the instances of the DSL. Furthermore, these tools have already a good level of maturity and provide integration with EMF.

- **Extend the domain of the DSPL**

For the implementation of the DSPL presented in this dissertation, we have limited the number of software artifacts that we are able to generate. In addition to that, we have chosen only one platform for the execution of the products. However, CAPucine could be extended to support other types of platforms, and domains. For example, we could increase the amount of code that is automatically generated depending on the chosen platform.

- **Graphic User Interfaces**

The examples provided in CAPucine only deal with the architecture of SCA applications and not with particular GUIs. Actually, in our experimentation for mobile devices, we have built a wrapper around SCA applications inside the Android platform, and the interfaces have been developed by hand in order to interact with the applications. An interesting work that could follow this research is related to the generation of code for mobile devices

like it is already being done by the Startup `UBINNOV`¹, which aims at automating the development of mobile applications. In particular they focus on automating the development process through software product lines. They target mobile platforms like iOS from Apple and Android from Google.

- **Add more information for the decision making**

The decision making mechanism presented in Chapter 6 only uses the information of the variability model to guarantee the correctness of the reconfiguration. Furthermore, if the constraints in the variability model or in the aspect dependencies are not respected, the adaptation does not take place. To overcome this limitation, it would be necessary to include other sources of information to reason about the best configuration for a particular set of context events. There can be several configurations that respect the variability and aspect models for a particular set of context events. Hence, using extra information like quality of service (QoS) or the complexity of the reconfiguration itself, one should be able to choose the most appropriate configuration that maximizes the quality functions and that respects the SPL constraints. A possible workaround can be found in the work of Romero [Rom11]. He describes the notion of ubiquitous feedback control loops which among their components, include a `Planning` mechanism that uses information like QoS to find the best reconfiguration for self-adaptive systems.

- **Finer-grained Adaptations**

The adaptations defined in the examples and in generally the ones presented throughout the work presented in this dissertation focus on rather *coarse-grained* adaptations. Concretely, we have presented examples that manipulate components, references, services, and in general the elements defined in the architectural metamodel. However, the modification of smaller elements in the architecture of an application (e.g., objects, properties, methods, etc.) are not covered. A short-term improvement to the overall approach is to consider finer-grained adaptations by creating aspects that enable the modification of the source code inside the `Operator` elements.

8.2.2 Long-term perspectives

In order to further improve the DSPL and to overcome some of the limitations with regard to the results of this research, several perspectives for long-term work can be foreseen:

- **Behavior Adaptation**

We have focused on the adaptation of the structure of products based on SCA, either at design time or at runtime. Even if our architecture metamodel includes several concepts related to the behavior, they have not been exploited for the adaptation of applications. We consider that the behavior elements already modeled can be enriched with the notions of execution processes like BPEL. This would bring a new level of adaptation to the products by allowing modifications of the sequence of calls between services and references, that extend the already supported addition or suppression of certain components or connections between components.

¹UBINNOV home page: <http://ubinnov.lille.inria.fr/>

- **Support multi-staged configurations**

An interesting problem in SPL engineering refers to multi-staged derivation processes. In CAPucine, we have assumed that a single-staged configuration and the associated product derivation is always possible. However this may not be true for different contexts where external restrictions require a product to be derived in a sequence of smaller and independent steps. This kind of derivations raises new challenges for the product derivation process and specially for the constraint analysis introduced in Chapter 5.

For example, suppose that a feature model includes two features *A* and *B*. In a multi-staged derivation process we may have restrictions like, for instance, being able to add only one feature at a time. In such a case, in order to add the two features, we have two possibilities: (1) add *A* first and then *B*; or (2) add *B* first and then *A*. In this simple example without relationships between the features or the aspects, the constraint analysis will not make any difference between the two options and return the same order from left to right as received (*A* and then *B*). However, adding *A* in the first place may violate external restrictions like for example budget (i.e. adding the feature *A* is too expensive and has to be delayed), and time (i.e. feature *B* takes less time to be integrated in the product). If we add to this problem, the constraints between features (e.g. if *A* requires *B*), the derivation process gets more complex and a solver is needed to define the best *path* of derivation (if any), that respects the external restrictions. This constitutes an important and interesting trend for improvement of the approach, where CAPucine could benefit from other works in the area that specifically deal with the multi-staged configuration problem.

- **Traceability and Reverse Engineering**

As a complement to the transformation processes, and in general to the derivation of software products presented in this dissertation, the traceability management is a field that is yet to explore. We have already implemented a small *top - down* mechanism for traceability with the definition of the `Configurator` component in every product of the software product line. Such component keeps the configuration of the product in terms of selected variants at runtime. However, there is no way to go the other way around (*bottom - up*) to reify a model of aspects or a feature diagram from a composite structure in an SCA application. Having such a tool, would allow one to analyze and understand bigger and complex applications by creating high abstract representations in terms of variability. This could also be used in the process of creating new assets from already implemented code, and integrating them inside the architecture of the SPL.

Traceability could also be useful to support the unweaving process of aspects that delete elements. This was one of the main limitations of the approach discussed in Chapter 7. In order to support this kind of adaptations, we need to store a trace of the history of modifications that have been done over a single product at runtime. This would mean that, every time an adaptation takes place, we would save the the elements deleted, in order to add them again if the adaptation has to be rolled-back. Several challenges have to be faced in order to support this kind of behavior mainly related to the order in which we can unweave aspects, and the constraints in the feature model and in the aspects. One possibility to face those challenges is the implementation of a traceability mechanism at runtime. Traceability can be used to *annotate* the elements with extra information about the relationships with the aspects where they belong and to a higher level, the features that are implemented by such aspects. This information could be used to evaluate if a given unweaving operation has an impact on the product at the level of features (*legal*) or aspects (*composable*).

- **Formal aspects of the approach**

The CAPucine framework is focused on the derivation process and the development of reusable assets throughout the life cycle of each product. Nevertheless, several parts of the approach could be enriched and improved by applying well founded theoretical studies. For instance, it could be interesting to redefine the constraint analysis using satisfaction of boolean expressions. This would allow us to express constraint algorithms between features and aspect models in a formal way and could open the doors to reuse them in different applicaiton contexts. In the same idea, it would be also interesting to explore the applicability of binary decision diagrams for the optimization of legal and valid configurations that we use through pre-calculated tables for the dynamic adaptation.

Finally, it would be also interesting to evaluate the impact of extending the proposed the feature model, to include the notions found in other approaches like cardinalities, feature groups, and attributes for the features. This would have an impact specially in the constraint analysis presented in Chapter 5, and in the algorithms for obtaining the product family. In that sense, it would be interesting to evaluate if the extra information added to the feature model is useful with regard to the aspect model dependencies and the derivation itself, or if we can rather build transformations towards languages that focus on the problem of feature configuration (e.g. *FAMILIAR*, *TVL*).

- **Discovery of Context Information**

Since we confine the definition of context-information to the design of an aspect model, new context observables, that were not initially specified by an aspect model, cannot be taken into account. This means that so far, aspects are defined before the execution, which limits the adaptation possibilities to the set of foreseen observables. Our approach does not support (yet) the on-the-fly introduction of new observables and their corresponding adaptation rules. We would like to enhance CAPucine by allowing the products to process new pieces of context-information at runtime.

Bibliography

- [ACLF10] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. FAMILIAR, a Language and its Environment for Feature Model Management. In *Journée Lignes de Produits. Maîtriser la Diversité*, , page 12. Université Paris 1 - Panthéon Sorbonne, October 2010. 16
- [AGG⁺08] Nicolas Anquetil, Birgit Grammel, Ismenia Galvao Lourenco da Silva, Joost Noppen, Safoora Shakil Khan, Hugo Arboleda, Awais Rashid, and Alessandro Garcia. Traceability for model driven, software product line engineering. In *ECMDA Traceability Workshop Proceedings, Berlin, Germany*, pages 77–86, Norway, June 2008. SINTEF. 17
- [AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of Caesar]. *Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I*, pages 135–173, 2006. 42
- [ALS08] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008. 40, 43
- [AOM] Aspect-oriented modelling workshops series. <http://www.aspect-modeling.org/>. 2
- [ARCR09] Hugo Arboleda, Andrés Romero, Rubby Casallas, and Jean-Claude Royer. Product derivation in a model-driven software product line using decision models. In Antonio Brogi, João Araújo, and Raquel Anaya, editors, *CibSE*, pages 59–72, 2009. 34, 37
- [Bar06] Olivier Barais. SpoonEMF, une brique logicielle pour l’utilisation de l’IDM dans le cadre de la réingénierie de programmes Java5. In *Journées sur l’Ingénierie Dirigée par les Modèles (IDM)*, June 2006. Poster. 69, 123, 158
- [Bar07] Graham Barber. What is SCA?, August 2007. <http://www.osoa.org/pages/viewpage.action?pageId=46>. 27
- [BBB⁺07] Michael Beisiegel, Henning Blohm, Dave Booz, Mike Edwards, Oisin Hurley, Sabin Ielceanu, Alex Miller, Anish Karmarkar, Ashok Malhotra, Jim Marino, Martin Nally, Eric Newcomer, Sanjay Patil, Greg Pavlik, Martin Raepple, Michael Rowley, Ken Tam, Scott Vorthmann, Peter Walker, and Lance Waterman. SCA Service Component Architecture. Assembly Model Specification , March 2007. 27, 28

- [BCFH10] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing TVL, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, Linz, Austria, January 27-29, pages 159–162. University of Duisburg-Essen, January 2010. Acceptance rate: 54¹⁶
- [BCL⁺06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006. ²⁵, ⁹³
- [Béz05] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, pages 171–188, May 2005. ¹⁸, ¹⁹
- [BJ95] Frederick P. Brooks Jr. The mythical man-month: After 20 years. *IEEE Software*, 12(5):57–60, 1995. ¹
- [BJC05] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *ECSA*, pages 1–17. LNCS, 2005. ³⁸, ⁴⁰
- [BJS09] Goetz Botterweck, Mikolas Janota, and Denny Schneeweiss. A design of a configurable feature model configurator. In *3rd International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2009)*, january 2009. ¹⁶
- [BMH06] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0*. O'Reilly, Beijing, 5. edition, 2006. ²⁵
- [BNT08] Leire Bastida, Francisco Javier Nieto, and Roberto Tola. Context-aware service composition: a methodology and a case study. In *SDSOA '08: Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 19–24, New York, NY, USA, 2008. ACM. ⁴⁰, ⁴³
- [Bro96] Peter J. Brown. The stick-e document: a framework for creating context-aware applications. In A. Brown, A. Brüggemann-Klein, and A. Feng, editors, *Special Issue: Proceedings of the Sixth International Conference on Electronic Publishing, Document Manipulation and Typography, Palo Alto*, volume 8, pages 259–272, John Wiley and Sons, June 1996. ¹, ³⁸, ⁸⁶
- [Bru10] Hugo Bruneliere. MoDisco, Model Discovery, 2010. <http://www.eclipse.org/gmt/modisco/>. ⁶⁹
- [BSBG08] Nelly Bencomo, Pete Sawyer, Gordon Blair, and Paul Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *2nd International Workshop on Dynamic Software Product Lines (DSPL 2008)*, Limerick, Ireland,, 2008. ³⁸, ⁴⁰
- [CA05] Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In Robert Glück and Michael R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2005. ³⁴, ³⁷
- [CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches, 2003. ²⁰, ²¹

-
- [Cha07] David Chappell. Introducing SCA, July 2007. 27
- [Cla02] Siobhán Clarke. Extending standard uml with model composition semantics. *Sci. Comput. Program.*, 44(1):71–100, 2002. 34, 37
- [CN01] Paul Clements and Linda Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, August 2001. 2
- [CTPRC08] Carlos Cetina, Pablo Trinidad, Vicente Pelechano, and Antonio Ruiz-Cortés. An architectural discussion on dspl. *2nd International Workshop on Dynamic Software Product Line (DSPL08)*, 2008. 43
- [DAS01] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, 16(2):97–166, 2001. 1, 38, 86
- [DL05] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, New York, NY, USA, 2005. ACM. 39
- [DL06] Pierre-Charles David and Thomas Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2006. 3, 39, 40
- [DLLC09] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annals of Telecommunications*, 64(1-2):45–63, February 2009. 92
- [DMFM10] Tom Dinkelaker, Ralf Mitschke, Karin Fetzter, and Mira Mezini. A Dynamic Software Product-Line Approach using Aspect Models at Runtime. In *Fifth Domain-Specific Aspect Languages Workshop*, 2010. 39, 40
- [Don98] Don Box. *Essential COM*. Addison Wesley, 1998. 25
- [dS07] Tijs Van der Storm. Generic feature-based software composition. In Markus Lumpe and Wim Vanderperren, editors, *Proc. of the 6th International Symposium on Software Composition (SC'2007) - Revised Selected Papers*, volume 4829 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2007. 36, 37
- [DSB04] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Experiences in software product families: Problems and issues during product derivation. In *SPLC*, pages 165–182, 2004. 2
- [Fal10] Jean-Remy Falleri. Minjava, A Java Reverse Engineering Software, 2010. <http://code.google.com/p/minjava/>. 69
- [FEBF06] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino. *L'ingénierie dirigée par les modèles au-delà du MDA*. Lavoisier, 2006. 18
- [FECA05] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005. 22, 23

- [FHMP⁺09] Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen, Gøran K. Olsen, Andreas Svendsen, and Xiaorui Zhang. A generic language and tool for variability modeling. Technical report, University of Oslo, Oslo, Norway, December 2009. 16
- [FJ09] Robert France and Jean-Marc Jézéquel. Editorial for the special issue on aspects and model-driven engineering. *Transactions on Aspect-Oriented Software Development*, 2009. 2, 23, 60
- [FJF⁺07] Steven Fraser, Frederick P. Brooks Jr., Martin Fowler, Ricardo Lopez, Aki Namioka, Linda M. Northrop, David Lorge Parnas, and Dave A. Thomas. "no silver bullet" reloaded: retrospective on "essence and accidents of software engineering". In *ACM SIGPLAN OOPSLA'07 Companion*, pages 1026–1030, 2007. 1
- [Gis01] Dan Gisolfi. Web services architect: Part 1. an introduction to dynamic e-business, April 2001. <http://www.ibm.com/developerworks/webservices/library/w-ovr/>. 26
- [HHPS08] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008. 2, 32
- [IBM06] IBM. An architectural blueprint for autonomic computing. white paper, June 2006. White Paper. 32
- [INR10] INRIA Project-Team Triskell. Kermeta tool suite, 2010. <http://www.kermeta.org/>. 69
- [Jéz08] Jean-Marc Jézéquel. Model driven design and aspect weaving. *Software and System Modeling*, 7(2):209–218, 2008. 60
- [Jos07] Nicolai Josuttis. *Soa in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007. 25, 26
- [JZ05] Waraporn Jirapanthong and Andrea Zisman. Supporting product line development through traceability. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 506–514, Washington, DC, USA, 2005. IEEE Computer Society. 17
- [KAAK09] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2009. ACM. 3, 35, 37
- [KCH⁺90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990. 12, 15, 59, 62, 156
- [KHH⁺01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag. 33
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997. 2, 22, 60

-
- [Kur05] Ivan Kurtev. *Adaptability of model transformations*. PhD thesis, University of Twente, Enschede, May 2005. 21
- [KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 18, 19
- [LBT09] Kwanwoo Lee, Goetz Botterweck, and Steffen Thiel. Aspectual separation of feature dependencies for flexible feature composition. In *Proc. of the 33rd Annual IEEE International Computer Software and Applications Conference*, pages 45–52. IEEE CS, 2009. 36, 37
- [LSO⁺07] Sten A. Lundesgaard, Arnor Solberg, Jon Oldevik, Robert B. France, Jan Øyvind Aagedal, and Frank Eliassen. Construction and execution of adaptable applications using an aspect-oriented and model driven approach. In Jadwiga Indulska and Kerry Raymond, editors, *DAIS*, volume 4531 of *Lecture Notes in Computer Science*, pages 76–89. Springer, 2007. 3, 41, 43
- [MBJ08] Brice Morin, Olivier Barais, and Jean-Marc Jezequel. K@rt: An aspect-oriented and model-oriented framework for dynamic software product lines. In *Proceedings of the 3rd International Workshop on Models@Runtime, at MoDELS'08*, Toulouse, France, oct 2008. 41, 43
- [MBN09] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming Dynamically Adaptive Systems with Models and Aspects. In *31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009. 41, 43
- [MFB⁺08] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon S. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, volume 5301 of *Lecture Notes in Computer Science*, pages 782–796. Springer, 2008. 3, 41, 43
- [MM09] John D. McGregor and Dirk Muthig. Splc '09: Proceedings of the 13th international software product line conference, 2009. 12
- [MSKC04] Philip K. McKinley, Seyed M. Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical report, Department of Computer Science and Engineering, Michigan State University, 2004. 33
- [nat68] Nato software engineering conference, 1968. 1
- [ND08] Carlos Noguera and Laurence Duchien. Annotation framework validation using domain models. In *Fourth European Conference on Model Driven Architecture Foundations and Applications*, pages 48–62, Berlin, Germany, June 2008. 69, 158
- [NnNG09] Angel Núñez, Jacques Noyé, and Vaidas Gasiūnas. Declarative definition of contexts with polymorphic events. In *International Workshop on Context-Oriented Programming, COP '09*, pages 2:1–2:6, New York, NY, USA, 2009. ACM. 42, 43
- [Obj06] Object Management Group. Meta Object Facility (MOF) Core Specification, 2006. <http://www.omg.org/spec/MOF/2.0/PDF>. 19

- [Obj10] Object Management Group. The Object Management Group, 2010. <http://www.omg.org/>. 17
- [OGT⁺99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimburger, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999. 32
- [OMG04] OMG. CORBA component model, 2004. <http://www.omg.org/technology/documents/formal/components.htm>. 25
- [OMT08] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *ICSE Companion*, pages 899–910, 2008. 32
- [Ope07a] Open Service Oriented Architecture Collaboration (OSOA). *Service Component Architecture*, November 2007. 69
- [Ope07b] Open SOA. Service component architecture specifications, November 2007. www.osoa.org/display/Main/Service+Component+Architecture+Home. 55, 158
- [Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. 22, 24
- [PBj98] Frantisek Plasil, Dusan Balek, and Radovan Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. *Configurable Distributed Systems, International Conference on*, 0:43, 1998. 25
- [PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005. 12, 14, 15, 46, 62, 156, 157
- [PK10] An Phung-Khac. *A Model-driven Feature-based Approach to Runtime Adaptation of Distributed Software Architectures*. PhD thesis, Université Européenne de Bretagne., November 2010. 41, 43
- [PKGJ08] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jézéquel. Reconciling automation and flexibility in product derivation. In *12th International Software Product Line Conference (SPLC 2008)*, pages 339–348, Limerick, Ireland, September 2008. IEEE Computer Society. 3, 35, 37
- [Plu93] Detlef Plump. *Hypergraph rewriting: critical pairs and undecidability of confluence*, pages 201–213. John Wiley and Sons Ltd., Chichester, UK, 1993. 73
- [PNP06] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA, may 2006. 69
- [PRS04] Renaud Pawlak, Jean-Philippe Retailé, and Lionel Seinturier. *Programmation orientée aspect pour Java/J2EE*. Eyrolles, 2004. 22
- [PSDC08] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien, and Thierry Coupaye. A component-based and aspect-oriented model for software evolution. *IJCAT*, 31(1/2):94–105, 2008. 39, 40

-
- [RCS08] Romain Rouvoy, Denis Conan, and Lionel Seinturier. Software architecture patterns for a context-processing middleware framework. *IEEE Distributed Systems Online*, 9(6):1, 2008. 86
- [RGF⁺06] Raghu Y. Reddy, Sudipto Ghosh, Robert B. France, Greg Straw, James M. Bieman, N. McEachen, Eunjee Song, and Geri Georg. Directives for composing aspect-oriented design class models. *T. Aspect-Oriented Software Development I*, 3880:75–105, 2006. 3, 36, 37
- [Rom11] Daniel Romero. *Towards the conception of a service-oriented middleware for context-aware environments*. PhD thesis, Université de Lille 1, 2011. 139
- [SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *In Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society, 1994. 1
- [Sch06] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006. 2, 60
- [SHT06] Pierre-Yves Schobbens, Patrick Heymans, and J-C Trigaux. Feature diagrams: A survey and a formal semantics. In *Requirements Engineering Conference, 2006. RE 2006. 14th IEEE International*, pages 136–145, 2006. 2, 62, 156
- [SLFG09] Pablo Sánchez, Neil Loughran, Lidia Fuentes, and Alessandro Garcia. Engineering languages for specifying product-derivation processes in software product lines. In *Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, pages 188–207, Berlin, Heidelberg, 2009. Springer-Verlag. 36, 37
- [SMF⁺09] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable sca applications with the frascati platform. In *6th International Conference on Service Computing (SCC'09)*, pages 268–275, sep 2009. 93
- [Sof10] Software Engineering Institute. A Framework for Software Product Line Practice, 2010. 12
- [SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Softw., Pract. Exper.*, 35(8):705–754, 2005. 2, 93
- [Szy02] Clemens A. Szyperski. *Component Software — Beyond Object-Oriented Programming*. Addison Wesley, second edition edition, 2002. 24, 65
- [TCn07] Pablo Trinidad, Antonio Ruiz Cortés, and Joaquín Peña. Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. In *International Workshop on Dynamic Software Product Line*, 2007. 39, 40
- [The10] The Eclipse Foundation. Eclipse Modeling Framework Project (EMF), 2010. <http://www.eclipse.org/modeling/emf/>. 62
- [TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, New York, NY, USA, 1999. ACM. 2, 60

- [VG07] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society. 36, 37
- [Wag08] Dennis Wagelaar. Composition techniques for rule-based model transformation languages. In *ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, pages 152–167, Berlin, Heidelberg, 2008. Springer-Verlag. 36, 37
- [Wit96] James Withey. Investment analysis of software assets for product lines. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, November 1996. 13
- [Zam95] Andreas Zamperoni. Graph-based, integrated development of software: Integrating different perspectives of software engineering. In *In Proceedings of the 18th International Conference on Software Engineering*, pages 48–59. IEEE Computer Society Press, 1995. 60
- [ZC06] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM. 39, 40

Appendixes

Vers les Lignes de Produits Logiciels Dynamiques

A.1 Introduction

Ces dernières années, de nombreux progrès dans le domaine de la mobilité ont été faits. Les téléphones mobiles sont maintenant équipés de multiples capteurs et d'interfaces d'accès aux réseaux qui ouvrent de nombreuses possibilités d'usage.

Pour profiter de toutes ces capacités matérielles et fournir aux utilisateurs une utilisation simplifiée, les logiciels pour ces téléphones mobiles doivent devenir sensibles au contexte, c'est-à-dire, qu'ils doivent surveiller les événements et les informations provenant de leur environnement et réagir en conséquence. De plus, nous remarquons qu'un nombre important des applications mobiles partagent des caractéristiques communes concernant l'architecture, les moyens de communication, la capacité de stockage et les interfaces. Pour cela, nous considérons que les logiciels sensibles au contexte peuvent bénéficier d'une approche basée sur les Lignes de Produits Logiciels (LPL) (Software Product Line en anglais). Les LPL ont été définies pour exploiter les points communs par la définition d'éléments réutilisables, ceci afin d'automatiser la dérivation de multiples produits. Néanmoins, les LPLs ne prennent pas en compte les modifications à l'exécution des applications.

Cette thèse propose une ligne de produits logiciels dynamiques (LPLD) (Dynamic Software Product Line en anglais) appelée CAPucine (pour Context-Aware software Product line). Celle-ci étend une LPL classique en fournissant des mécanismes pour adapter les produits à l'exécution, ceci pour faire face aux changements dynamiques imposés par la sensibilité au contexte. Notre objectif principal est d'unifier les adaptations à la conception et à l'exécution en utilisant des artefacts logiciels de haut niveau. Ces artefacts peuvent ensuite être réutilisés pour mettre en œuvre une LPLD en définissant les processus nécessaires pour produire des produits logiciels à la conception et les adapter pendant son exécution.

Concrètement, la première contribution de cette thèse introduit un modèle de variabilité et un modèle de composition. Avec le modèle de variabilité, nous cherchons à définir une famille de produits et à identifier les points communs et les différences pour un ensemble d'applications

mobiles. Le modèle de composition, quant à lui, est fondé sur les idées du développement logiciel par aspects (Aspect Oriented Software Development en anglais). Nous utilisons le modèle de composition pour modulariser les produits sous forme de modèles d'aspect qui permettent d'obtenir une représentation indépendante de la plate-forme. Chaque modèle d'aspect est formé de trois parties : le modèle d'architecture qui représente les parties de logiciel à ajouter, les advicees qui contiennent un ensemble de modifications demandées et enfin, le point de coupe qui identifie les endroits où les modifications seront effectuées.

Comme seconde contribution, nous proposons deux processus de dérivation du produit : un au moment de la conception et un au moment de l'exécution. Le premier processus vise à construire un produit. Le processus à l'exécution vise à adapter un produit en cours d'exécution. Les deux processus utilisent les modèles de variabilité et de composition définies précédemment. De cette façon, nous permettons aux développeurs d'utiliser les mêmes artefacts logiciels, à la fois pour la conception d'un produit logiciel et pour l'adaptation à l'exécution. Pour la conception, nous nous basons sur une approche dirigée par les modèles, où les transformations et la génération de code sont utilisées pour construire le produit logiciel à partir d'un ensemble de modèles. Pour l'exécution, nous utilisons FraSCAti, une plate-forme d'exécution des applications à base de services et de composants qui comprend des propriétés dynamiques, de façon à pouvoir exécuter les adaptations. Nous utilisons également un gestionnaire de contexte pour traiter les événements provenant de l'environnement et prendre des décisions sur l'adaptation.

Ce travail de recherche s'est déroulé dans le cadre du projet FUI CAPPUCINO, qui a pour objectif de créer des applications mobiles pour des environnements ubiquitaires. Pour valider notre approche, nous avons défini une LPLD pour une étude de cas de vente d'un hypermarché, qui est composée des deux processus de dérivation à la conception et à l'exécution. Le scénario démontre les avantages de notre approche et, en particulier, l'unification réalisée par les modèles d'aspect utilisés au moment de la conception et lors de l'exécution.

A.2 CAPucine : Vue Globale

Nous proposons dans cette section une organisation de la chaîne de production logicielle, appelée CAPucine. Un produit logiciel sensible au contexte dans le cadre de CAPucine contient au final des composants de type SCA, des classes Java, et des scripts d'adaptation. Pour construire ces différents éléments, nous utilisons une chaîne de production de logiciels dont nous décrivons l'organisation dans cette section. Cette ligne de production a la particularité de proposer des points de variation qui dépendent du contexte dans lequel le logiciel sera exécuté. Une SPL dynamique telle que CAPucine présente plusieurs différences vis-à-vis d'une SPL traditionnelle, en ce qui concerne l'architecture, et le cycle de développement. De telles différences sont reflétées en termes : (1) de rôles et responsabilités différents, (2) de type d'artefacts (ou assets) identifiés et construits dans le processus de l'ingénierie du domaine, et (3) de processus de dérivation. La figure A.1 présente une vue globale de notre DSPL avec les phases et les éléments présents dans les processus de l'ingénierie du domaine et de l'ingénierie d'application. Dans le cas de l'ingénierie de domaine, les artefacts sont représentés comme des méta-modèles et modèles pour l'expression de la variabilité ((Features), l'application (Aspects), et la plate-forme (SCA and Java). Pour l'ingénierie d'application, il y a deux types de dérivation de produit : à la conception et à l'exécution. La dérivation au moment de la conception se concentre sur la construction d'un produit à partir de zéro, tandis que la dérivation au moment de l'exécution est prévue pour modifier un produit existant.

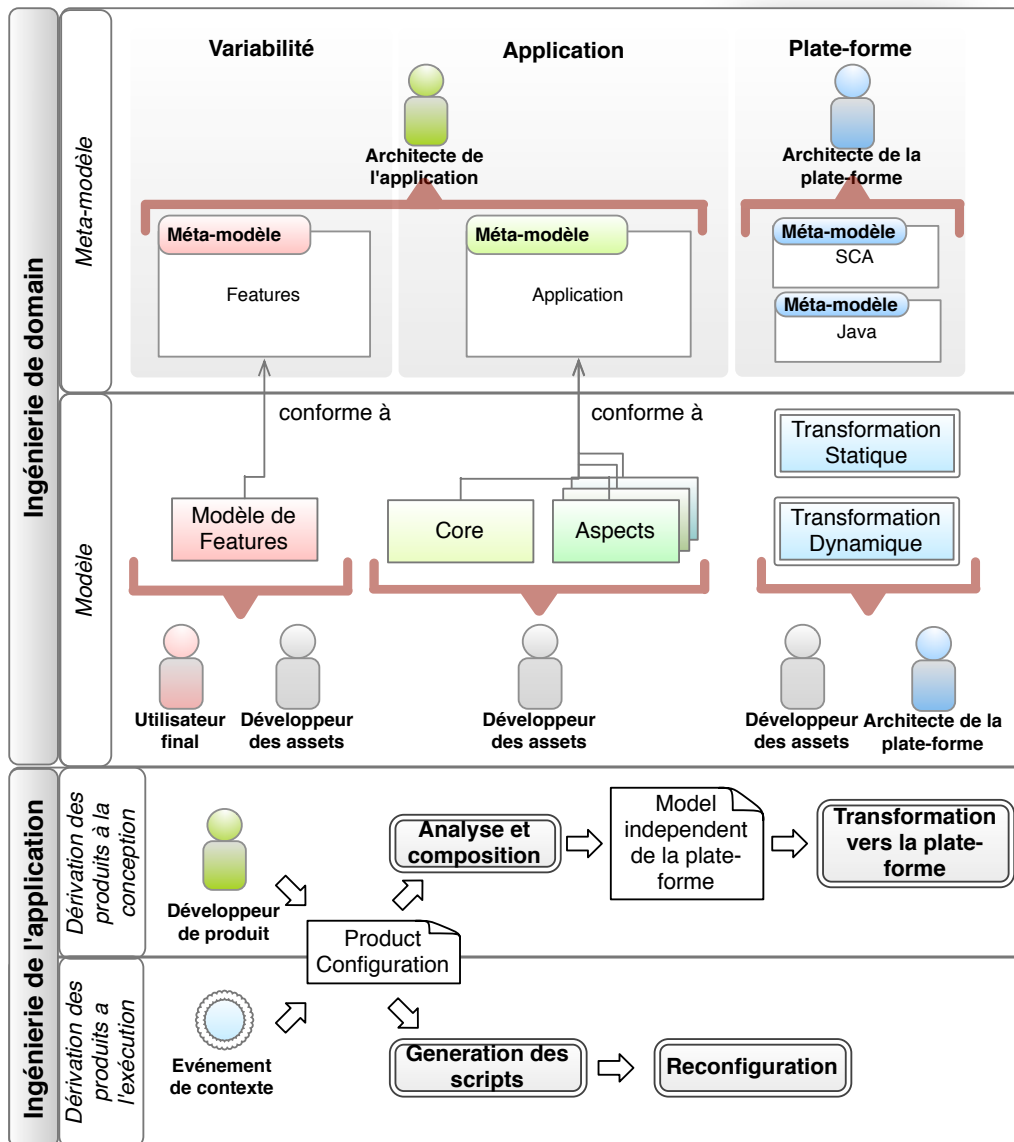


Figure A.1: Ingénieries du domaine et de l'application.

A.2.1 Ingénierie de domaine

Le processus de l'ingénierie de domaine concerne la définition et la construction des assets. Dans notre cas, nous suivons une approche de l'ingénierie Dirigée par les modèles (IDM). En conséquence, les assets correspondent principalement à des méta-modèles, des modèles, et des transformations. Nous divisons la création des assets en deux catégories différentes : assets abstraits, et assets concrets. Les assets abstraits sont nécessaires afin de créer les assets concrets qui sont employés pendant la dérivation de produit pour construire les logiciels. Les assets abstraits correspondent principalement aux méta-modèles, alors que les assets concrets incluent les modèles, les transformations de modèle et les générateurs de code. La responsabilité de développer

chaque type d'asset est partagée entre quatre rôles différents : architecte d'application, architecte de plate-forme, utilisateur final, et développeur d'assets.

A.2.2 Ingénierie de l'application

Le processus de d'ingénierie d'application concerne le développement des différents produits par choix d'un sous-ensemble de caractéristiques et par combinaison des assets définis pendant le processus de l'ingénierie du domaine. Une caractéristique particulière qui distingue une SPL dynamique d'une SPL traditionnelle est qu'il y existe deux rôles supplémentaires lors de la dérivation de produit (Figure A.1) : le développeur de produit qui veut construire un produit en choisissant un ensemble de caractéristiques (features), et la notion d'événement de contexte qui crée également une configuration de produit mais cette fois au moment de l'exécution. Pour chaque rôle particulier, il y a un processus différent de dérivation de produit qui lance les deux phases différentes : la conception et l'exécution. Les deux phases partagent les mêmes abstractions pour construire un produit pour une configuration donnée, mais la réalisation et les technologies employées pour les mettre en oeuvre sont différentes.

A.3 Phase de Conception

Afin de permettre la dérivation automatisée d'un produit, il est nécessaire de spécifier les éléments composables qui réifient chaque feature. Chacun de ces éléments composables doit inclure toute les informations exigées pour la composition parmi lesquelles nous trouvons : (1) les endroits affectés par la composition, (2) les éléments complémentaires devant être composés et (3) les changements à exécuter afin d'ajouter le feature associé. Pour relever un tel défi, nous présentons la notion de modèle d'aspect qui est une manière de réaliser un élément composable. L'utilisation des modèles d'aspect permet la dérivation des produits au moyen d'une composition logicielle basée sur les features. La définition des modèles d'aspect se fonde sur la Modélisation Orientée Aspect (MOA ou en anglais *Aspect-Oriented Modeling*). De la même façon, le processus de dérivation couvre la configuration de produit, l'analyse de contraintes, le tissage d'aspect, et finalement la transformation vers la plate-forme d'exécution.

A.3.1 Méta-modèles

Méta-modèle de Features

Plusieurs travaux sur la modélisation de features ont proposé de multiples extensions aux travaux sur les diagrammes de features (FD) introduits par [KCH⁺90]. Dans [SHT06] Schobbens *et al.* décrivent différentes approches pour modéliser les features et définissent une syntaxe abstraite pour les diagrammes de features qui élimine les ambiguïtés des précédentes propositions. Ils utilisent une notation mathématique pour définir les relations entre les features. Une approche différente pour traiter l'ambiguïté dans les FDs est de définir un méta-modèle comme celui proposé par Pohl *et al.* [PBL05]. Ce méta-modèle présente deux concepts principaux : les *points de variation* et les *variants*. Un point de variation est une représentation d'un sujet de variabilité, par exemple, le type d'une interface utilisateur qu'une application fournit. Un variant identifie une option simple d'un point de variation. En utilisant le même exemple, chaque interface utilisateur simple qui peut être choisie pour l'application (par exemple, riche, léger, basé sur le web, mobile)

est représentée par un variant. Le méta-modèle présenté dans [PBL05] spécialise alors les relations entre les points de variation et les variants, en classifiant les types de relations qui peuvent exister. Ils définissent les dépendances (*facultatif* et *obligatoire*) et des contraintes (*requires* et *excludes*). Dans notre cas, nous avons défini un méta-modèle de features inspiré des concepts que Pohl *et al.* ont identifiés. Nous définissons les mêmes concepts et les relations en utilisant Eclipse Modeling Framework (EMF), mais nous changeons la manière dont ils sont modélisés, puisque l'EMF ne supporte pas la spécialisation ou l'héritage des relations entre deux méta-classes différentes. Notre méta-modèle de features est montré dans la figure A.2.

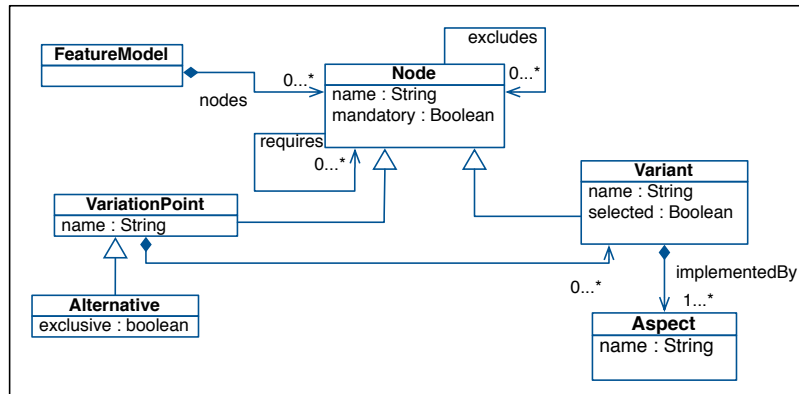


Figure A.2: Métamodèle de Features.

La racine du métamodèle est la méta-classe `FeatureModel`. Elle contient une ensemble de `Nodes`. Chaque `Node` a deux relations : `requires` ou `excludes`. De telles relations tiennent compte de la définition des contraintes entre tous les types de `Node`. Le `Node` peut également être obligatoire ou pas. Nous supportons deux types de `node` : le `VariationPoint`, et le `Variant` qui sont les équivalents des concepts du point de variation et du variant discutés préalablement. En plus, un point de variation peut être alternatif, ainsi cela signifie qu'à partir de tous les variants alternatifs, seulement un doit être choisi. En conclusion, toutes les variants ont au moins un `Aspect` qui les implémente.

Méta-modèle d'Aspects

Le méta-modèle d'aspect est utilisé pour construire les assets qui seront composés pour produire n'importe quel produit de CAPucine. Nous utilisons ce méta-modèle pour définir à la fois le modèle de base (e.g., le cœur ou *core*) qui contient les éléments communs de la famille de produits (e.g. les features obligatoires), et les aspects contenant la variabilité (e.g. les features optionnels) qui peuvent être tissés sur le modèle de base.

Le métamodèle d'aspects (voir figure A.3) est formé de quatre parties : les éléments à tisser (`Model`), les endroits où le tissage doit être réalisé (`Pointcut`), les modifications à apporter via l'aspect (`Advice`), et de façon optionnelle, le moment à l'exécution où l'aspect peut être tissé (`Event`).

- *Model* : la partie `Model` du métamodèle est utilisée pour définir le modèle *core*. Pour modéliser le *core*, nous avons créé un métamodèle inspiré des éléments issus des modèles à

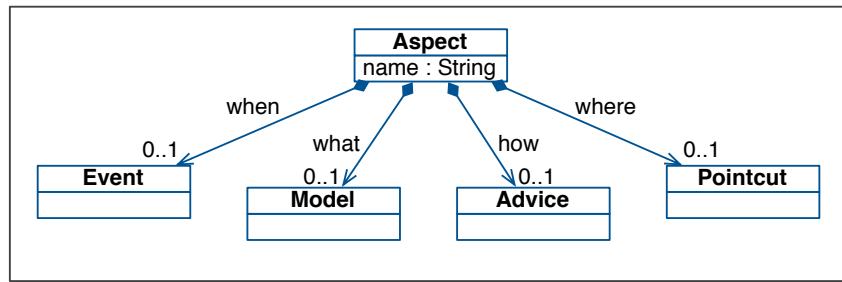


Figure A.3: Méta-modèle d'aspects: vue simplifiée.

services et à composants (SOA et CBSE), et plus particulièrement SCA (Service Component Architecture) [Ope07b].

- *Pointcut* : nous considérons le *Pointcut* comme une requête qui retourne tous les éléments du modèle qui doivent être présents dans le modèle pour qu'un aspect soit tissé. Nous proposons une modélisation des pointcuts sous la forme d'expressions.
- *Advice* : nous considérons l'*Advice* comme une séquence de modifications atomiques. Nous supportons deux types de modifications: *Add* et *Delete*. Dans le premier cas, pour ajouter un élément, chaque élément *Add* lie un élément du modèle, et un résultat de l'exécution d'un pointcut, qui représente l'endroit où l'élément va être ajouté.
- *Event* : même si la notion du temps n'est seulement utilisée que pendant l'exécution d'une application, il est nécessaire de la modéliser. Pour faire ceci, nous employons des événements contextuels. Par contexte nous entendons chaque type d'information pouvant affecter une application. Les exemples de telles informations correspondent à la disponibilité des ressources, des services, ou encore à des informations telles que la température, le lieu, ou les restrictions liées au matériel comme la mémoire. En conséquence, un événement de contexte est considéré comme un changement de l'information de contexte.

Plate-forme et langage

Nous utilisons FraSCaTi comme plate-forme d'exécution des applications. Pour la transformation vers cette plate-forme, et la génération de code, nous utilisons deux méta-modèles différentes. Un qui définit les architectures à base des services et composants (SCA), et un qui définit les éléments classiques des implémentations en Java. Concrètement, nous avons employé le méta-modèle défini par le groupe d'OSOA pour SCA, et le méta-modèle Java proposé dans le projet Spoon EMF [ND08, Bar06] pour le langage Java.

A.3.2 Derivation

Analyse de contraintes

Le but de l'analyse statique de contraintes est d'identifier et de prévenir les conflits qui peuvent exister entre deux mondes différents, le monde de la variabilité exprimé par un modèle de variabilité et le monde des aspects représenté avec un *core* et plusieurs aspects.

Dans notre approche nous proposons une analyse des contraintes dans le diagramme de features (FD) et des dépendances entre les éléments composables correspondants.

Le processus d'analyse de contraintes a lieu une fois que le développeur a configuré un produit particulier. La sélection de features est représentée comme un ensemble de variants. Basée sur cette sélection, l'analyse de contraintes vise (1) à vérifier que les contraintes définies dans le FD sont conformes aux dépendances correspondantes entre les aspects, (2) à identifier les conflits implicites de composition, et (3) à dériver l'ordre le plus approprié de composition.

Composition de modèles

À la fin de l'analyse de contraintes, nous avons comme résultat l'ordre de tissage pour les modèles d'aspect qui font partie de la configuration sélectionnée par le développeur du produit. L'étape suivante consiste à composer ces modèles pour avoir une représentation de haut niveau du produit que nous sommes en train de construire. D'une façon générale, la composition de modèles d'aspects se compose des appels successifs à une transformation simple de modèles génériques (tissage). Cette transformation prend comme entrée le modèle M du core et un aspect A à tisser, et renvoie un modèle simple représentant la composition du core et de l'aspect. La transformation est exécutée autant de fois qu'il y a d'aspects à tisser car les aspects doivent être tissés dans l'ordre défini par l'analyse de contraintes. Le tisseur utilise les informations dans chaque aspect pour faire la composition. Plus précisément, le tisseur utilise les expressions dans le pointcut pour trouver les endroits où il doit modifier le core, et ensuite il exécute ces modifications tels qu'elles sont exprimées dans l'advice.

Transformation et génération de code

Afin de produire le code source correspondant à ce produit, nous suivons une approche classique MDE où le modèle composé est l'entrée d'une transformation qui prend en compte les concepts de la plate-forme à services (SCA) et du langage de programmation (Java). Chaque transformation se compose d'un ensemble de règles qui mettent en correspondance les concepts du modèle avec les éléments Java et SCA. Par exemple, un élément sera mis en correspondance dans (1) un composant SCA, (2) une interface de Java définissant ses services, et (3) des classes Java qui implémentent l'interface de Java et représentent le composant de SCA.

La figure A.4 représente une vue conceptuelle montrant comment les modèles, les transformations et la génération de code sont traités. En transformant d'abord les concepts des modèles en modèles Java et SCA, nous pouvons vérifier la cohérence de ces nouveaux modèles. Ce ne serait pas possible si nous avions directement produit du code source à partir du méta-modèle d'aspect. En conclusion, le code est généré à partir des modèles SCA et Java pour obtenir les descripteurs composites et le code Java respectivement.

A.4 Phase d'exécution

Nous avons présenté un processus de dérivation de produit lors de la conception utilisant des modèles d'aspect. Nous avons montré aussi un processus d'analyse et de dérivation de contraintes conduit par les sélections de features. Dans cette section nous allons présenter une prolongation de ce processus de dérivation au moment de l'exécution. Nous montrons les mécanismes requis pour utiliser les modèles d'aspect dans la reconfiguration à l'exécution.

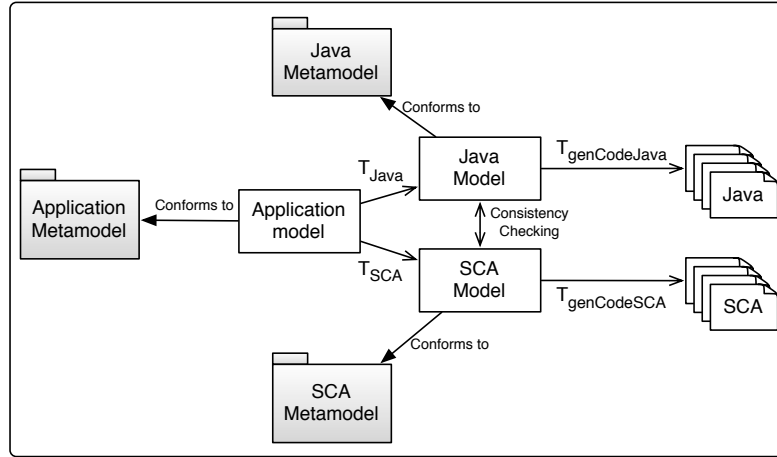


Figure A.4: Schéma de transformation.

A.4.1 Adaptation en CAPucine

Pour l'adaptation dynamique en CAPucine, nous définissons un processus de tissage à l'exécution, que transforme le modèle d'aspect défini précédemment dans une nouvelle représentation appropriée pour la plate-forme d'exécution.

La figure A.5 illustre le processus défini pour le cycle d'adaptation. Il commence par un événement qui est traité par le gestionnaire de contexte. À la fin de l'agrégation, une information de contexte est représentée comme un observable. L'observable déclenche l'adaptation. La première partie de l'adaptation vise à définir les variants du diagramme de features (FD) qui doivent être présents dans le produit en fonction des valeurs courantes des différents observables. Cette liste est alors vérifiée par le validateur qui compare la nouvelle sélection avec les contraintes du FD. L'algorithme employé pour vérifier les contraintes correspond aux analyses de contraintes de la phase de conception. S'il n'y a aucun conflit, le générateur de scripts effectue la différence entre la nouvelle configuration obtenue à partir du validateur et la configuration courante du produit. Le résultat de cette différence donne deux ensembles : un ensemble de variants qui doit être retiré du produit, et un ensemble de variants qui doit être ajouté. Pour que chaque variant soit enlevé ou ajouté, des scripts sont produits pour l'aspect associé et qui peuvent être exécutés sur la plate-forme FraSCAti.

Adaptateur

L'Adaptateur vise à trouver une nouvelle configuration, exprimée en termes de variants choisis dans le FD. Pour décider quels sont les variants à employer, il utilise l'information disponible uniquement pendant l'exécution du produit. Néanmoins, comme la notion de variant appartient aux étapes initiales de conception, une des premières tâches pour rendre la SPL dynamique est d'apporter la notion de variant au produit pendant l'exécution. Sans cette information, raisonner en termes de variants choisis n'est pas possible. Pour tracer la notion du feature à l'exécution, nous ajoutons un composant supplémentaire au noyau de chaque produit généré par le DSPL. Un tel composant est responsable de maintenir une représentation de l'état courant du produit en termes de features. Il expose un service avec deux opérations, une pour obtenir la configuration

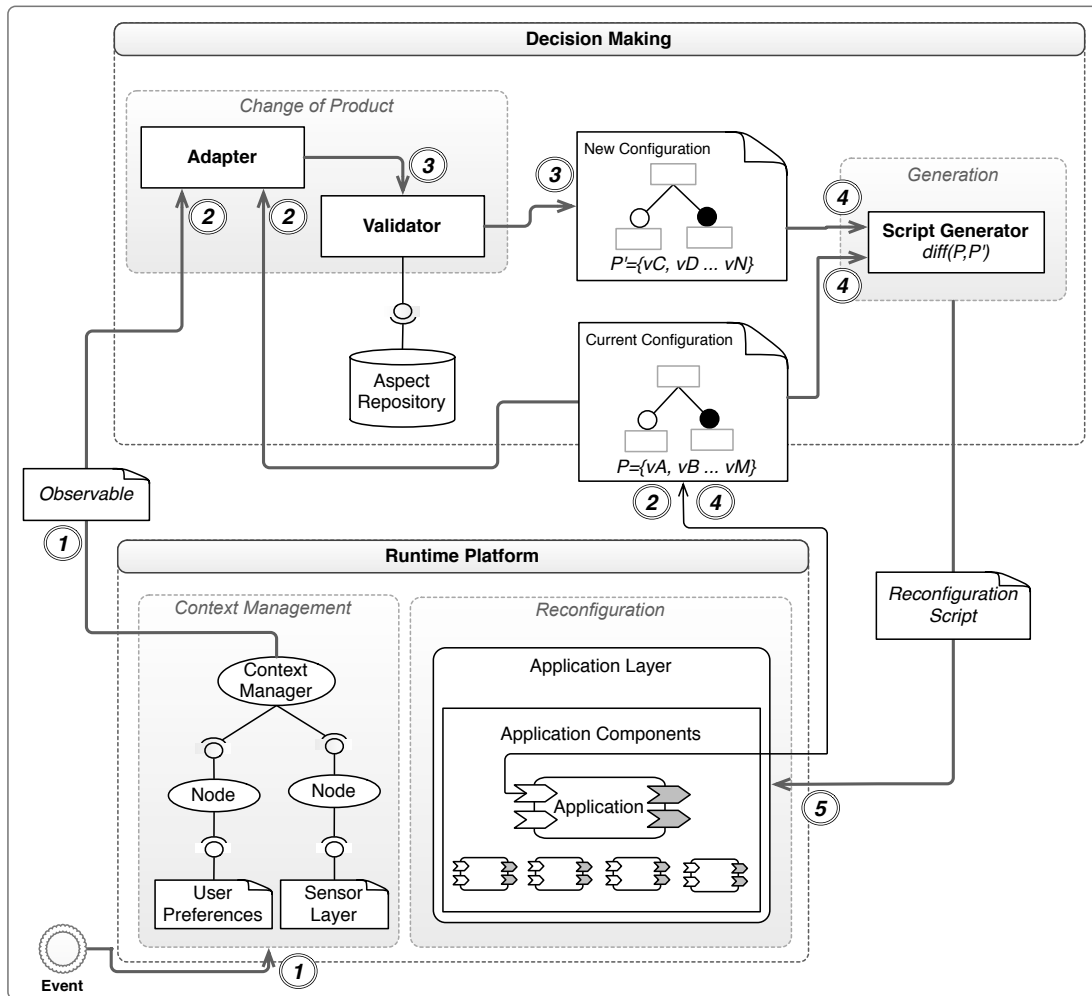


Figure A.5: Dérivation d'un produit à l'exécution.

courante, et une pour mettre à jour la configuration. Tous les deux sont employés pendant la reconfiguration pour obtenir la configuration courante et mettre à jour la nouvelle configuration une fois que le produit a été adapté. Chaque aspect lié à un variant a plusieurs observables qui indiquent quand le tisser. À chaque mise à jour de contexte, les observables des aspects affectés sont réévaluées pour décider si l'aspect doit être tissé/détissé.

Valideur

Le valideur est quant à lui chargé de vérifier la cohérence du produit. Pour cela il utilise les résultats des analyses réalisées au moment de la conception, et notamment, il prend en compte l'ensemble des configurations que nous obtenons avec les algorithmes d'analyse de contraintes. A partir de cette information, il prendra alors la décision d'appliquer les modifications nécessaires et de faire appel au générateur de scripts.

Générateur de scripts

Le générateur de scripts calcule la différence entre les modèles de features. Le résultat de la différence donne une liste d'aspects à détisser, et une liste d'aspects à tisser. A ce moment, l'information de chaque aspect dans la configuration est employée pour obtenir les scripts de tissage/détissage. Ces scripts sont ensuite exécutés.

Puisque les aspects sont des modèles, le but avec la génération de scripts est de créer les scripts pour les adaptations demandées en vue de tisser ou détisser un aspect, et d'exécuter ces scripts à l'exécution. Pour cela nous construisons une transformation qui prend comme entrée chaque modèle d'aspect qui produit en sortie les scripts nécessaires de reconfiguration, en utilisant les langages FPath et FScript. Ces deux langages sont utilisés pour parcourir et déterminer des endroits précis dans une application à composants (i.e. l'équivalent du pointcut), et ensuite, pour définir des modifications sur l'application sur ces endroits (i.e. l'équivalent de l'advice).

A.5 Conclusion

Cette recherche apporte de nouvelles idées sur le domaine des lignes de produits logicielles dynamiques. En particulier, nous avons relevé les défis du développement d'applications adaptables selon les principes SPL.

Nous avons défini un méta-modèle pour l'expression de la variabilité des produits logiciels. Grâce à cette variabilité, il est possible de fabriquer des produits différents qui partagent un ensemble de fonctionnalités, mais qui se différencient par un ensemble de features (variabilité). Nous avons également défini un méta-modèle qui permet d'avoir une représentation architectural de haut niveau des applications basées sur les principes de SOA et CBSE. Ce méta-modèle s'inspire des notions d'aspects, en définissant par exemple des pointcuts et des advices. Avec ce méta-modèle d'aspects, nous avons été capables de diviser l'architecture des applications en deux parties: la première partie qui contient les éléments obligatoires pour tout produit, et la seconde partie qui contient un ensemble de modèles d'aspect indépendants représentant les éléments facultatifs (variabilité). Le méta-modèle et les modèles dérivés ont été ensuite utilisés pour la dérivation des produits, en associant des aspects à chacune des features identifiées dans le modèle de variabilité. Pour compléter la dérivation des produits, nous avons défini des transformations vers une plate-forme cible.

En outre, nous avons défini un deuxième processus de dérivation de produits qui est pris en compte au moment de l'exécution des produits logiciels. Pour cela, nous avons enrichi le méta-modèle d'architecture en ajoutant la notion d'observable, qui représente un événement de contexte. Les produits peuvent alors être adaptés en fonction des conditions spécifiques d'exécution en utilisant les mêmes modèles d'aspect utilisés pour leur conception.

Grâce au méta-modèle d'architecture basée sur les aspects, notre approche *unifie* les adaptations des applications sensibles au contexte, en utilisant deux processus de dérivation différents au moment de la conception et à l'exécution. Les aspects peuvent à la fois être utilisés pour représenter les sélections des développeurs au moment de la conception et ils peuvent aussi prendre en compte les informations provenant de l'environnement et les utiliser pour déclencher une adaptation dynamique. Par conséquent, les développeurs doivent définir une seule fois les modèles d'aspect pour ensuite les utiliser aussi bien à la conception que à l'exécution.

