FieSta: An approach for Fine-Grained Scope Definition, Configuration and Derivation of Model-Driven Software Product Lines

Hugo Arboleda

FieSta: An approach for Fine-Grained Scope Definition, Configuration and Derivation of Model-Driven Software Product Lines

Universidad de Los Andes

Hugo Fernando Arboleda Jiménez

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor in Engineering in the School of Engineering, Department of Systems and Computing

Supervised by
Rubby Casallas
and
Jean-Claude Royer

Jury

Rubby Casallas - Director
Jean-Claude Royer - Director
Laurence Duchien - Reviewer
Juan Francisco Díaz - Reviewer
Dario Correal - Reviewer
Dirk Deridder - Reviewer

October 2009

UNIVERSITÉ DE NANTES
UFR SCIENCES ET TECHNIQUES

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATHÉMATIQUES

Année 2009

FieSta: An approach for Fine-Grained Scope Definition, Configuration and Derivation of Model-Driven Software Product Lines

THÈSE DE DOCTORAT
Specialité: Informatique et Applications
Présentée
et soutenue publiquement par

Hugo Fernando Arboleda Jiménez

Le 28 Octobre 2009 à Bogotá, devant le jury ci-dessous

Président	:		
Rapporteurs	:	Laurence Duchien	Professeur, Universit de Lille 1
		Juan Francisco Díaz	Professeur, Universidad del Valle
Examinateurs	:	Rubby Casallas	Professeur, Universidad de Los Andes
		Jean-Claude Royer	Professeur, École des Mines de Nantes
		Dario Correal	Matre Assistant, Universidad de Los Andes
		Dirk Deridder	Postdoctoral Researcher, Vrije Universiteit Brussel

Directeures de thèse : Rubby Casallas et Jean-Claude Royer

Laboratoire : UMR Laboratoire informatique de Nantes Atlantique (LINA)
Etablissement(s) d'accueil : École des Mines de Nantes-INRIA, LINA
Addresse : La Chantrerie – 4, rue Alfred Kastler – 44307 Nantes cedex 3 – France

ED: 503-029

to Lina, my lovely wife

a Lina, mi encantadora esposa

Abstract

We present FieSta, an approach based on Model-Driven Development ideas to create Software Product Lines (SPLs). In Model-Driven SPL approaches, the derivation of a product starts from a domain application model. This model is transformed through several stages reusing model transformation rules until a product is obtained. Transformations rules are selected according to variants included in *configurations* created by product designers. Configurations include *variants* from *variation points*, which are relevant characteristics representing the variability of a product line. FieSta (1) provides mechanisms to improve the expression of variability of Model-Driven SPLs by allowing designers to create *fine-grained* configurations of products, and (2) integrates a product derivation process which uses *decision models* and Aspect-Oriented Programming facilitating the reuse, adaptation and composition of model transformation rules.

We introduce *constraint models* which make it possible for product line architects to capture the scope of product lines using the concepts of *constraint*, *cardinality property* and *structural dependency property*. To configure products, we create domain models and binding models, which are sets of bindings between model elements and variants and satisfy the constraint models.

We define a decision model as a set of *aspects*. An aspect maintains information of when transformation rules that generate commonalities of products must be intercepted (*joinpoints*) and what transformation rules (*advices*) that generate variable structures must be executed instead. Our strategy maintains uncoupled variants from model transformation rules. This solves problems related to modularization, coupling, flexibility and maintainability of transformations rules because they are completely separated from variants; thus, they can evolve independently.

Resumen

Presentamos FieSta, un enfoque basado en el desarrollo dirigido por modelos para la creación de líneas de producto de software. En los enfoques de creación de líneas de producto basados en el desarrollo dirigido por modelos, la derivación de un producto parte de un modelo de dominio. Este modelo es transformado en varias etapas, hasta que se obtiene un producto, reusando reglas de transformación de modelos . Las reglas de transformación son seleccionadas de acuerdo con variantes incluidos en *configuraciones* creadas por diseadores de producto. Las configuraciones incluyen *variantes* de *puntos de variación*, los cuales son características relevantes que representan la variabilidad de una línea de producto. FieSta (1) proveé mecanismos para mejorar la expresión de variabilidad de líneas de producto creadas usando el paradigma de desarrollo dirigido por modelos, permitiendo a diseadores de producto crear configuraciones *finas*, e (2) integra un proceso de derivación que usa *modelos de decisión* y programación orientada por aspectos facilitando el reuso, adaptación y composición de reglas de transformación de modelos.

Introducimos los *modelos de restricciones*, los cuales hacen posible que arquitectos de líneas de producto capturen el alcance de líneas de producto usando los conceptos de *restricción*, *propiedad de cardinalidad*, y *propiedad de dependencia estructural*. Para configurar productos, creamos modelos de dominio y modelos de relaciones, los cuales son conjuntos de relaciones entre elementos de modelos y variantes, y satisfacen los modelos de restricciones.

Definimos un modelo de decisión como un conjunto de *aspectos*. Un aspecto mantiene información de qué y cuándo reglas de transformación que generan características comunes de products deben ser interceptadas (*joinpoints*) y qué reglas de transformacin (*advices*) que generan estructuras variables deben

ser ejecutadas en su lugar. Nuestra estrategia mantiene desacoplados los variantes de las reglas de transformación. Esto resuelve problemas relacionados con modularización, acoplamiento, flexibilidad y mantenibilidad de reglas de transformación debido a que estas últimas permanecen completamente separadas de los variantes; asi, ellas pueden evolucionar independientemente.

Résumé

Nous présentons FieSta, une approche basée sur les idées de l'ingénierie dirigée par les modles pour créer des lignes de produits logiciels. Dans les approches dirigée par les modles pour créer lignes de produits logiciels, la dérivation d'un produit commence par un modle d'application du domaine. Ce modle est transformé en plusieurs étapes en utilisant des rgles de transformation de modles, jusqu' ce qu'un produit final soit obtenu. Les rgles de transformations sont choisies selon les variantes incluses dans les configurations créés par le concepteur des produits. Les configurations comportent des variantes associées des points de variation, qui sont des caractéristiques représentant la variabilité d'un ligne de produit. FieSta (1) fournit des mécanismes pour améliorer l'expression de la variabilité des lignes de produits dirigées par les modles en permettant des concepteurs de créer des configurations grain fin des produits, et (2) intgre un processus de dérivation des produits qui emploie des modles de décision et la programmation dirigé par les aspects pour faciliter la réutilisation, l'adaptation et la composition des rgles de transformation des modles. Nous présentons les modles de contraintes qui permettent aux architectes du produit de capturer le domaine des produits en utilisant les concepts de contrainte, de propriété de cardinalité et de propriété de dépendance structurale. Pour configurer les produits, nous créons les modles de domaine et les modles de décision, qui sont des ensembles de liens entre des éléments et des variantes et satisfont les modles de contraintes. Nous définissons un modle de décision comme un ensemble d'aspects au sens de la programmation par aspects. Un aspect mémorise l'information concernant quand, o et comment intercepter une rgle de transformation produisant la base commune du produit. Ces aspects détectent les points de jonctions o de nouvelles rgles de transformation, gérant la variabilité, doivent tre exécutées. Notre stratégie maintient la création des variantes découplé des rgles

de transformation pour les parties communes. Ceci résout des problmes liés la modularisation, l'appariement, la flexibilité et la maintenance des rgles de transformations. Parceque les rgles communes sont compltement séparées des variantes, elles peuvent plus facilement évoluer indépendamment.

Acknowledgement

The work presented in this thesis was funded in a large part by COLCIENCIAS - "Departamento Administrativo de Ciencia Tecnología e Innovación" and by The European Commission STREP Project AMPLE IST-033710.

Thank you to the members of the ASCOLA/OBASCO Group at the École des Mines de Nantes for giving me the opportunity to do my thesis with them.

Thank you to the members of the Software Construction Group at the University of Los Andes. A very special thanks to Andrés Romero for his collaboration on tool support and case study development, in addition to the enriching discussion and feedback he provided me.

I owe my gratitude to my Ph.D. committee members for taking the time to read this dissertation in detail, and for providing me with valuable comments. Apart from my advisors, Rubby and Jean-Claude, the committee members are Prof. Dr. Laurence Duchien, Prof. Dr. Juan Francisco Díaz, Prof. Dr. Dario Correal and Prof. Dr. Dirk Deridder.

Thank you to all my friends in France, especially Daniel (and his wife Maury), Stephane, Fabricio, Joost, Ali, Angel, and Veronica, who ensured that our stay at Nantes was wonderful. I would also like to thank my colleagues and friends, Oscar Gonzales and Andrés Yie, who have made my job more pleasant and were always available for interesting discussions. I'd like to thank my closest friends, Yulder (now in Pittsburgh, US), Lucho, Jose, Mauricio (now in Madrid, ES), Julian Valencia, Juian Cifuentes (now in Buenos Aires, AR), Fernando, and Jonas, for keeping in touch with me after I moved to Bogotá and Nantes. Despite not seeing one another for a long time, they have consistently been in touch with me to give me their support.

I would like to thank Nelson Perlaza and Aura Rosa de Perlaza for offering their support to me, and for sharing their friendship with my parents.

I would like to thank my parents, Hugo and Elsa, who have always supported my studies from the very beginning. I would also like to thank my sister, Angela, and her husband, Luis Fernando, my brother, Gabriel, as well as my aunt, Georgina, for all their help with our home and love they showed to me and my brothers. I would not be who I am without them.

All my love and gratefulness goes to my wife, Lina, for giving me emotional support during my years of studies. She was always unconditionally there when I needed her the most.

My deepest thanks goes to my advisors. Rubby Casallas, since the time of my Masters' thesis, believed in me and provided me with guidance and encouragement. Jean-Claude Royer (and his wife Zara) welcomed Lina and me warmly during our stay in France and supported me with cooperative supervision. Thank you, Jean-Claude and Zara, for making us feel at home.

Agradecimientos

El trabajo presentado en esta tesis fue auspiciado en gran parte por COLCIENCIAS - Departamento Administrativo de Ciencia Tecnología e Innovación y por The European Commission STREP Project AMPLE IST-033710.

Gracias a los miembros del grupo ASCOLA/OBASCO en la École des Mines de Nantes por darme la oportunidad de realizar esta tesis con ellos.

Gracias a los miembros del Grupo de Construcción de Software en la Universidad de Los Andes. Quiero agradecer especialmente a Andrés Romero por su colaboración en el desarrollo de las herramientas de soporte y los casos de estudio que acompaan esta tesis, además de las enriquecedoras discusiones y retroalimentación.

Debo mi gratitud a los miembros del comité evaluador de mi tesis, por tomar el tiempo para leer mi disertación en detalle y por darme sus valiosos comentarios. Aparte de mis asesores Rubby y Jean-Claude, los miembros del comité son Prof. Dr. Laurence Duchien, Prof. Dr. Juan Francisco Díaz, Prof. Dr. Dario Correal y Prof. Dr. Dirk Deridder.

Gracias a todos mis amigos en Francia, especialmente a Daniel y su esposa Maury, Stephane, Fabricio, Joost, Ali, Angel y Veronica, quienes ayudaron a que nuestra estadía en Nantes fuera muy agradable. Quisiera agradecer también a mis colegas y amigos Oscar Gonzales y Andrés Yie, quienes hicieron mi trabajo más placentero y siempre estuvieron ahí para interesantes discusiones. Quisiera agradecer a mis amigos más cercanos, Yulder (ahora en Pittsburgh, US), Lucho, Jose, Mauricio (ahora en Madrid, Es), Julian Valencia, Julian Cifuentes (ahora en Buenos Aires, Ar), Fernando y Jonas, por mantenerse en contacto conmigo luego de que me moví a Bogotá y a Nantes. Incluso luego de no vernos por mucho tiempo, siempre han estado en contacto

para darme su apoyo.

Un agradecimiento muy especial a Nelson Perlaza y Aura Rosa de Perlaza por ofrecerme su apoyo, y por compartir su amistad con mis padres.

Quisiera agradecer a mis padres, Hugo y Elsa, quienes siempre han apoyado mis estudios desde el principio. Tambien quisiera agradecer a mi hermana Angela y su esposo Luis Fernando, mi hermano Gabriel, así como a mi tia Georgina por toda su ayuda con nuestro hogar y su amor para mis hermanos y para mí. Yo no sería quien hoy soy si no hubiera contado siempre con ellos.

Todo mi amor y gratitud para mi esposa, Lina, por darme soporte emocional durante mis años de estudio. Ella siempre estuvo incondicionalmente ahí cuando más la necesité.

Mis más profundos agradecimientos a mis asesores. Rubby Casallas, quien desde el tiempo de mi tesis de mestría creyó en mi, me brindó su guía y me alentó para seguir adelante. Jean-Claude Royer (y su esposa Zara), quien nos acogió calurosamente a Lina y a mi durante nuestra estadía en Francia, quien compartió conmigo importantes consejos y siempre me apoyó con su supervisión. Gracias Jean-Claude y Zara por hacernos sentir en casa.

Contents

List of Figures

List of Tables

List of Abbreviations

AOM	Aspect Oriented Modeling
AOP	Aspect Oriented Programming
API	Application Programming Interface
ATL	ATLAS Transformation Language.
DSM	Domain Specific Modeling
DSML	Domain Specific Modeling Language
EMF	Eclipse Modeling Framework
FODA	Feature-Oriented Domain Analysis
GUI	Graphical User Interface
MDD	Model Driven Development
MD-SPL	Model Driven Software Product Line
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
oAW	openArchitectureWare
QVT	Query/View/Transformation Language
SEI	Software Engineering Institute
SPL	Software Product Line

UML	Unified Modeling Language
OWL	Web Ontology Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Part I. Introduction

Chapter 1

Introduction

1.1 Context

Software engineering aims at speeding up software development and maintenance processes, decreasing costs, and improving productivity and quality. Addressing those objectives, Software Product Line (SPL) Engineering searches to develop software products using already developed artifacts well tested and improved [CNN01, Bos00]. Thus, products should be rapidly developed, and their quality should be as good as the quality of the artifacts used for their construction. A Software Product Line is defined as a set of similar products, in the context of one specific application domain, created from reusable artifacts [CE00]. In SPL Engineering, product designers configure and derive products by reusing the available artifacts created by product line architects.

The description of the set of products which are part of an SPL is called the scope of the product line [Cle02]. To capture and express the scope of SPLs, product line architects first determine the commonalities, *i.e.* the characteristics all products in a product line share, and then the ways in which they can vary (variability). Variability models include *variation points* and *variants*. Variation points are relevant characteristics that can have different values or variants according to the variability of a product line [PBvdL05].

Many approaches to create SPLs have emerged based on Model-Driven Devel-

opment (MDD), *e.g.* [VG07b, Wag05]. These are called *MDD-based SPL approaches* or *MD-SPL approaches*. MDD conceives the whole software development cycle as a process of creation, iterative refinement and integration of models. An MD-SPL is a set of products developed from domain application models, and derived from a set of reusable model transformation rules. For many in the domain (*e.g.* [VG07b]), including us [ARCR09, ACR09], these model transformations may require several stages. At each stage, domain application models are automatically transformed to include more implementation details. It means, models including only problem space concerns are incrementally transformed to include the solution space, *i.e.* concerns of software design and/or technological platforms. At the end of a staged model transformation process, models including all the implementation details are transformed into source code of software systems.

There are two major processes related to our work. On the one hand, there is the process of capturing and expressing variability in MD-SPLs, which impacts consequently the process of configuring product line members. On the other hand, there is the process of deriving products reusing and composing model transformations based on product configurations.

Most of the current MD-SPL approaches [VG07b, Wag05, LSGF, SLFG08] create *separately* domain application metamodels and variability models to capture and express variability. For configuring a particular product, product designers create *configurations* that consist of (1) domain application models and (2) *instances* of variability models. An instance of a variability model includes a selection of variants from the variability model. MD-SPL approaches using multi-staged model transformations also easy the configuration of products before each model transformation stage starts by creating specific instances of variability models. For example, product designers can select software architectural details before executing model transformations in charge of adding architectural information. Therefore, the staged transformation of a domain application model may derive products with different software architecture or products to run on different technological platforms.

During the product derivation process, the instances of variability models are used to decide what transformation rules must be applied. Thus, from different instances of variability models, different products can be derived from a same domain application model.

Figure 1.1 summarizes the process of creating an MD-SPL's example. Each

product line member manages its data by means of a relational database schema. In this example, product line architects have chosen to use the UML `Class Metamodel` to capture and express the variability related to problem space concerns. Thus, product designers are able to start the configuration process of products by creating diverse `class models`. To capture variability in the context of relational database schemas, product line architects create a variability model which includes one variation point, `Primary Key Structure`, which has two alternative variants, `With Primary Key` and `Without Primary Key`. Additionally, the architects relate one different model transformation rule to each variant. The `Rule One` is related to the variant `With Primary Key` and the `Rule Two` is related to the variant `Without Primary Key`. Product designers complete the configuration process of products by creating instances of the variability model. If the variant `With Primary Key` is selected in an instance of the variability model, using the `Rule One`, all the class elements in a `Source Class Model` are transformed into table elements with one primary key. If the variant `Without Primary Key` is selected in other instance of the variability model, using the `Rule Two`, all the class elements in a `Source Class Model` are transformed into table elements without a primary key.

1.2 Problem Statement

P1. MD-SPL approaches limit the expression of variations between product line members only to coarse-grained ones.

Most of the current MD-SPL approaches capture and express the possible variations between product line members by creating separate metamodels and variability models. When variants are associated to metaconcepts to denote possible variations, we call it *coarse-grained variations*. This is because during models transformation processes the association of a selected *variant* to a *metaconcept* will affect *all* the model elements that conform to such a metaconcept. For example (see Figure 1.1), when we associate the variant `With Primary Key` to the metaconcept `Class` we are denoting a coarse-grained variation. This is because during the model transformation process of class models into table models *all* the `Table` elements will be generated with a primary key.

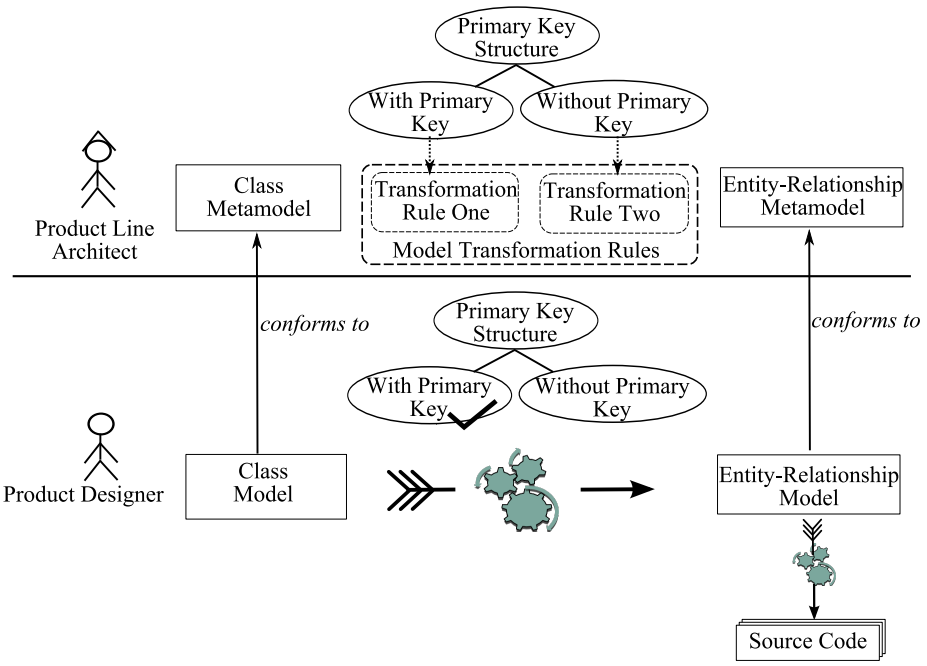

Figure 1.1: Creation Process of an MD-SPL.

When *variants* are associated to *model elements* instead of metaconcepts to denote possible variations, we call it *fine-grained variations*. MD-SPL approaches lack of mechanisms that allow product line architects and product designers to capture fine-grained variations to express, for instance, that products can vary in the particular tables that have a primary key. For example, a fine-grained variation must allow a product designer to indicate that the feature `With Primary Key` affects individually a class `Student`, while the feature `Without Primary Key` affects individually a class `Professor`. It is also required a mechanism to restrict the valid fine-grained variations. For example to indicate that the features `With Primary Key` and `Without Primary Key` could affect `Class` elements individually, but it is not valid that they affect `Attribute` elements from class models.

These are all problems in application domains where (1) model elements must be configured individually and (2) products must be configured in multiple stages, sometimes by designers with different domain knowledge.

P2. The mechanisms used by MD-SPL approaches to derive prod-

ucts make difficult the maintenance, reuse and evolution of reusable core assets of MD-SPLs.

During the process of deriving products, model transformation rules must be composed in order to derive configured products. The composition is done based on each configuration. Most of the existing MD-SPL approaches maintain the information of relationships between variants and their related transformation rules coupled inside the source code of the transformation rules. This makes difficult to maintain and reuse transformation rules and/or variability models.

Additionally, the abstraction level at which current MD-SPL approaches can (fully) adapt the required transformation rule's composition is too low (*e.g.* using Ant scripts). High-level mechanisms to adapt the execution scheduling of model transformations is then required.

Finally, MD-SPL approaches only provide mechanism to create coarse-grained configurations and derive products based on them. It lacks some mechanism to derive products based on fine-grained configurations.

1.3 Research Objectives

RO1. To provide Model-Based mechanisms for extending the power of expression of variability and to extend the scope of MD-SPLs.

It is our first objective to extend the power of expression of variability in MD-SPL approaches in such a way that new and more detailed products can be configured. We plan to achieve this objective in two stages:

- Introducing a mechanism that allows product line architects to capture and express the possible fine-grained variations between members of a product line.
- Developing a mechanism that allows product designers to create fine-grained configurations which represent *valid* products. We define a valid product as a runnable system that accomplish the requirements that product designer specify by means of configurations.

RO2. To provide a Model-Based mechanism for deriving products that facilitates the maintenance, reuse and evolution of model transformations and variability models in MD-SPL approaches.

We plan to achieve this objective in two stages:

- Developing a strategy to capture separately (1) the model transformation rules used to derive product line members, (2) the variants included in variability models, and (3) the relationships between model transformations and variants.
- Developing a mechanism to compose model transformation rules and adapt their execution ordering according to configurations. This must be a high-level mechanism that facilitates maintenance and evolution of MD-SPLs' core assets.

RO3. To create new tool support for deriving MD-SPLs

Our aim is to create Model-Based tool support which implements facilities to (1) capture fine-grained variations between members of product lines, (2) configure new and more detailed products, and (3) derive fine-grained configured products.

1.4 Approach - in a nutshell

We propose FieSta, an approach to create SPLs based on MDD. FieSta (1) provides mechanisms to extend the power of expression of variability in MD-SPLs by using coordinately metamodeling and feature modeling, and (2) integrates a product derivation process which uses *decision models* and Aspect-Oriented Programming facilitating the reuse, adaptation and composition of model transformation rules. Figure 1.2 presents an activity diagram summarizing the processes involved in FieSta.

During the domain engineering process, product line architects create domain application metamodels, feature models and *constraint models* to capture the variability and commonalities of MD-SPLs.

Metamodels define the common and variable structure of sets of models and serve as a vocabulary that is familiar to the practitioners of a specific ap-

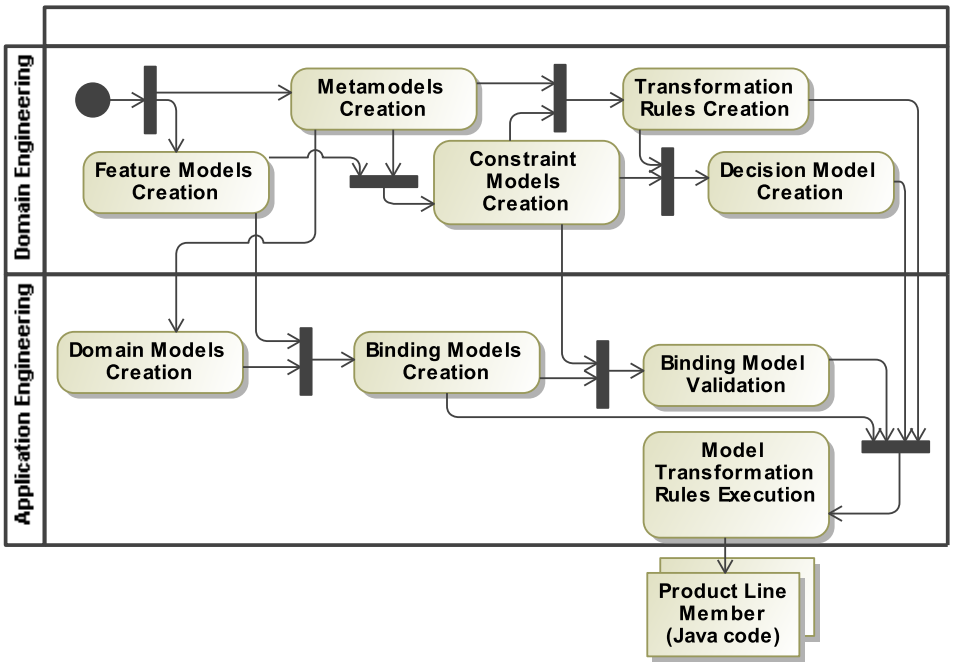

Figure 1.2: General Process.

plication domain. Feature models, which are probably the most well-known and accepted notation for specifying variability of a product line, represent variation points and variants allowing to configure products only by selecting features.

We introduce constraint models which make it possible for product line architects to capture and express the valid fine-grained variations between product line members using the concepts of *constraint*, *cardinality property* and *structural dependency property*. A constraint model is a set of *constraints*. A constraint $C = [M, F, A, D]$ is a tuple composed of a metaconcept M, a feature F, and two properties A and D. A constraint C expresses the fact that during the product configuration process, model elements that conform to the metaconcept M can be bound to the feature F to create product fine-grained configurations.

During the domain engineering process, product line architects create model transformations which consist of sets of transformation rules. Each trans-

formation rule is responsible for producing a part of a final product. Model transformation rules implement algorithms to transform domain application models into refined models (or source code) including concerns from a different abstraction level. Two groups of transformation rules are created: transformation rules to generate commonalities of products, and transformation rules to generate variability of products.

Product line architects also create *decision models*. Decision models are the base of our mechanism to derive products including variability. They capture the execution ordering of transformation rules to be performed by the model transformation engine to derive configured products. We use Aspect-Oriented Programming (AOP) to build the scheduling of the transformations rules, *i.e.* the order in which transformation rules are going to process model elements to accomplish the desired derivation. Thus, we define a decision model as a set of *aspects*. An aspect maintains information of what and when transformation rules that generate commonalities of products must be intercepted (*joinpoints*) during the product derivation process which is driven by a product configuration, and what transformation rules (*advices*) that generate variable structures must be executed instead.

To configure a product during the application engineering process, product designers create (1) domain application models that conform to domain application metamodels, and (2) *binding* models, which are sets of bindings between model elements and features. After a binding model is created, we validate this against a set of OCL-Type sentences derived from its respective constraint model.

To derive a complete product according to a binding model, we dynamically adapt the parameters of model transformation executions. We achieve it using model transformation rules which are selected from the binding model and the pre-created decision models.

1.5 Contributions

C1. A Model-Based mechanism that allows extending the power of expression of variability in MD-SPLs and consequently extending the scope of products that can be fine-grained configured.

We have created a mechanism that allows product line architects to capture the possible fine-grained variations between members of MD-SPLs by creating *constraint models*. Constraint models facilitate to capture constraints that product configurations must satisfy. Our mechanism includes facilities to generate OCL-type expressions from constraint models, and then to validate product configurations against the OCL-type expressions. This work has been presented in [ACR09].

Regarding product configuration, We have created a mechanism that includes a configuration process which allows product designers to create fine-grained configurations of products by means of *binding models*. We present how binding models facilitate staged-configuration of products by binding, in different stages, model elements from domain application models to variants from variability models. We first introduce our mechanism for creating fine-grained configurations in [GPA+07], then we used it in [ACR09, ACR07b, AGGa+08, ACR07a].

C2. A mechanism to derive fine-grained configured products that facilitates the maintenance, reuse and evolution of core assets from MD-SPLs.

We have created a mechanism that allows deriving product by adapting model transformation rules according to binding models and decision models. We introduced this work in [ACR08, ARCR09], and we used it in [ACR09].

C3. Tool support.

We have developed a toolkit, named FieSta Toolkit. This toolkit assists: (1) product line architects during the domain engineering process to create feature models, constraint models and decision model; and, (2) product designers during the application engineering process to create binding models and validate them against constraint models.

In addition, we have added components to the openArchitectureWare framework (oAW) [BBM03] for allowing product derivation based on binding models. Our oAW components are described in [ACR08].

The FieSta Toolkit and the oAW components are available at `http://qualdev.uniandes.edu.co/wikiMain/doku.php?id=projects:md-slp_engineering:toolkit`.

C4. Two case studies of MD-SPLs.

We have created two case studies of SPLs that have been developed by using our mechanisms and tool support. One case study refers to a product line of Smart-Home systems. The other one refers to a product line of stand-alone applications to manage data collections. The case studies, including detailed documentation of metamodels and source code are available at `http://qualdev.uniandes.edu.co/wikiMain/doku.php?id=projects:md-slp_engineering`.

1.6 Thesis Structure

Figure 1.3 presents the structure of this document, which is organized in for parts: Introduction, State of The Art, Proposal and Conclusion. Following this chapter, a background chapter on the Model-Driven Development and a chapter including background and State-of-the-Art on Model-Driven Software Product Line Engineering provide the base for our approach. The next chapter discusses our approach itself. Following, the next chapter presents the results we obtained creating products of MD-SPLs and the tools we developed for supporting the MD-SPL Engineering mechanisms we introduced. The final chapters present the conclusion of this thesis, including a discussion and considering future work. We will now give a detailed description of each chapter.

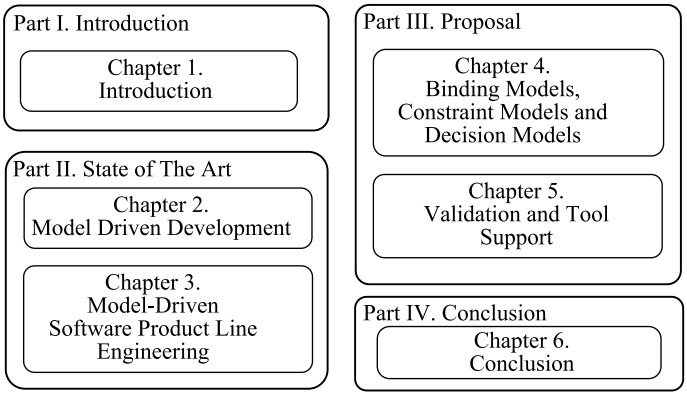

Figure 1.3: Structure of the Document.

Chapter 2: Model-Driven Development. This chapter introduces the main concepts about MDD: model, meta-model and model transformations. Regarding models, we introduce some definitions and we explain the concept of *separation of concerns* of a system in different models. We also discuss the concept of *level of abstraction* of models, and we classify levels of abstraction as a particular case of separation of concerns. We explain the general concepts of metamodeling: Domain Specific Modeling (DSM), the relation of conformity and the four-layer metamodeling framework. We also introduce the Eclipse Modeling Framework (EMF), which is a metamodeling and modeling framework. Finally, we define the concept of model transformations and we classify them into four major categories: model-to-model, model-to-text, horizontal and vertical transformations. We introduce the Xpand and the Xtend model transformation languages, which are languages included in the openArchitectureWare framework.

Chapter 3: Model-Driven Software Product Line Engineering. This chapter introduces the Software Product Line (SPL) Engineering. The major stages in SPL development are discussed: (1) the domain engineering process and (2) the application engineering process. Feature modeling is introduced as a mechanism for expressing product line variability and configuring products. Decision models are included as artifacts used to relate reusable core assets and variants from product lines, and support the product derivation process based on product configurations. The development of SPL based on MDD is the most interesting part for our work. This topic is discussed in detail and different MDD approaches to create SPLs are presented and compared.

Chapter 4: Binding Models, Constraint Models and Decision Models. The previous chapters presented the background for this chapter in which FieSta, our approach to create SPLs based on MDD, is introduced. This chapter starts introducing one case study which is used to illustrate the different axes of our approach. Constraint models, which are reusable artifacts we build to capture the scope of Model-Driven SPLs, are presented and their use is illustrated in the context of our case study. Binding models, which serves to configure products and are sets of bindings between model elements and features that satisfy the constraint models, are explained and also illustrated with our case study. We then show how we derive products based on binding models and decision models, which are sets of aspects we

use to adapt model transformations required to derive configured products. Finally, we present limitations of our approach for deriving products based on decision models.

Chapter 5: Validation and Tool Support.

In this chapter we aim to validate FieSta, our MD-SPL approach, by presenting examples of products that we are able to derive using our MD-SPL mechanisms. We present results of configuring and deriving products of two MD-SPLs. We also present the implementation strategy for FieSta. The implementation strategy defines the general process for the implementation of our MD-SPL engineering mechanisms for creating product lines. Our implementation strategy includes (1) the required activities to create products, and (2) the tools we created to support these activities. We present the tool support for expressing variability and configuring products, and the tool support for deriving configured products.

Chapter 6: Conclusion. This chapter concludes this thesis presenting (1) a summary of our work, (2) a reflection taking into account the contributions we do to the field of Model-Driven Software Product Line Engineering to rich the research objectives we considered, and (3) future research directions.

Part II. State of The Art

Chapter 2

Model-Driven Software Development

2.1 Introduction

The Model Driven Development (MDD) paradigm proposes a framework, using models as first engineering artifacts, to i) define software development methodologies, ii) develop systems at any level of abstraction, and iii) organize and automate testing and validation [FS04]. Thus, MDD conceives the whole software development cycle as a process of creation, iterative refinement and integration of models [GSCK04, SVC06]. During the software lifecycle, stakeholders create models and use model transformations to derive products.

This chapter presents the main concepts involved in the MDD paradigm: models, metamodels and model transformations. Additionally, this chapter introduces some representative modeling frameworks and model transformation languages. These MDD-frameworks and the transformation languages provide specific functionalities to create and process models based on the MDD principles.

2.2 Models and Metamodels

MDD uses models as first-class entities during the whole software development process. There is no a standard definition of what a model is, even in the software engineering field. There is, however, a common consensus between many definitions about one fundamental characteristic: a model is an abstraction of a system and/or its environment. The MDA guide [OMG03] defines a model of a system as follows: "A model of a system is a description or specification of that system and its environment for some certain purpose."

2.2.1 Domain Specific Modeling and Metamodels

Domain-Specific Modeling (DSM) is a way of developing software systems that involves the use of Domain-Specific Modeling Language (DSML) to represent different concerns of an application domain. Such languages tend to support high-level abstractions which are closer to the problem domain than to the implementation domain [MRV08].

Defining a Domain-Specific Modeling Language (DSML) involves at least three aspects: (1) a notation for the construction of models, which is defined by a concrete syntax, (2) a description of the vocabulary (concepts, relationships, and integrity constraints) of the domain concepts, which is defined by an abstract syntax, and (3) the way to use the domain concepts to create well-formed models, which is defined by the semantic domain. The semantic domain is usually defined by means of some mathematical formalism in terms of which the meaning of the models is explained [ESB04]. This can be also defined using OCL expressions.

The standard way to define the abstract syntax of the language is by means of metamodels. A metamodel describes the concepts of the language, the relationships between them, and the structuring rules that constrain the model elements and combinations in order to respect the domain rules [MRV08].

The relation between a model and its reference model is called conformance [Bó5]. Thus, we normally say that a model *conforms* to its metamodel, *i.e.* that a model is written in the language defined by its metamodel. The relation of conformance is neither injective (several model elements may

be associated to the same metaconcept) nor surjective (not all metaconcepts need to be associated to a model element) [Bó5, JB06].

Figure 2.1 presents a sample metamodel for UML class models. This metamodel is expressed as a UML class diagram and includes the abstract metaconcept of `Classifier`, which comprises the concrete metaconcepts `PrimitiveDataType` and `Class`. A `Package` is composed by `classes`, and a `Class` contains `attributes`.

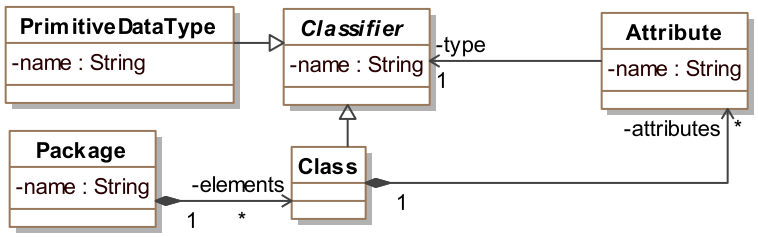

Figure 2.1: Metamodel for Class Models.

Figure 2.2 presents a class model that conforms to the metamodel for class models. The concrete syntax we used to create this model presents model elements as stereotyped boxes. Each stereotype indicates the metaconcepts to which the model element conforms to. Values for element's properties are displayed inside each box. Relationships between model elements are represented by standard class model's arrows (directed-composition or directed-association arrows). Thus, the class model has one package, `School`, containing two classes, `Student` and `Program`. `Student` has two attributes, `studentName` of type `String` and `registeredProgram` of type `Program`. `Program` has one attribute, `programName`, which is of type `String`.

2.2.2 The 4-Level Metamodeling Framework

Since metamodels are also models, they need to be written in another language, which is described by its meta-metamodel. This recursive definition normally ends at that level, since meta-metamodels conform to themselves [Bó5, OMG06b].

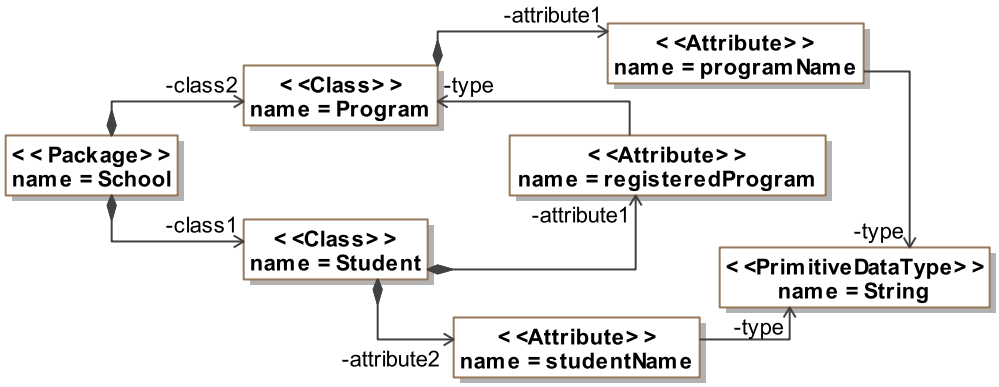

Figure 2.2: Class Model Example.

The OMG has introduced the Meta Object Facility (MOF), a 4-level meta-modeling framework that removes ambiguities from the term meta [OMG06b]. This framework is based on a four-layer metadata architecture used to conceptualization of the relationships between data and descriptions of them. These layers are `System`, `Model`, `Metamodel` and `Meta-metamodel`. The `System` layer comprises the data to describe. The `Model` layer contains metadata that describe the data in the information layer. The `Metamodel` layer is composed of descriptions that define the structure and semantics of metadata. The `Meta-metamodel` layer is composed of the descriptions of the structure and semantics of meta-metadata. Figure 2.3 presents the four-layer metadata architecture.

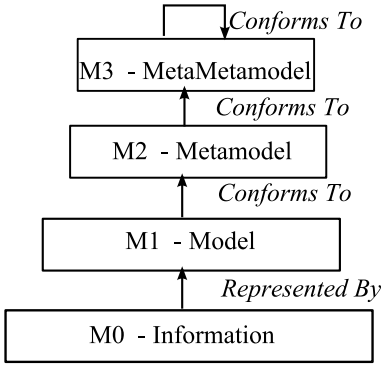

Figure 2.3: The Four-Layer Metadata Architecture.

MOF is the meta-metamodel proposed by the OMG [OMG06b]. As defined in its specification v2.0, MOF provides among others the following four basic meta-metaconcepts for creating metamodels:

- Classes are types. Metaconcepts that conform to Class have identity, state, and behavior. The state of a Class metaconcept is expressed by its Attributes and Constants, and its behavior is governed by Operations and Exceptions.
- Associations describe binary relationships between Classes. They may express composite or non-composite aggregation semantics. MOF associations have no object identity.
- Packages are nestable containers for modularizing and partitioning metamodels into logical subunits. Generally, a non-nested Package contains all of the elements of a metamodel.
- Constraints specify the well-formedness rules that govern valid domain models. MOF provides several features for metamodel composition, extension, and reuse, including Class inheritance, Package inheritance, Class importation, and Package importation.

As part of our work we have used Ecore as meta-metamodel. Ecore is a core subset of the MOF model. Ecore and the Eclipse Modeling Framework (EMF) [BBM03], which is a framework that aims to follow the MOF standard and use Ecore as meta-metamodel, are explained in detail in Section 2.4.

2.2.3 The Nature of Models

An intrinsic characteristic of MDD is the separation of concerns of a software system in different models. In MDD it is possible to create and process simultaneously several models from the same system, regarding different perspectives or point of views of different stakeholders.

The models describing a system can be classified in terms of their *level of abstraction*. The level of abstraction of a model refers to the amount of implementation details that the model has or, in other words, it indicates how close to the problem space the model is. Closer to the problem space, higher the level of abstraction; closer to the solution space, lower the level of abstraction.

For example, stakeholders may create high-level abstraction models which include only domain-specific application details or only concepts regarding the problem. Other stakeholders may create, or interact, with models including details of software design. These models can be considered as medium-level abstraction models. Finally, stakeholders could process models including details of the technological platforms used to implement the system. These models are considered low-level abstraction models. Thus, we conceive software development as a chain of modifications (enhancements) where models of a system are transformed through different levels of abstraction starting at the problem space and finishing at the solution space

The model presented before in Figure 2.2 is an example of a high-level abstraction model including only concepts regarding the problem space. Figure 2.4 presents a lower-level abstraction model. This model includes software design concerns to represent `EJBSession` and `EJBEntity` elements. Thus, this model is closer to the solution space, *i.e.* it includes more implementation details than the model presented in Figure 2.2.

Figure 2.4: Low-Level Abstraction Class Model.

The separation of concerns of a system in different models according to the level of abstraction is only one of the criteria that stakeholders can use to separate models. At each different level of abstraction of a system, different stakeholders may have different points of view of the system. Figure 2.5 presents an example of a high-level class model including an extra property, `isPersistent`, related to `Class` elements. This property allows stakeholders marking the `Class` elements whose data require to be maintained in a data base repository in the final software system.

2.3 Model Transformations

Model transformation appears to be one of the most useful operations on models. Model transformations are software artifacts that implement algorithms to transform models that conform to source metamodels into either models that conform to target metamodels or source code.

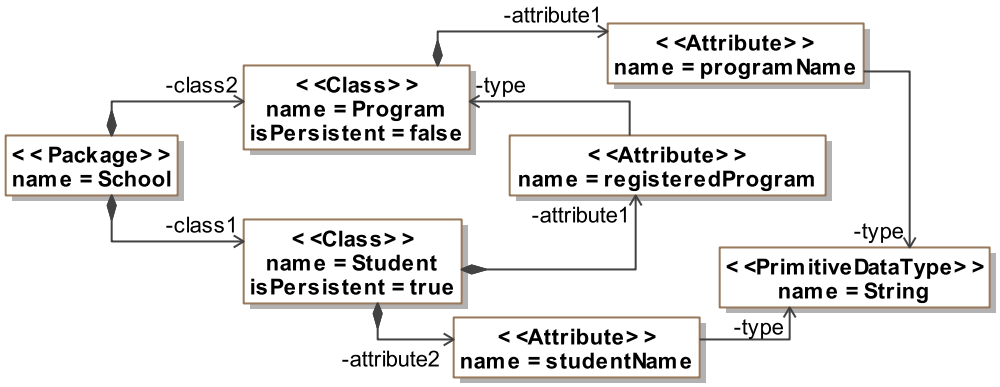

Figure 2.5: Class Model with Persistence's Properties.

Figure 2.6 presents the scenario of a model transformation with one source model and one target model. Note that (1) each model conforms to its respective metamodel and (2) the model transformation refers the source and target metamodels. Metamodels are used in model transformation to navigate models by using *transformation rules*. Transformation rules are considered as functions or procedures implementing some transformation step. They are the smallest units of model transformations [CH06]. Finally, a transformation engine is in charge of executing the model transformation on the source model to derive the target model.

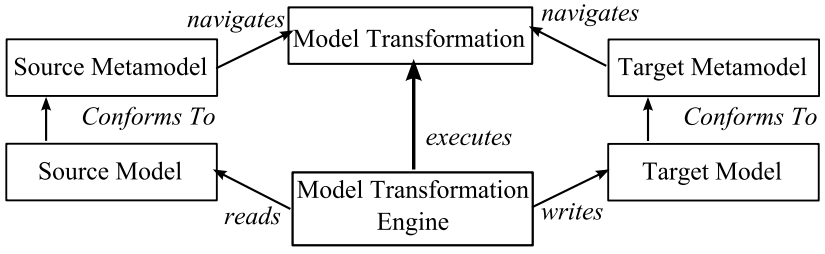

Figure 2.6: Model Transformation Scenario.

2.3.1 Scheduling of Transformation Rules

As said before, transformation rules are the smallest units of model transformations [CH06]. To transform source models into target models several transformation rules are required as well as an execution ordering. Czarnecky and Helsen name *scheduling of transformation rules* the execution ordering of a set of transformation rules [CH06]. Basically the scheduling of transformation rules is a *call graph* in the context of routines to transform models. *A call graph is a directed graph that represents calling relationships between subroutines in a program. Each node represents a procedure and each edge (f,g) indicates that procedure f calls procedure g* [Ryd79].

The manner to describe the scheduling of transformation rules depends of the paradigms followed by the model transformation language chosen to write the transformation rules. Current model transformation languages use well known paradigms for programming languages. The most common paradigms used actually in model transformation languages are the *declarative* and the *imperative* paradigms [JK05].

In declarative programming the logic of a computation is expressed without describing its control flow. Model transformation languages applying declarative programming, *e.g.* ATL [JK05] and Tefkat [LR07], attempt to minimize or eliminate side effects by describing what the program should accomplish, rather than describing how to go about accomplishing it. In imperative programming computations are described in terms of statements that change a program state. Imperative transformation rules define sequences of commands to perform on source models, and require a detailed description of the algorithm to be run and the scheduling of transformation rules. Examples of model transformation languages applying imperative programming are Xtend and Xpand [OAW09b].

We selected an imperative model transformation language in the implementation of the approach we describe in Chapter 4. One of the reasons we had for selecting an imperative model transformation language is that we can have always control on the call graph of transformation rules; thus we can manipulate it when required.

2.3.2 Model Transformation Patterns

Transformation rules are written in terms of the source and target metamodels. It means that models are transformed following transformation patterns defined in terms of metaconcepts of the source and target metamodels. Figure 2.7 presents an example to illustrate this characteristic of model transformation rules. In the example, `Class` elements are transformed into `Table` elements using a transformation rule that is written in terms of the metaconcepts `Class` and `Table`.

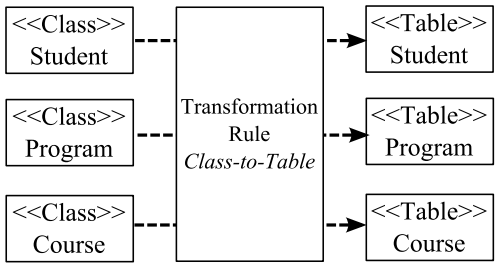

Figure 2.7: Example of a Model Transformation Pattern.

This characteristic of transformation rules implies that several transformation rules must be written when model elements that conform to the same metaconcept must be transformed following several (different) transformation patterns. For example, we can write a transformation rule `ClassToPersistentClass` to transform elements that conform to the `Class` metaconcept from Figure 2.2 into elements that conform to the `Class` metaconcept from Figure 2.5, which has a boolean property `isPersistent`. `ClassToPersistentClass` transforms any source `Class` element following a transformation pattern which creates a target `Class` element with the property `isPersistent` set to *true*. If we need to transform source `Class` elements into target `Class` element with the property `isPersistent` set to *false*, we must create another transformation rule.

In Section 3.6 we present some mechanisms which allows to select the transformation rules that must be executed according to particular requirements of stakeholders. For instance, if a stakeholder needs to create a target `Class` element with the property `isPersistent` set to *true*, the rule `ClassToPersistentClass` is automatically selected. These mechanisms also

include strategies to modify the scheduling of transformation rules and thus to derive several (different) products.

2.3.3 Classification of Model Transformations

It is possible to classify model transformations according to several criteria. Given the particular interest of our work, we focus on two general classifications.

On the one hand, Czarnecky and Helsen have classified model transformations establishing as their major categories model-to-model and model-to-text transformations [CH03, CH06]. The reason for this distinction is that the techniques, languages and tools used for both categories are different. Model-to-model transformations are used to transform models that conform to source metamodels into models that conform to target metamodels. Model-to-text transformations are mostly utilized for transforming low-level abstraction models into source code of a specific programming language, and also for generating low-level artifacts including technology implementation details such as deployment descriptors or configuration files.

On the other hand, France and Bieman categorize model transformations along vertical and horizontal dimensions [FB01]. Vertical transformations occur when a source model is transformed into a target model at a different level of abstraction. A horizontal transformation involves transforming a source model into a target model that is at the same level of abstraction as the source model. The next two subsections extend these explanations and present some examples.

2.3.4 Vertical Model Transformations

Vertical transformations transform models between different abstraction levels. This type of model transformation is classified in *refinement* and *abstraction* transformations [FB01]. Refinement transformations transform models at a higher abstraction level into models at a lower abstraction level, whereas abstraction transformations transform models at a lower abstraction level into models at a higher abstraction level.

Figure 2.8 presents an example of a refinement transformation. On the left, the high-level abstraction model presented before in Figure 2.2 is transformed into the lower-level abstraction model from Figure 2.4. In this example `Package` elements are transformed into `Model` elements and `Class` elements still remain as `Class` elements. One `Controler` element and one `View` element are created from each `Package` element and associated to the corresponding `Model` element. Thus, the target model includes software design concerns to represent a basic Model-View-Controller (MVC) architectural design pattern.

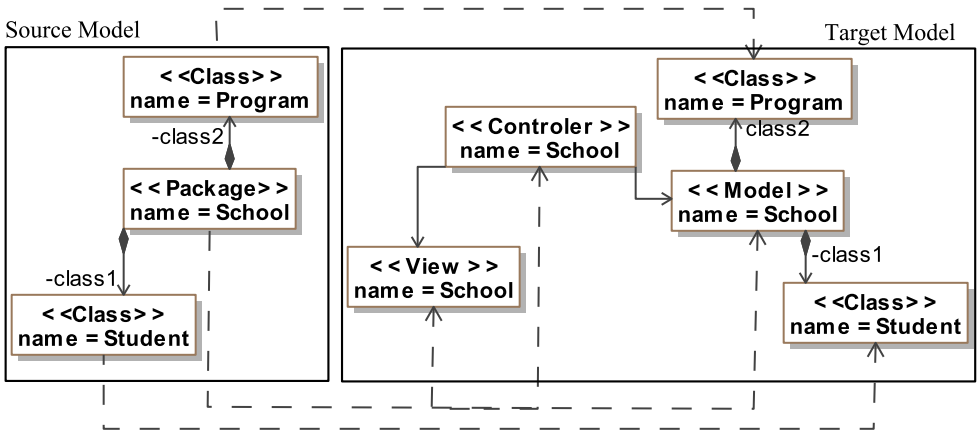

Figure 2.8: Example of Vertical Transformation.

2.3.5 Horizontal Model Transformations

Horizontal transformations relate or integrate models covering different aspects or domains within a system, but at the same level of abstraction. Horizontal transformations are classified in *migration*, *merge* and *identification* transformations [FB01]. Migration transformations transform one model that conforms to a source metamodel into another model that conforms to a target metamodel. The source and target metamodels can be the same metamodel. Merge transformations combine individual models, seen as different views, to form a complete model. Finally, identification transformations create target models including subsets of elements from the source models; for this, a selection filter is used.

As part of the approach we introduce in Section 4, on the one hand we use vertical (refinement) transformations to incrementally add implementation details to high-level abstraction models until to derive software systems. On the other hand, we use horizontal (*migration* and *merge*) transformations for adding to models various concerns from the same abstraction level in different model transformation stages.

2.4 Modeling Frameworks

The Eclipse Modeling Framework (EMF) [BBM03] is the main academic and industrial reference of modeling frameworks. Other modeling frameworks extend the facilities that EMF provides as is the case of the Topcased toolkit [Pttt07]. Through our work, we use the Topcased facility to create model editors. This section introduces EMF and Topcased.

2.4.1 The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [BBM03] is a modeling framework and code generation facility for building tools and other applications based on models. EMF started as an implementation of the Meta Object Facility (MOF) specification and currently it uses Ecore as meta-metamodel, which is a core subset of the MOF model.

EMF offers editing tools for creating and manipulating metamodels that conform to Ecore, and models that conform to such metamodels. This support includes reusable classes for building model editors and code generation capabilities. EMF also offers runtime support for operations with models, including change notification, persistence support with XML Metadata Interchange (XMI) serialization, and a reflective API for manipulating EMF objects.

Figure 2.9 presents a subset of the Ecore meta-model. Ecore prefixes an `"E"` before all its meta-classes. This helps for example to distinguish between Ecore metaconcepts and UML metaconcepts. It also makes a distinction between `EAttribute` and `EReference`. The difference is that the type of an `EAttribute` is always a primitive type, such as String or integer, while the

type of an `EReference` is always an `EClass`. Associated `EReferences` are related to each other using the `eOpposite` property.

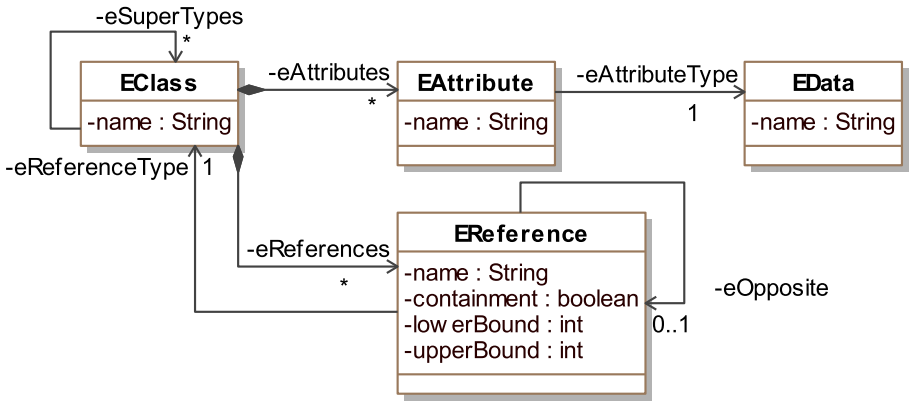

Figure 2.9: The Ecore Meta-Metamodel.

Ecore models, *i.e.* metamodels that conform to the Ecore meta-metamodel, can be defined in at least three ways: creating (1) Java Interfaces, (2) UML-type Class Diagrams and (3) XML Schemas. Once a model is created using one of the three different ways, EMF can generate the others. Figure 2.10 presents the EMF editor to create Ecore models using UML-type Class Diagrams. On the left, a sample Ecore model that correspond to a part of the class metamodel shown in Figure 2.1 is presented. On the right, the "palette" of options to create Ecore models is displayed.

EMF also provides facilities to create models that conform to Ecore models, the syntax used to do it is a general tree structure. A tree structure is a way of representing the hierarchical nature of a model. Figure 2.11 presents an example where the EMF model editor is used to create a class model that conforms to the class metamodel. The root of the tree is a package, `School`. This package contains two classes, `Student` and `Program`, which in turn have one attribute each one.

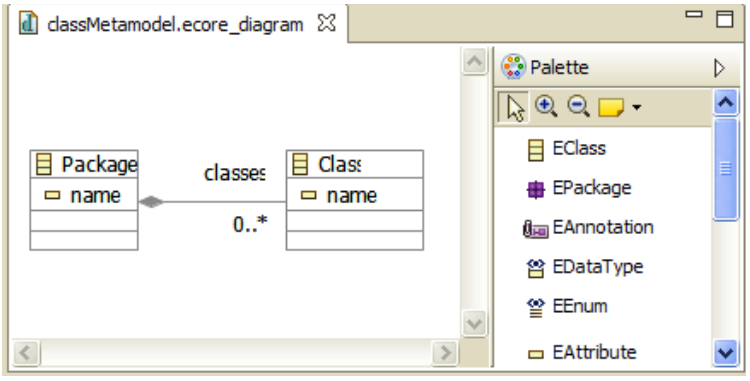

Figure 2.10: Ecore Class Model Example.

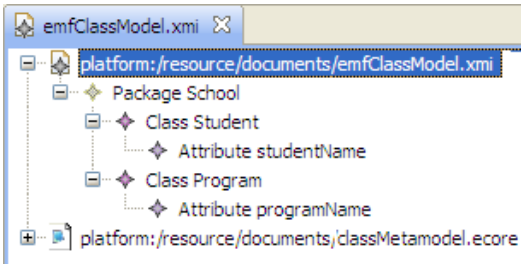

Figure 2.11: EMF Models' Editor.

2.4.2 The Topcased toolkit

The Toolkit In OPen source for Critical Applications and SystEms Development (TOPCASED) [Pttt07] is an integrated model-oriented System/Software engineering toolkit. It covers the stages from requirements analysis to implementation, as well as some transversal activities such as version control, and requirements traceability. Topcased provides model editors, model checkers and model transformations.

Topcased also provides a generative component for developing graphical editors based on Ecore models. Thus, the toolkit allows DSML developers creating and associating concrete syntax to particular metamodels instead of using the general model editor provided by EMF. Figure 2.12 presents an example of a model editor to create class models. On the left, the figure presents the customized palette of options to create models that conform to the class metamodel from Figure 2.10, and, on the right a class model

example.

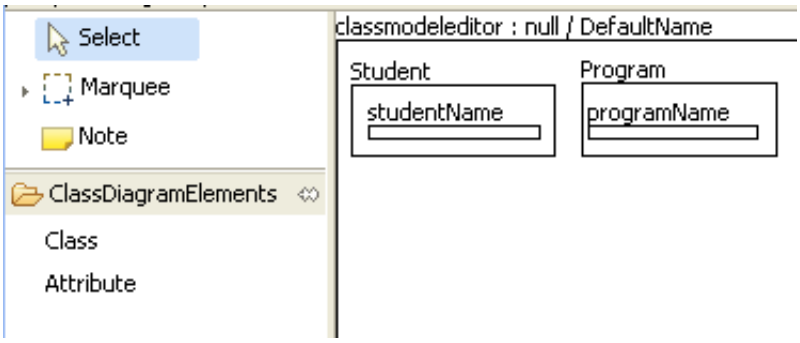

Figure 2.12: Topcased Model Editor Example.

2.5 Model Transformation Languages

OMG proposes MOF-QVT (Query/View/Transformation) [OMG06a] as the standard language for specifying model transformations. QVT exists as an OMG specification, however, even when there are some implementations for the concrete syntax of QVT such as SmartQVT [Tel09], at the moment of this writing there is not an official reference implementation.

There are several implemented MOF-based model-to-model transformation languages such as ATL [JK06] and Kermeta [MFJ05]. Similarly, the openArchitectureWare (oAW) framework [OAW09b] provides a textual language to support the activities of model-to-model transformations, the *Xtend* language, but also a language to support the activities of code generation, the *Xpand* language.

Xtend and Xpand are built up on a common type system and expression language. Therefore, they can operate on models, metamodels and metametamodels by using the same syntax. We have implemented the approach we present in Chapter 4 using and extending oAW. Consequently, we have selected Xtend and Xpand as our transformation languages. In the following subsections we introduce oAW, the type system and the expression language used by the Xtend and Xpand languages, and the Xtend languages it self. Given that the Xpand language use the same type system, the expression

language, and the general facilities that Xtend uses, in this section we do not include a particular description of Xpand. For details please refer to the oAW manual reference [OAW09a].

2.5.1 The openArchitectureWare Framework

openArchitectureWare (oAW) [OAW09b] is an MDD framework integrated into Eclipse. oAW offers facilities to transform models-to-models and models-to-text (source code). At the core of oAW, there is a workflow engine allowing the definition of model transformation *workflows* by sequencing diverse *workflow components*. A workflow component specifies a step in a model transformation chain.

oAW has some pre-built workflow components that facilitate the reading and instantiation of models, checking them for constraint violations, and transforming them into other models or source code. Transformation workflows are built using XML files that describe the steps needed to be executed in a generator run.

oAW provides support for Aspect Oriented Modeling (AOM) and Aspect Oriented Programming (AOP) in the context of MDD. In Section 2.5.2 we will illustrate how AOP is integrated into MDD. This characteristic is specially useful to create SPLs using the MDD principles [Voe05, GSV08, VG07a, VG07b], and it is one of the main reasons why we had selected oAW as the implementation framework for our approach.

The oAW Type System.

In the oAW generator framework every object (*e.g.* metaconcepts, model elements, values, etc) has a type. Every type has a simple name (*e.g.* String) and an optional namespace used to distinguish between two types with the same name. Thus, a fully qualified name looks like this: `my::fully::qualified::typeName`.

The type system provides access to built-in types such as `String`, `Object`, `Collection`, `List` or `Set`. Each type contains properties and operations. For instance, the `String` type has a library which is especially important for code generation. The type system supports the `'+'` operator for concatenation,

the usual `java.lang.String` operations and some special operations like `toFirstUpper()` and `toFirstLower()`.

The type system is also extensible allowing for accessing types corresponding to models or metamodels created by MDD developers. For example, an MDD developer can register in the type system the class metamodel from Figure 2.1 and then having access to the types `Package`, `Class`, and `Attribute`.

The oAW Expression Language.

The oAW expression language is a syntactical mixture of Java and OCL. For instance to access a model element property the following syntax is used: `myModelElement.property`. Respectively, a boolean expression looks like this: `!("textExample".startsWith('t') && ! false)`.

The expression language provides several literals for built-in types, for example, the boolean literals are `true` and `false`. Like OCL, the expression language also defines several special operations on collections such as `select`, `collect`, `reject`, `forAll` and `exist` between others. For instance, the `forAll` operation allows specifying a boolean expression, which must be `true` for all objects in a collection in order for the `forAll` operation to return `true`: `collection.forAll(v | boolean-expression-with-v)`.

The expression language includes conditional expressions (if and switch expressions), expressions to instantiate new objects (create expressions) and expressions to define local variables (let expressions) among others.

2.5.2 The Xtend Language

The Xtend language is a textual and functional transformation language. As said before, Xtend is built up on the common type system and expression language of oAW. Listing 2.1 presents an example of an Xtend file including transformation rules to transform models that conform to the class metamodel from Figure 2.1 into models that conform to a metamodel of relational database schemas for a relational database management system. The relational database schemas' metamodel has two metaconcepts, `Table` and `Column`. A `Table` contains `columns` and both `Table` and `Column` have a `name` property.

```
1  import classMetamodel;
2  import relationalDatabaseMetamodel;
3
4  create Table class2ER (Class myClass):
5    this.setName(myClass.name)->
6    myClass.attributes.createColumn(this)->
7    this;
8
9  create Column createColumn(Attribute myAtt, Table myTable):
10   this.setName(myAtt.name)->
11   myTable.add(this)
12   this;
```

Listing 2.1: Example of an Xtend Model Transformation.

In line 1 and line 2 of Listing 2.1, `import` statements are used to import the name spaces of several types, in this case the types corresponding to meta-concepts of the `classMetamodel` and the `relationalDatabaseMetamodel`. In line 4 a transformation rule appears. This transformation rule receives one `Class` element as parameter, `myClass`, and returns a `Table` element. As soon as this transformation rule starts its execution, a `Table` element is created. In line 5 the `name` property of `myClass` is assigned to the `name` property of the created `Table` element. In line 9 the transformation rule `createColumn(Attribute myAtt, Table myTable)` is called for each attribute of `myClass`. This transformation rule receives an `Attribute` element and a `Table` element, creates a `Column` element from the received attribute, adds it to the collection of attributes of the received `Table` element, and returns the created `Column` element.

In Xtend a function is evaluated only once for each unique combination of parameters. Thus, one can call the same function with the same number of arguments multiple times, and it will only be evaluated the first time. This is an indispensable feature when working with graph transformations, especially, if they contain circular references.

The Xtend language also provides the possibility to define libraries of independent operations and non-invasive metamodel extensions based on either Java methods or oAW expressions. Those libraries can be referenced from all other textual languages that are based on the expressions framework such

```
1  <component class="oaw.xtend.XtendComponent">
2
3    <metaModel class="oaw.type.emf.EmfMetaModel">
4      <metaModelFile value="classMetamodel.ecore"/>
5    </metamodel>
6
7    <metaModel class="oaw.type.emf.EmfMetaModel">
8      <metaModelFile value="erMetamodel.ecore"/>
9    </metaModel>
10
11   <invoke value="my::path::class2ER(sourceModel)"/>
12   <outputSlot value="transformedErModel"/>
13 </component>
```

Listing 2.2: Example of a Workflow Configuration of the Xtend.

as Xpand.

Workflow Components.

To run the oAW model transformation engine, we have to define a workflow. It controls which steps (loading models, checking and transforming them, generating code, etc) the engine executes. To transform models-to-models Xtend can be invoked within a workflow. An example of a workflow configuration of the Xtend component is presented in Listing 2.2. In line 4 and line 8, the source and target metamodels are registered in the execution context. Thus, the types from the metamodels are added to the set of types available in the type system. In line 11 the root transformation rule, `create Table class2ER (Class myClass)` is invoked and its result is left in the `outputSlot` (line 9).

Aspect-Oriented Programming in Xtend.

In the oAW context, aspect orientation is about weaving code into different points inside the call graph of a program. Such points are called *join points*. One specifies on which join points the contributed code should be executed by specifying a *pointcut*, which is a set of join points. Whenever the program execution reaches one of the join points described in the pointcut, a piece of code associated with the pointcut (called advice) is executed.

```
1 around my::path::createColumn(Attribute myAtt, Table myTable):
2   log("Invoking" + ctx.name) -> ctx.proceed();
```

Listing 2.3: Example of an Xtend Advice.

In Xtend the join points are the invocations to transformation rules. Xtend provides a mechanism to define and use *around advices*. Thus, it is possible to reuse available transformation rules changing part of their behaviour without modifying any code.

Listing 2.3 presents an example of an advice which is weaved around every invocation of the transformation rule `createColumn(Attribute myAtt, Table myTable)`. This advice is saved as any other Xtend file with extension `.ext`, for instance `myAdvice.ext`. Note that the parameters of the transformation rule must be also specified in the point cut. Inside the advice (line 2) we call the underlying transformation rule. This is done using the implicit variable `ctx` that provides an operation `proceed()`, which invokes the underlying transformation rule with the original parameters. Thus, the advice adds an entry to the execution log indicating which underlying transformation rule is invoked, and then it invokes the transformation rule.

To weave the defined advice into the selected join points, one need to configure the `XtendComponent` indicating the (fully qualified) name of the Xtend file containing the advice. Listing 2.4 presents an example of such a configuration. Note that in line 13, the workflow definition from Listing 2.2 now includes the name of the Xtend file containing the advice.

2.6 Summary

In this chapter we have introduced the Model Driven Development (MDD) paradigm, which conceives the whole software development cycle as a process of creation, iterative refinement and integration of models. We presented Domain-Specific Modeling (DSM) as a way of developing software systems that involves the use of Domain-Specific Modeling Language (DSML) to represent the different concerns of an application domain. In the context of MDD and DSML we introduce the concept of metamodels, the relation of conformance between models and metamodels, and the MOF 4-level Meta-

```
1  <component class="oaw.xtend.XtendComponent">
2
3    <metaModel class="oaw.type.emf.EmfMetaModel">
4      <metaModelFile value="classMetamodel.ecore"/>
5    </metamodel>
6
7    <metaModel class="oaw.type.emf.EmfMetaModel">
8      <metaModelFile value="erMetamodel.ecore"/>
9    </metaModel>
10
11   <invoke value="my::path::class2ER(sourceModel)"/>
12   <outputSlot value="transformedErModel"/>
13   <value="my::Advices::myAdvice"/>
14
15 </component>
```

Listing 2.4: Example of a Workflow Configuration including Advices.

modeling framework. We have explained how MDD uses model transformations to achieve the transition of models between several levels of abstraction by means of vertical transformations. We have also presented horizontal transformations as the mechanism to transform models at the same level of abstraction but integrating several concerns or point of views of an application domain.

We introduced in particular the Eclipse Modeling Framework (EMF), the Topcased toolkit and the openArchitectureWare (oAW) framework, since the implementation strategy of our approach for creating MD-SPLs builds on these technologies. Along with the presentation of oAW, we introduced Xtend and Xpand, which are the model transformation languages provided by oAW. We also introduced the main characteristics of Xtend including their type system and the facilities for including Aspect Oriented Programming in MDD.

The next chapter presents the Software Product Line (SPL) Engineering, and how MDD is used to support the creation of SPLs.

Chapter 3

Model-Driven Software Product Line Engineering

3.1 Introduction

Software systems are complex and their development is time-consuming and error-prone. The strategy of reusing software artifacts has been seen as a means to alleviate these and other problems associated with software development. Reuse of software artifacts facilitates the composition of products from a set of artifacts already developed and tested, instead of the construction of products from scratch.

Software Product Line Engineering is a paradigm that provides a means to incorporate the reuse strategy as a central part of software development [CNN01, Bos00]. A Software Product Line (SPL) is a set of software products that share many common properties to be built from a common set of assets [CE00]. Approaches to create SPLs have emerged based on MDD, *e.g.* [VG07b, Wag05]. These are called MDD-based SPL approaches or *MD-SPL approaches*. MD-SPLs are developed from domain application models which conform to domain application metamodels using reusable model transformation rules. Two international events have been recently created focus on MD-SPL approaches [Mez09, Goe09].

In this chapter we first introduce the basis of SPL Engineering, including the

main processes involved in the creation of SPLs: *domain engineering process*, Section 3.4, and *application engineering process*, Section 3.5. Then, we introduce the MD-SPL Engineering paradigm and we present a State-of-the-Art of it, Section 3.6. At the end of Section 3.6, we present a discussion emphasizing on the advantages and drawbacks of representative MD-SPL approaches with respect to two aspects. The first one is related to the mechanisms the approaches use for expressing variability and configuring products. The second one is related to the core assets development and the mechanisms for deriving products. These two aspects deserve our attention given that they are at the core of the research problems exposed in this thesis.

3.2 Software Product Line Engineering

The Software Engineering Institute (SEI) [Car09], which has been the most important promoter of the paradigm, provides the following definition of what an SPL is: *"A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"* [CNN01]. This definition uses the term core asset that are reusable artifacts considered as building blocks in SPL Engineering; these reusable artifacts can be models, common components, documentation, requirements, test cases and so on.

In order to obtain benefits from the creation of reusable common assets, it is important to be able to derive from the assets many products. In SPL Engineering, the description of the set of products which are part of an SPL is called the *scope of the product line*. To achieve a profitable SPL, its scope must be neither very large nor very small. If the scope is very large, then the core assets will loose their ability to satisfy the variability, economies of product derivation will be lost, and the product line will fall down into the traditional style of one-product-per-time. If the scope is very small, then the core assets might not be built in a generic enough way, and the return on investment will never be achieved [Cle02, CNN01].

To capture the scope of SPLs, product line architects determine the commonalities, *i.e.* the characteristics that all products in a product line share, and the ways in which they can vary (variability). The management of variability

is the most important activity in SPL development. Variability management is a transversal activity performed during the whole product line development cycle.

3.3 Variability Management in SPL Engineering

Variability management in SPL Engineering is the set of activities related to the identification, expression, and binding of common and variable features included in the scope of product lines. The management of variability is of primary importance for product line development. The effectiveness of a product line approach depends on how well it manages the variability throughout the development life cycle, from early analysis to final derivation of products [SVC06]. The management of variability in SPLs is the most general and important topic concerning our work and it is at the core of the approach we present in Chapter 4.

Different definitions related to variability management can be found in the literature. Here we list two of them which refer *variability* and *variability management*.

Variability is the ability of a software system or artifact to be changed, customized or configured for use in a particular context [vGB02].

Variability management encompasses the activities of explicitly representing variability in software artifacts throughout the lifecycle, managing dependencies among different variability, and supporting the instantiations of the variability [SJ04].

Pohl et al. [PBvdL05] define variability management as the set of activities for defining and exploiting variability throughout the SPL development lifecycle. The concept covers the following issues: (1) supporting activities concerned with variability and commonality analysis which includes identification and documentation of variability, and (2) supporting activities concerned with variability binding and variability realization which includes configuration of product line members and derivation of these products.

Typically, (1) the variability and commonality analysis is performed during the *domain engineering process*; (2) the variability binding and variability realization is performed during the *application engineering process* [CE00, PBvdL05, WL99, vdL02]. Figure 3.1 summarizes the main four activities involved in the two SPL engineering processes. Next section explains these activities.

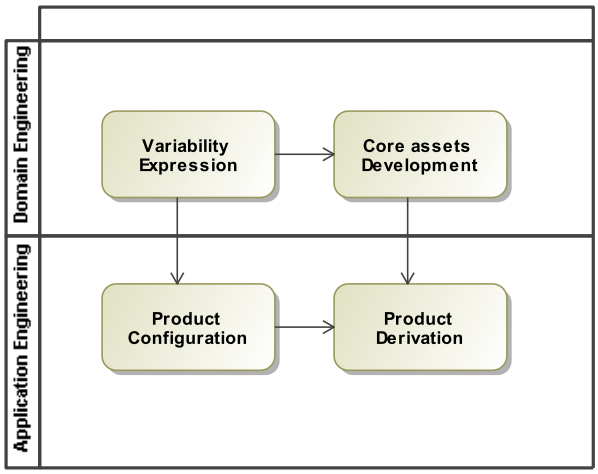

Figure 3.1: The Processes of Domain and Application Engineering.

3.4 The Domain Engineering Process

Domain engineering is the process of SPL Engineering in which the commonality and the variability of the product line are defined [PBvdL05]. The development of an SPL starts with the analysis and modeling of common and variable features of the product line. First, (i) variability is identified, classified and documented. Second, (ii) reusable core assets are built to fulfill the identified and classified variations.

3.4.1 Expressing Variability

There are large number of methods for classifying and documenting variability in software product lines [CBA09]. Several approaches for classifying

and documenting variability center their attention on the use of variability models [SD07, Bay06]. The main concepts regarding variability models are *variation point* and *variant*. Variation points are relevant characteristics that can have different values or variants according to the variability of a product line.

At the present, there is no a standard way to represent variation point and variants in variability models. However, one of the most used methods to represent variation point and variants is by means of feature models. Feature modeling deserves special attention for our work.

Feature Modeling.

Feature modeling is a method and notation for capturing commonalities and variability in product lines [KCH+90, KKL+98, RBSP02, CHE05, VG07b]. Features describe the common and variable functionality of a system under development. The feature modeling approach eases the construction of a hierarchical decomposition of features into a tree structure which represents variation points and variants. As said before, a variation point is a relevant characteristic of a system, for example the operative system under which a system can run. A variation point can have different values or variants according to the variability of a product line, for instance, variants of the operative system variation point can be Linux and MS-Windows.

Feature modeling was first introduced by Kang et al. as Feature-Oriented Domain Analysis (FODA) [KCH+90]. FODA is described as a domain analysis method for identifying prominent and distinctive features of a set of systems in a specific domain. In FODA the features are used to define a specific domain in terms of their *mandatory*, *optional*, or *alternative* characteristics. After Kang et al. other authors extended the concepts regarding feature modeling. Among these extensions there are the concepts of feature cardinality [CE00], groups and group cardinality [RBSP02], and attributes for features [CBUE02] between others. The purpose of these extensions is to restrict the set of variants that can be selected from feature models to create particular configurations.

One of the most cited works on feature modeling is the presented by Czarnecki et al. [CHE04], where the authors propose a *cardinality-based* notation for feature modeling including *solitary*, *group* and *grouped* features. This approach integrates a number of existing extensions of previous approaches. Figure 3.2

presents an example to illustrate the concepts introduced by Czarnecki et al. by using a feature model of an operating system security profile [CHE04].

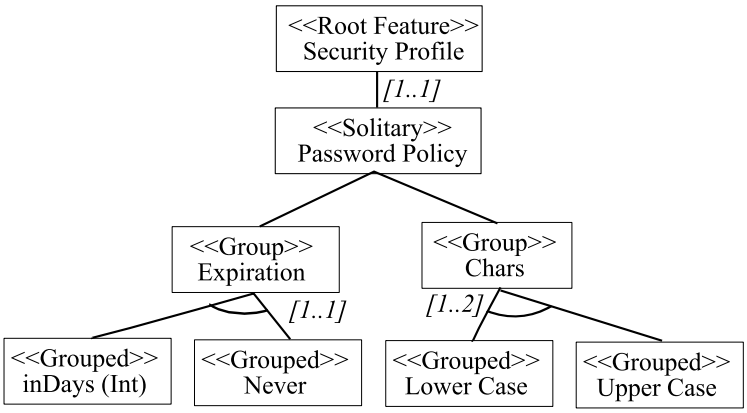

Figure 3.2: Feature Model Example.

The `Password Policy` of the `Security Profile` has associated a policy to manage the password `Expiration` date. For the `Password Policy` a `solitary` feature has been created. In this case, the `solitary` feature has associated the cardinality `[1..1]`, which means that one and only one `Password Policy` can be defined for a particular system under development. For the `Expiration` date a `group` feature is created. A `group` feature has a set of `grouped` features. In this example the `Expiration` date has two `grouped` features, `inDays` and `Never`. Thus, passwords can be set to expire after a given number of days, or never expire. The number of days a password remains valid can be set in an integer attribute associated to the `inDays` feature. The constraints on the number of policies for the `Expiration` date are captured in the cardinality associated to the `group` feature. In this case the `Expiration` date has the cardinality `[1..1]`, which means that one and only one policy for expiration date can be selected.

The feature model also takes into account the possible requirements on the characters to be used in a password. The constraints on characters required in a password are specified by a `group` feature, `Chars`, with cardinality `[1..2]`. This means that any actual password policy must specify between one and two requirements on characters (`Chars`) in a password, `Upper Case` and/or `Lower Case`.

Figure 3.3 presents the Czarnecki et al.'s feature metamodel [CHE04].

`FeatureGroup` expresses a choice over the set of `GroupedFeatures` in the group and its `groupCardinality` defines the restriction on the number of choices. A `GroupedFeature` does not have cardinality and a `SolitaryFeature` is a feature that is not grouped by any `FeatureGroup`. The cardinality of a `SolitaryFeature` specifies the maximum number of times this feature can appear in a final feature configuration. Thus for example, if a `SolitaryFeature` has cardinality `[1..2]`, this feature can appear between one and two times in a feature configuration. The process of creating several features in feature configurations from one `SolitaryFeature` is called *cloning*, and the features created from `SolitaryFeature` in a feature configuration are called *clones*. Finally, features may have `Attributes` of different type and references (`FDReference`) to other features. The values for the attributes related to clones can be different for each clone.

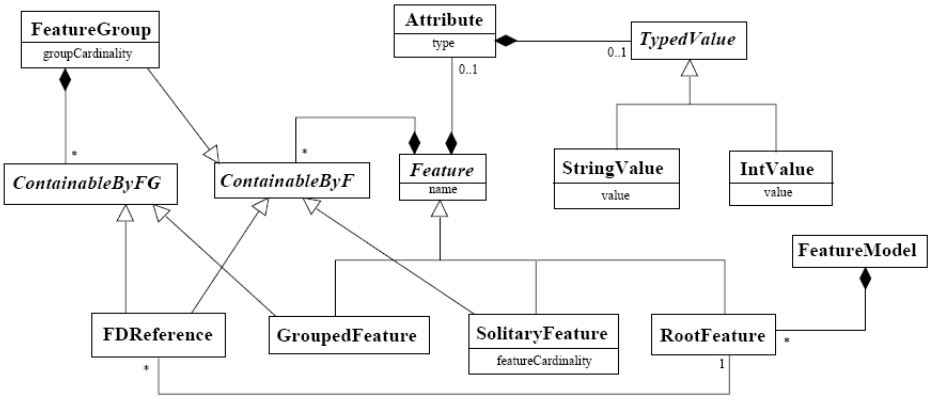

Figure 3.3: The Czarnecki et al.'s Feature Metamodel [CHE04].

Staged Expression of Variability.

Currently in SPL Engineering there is a trend to separate in several variability models different concerns involved in a product line. For example, it can be desirable to create a variability model including software design's concerns separately from technological platform's concerns. This separation of concerns facilitates to product line architects focusing on particular concerns at different times. Similarly, when products are configured, staged feature configurations could be created by different groups of product designers focusing on particular concerns.

To facilitate the separation of concerns, and then the staged configuration of products, separated feature models can be created. Czarnecki et al. motivate the concept of staged configuration and stepwise specialization of feature models [CHE05]. They propose to create separated feature models but maintaining relationships between them. Thus, they avoid a breakdown between the different concerns of individual feature models. For instance, they create a feature model including software architectural concerns and another feature model including technological platform concerns. They maintain relationships between these feature models. For example, one relationship indicates that only if the feature *ObservablePattern* of the architectural feature model appears in a feature configuration, then the feature *OSGi-PeriodicComponent* of the platform feature model will be available to be selected.

In Chapter 4 we present how we have introduced the concept of staged expression of variability and staged configuration of products in our MD-SPL approach.

3.4.2 Core Assets Development

Core assets are reusable artifacts considered as building blocks in SPL Engineering; these artifacts include reusable common models, components, documentation, requirements and test cases among others. Product line architects create core assets according to the variants identified and documented during the activity of variability expression. For instance, for the `Expiration` feature from Figure 3.2, a product line architect creates two (different) software components, one for each grouped feature of the group, `inDays` and `Never`. The first software component has services for checking that passwords are changed each defined number of days, and for supporting the requirement of changing a password. This software component is created for the `inDays` feature. The second software component, created for the `Never` feature, only has one service to inform that passwords cannot be changed.

In practice there is a significant gap between variability at a conceptual level (variation points and variants) and variability at the implementation level (concrete core assets). *Decision models* [ABM00, BFG00, FMP08, DGR08] intend to close that gap.

A decision model is defined as a model that captures variability in a prod-

Table 3.1: Example of a Textual Decision Model

Decision	Resolution	Effect
What policy for Password expiration Will be used?	Password will expire in a determinated number of days	The PasswordExpire component is deployed with the rest of the common components.
	Password will never expire.	The PasswordNeverExpire component is deployed with the rest of the common components.

uct line in terms of *open decisions* and *possible resolutions* [BFG00]. Each decision is expressed in terms of a selected variation point and associated to a set of possible resolutions, which in turn refer to variants of selected variation points. A set of *effects* is associated to each possible resolution. An effect indicates how a particular core asset is reused to create a product line member.

Decision model instances, also called *resolution models*, are created at configuration time of products. In resolution models all decisions must be resolved. As resolutions are related to variants and effects on particular core assets, a resolution model defines a product line member including (1) a subset of chosen variants, (2) the core assets required to derive the desired product, and (3) the adaptation that must be performed on the core assets to obtain a product line member.

Table 3.1 presents a decision model example to create an SPL which includes variants of the security profile from Figure 3.2. This decision model includes only one decision expressed in terms of the variation point `Expiration` date, which has been created as a `FeatureGroup`. This decision is associated with two possible resolutions, which in turn refer to variants from the `Expiration` date variation point, `inDays` and `Never`. One effect is associated to each resolution. Each effect indicates what software components must be deployed in case of selecting each particular resolution. Thus for instance, if a resolution model is created including the resolution `"Passwords will expire in a determinated number of days"`, then the `PasswordExpire` component is deployed with the rest of the common components.

Even when decision models help in the process of creating SPLs, there are still several remaining problems regarding the gap between variability at a

conceptual level and variability at the implementation level. These problems have special importance for us when they are taken into the field of MD-SPL approaches. We go deeply into these particular problems in Section 3.6.

3.5 The Application Engineering Process

Pohl et al. define application engineering as *"the process of SPL Engineering in which product line members are built by reusing core assets and exploiting the product line variability"* [PBvdL05]. During this process, product designers use the variability identified and the core assets created during domain engineering to ensure the correct derivation of desired products.

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 core assets (product derivation).

3.5.1 Product Configuration

In SPL Engineering during the product configuration activity, product designers are responsible for configuring particular product line members by choosing sets of valid combinations of variants identified at the domain engineering process.

When product designers select variants to appear in a particular *product configuration* is called *binding time of the variability* [BGJ+03, BFG+02, PBvdL05]. Some authors have identified the advantages of deciding very late on the binding time, and thus making the binding time variable [vO02, CHE05, AMS07]. The advantage of postponing the binding time is that decisions, *i.e.* design or technological decisions, may be open until very late in the configuration and derivation processes. This adds flexibility to the product line and decouples platform decisions from design decisions or functional requirements.

Product *configurators* are artifacts defined to support the creation of product configurations. The basic functionality of a configurator is to facilitate to product designers the creation of valid configurations from given

variability models. According to Asikainen et al., *"a configurator must make deductions based on the requirements the product designer has entered so far, and prevents or discourages the designer from making incompatible choices"* [AMS07].

Different configurators have been proposed to support configuration of products at different stages of the product configuration activity, *e.g.* [AMS07, AC04, Wag05, PM06]. One example of a product configurator using feature models is the *FeaturePlugin* [AC04]. The *FeaturePlugin* is a feature modeling plug-in for Eclipse. The tool supports configuration based on feature models that conform to the Czarnecki et al.'s feature metamodel from Figure 3.3. This configurator implements *cardinality-based* feature modeling, which includes feature and group cardinalities, and feature attributes. In Chapter 5 we present the product configurator we created to support our MD-SPL approach.

3.5.2 Product Derivation

Product derivation is the activity related to the manual or automated construction of product line members from the available assets. The requirement specifications of products, which are captured in product configurations, are the main input for the product derivation activity. Therefore, to derive products, it is necessary to *adapt* and *assemble* core assets according to the variants chosen from the variability models, and captured in product configurations.

As introduced in Section 3.4.2, there is a significant gap between the conceptual representation of variation points and variants, and the concrete assets that must be created to implement such variants. *Decision models* intend to close the gap capturing variability in terms of *open decisions* and *possible resolutions*.

Resolution models are created at configuration time of products to resolve all the decisions in decision models. A resolution model defines a product line member including (1) a subset of chosen variants, (2) the core assets required to derive the desired product, and (3) the required adaptation and assembly of such core assets. In practice, however, the actual adaptation and assembly of core assets still remains as an open issue. Some of the questions still open

are: how to derive a product when variants are scattered through several core assets? how to derive a product when variants have dependencies between them and core assets were created without taking it into account?

Several authors have introduced approaches to derive products based on product configurations and their particular decision models, *e.g.* [FSJ99, AMS07, ABM00, MO04]. Some approaches used to adapt and assemble core assets are based on traditional mechanisms such as *polymorphism*, *inheritance*, *interface definitions* or *directive's compilation*, *e.g.* [FSJ99]. Other approaches use Aspect Oriented Programming (AOP) as their main mechanism to adapt the call graph of a program [MO04]. In Chapter 4 we present how we deal with the problem of adapting and assembling core assets in the field of MD-SPL approaches, and how we use proven mechanisms such as AOP.

3.6 Model-Driven Software Product Lines

MDD-based SPLs, or *MD-SPLs* for short, are product lines which are created based on MDD principles (see Chapter 2).

A product line member of an MD-SPL is created from a domain application model which (1) conforms to a domain application metamodel and (2) is transformed until to obtain the application by using model-to-model and model-to-text transformations. There is no reference framework for creating MD-SPLs. For many in the domain (*e.g.* [VG07b]), including us [ARCR09, ACR09], these model transformations may require several stages and may include horizontal and vertical transformations. At each transformation stage, domain application models are automatically transformed to include new concerns from a particular abstraction level or more implementation details from lower abstraction levels.

Several approaches to create SPLs have emerged that are based on MDD. In this section we discuss four of the most *representative* works presented in the area. These approaches are: Czarnecki and Antkiewicz's approach [CA05], Wagelaar's approach [Wag05, Wag08b, Wag08a], Loughran et al.'s approach [LSGF, SLFG08], and, Voelter and Groher's approach [VG07b].

We have chosen to present each work following two aspects, see Figure 3.4.

The first one, located at the *problem space* [CE00], is related to the mechanisms the approaches use for expressing variability and configuring products. The second one, located at the *solution space* [CE00], is related to the core assets development and the mechanisms for deriving products. These two aspects deserve our attention given that they are at the core of the research problems exposed in this thesis, which we aim to resolve with our proposal (Chapter 4).

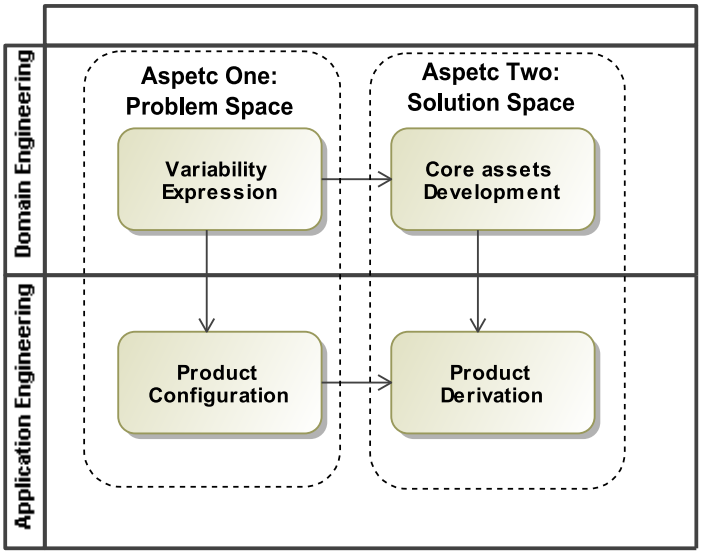

Figure 3.4: .

At the end of this section, we present a discussion emphasizing on the advantages and drawbacks of the different mechanisms used by the presented approaches. Therefore we will remark by comparison where our work presents a contribution to the domain.

3.6.1 The Czarnecki and Antkiewicz's Approach [CA05]

Problem Space: Expressing Variability and Configuring Products.

To express variability Czarnecki and Antkiewicz propose an approach where variation points and variants are captured by means of feature models. They extend the FODA approach by adding cardinality and attributes for features

between others, see Section 3.4.1. Products are configured by creating feature configurations.

Solution Space: Core Assets Development and Products Derivation.

The main core assets built by product line architects to derive products in Czarnecki and Antkiewicz's approach are *template models* and model transformations.

Template models are expressed using UML and represent all the possible elements required to create product line members. For example, to represent a family of UML 2.0 activity models, both the model template and the template instances are expressed using the UML 2.0 activity modeling notation. A template model is a superimposition of all the possible model elements required to derive diverse products according to feature configurations.

Template models are annotated by product line architects using *presence conditions* and *meta-expressions*. The annotations are defined in terms of features from a feature model which capture the variability of the product line under development. Presence conditions indicate whether an element should be present in or removed from a template instance because of the presence of a particular feature in feature configurations. Meta-expressions indicate how to compute attributes of model elements, such as the name of an element or the return type of an operation, based on values assigned to feature attributes in feature configurations.

Product line architects also create model-to-model transformations to instantiate automatically the template models and thus to derive configured product line members. In these model-to-model transformations both the input and output models conform to the UML 2.0 metamodel. Several model transformations are created, each one is in charge of removing elements from the template model and/or compute attributes of model elements according to the annotations in the template model. Thus, based on a feature configuration, a template model can be instantiated automatically by using the model transformations.

Decision models are not explicitly created to support the product derivation process. The *resolution* of variability is performed by product designers creating feature configurations. However, the *effects* on UML models are

specified in the model annotations. This produces a high coupling between the core assets and the required *effects* to create products.

Thus, products are derived from UML models executing the created model transformations. The execution order of the set of model transformations is pre-defined by product line architects. To assure the consistency of the created template instances after the model transformations are executed, the Czarnecki and Antkiewicz's approach proposes two additional processing steps: *patch application* and *simplification*. A patch is a transformation that automatically fixes a problem which may result from removing elements. It is defined for situations in which there exists a unique and intuitive solution to a problem created by element removal. Simplification involves removing elements that have become redundant after removing other elements.

Figure 3.5 [CA05] presents an example of a UML class diagram with annotations. In this example some of the annotations indicate the following: the class `Category` is present in a template instance if the feature `Categories` appears in a feature configuration, a containment hierarchy for `Category` is present if the feature `MultiLevel` is selected, the class `Asset` is present in a template instance if the feature `AssociatedAsset` is chosen, the feature `PhysicalGoods` implies the attribute `weight` in the class `Product`, and so on.

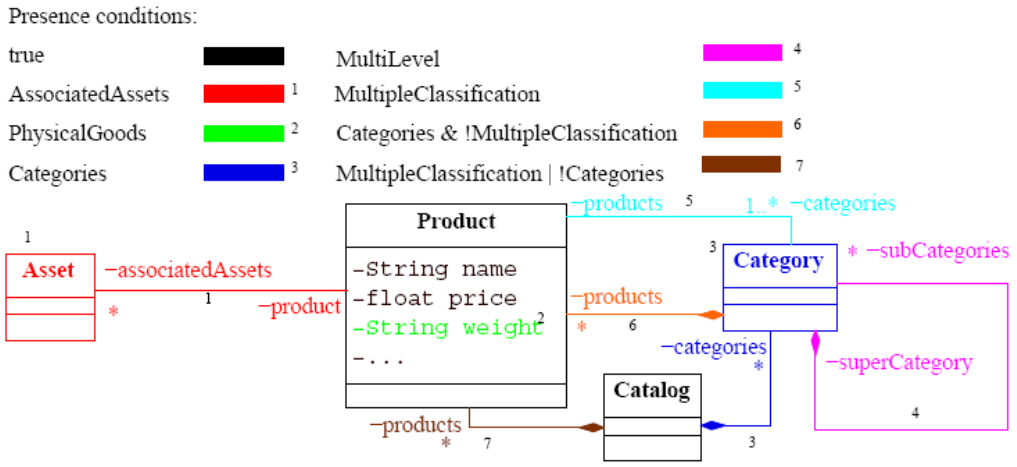

Figure 3.5: Example of a UML Class Diagram with Annotations [CA05].

3.6.2 The Wagelaar's Approach [Wag05, Wag08b, Wag08a]

Problem Space: Expressing Variability and Configuring Products.

The Wagelaar's approach focuses on variability related to technological platforms. The author proposes an explicit *platform model*, which serves as a vocabulary for describing technological platforms. The platform model is expressed using the Web Ontology Language (OWL) [Mic04].

Ontologies are commonly used to represent domain knowledge and to provide a controlled vocabulary in specific domains. OWL supports the necessary concepts of a general ontology language such as *classes*, *properties*, *individuals* and *relationships* between these individuals. In OWL domain concepts are generally represented as simple named classes, which can have subclasses. Class members or instances are called individuals. Properties allow us to assert general facts about members of classes and specific facts about individuals. A property is a binary relationship. Two types of properties are distinguished, *datatype* and *object* properties. Datatype properties describe relations between instances of classes and primitive data types. Object properties describe relations between instances of two classes.

To capture variation points and variants regarding particular technological platforms, the author creates instances of the platform model, or *platform instances* for short. Each platform instance is composed by a set of class members or OWL individuals of the ontology representing the platform model. Figure 3.6 [Wag08b] presents an example of a platform instance for describing Java runtime environments. The `JavaPackageManager` is a class member of the class `platform:PackageManager`, which is a class from the platform model. This class member, or individual, represents a variation point with three possible variants, `JavaWebApplet`, `JavaWebStart` and `JavaMIDlet`. Thus, a product line architect may create different platform instances for different technological platforms.

The author proposes creating *configuration metamodels* as a means to complement the expression of variability having into account concerns different from technological platforms. Figure 3.7 [Wag08b] presents a metamodel capturing possible variations of an instant messengers' SPL. In the figure, the `UserInterface` metaconcept represents a variation point with three vari-

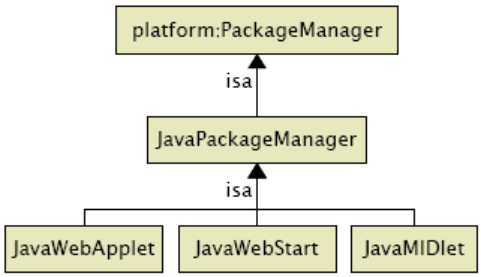

Figure 3.6: Example of a Platform Instance for Describing Java Runtime Environments [Wag08b].

ants, `AWTUserInterface`, `SwingUserInterface` and `LCDUIUserInterface`. The `Packaging` metaconcept represents a variation point with three variants, `WebAppletPackaging`, `IpkgPackaging` and `MIDletPackaging`. The `JabberTransport` metaconcept represents a variation point with two variants, `DefaultJabberTransport` and `MEJabberTransport`. Therefore, a product designer could, for instance, configure an instant messenger with a `SwingUserInterface`, while also s/he selects the `WebAppletPackaging` as packaging method and the `DefaultJabberTrasport` as selected jabber transporter.

The approach suggests extending configuration metamodels with annotations based on platform instances. This linking between metaconcepts and technological platform constraints allows imposing certain platform dependency constraints to the choices provided by the configuration metamodel. Products are configured by creating configuration models. Thus, whenever a model element is included in a configuration model, the platform dependency constraints related to the metaconcept to which such a model element conforms, apply.

Solution Space: Core Assets Development and Products Derivation.

Similarly as in the Czarnecki and Antkiewicz's approach, product line members are derived from UML class models which are created as *templates*. Each template model is created for a group of variants included in a configuration metamodel. A template model represents a superimposition of all the possible classes, properties and operations required to include their respective

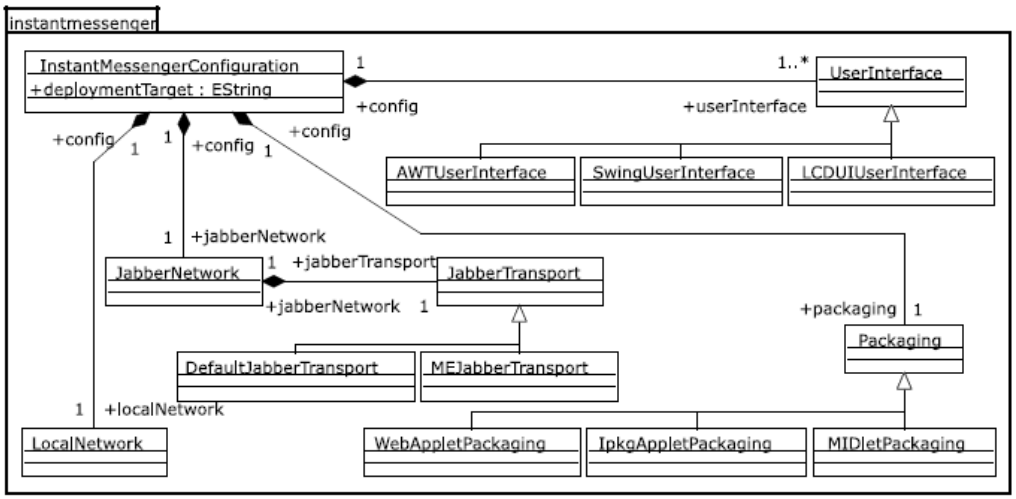

Figure 3.7: Example of a Configuration Metamodel in the Wagelaar's Approach [Wag08b].

variants in a final product. Figure 3.8 [Wag08b] presents an example of a template model. This template model is created for the `JabberTransport` variation point from Figure 3.7. Then, a template instance is derived from this template model according to the variant selected for a product designer: `DefaultJabberTransport` or `MEJabberTransport`. Some of the class elements, their properties and operations are annotated. These annotations are used during the process of transforming the template models into final products.

Product line architects create several groups of model transformations to derive products from template models. Each group is in charge of transforming one template model into a part of a final product that runs on a particular technological platform. Thus, when a product designer creates a configuration model and selects a target technological platform, the template models related to the selected variants are transformed using the respective group of model transformations created for the selected target technological platform.

Decision models are not explicitly created to support the product derivation process. The *resolution* of variability is performed by product designers creating configuration models and selecting a target technological platform. The *effects* on template models, which are used as starting core assets to

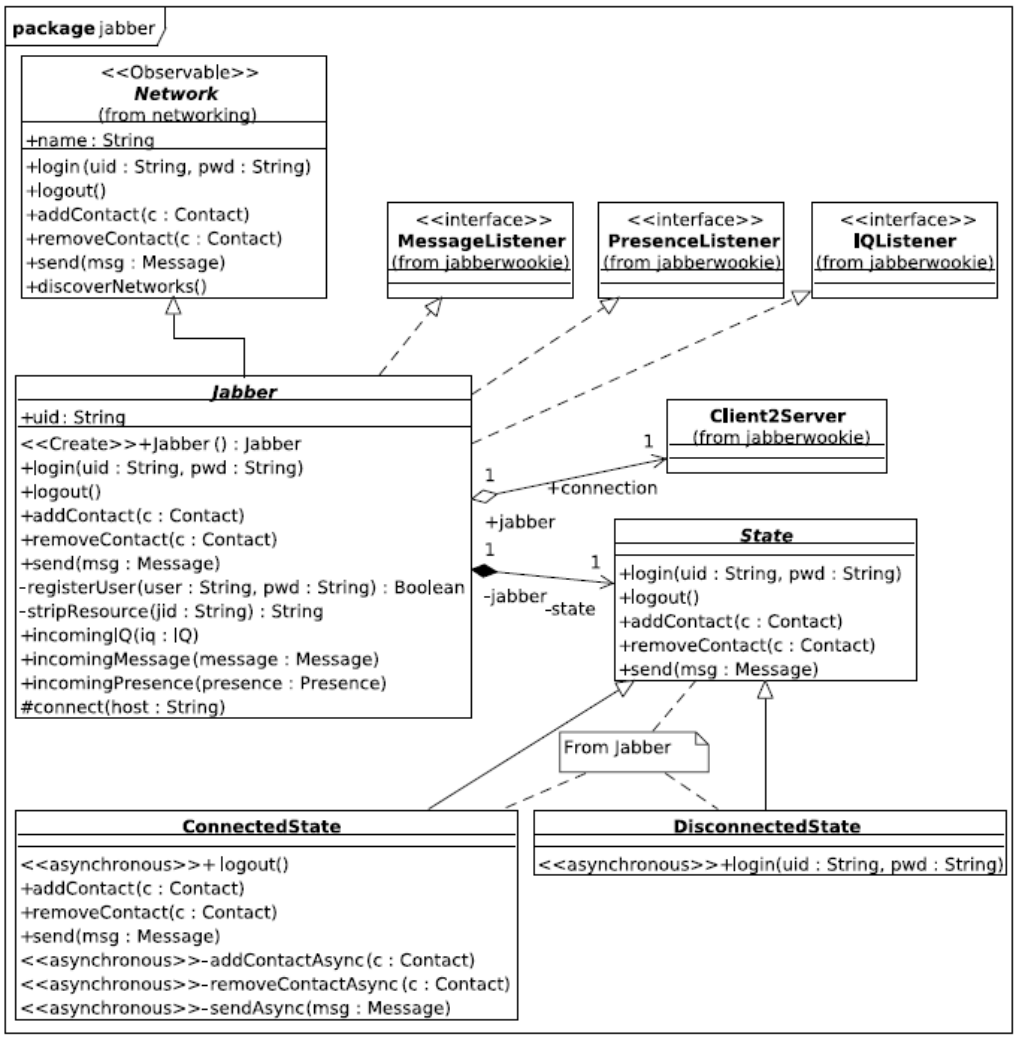

Figure 3.8: Example of a Template Model in the Wagelaar's Approach [Wag08b].

derive products, are specified in the model transformations. Therefore, the *effects* must be expressed in terms of the model transformations: what model transformations must be used?, and, what is the execution ordering required to include selected variants?

The selection of the groups of model transformations to be used is defined from the selected variants, *i.e.* the model elements included in the configuration models, and the selected target technological platform. The execution ordering of the model transformations is predefined by creating a type of *abstract* execution ordering. The *abstract* execution ordering defines the required sequence of calls to *abstract* transformation rules. The *concrete* transformation rules are executed once the groups of model transformations to be used are defined from the selected variants and the selected target technological platform.

To replace the abstract transformation rules by the concrete transformation rules at execution time of the model transformations, the authors propose a composition technique that they call *module superimposition*. To apply this technique, transformation rules must be grouped in *modules*. This technique allows modifying an execution ordering, which include transformation rules from a module "m-1", overriding it to include: (1) new calls to transformation rules from a module different to "m-1", and (2) calls to transformation rules with the same names and the same parameters that the included in the module "m-1", but from a module different to "m-1". This mechanism has been implemented to work on the ATLAS Transformation Language (ATL) [JK05].

3.6.3 The Loughran et al.'s Approach [LSGF, SLFG08]

Problem Space: Expressing Variability and Configuring Products.

Loughran et al. propose an approach where variability is expressed using cardinality-based feature models. Products are configured creating feature configurations.

The main purpose of Loughran et al. is to provide support for composition of software *components* based on feature configurations. Configuration of products could be performed by product designers in one or several stages. However, the authors only consider one configuration stage to capture domain (non-architectural) choices.

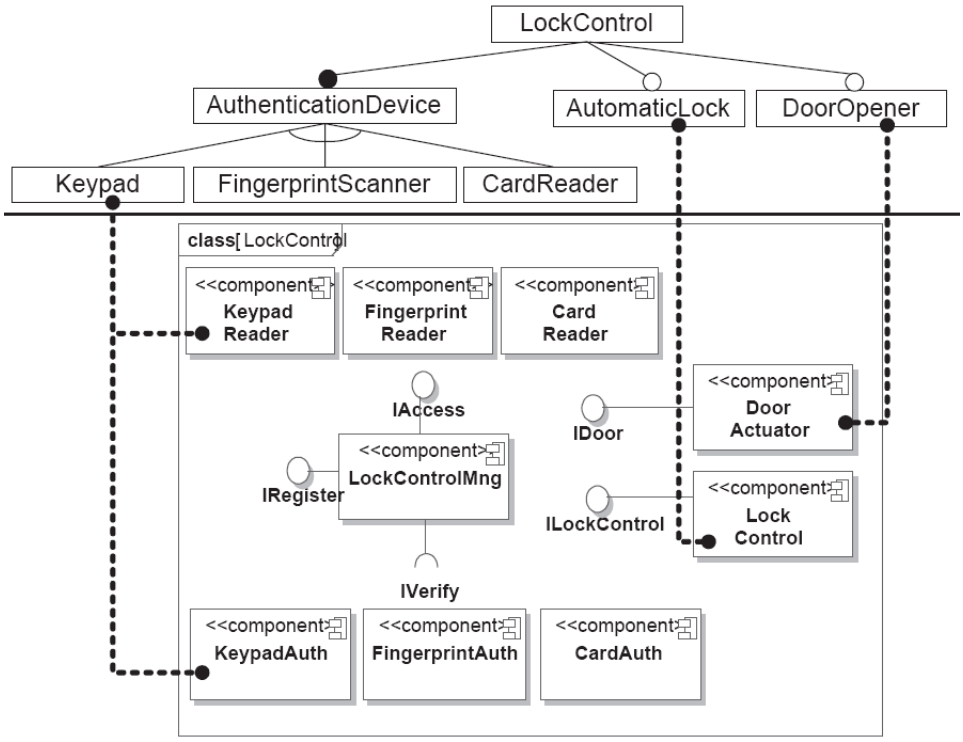

Figure 3.9: Example of a Reference Architecture in the Loughran et al.'s Approach [SLFG08].

Solution Space: Core Assets Development and Products Derivation.

Product line members are derived from component models of UML 2.0. A set of component models is created for each feature in the feature model. Additionally, a set of common components is created. Common components are present in every product of the product line. Figure 3.9 [SLFG08] presents an example of a feature model (top) and a reference architecture model (bottom) including the set of components related to the different features. Thus for example, if the `Keypad` feature is selected in a feature configuration, the `KeypadReader` component must be connected to the common component `LockControlMng` to derive a final product.

Loughran et al. propose a language, *VML*, to express how software compo-

```
1  Concern LockControl {
2    VariationPoint AuthenticationDevice {
3    Kind: alternative ;
4      Variant Keypad {
5      SELECT:
6        connect(KeypadReader,LockControlMng)
7          using interface(IAccess);
8      UNSELECT:
9        remove(KeypadReader);
10 }
```

Listing 3.1: Example of a VML Specification.

nents must be composed according to feature configurations. VML includes constructors that correspond to possible operations on components such as *connect(component-1, component-2)* or *disconnect(component-1, component-2)*. VML also supports to specify the links between features and components, indicating how the components of the reference architecture model must be composed according to features selected in feature configurations. For example, it is possible to specify that the `KeypadReader` component must be connected to the common component `LockControlMng` using the interface `IAccess` if the `Keypad` feature was selected in a feature configuration. Listing 3.1 presents the VML specification for this example.

Therefore, VML allows for creating decision models using its well structured constructors. Using the VML constructors it is possible to relate (1) a set of *effects* on the reference architecture and (2) features in a determinate state (selected/unselected). For instance, from Listing 3.1, if `SELECTED` the *Keypad* feature, then execute commands from line 6; if `UNSELECTED` the *Keypad* feature, then execute commands from line 8. Commands from line 6 and line 8 imply *effects* on the reference architecture. Using the VML constructors it is not possible, however, relating (1) a set of *effects* on the reference architecture model and (2) a subset of features in determinate states. For instance, if `SELECTED` the *Keypad* feature "and" `UNSELECTED` the *CardReader* feature, then execute a set of commands.

To transform VML specifications into a set of model transformations in charge of transforming reference architecture models into final products the authors have created a High Order Transformation (HOT). A HOT is a model

transformation that generates other model transformations. For executing the created HOT, the authors propose first to transform VML specifications into models that conform to a *VML metamodel*. Thus for instance, for the line 6 of the VML specification from Listing 3.1, the HOT generates a transformation rule to transform the reference architecture model from Figure 3.9 into a model including the connection between the `KeypadReader` and `LockControlMng` components by using the `IAccess` interface.

Thus, when a product designer creates a feature configuration, the generated model transformations are executed and the final product is derived. The execution ordering of the generated model transformations must be predefined, and they are fixed, to avoid inconsistencies in the final product.

3.6.4 The Voelter and Groher's Approach [VG07b]

Problem Space: Expressing Variability and Configuring Products.

The Voelter and Groher's approach proposes to create metamodels in conjunction with *cardinality-based* feature models to capture and express variability. This approach supports the explicit and separated modeling of variability in metamodels and feature models.

Product line architects create different metamodels during domain engineering; each metamodel captures concerns related to diverse concerns. One metamodel is the *domain metamodel* and serves as a vocabulary that is familiar to the practitioners of the system's domain. A domain model does not include concepts regarding details of the structure or processing of the system. Others metamodels are the *architectural metamodel*, which contains software architectural concerns, and the *platform metamodel*, which contains technological platform concerns.

Regarding feature models, only one feature model is created grouping different concerns.

This approach is particularly interested in staged-configuration and staged-derivation of products. To configure a product, first a product designer creates a model that conforms to the domain metamodel. After, another product designer selects features from the feature model including choices from concerns different to the general application domain.

Solution Space: Core Assets Development and Products Derivation.

Products are derived from (1) domain models, (2) reusable models that conform to the architectural and platform metamodels, (3) reusable pieces of source code, (4) and model transformations in charge of adapting the reusable models and pieces of code according to domain models and the valid feature configurations.

For each feature in the feature model the authors suggest to create a set of reusable models, source code and model transformations. Model transformations are created to transform (1) domain models into architectural models, (2) architectural model into platform models, and finally, (3) platform models into source code. Thus, if a feature is selected in a feature configuration, the domain model is incrementally transformed using the model transformations associated to the selected feature. The model transformations not only create new model structures in the architectural and platform domains, but they also take the reusable models and weave them to the new crated model structures. Similarly, in the latest transformation, the model-to-text transformations create source code and reuse pieces of code to create final products.

For the implementation of this approach, the authors use the oAW framework, including its AOP mechanism (see Section 2.5.2). Thus, decision models are created in form of textual descriptors, oAW workflows, to support the product derivation process. In these descriptors the authors indicate the model transformations that must be executed, and the required execution ordering according to selected features. For modifying the execution ordering having into account feature configurations, the authors have created a new oAW component. This component allows for querying a feature configuration at model transformation execution time, and weaving an oAW aspect if a particular feature appears in the configuration (selected or unselected).

Listing 3.2 presents an example of an oAW workflow using the created component. The `baseModelTransformation` from line 6 transforms a domain model into an architectural model. For this the first rule to be executed is `transformationRuleBase(domainModel)` (line 9). The normal call graph of this rule is modified if the `featureExample` is selected. This is specified in line 2. If the feature appears selected in a feature configuration the `transformationAdvice` is executed, modifying thus the base execution

```
1
2  <feature isSelected="featureExample">
3    <transformationAspect adviceTarget="baseModelTransformation">
4      <extensionAdvice value="transformationAdvice" />
5    </transformationAspect>
6  </feature>
7
8  <transform id="baseModelTransformation">
9    <invoke value="transformationRuleBase(domainModel)" />
10   <outputSlot value="architectureModel" />
11 </transform>
```

Listing 3.2: Example of a Workflow Using the Voelter and Groher's Component.

ordering. For details of how the AOP mechanism of oAW works see Section 2.5.2 and [OAW09a].

3.6.5 Discussion

In this section, we emphasize on the advantages and drawbacks of the presented approaches. We tackle this discussion regarding the mechanisms for expressing variability and configuring products, and, we review if the approaches consider (1) metamodeling and feature modeling, (2) multi-staged configuration of products and (3) expression of possible fine-grained variations between product line members, and fine-grained configuration of products.

Regarding the mechanisms to develop core assets and to derive products, we review how the approaches (1) create and use decision models, and (2) how they tackle the derivation of fine-grained configured products.

Metamodeling and Feature Modeling.

Metamodeling and feature modeling are the most common mechanisms used in MD-SPL approaches for capturing and expressing variability. Both metamodeling and feature modeling can be used for capturing not only structural but also behavioral variations. However they are different in many ways. On

the one hand, metamodels facilitate the modeling of variations at language level. Product designers who are domain experts are capable of configuring different products by creating diverse and rich domain application models. Thus, metamodeling implies a constructive approach that requires a high level of expertise. On the other hand, feature modeling allows configuring products only by selecting features, hiding the complexity of building complex models; this is a selection-based approach that requires only domain knowledge.

Regarding the use of alternative mechanisms for capturing and expressing variability, such as ontology models, they seem to be very useful and still require much exploration to become well exploited in the MD-SPL Engineering field. As part of our future work (see Chapter 6), we plan to explore how we can incorporate the use of ontology models into our approach.

Using feature modeling and metamodeling together, such as Voelter and Groher propose, gives to MD-SPL approaches the advantage of counting with the flexibility and power of expression of metamodels, and the simplicity, well-known and well-defined structure of feature models. To use feature modeling and metamodeling together it is necessary to establish relationships between them.

Multi-Staged Configuration of Products.

MD-SPL approaches such as those presented by Czarnecki and Antkiewicz, Wagelaar, and Loughran et al. only consider one-stage activity for configuration of products. The Loughran et al.s' approach focuses on capturing variations including only domain (non-architectural) concepts. The software architecture of products is predefined and it cannot vary. The Wagelaar's approach is more focused on technological platform variations and the Czarnecki and Antkiewicz's approach is only worry about problem space variations. All these approaches have the advantage to focused on the domain level where they are experts (*e.g.* architectural domain or platform technology domain), but this limits the scope of product lines because it is not possible to configure variations from other domains.

Voelter and Groher's approach supports the explicit and separated modeling of variability. Products may be configured at different binding times where at each stage specific variants are chosen creating domain models or feature configurations. Then, *e.g.* design or technology decisions may be left

open or postponed to the latest possible binding time in the configuration process. Furthermore, multi-stage configuration facilitates the intervention of product designers with different domain-knowledge at different binding times [CHE05]. For instance, assume we want to create an MD-SPL from diverse class models that are transformed into models of relational database schemas, and then into source code of product line members. To make the scope of the product line wider, a feature model is created with one feature group, `Primary Key Structure`, which groups two alternative grouped features, `With Primary Key` and `Without Primary Key`. Thus, two different product designers could configure products at different binding time by creating class models and feature configurations. If the feature `With Primary Key` is selected, all the class elements are transformed into table elements with one primary key. If the feature `Without Primary Key` is selected, all the class elements are transformed into table elements without a primary key.

Coarse- and Fine-Grained Variations and Configurations.

The MD-SPL approaches we have presented capture and express the possible variations between members of a product line by creating separate metamodels and/or variability models. This allows product line architects to capture and express *coarse-grained variations* between products. For example, using the above example of the MD-SPL created from diverse class models that are transformed into models of relational database schemas, a first product has a coarse-grained variation in relation with a second product if all the tables storing data of the first product have a primary key, and none of the tables storing data of the second product have a primary key.

Coarse-grained variations between members of a product line are obtained from the *coarse-grained configuration* that product designer can create using separate metamodels and variability models. A *coarse-grained configuration* consists of models that conform to metamodels, and instances of variability models. Thus, for instance, a first product can be coarse-grained configured by creating a class model and selecting the variant `With Primary Key`. A second product can be coarse-grained configured by using the same class model created to configure the first product, and selecting the variant `Without Primary Key`. When products are derived, a coarse-grained variation between them appears: all the tables storing data of the first product will have a primary key, and none of the tables storing data of the second

product will have a primary key.

The presented MD-SPL approaches lack of mechanisms to capture and express *fine-grained variations* between products. For instance, a first product has a fine-grained variation in relation with a second product if tables storing data of both products have primary key, but the two products differ in their particular tables which have primary key.

Along with the need of expressing the possible fine-grained variations between members of a product line is the need of defining a mechanism for creating *fine-grained configurations*. A fine-grained configuration must allow product designers to configure model elements individually based on variability models. For example, a fine-grained configuration must allow indicating that the feature `With Primary Key` affects individually a class `Student`, while the feature `Without Primary Key` affects individually a class `Professor`. This also requires a mechanism to restrict the valid fine-grained configurations. For example to indicate that the features `With Primary Key` and `Without Primary Key` could affect `Class` elements individually, but it is not valid that they affect `Attribute` elements from class models.

In the next chapter we present the mechanisms we propose for dealing with fine-grained variations, fine-grained configurations and for constraining their creation.

Core Assets Development and Decision Models.

Approaches such as the Czarnecki and Antkiewicz's approach and the Wagelaar's approach couple their core assets and their variability models. The Czarnecki and Antkiewicz's approach propose to annotate UML models with features. The Wagelaar's approach relates the model transformations with the platform instances created to express possible variations in particular technological platforms. The coupling of core assets and variants makes difficult the maintenance and reuse of transformation rules, other core assets such as UML models, and variability models. For avoiding this problem decision models are a good proven solution.

The Loughran et al.s' approach and the Voelter and Groher's approach are representative examples of approaches using explicit decision models. The Loughran et al.'s approach uses its defined language VML to create decision models. The Voelter and Groher's approach takes advantage of the oAW

framework for creating decision models by means of oAW workflows. Both the Loughran et al.'s approach and the Voelter and Groher's approach, however, have some limitations to create decision models.

On the one hand, the Loughran et al.'s approach only has into account the individual selection of features to adapt the architectural model of product line members. This approach does not take into account that several features selected together may imply different adaptation than the required when features are selected separately. Thus, this approach does not study the effects that different combinations of features may have in reference architectural models. On the other hand, the Voelter and Groher's approach is restricted to use a platform-dependent language to create decision models. This is the Xtend language provided by oAW. This limits the portability of the approach.

Product Derivation.

An important characteristic required in MD-SPL approaches is the ability to select transformation rules and modify their execution ordering according to selected variants. The Czarnecki and Antkiewicz's approach does not provide this characteristic. Instead, the approach proposes executing always a predefined set of transformation rules which from source annotated models generate target models. Since the approach does not provide a mechanism to indicate a required execution ordering, they have to use a post-transformation process to eliminate inconsistencies in target models. This mechanism is very restricted dealing only with well-identified inconsistencies in UML models.

The Wagelaar's approach and the Voelter and Groher's approach provide mechanisms for selecting transformation rules and modifying their execution ordering according to selected variants. These mechanisms are however attached to particular model transformation languages. On the one hand, the Wagelaar's approach implements its mechanism using the ATL language. On the other hand, the Voelter and Groher's approach implements its mechanism using the Xtend language.

The Loughran et al.'s approach in turn provide a mechanism for selecting transformation rules according to selected features, but it does not provide a mechanism for modifying their execution ordering. Thus, transformation rules are always executed in a predefined ordering. This mechanism works well in systems where the execution ordering of transformation rules does

not affect the resultant target models. This is however a characteristic that limits the generation of real software systems, where the selection of different variants necessarily requires adaptation of the execution ordering of model transformations.

Comparison Summary.

Table 3.2 presents a comparison summary based on the discussion presented in this section.

3.7 Summary

In this chapter we have introduced the Software Product Line Engineering paradigm. We were focus on the management of variability through the whole development lifecycle of SPLs. We described the Domain Engineering process and the Application Engineering process as the core processes in SPL Engineering. On the one hand, we explained how the Domain Engineering process involves activities for capturing and expressing variability in SPLs, and for developing the core assets which are reused for derivation of products. On the other hand, we explained how the Application Engineering process involves activities for configuring and deriving products.

We introduced MDD-based SPLs (MD-SPLs), which are the type of SPLs on which we are specially interested. We have presented four approaches for creating MD-SPLs explaining their mechanisms for expressing variability and configuring products, and developing core assets and deriving products. At the end of this section, we presented a discussion remarking the advantages and drawbacks of these approaches.

The next chapter presents our proposal of MD-SPL approach including the mechanisms we developed for sorting out the drawbacks found in related work.

Table 3.2: Related Work's Comparison Table

	Czarnecki and Antkiewicz	Wagelaar	Loughran et al.	Voelter and Groher
Metamodeling for expressing variability and modeling for configuring products	No	Yes	No	Yes
Multi-staged configuration of products	No	No	No	Yes
Expression of *fine-grained* variations and creation of *fine-grained* configurations	No	No	No	No
Creation of explicit decision models	No	No	Yes	Yes
Decision models take into account the effects that possible feature combinations may have in final products	n/a	n/a	No	Yes
Decision models are independent of particular implementation languages	n/a	n/a	Yes	No
Selection of transformation rules according to selected variants	No	Yes	Yes	Yes
Modification of transformation rules' execution ordering according to selected variants	No	Yes	No	Yes
Mechanisms for modifying execution ordering of transformation rules independent of particular model transformation languages	n/a	No	n/a	Yes

Part III. Proposal

Chapter 4

Binding Models, Constraint Models and Decision Models

4.1 Introduction

In Chapter 3, we have presented how MDD can be used to enhance SPL Engineering. We have shown that models and model transformations can be used to support respectively the configuration and derivation of product line members. We have also discussed how current MD-SPL approaches (1) have some limitations to express variability and configure products, and (2) do not provide appropriated mechanisms to derive products that facilitate the maintenance, reuse and evolution of reusable core assets such as transformation rules.

This chapter first introduces a case study which is used through the description of our approach, Section 4.2. We then present the base strategy we use in the processes of expressing variability and configuring products supported on metamodels and feature models, Section 4.3. After, we present FieSta, our MD-SPL approach. We present our proposal to improve the power of expression of variability, Section 4.4, where we introduce our mechanism to capture and express *fine-grained variations* between products of a MD-SPL. Finally, we present our mechanisms for deriving configured products supported on decision models, Section 4.5 and Section 4.6. At the end of this chapter, we present limitations of FieSta, Section 4.7.

4.2 Case Study

In order to better explain the general concepts, the problems related to this thesis and our approach to achieve the thesis objectives and validate our results, through this document we use an example that is part of a product line of Smart Home systems.

The Smart Home case study is taken from the domain of home automation. *"A smart home is a building for living equipped with a set of electrical and electronic sensors and actuators in order to allow for an intelligent sensing and controlling of the building's devices: windows, heaters, lights, etc."* [EFG+08].

The objective of this case study is to generate software that can be tested in our own simulation environment. This environment serves for demonstration and validation of the model-driven product line engineering mechanisms and tools developed as part of this thesis, which has been developed in the context of the AMPLE project [AMP09].

4.2.1 Smart-Home System's Domain

Currently homes are equipped with a wide range of electronic and electrical devices such as light arrays, temperature sensors and thermostats, electrically steered blinds and windows, door sensors and door openers etc. A Smart-Home software system coordinates and controls such devices enabling inhabitants to manage them from a common user interface.

A Smart Home system shall offer high level functionality in which several sensors and actuators are working together. Sensors are physical devices that measure properties of the environment and make them available to the Smart-Home system. Actuators activate devices whose state can be monitored and changed. All installed devices, including sensors and actuators, are part of the Smart-Home network. The status of devices can either be changed by inhabitants via the user interface or by the system using predefined policies. Policies let the system act autonomously in case of certain events. For example, in case of low indoor temperature, windows get closed automatically.

The architectural structure of buildings (*e.g.* the number of floors, rooms or windows), the sensors and actuators as well as other devices, their location inside the buildings, and the policies that let the system act, are particular for each Smart-Home system and must be defined by Smart-Home system's designers. Thus, Smart-Home systems can be created with the necessary and sufficient software components to respond to particular requirements of Smart-Homes' owners. At run time of a Smart-Home system, the system is then ready to respond to external or internal stimulus depending on the defined structure of the building, its devices and the policies acting on them.

Some of the functions that Smart-Home systems must provide are the following:

- **Climate control system.** Climate control devices must be orchestrated to keep a preferred temperature in the rooms of the house.
- **Security system.** Door and window sensors and motion detectors should be used to detect if people who are not allowed to enter the house try to do it. If the house detects any attempt of intrusion, it should take emergency actions.
- **Energy saving.** House devices should be orchestrated to use the least amount of energy as possible.

4.2.2 Case Study Requirements

Several types of houses, different customer demands, the need for short time-to-market, and saving of costs are the main causes for variability and motivate the need for product lines of Smart-Home systems. For our particular case study, we characterize a Smart-Home system's product line according to the following three sources of variability:

- **Architectural structure.** Each house may have a particular architectural structure with several floors, rooms, stairs, doors and windows.
- **Smart-Home's Facilities.** Each house may be equipped with several facilities related to controlled devices.
- **Software Architecture.** Each Smart-Home system has a technology platform integrating their devices under different software archi-

tectures.

The objective of this case study is to develop diverse Smart-Home systems which (1) are able to manage particular variants of Smart-Homes and (2) only include the necessary software components to satisfy the requirements of Smart-Homes' owners. It is not our interest to develop only one Smart-Home system which can be dynamically configured to support the considered variability. The following subsections describe in more detail the particular variants related to each of the three source of variability our product line considers.

Architectural structure.

The structure of houses is the most evident source of variation. The description of a house includes structural elements as floors and rooms. In our case study, we take into account the following structural elements: floors, rooms, staircases, doors and windows. Thus, houses can have different number of floors; floors can have different number of rooms; rooms can have different number of windows or doors; staircases connect the different floors in the house, and so on. Therefore, the configuration of the architectural structure of houses must be performed by *building architects*.

Smart-Home's Facilities

We take into account the need of incorporating houses automation facilities that are orthogonal to the house structure. By orthogonal we mean facilities that affect multiple structural entities. These facilities let the house act autonomously according to defined policies.

Thus, houses include electrical and electronic devices such as automatic lights, electric windows, security devices as alarms, security systems for authentication, among others. These devices, and therefore their behavior, are related to optional facilities that the designer has to select and bind to other elements that already exist in the house. For instance, the automatic lights can be bound to all rooms in the house or the security alarm system can be bound only to the main door entrance.

For this case study we consider two groups of facilities. The first group is related to the access control facilities. The second one is related to environmental control facilities.

- **Access Control.** This facility should assure that only inhabitants and authorized visitors may go into the house. Two alternative options must be provided to control the access of inhabitants: (1) keypad authentication and (2) fingerprint authentication.
- **Environmental Control.** This facility must add the capability of measuring the indoor temperature and take some actions according to predefined rules. Two alternative options must be provided to environmental control: (1) automatic windows and (2) air conditioning. Automatic windows must be automatically opened if the temperature in a room rises above a certain threshold and closed if the temperature falls below a certain threshold. Similarly, air conditioning is turned on if the temperature in a room rises above a certain threshold and turned off if the temperature falls below a certain threshold.

The configuration of Smart-Homes' facilities must be in charge of experts who know the domain of configuring houses including devices such as sensors and actuators. *Facilities designers* must also support houses' owners to take decisions about distribution of devices, for example to save costs of construction and maintenance of Smart-Home systems.

Software Architecture.

We build Smart-Home systems using a component-based development strategy. We create components to manage the different devices included in Smart-Home systems. For our case study we use OSGi (Open Services Gateway Initiative) [OSG09] as our base components integration platform because it is currently the preferred platform for home automation.

We classify software components according to their type, *Periodic* or *Service* components, and their instantiation mode, on *Deployment* or on *Invocation*. Thus, regarding the architecture of Smart-Home systems, this can vary depending of the type of components we create to manage the devices, and the instantiation mode the components implement.

- **Components Type: Periodic or Service Components.** On the one hand, periodic components are *active* components offering exactly one service. The infrastructure invokes this service periodically after a configurable time period. Periodic components have their own threads of control and they are started once the component is activated. On the

other hand, service Components are *passive* components offering services to other components. The type of a component depends on the particular services the component will provide. For instance, a component providing a service to open/close automatic windows according to the temperature of rooms is a good candidate to be a periodic component. This component could check periodically the temperature of rooms to open/close the automatic windows. A component providing services to open/close doors only when inhabitants arrive is a better candidate to be a service component.

- **Instantiation Mode: on Deployment or on Invocation.** A component can be instantiated either when it is deployed or when one of their services is invocated.

The configuration of the software architecture of Smart-Home systems must be in charge of *software architects* who have experience in taking software design decisions.

Thus, for each different source of variability, one expert with particular skills is required to configure a Smart-Home system. In our case study these are the *building architect*, the *facilities designer* and the *software architect*. As presented before in Section 3.4.1, this is one of the main reasons that imply the use of staged-configuration mechanisms. In each stage, an expert with particular skills must be involved in the configuration process.

4.3 Variability Expression and Product Configuration

As presented in Chapter 3, the most common mechanisms used to capture variability and configure products in MD-SPLs are metamodels and feature models. As part of our approach to create MD-SPLs, playing the role of product line architects we use metamodels and feature models as our base core assets. Playing the role of product designers, we configure products creating models that conform to metamodels and feature configurations.

4.3.1 Metamodels

Because we use a multi-staged approach for configuration and derivation of products, we separate domain-specific concepts in several metamodels. The first one is the domain metamodel which serves as a vocabulary that is familiar to the practitioners of the system's domain. A domain model does not include concepts regarding details of the structure or processing of the system. The other metamodels contain facilities and architectural concepts which are orthogonal to the concepts in the domain metamodel. This concepts represent variability that affect multiple domain concepts and their subsequent processing (*i.e.* transformation and generation) stages.

Each metamodel has as main objective to capture the variability that characterizes a product line; however, they play different roles during the product line development lifecycle. Product designers use the first metamodel, the *domain metamodel*, during the configuration process. This metamodel is the reference to create domain models, which are the starting point to derive product line members.

We create three metamodels to capture, separately, the three sources of variability that characterize our Smart-Home system's product line (see Section 4.2.2):

- **Domain Metamodel.** This metamodel includes concepts regarding architectural structure of houses.
- **Smart-Home's Facilities Metamodel.** Each house may be equipped with several facilities related to controlled devices.
- **Software Architecture.** Each Smart-Home system has a technology platform integrating their devices under different software architectures.

Besides these three metamodels, we create another metamodel: *the component metamodel*. This metamodel includes only concepts regarding component-based development. This metamodel is important to represent the problem domain in terms of software components.

The first stage to configure a product starts with the creation of domain model; *i.e.*the model that represents a particular building. Figure 4.1 presents the staged-transformations a domain model suffers after it is cre-

ated by a product designer, in this case a building architect. The model transformation rules are used in four stages. The first set of rules is defined from the domain metamodel to the facilities metamodel. The second set is defined from the facilities metamodel to the components metamodel. The third set is defined from the components metamodel to the architecture metamodel. Finally, the fourth set of rule transformations includes model-to-text transformations which produce the source code of product line members. We create model-to-text transformation rules from the facilities and the architecture metamodel to Java source code. Next sections present the details about the model transformations, and the possible variations these model transformations can have according to the SPL variability.

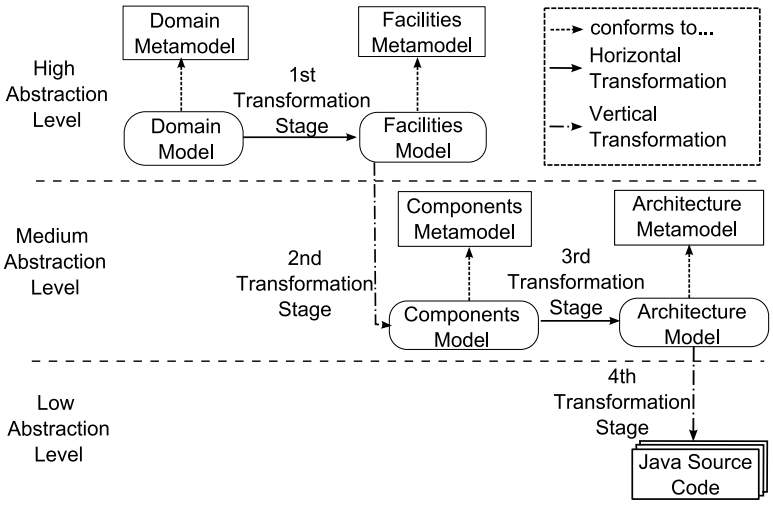

Figure 4.1: Staged-Transformations to Derive SPL Members.

In the following we present our four metamodels in detail.

Domain Metamodel.

The first metamodel is the domain metamodel which includes domain application concepts that facilitate the creation of models representing the structure of houses. Figure 4.2 presents the domain metamodel. Using this metamodel we can create houses with several architectural structures including several `Floor`, `Room`, `Door` and `Window` elements.

Figure 4.3 shows a domain model example that conforms to the domain metamodel. The model defines `firstFloor` and `secondFloor`. These conform to

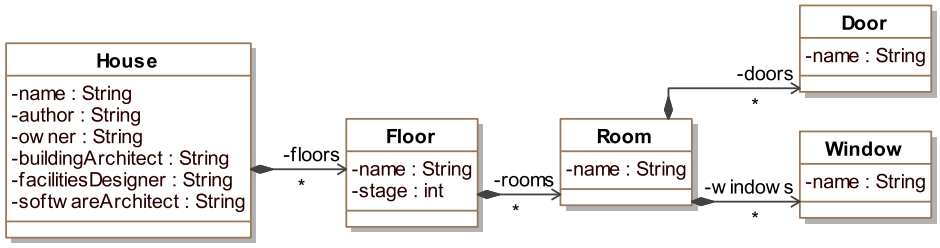

Figure 4.2: The Domain Metamodel.

the `Floor` metaconcept. In the `firstFloor` there are two rooms, `livingRoom` and `kitchen`. In the `secondFloor` there is another room, `mainRoom`, which has two windows, `mainRoomW1` and `mainRoomW2`. There are also two doors. The first door, `livingRoomD1`, is in the `livingRoom`. The second door, `mainRoomD2`, is in the `mainRoom`.

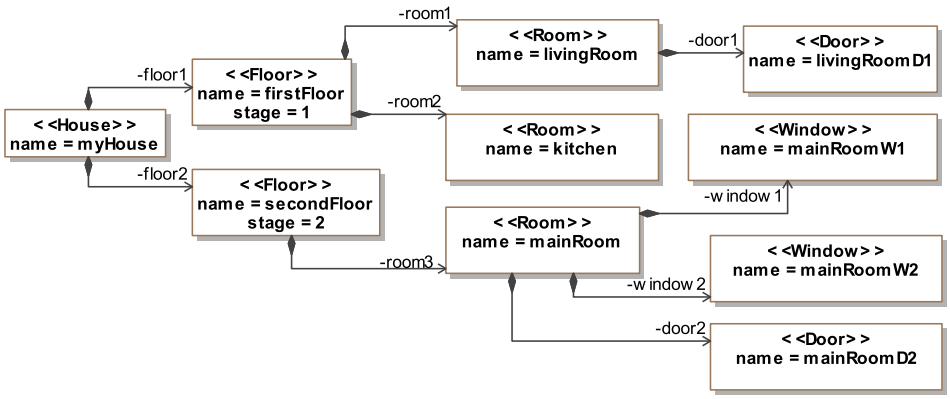

Figure 4.3: Example of a Domain Model.

Facilities Metamodel.

The facilities metamodel is presented in Figure 4.4. This metamodel is at the same level of abstraction as the domain metamodel. The facilities metamodel includes, however, metaconcepts of Smart-Homes' facilities such as environmental control and authentication devices. Based on this metamodel it is possible to add facilities to Smart-Homes. The `Window` metaconcept is now specialized in `Automatic` and `Manual` metaconcepts. Thus, windows can be configured as automatic or manual windows. The `Room`

metaconcept contains one `EnvironmentalControl` metaconcept, which is specialized in the `WindowsController` and `AirConditioning` metaconcepts. Thus rooms can be configured to manage air conditioning or automatic windows as environmental control. Finally, the `Door` metaconcept contains the `LockDoorControl` metaconcept, which is specialized in the `Fingerprint` and `Keypad` metaconcepts. Thus, doors can be configured to manage fingerprint or keypad as lock door control.

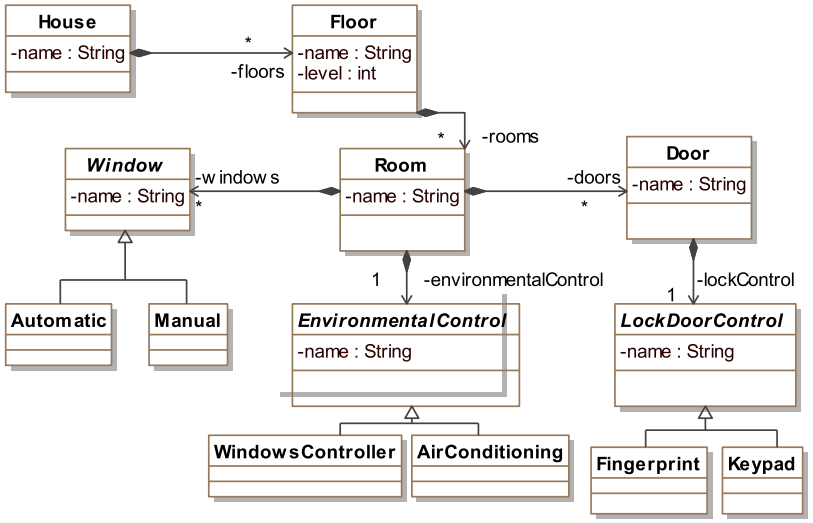

Figure 4.4: The Facilities Metamodel.

Components Metamodel.

The (software) components metamodel, is used to represent concepts of Component-Based development. This metamodel includes more implementation details than the domain metamodel and the facilities metamodel. That is because we said this metamodel is at a lower abstraction level than the two previously presented metamodels.

Figure 4.5 presents our components metamodel. We take the basic concepts from the UML2 metamodel to create a simplified metamodel of components. A `Component` is a modular, replaceable, and deployable piece of software which interacts with its environment via interfaces or ports [LH05]. We specialize the `Component` metaconcepts in the `Periodic` metaconcept. Thus, we can create `Periodic` (Component) elements when the component is can-

didate to be a periodic component in the final software architecture of a Smart-Home system. A `Port` serves as a contract between the elements it connects. Ports are usually of the type `Interface`. In UML2, interfaces can be either `Provided` or `Required` ones. `Provided` interfaces specify the `providedOperations` that a component offers to their clients. `Required` interfaces specify the `requiredOperations` that a component needs to perform its functions. In UML2 `Provided` and `Required` interfaces are related using Connectors. To simplify our metamodel we connect `Provided` and `Required` interfaces creating a directed relationship between them, `useProvided`. Finally, a `Component` owns a unique identifier, `componentName`, and a set of `Property` elements.

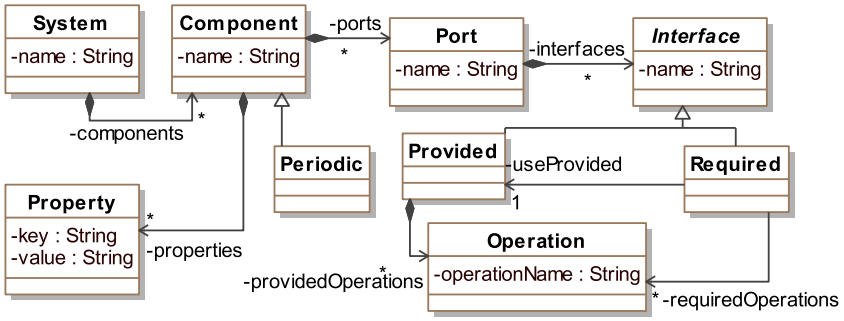

Figure 4.5: The Components Metamodel.

The Architecture Metamodel.

The architecture metamodel is at the same level of abstraction than the components metamodel is. However, the architecture metamodel includes new metaconcepts to represent the variants identified regarding architectural design. Figure 4.6 presents the architecture metamodel. The `Component` metaconcept is now specialized also in `Service`, thus components can be configured to be either periodic or service components. Furthermore, the `Component` metaconcept includes the property `instantiationMode` to indicate when a component is instantiated, `ON_INVOCATION` or `ON_DEPLOYMENT`.

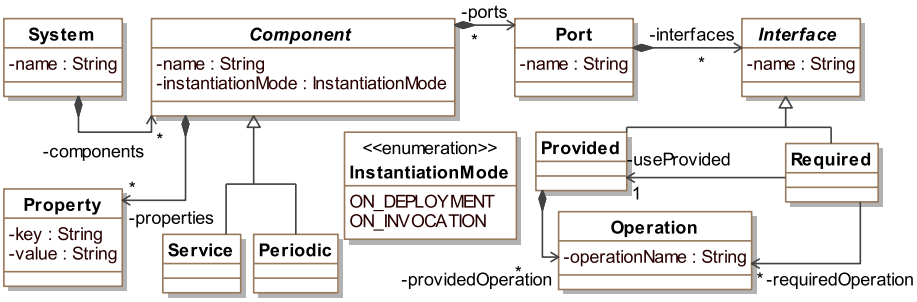

Figure 4.6: The Architecture Metamodel.

4.3.2 Feature Models

Because of the different sources of variability, MD-SPL approaches must allow product designers to configure a product giving its domain model, and selecting variants from the sources of variability. For instance, in our case study those are the variants from Smart-Home's facilities and software architecture.

Figure 4.7 presents an example of how limited the configuration and the derivation of Smart-Home systems will be in case of only allowing product designers to configure a product by means of a specific domain model. In the example, from a building representing the architectural structure of a Smart-Home, only one possible Smart-Home system could be derived, without including variants from concerns different to buildings' structure.

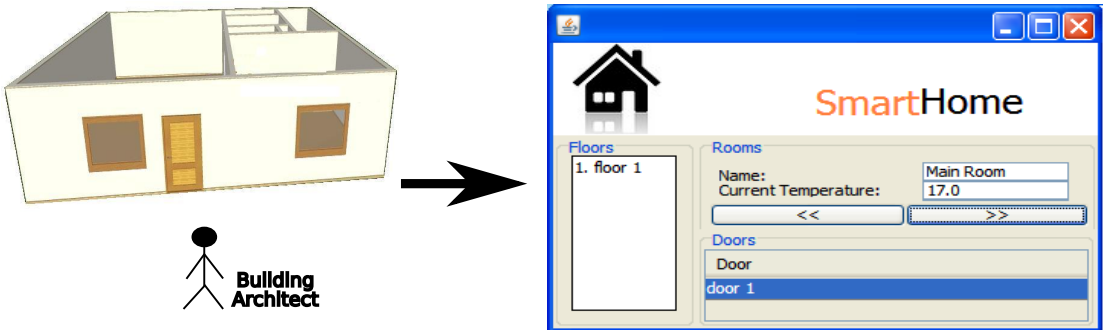

Figure 4.7: Example of Configuration without Variability Models.

We use feature modeling to allow product designers configuring products from sources other than the application domain. Likewise metamodeling, feature modeling can be used for capturing not only structural but also behavioural variations. Metamodeling facilitates the configuration of products by creating rich models using a constructive approach that requires a high level of expertise. Feature modelling facilitates the configuration of products by selecting features, hiding the complexity of building models from scratch; this is a selection-based approach that requires only domain knowledge.

We create our feature models based on the Czarnecki et al.'s metamodel [CHE04], which is itself based on FODA [KCH+90].

Figure 4.8 presents our simplified feature metamodel. Such as in the Czarnecki et al.'s metamodel, a `FeatureGroup` expresses a choice over the set of `GroupedFeatures` in the group and its cardinality defines the restriction on the number of choices. A `GroupedFeature` does not have cardinality and a `SolitaryFeature` is a feature that is not grouped by any `FeatureGroup`. Examples of feature models are given in the next subsection using our case study.

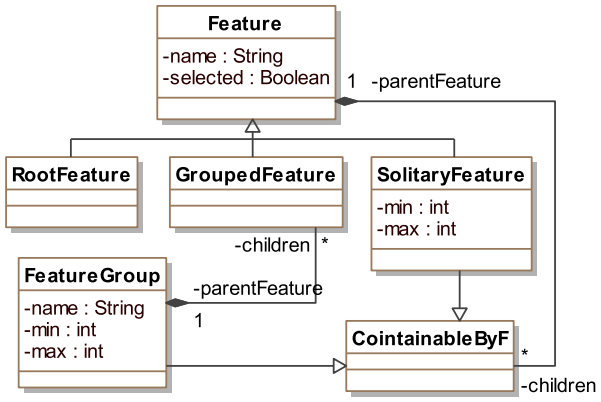

Figure 4.8: Simplified Feature Metamodel.

For our case study's SPL, playing the role of product line architects, we create a feature model that represents variants of Smart-Homes' facilities, and another one that represents variants of architectural (software) design. Thus, product designers are able to configure products by creating feature configurations including choices of Smart-Homes' facilities and (software) ar-

chitecture. These feature configurations are inputs to the product derivation process. They are used to select the transformation rules to be used in each stage of the model transformation chain.

The Facilities Feature Model.

As we introduced in Section 4.2.2, we take into account the need of incorporating the house automation facilities that are orthogonal to the house structure. We consider particularly two groups of facilities: access control facilities and environmental control facilities.

Figure 4.9 presents our Smart-Homes' facilities feature model. One `FeatureGroup` appears for each group of facilities. The `Lock Door Control` feature groups the features `Fingerprint` and `Keypad` and has cardinality `[0..1]`, which *implicitly* means that `Door` elements can have either keypad, fingerprint, or none of them as lock door control mechanism. The `Environmental Control` feature groups the features `Air Conditioning` and `Automatic Windows` and also has cardinality `[0..1]`, which *implicitly* means that `Room` elements can have either automatic windows, air conditioning, or none of them as lock environmental control mechanism. We say *implicitly* because there is no semantics in traditional feature models, neither in metamodels, to formally denote that features represent variants that affect particular model elements.

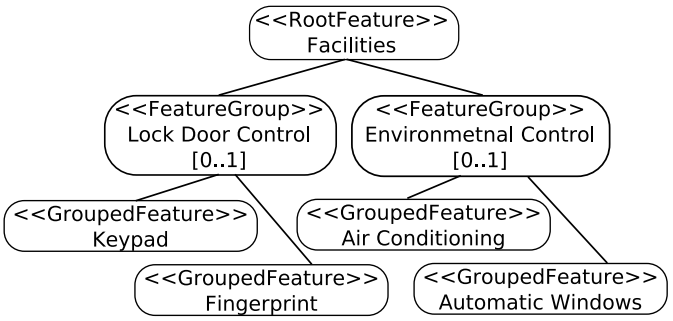

Figure 4.9: Smart-Homes' Facilities Feature Model.

The Architecture Feature Model

Figure 4.10 presents our architecture feature model. Given that we have classified software components according to their type, and their instantiation

mode, we create one `FeatureGroup` for each classification. The `Component Type` feature groups the features `Periodic` and `Service` and has cardinality `[1..1]`, which *implicitly* means that `Component` elements can be either periodic or service components. The `Instantiation mode` feature groups the features `Deployment` and `Invocation` and also has a cardinality `[1..1]`, which *implicitly* means that `Component` elements can be instantiated either on deployment or on invocation.

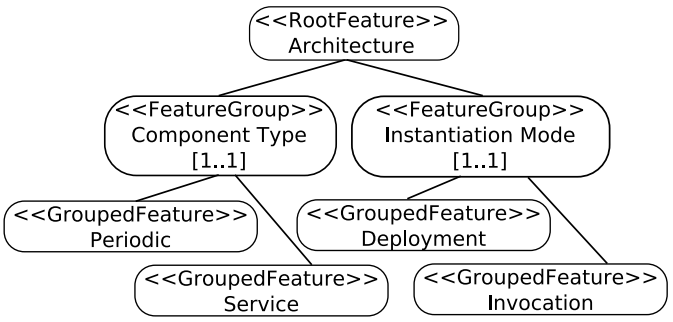

Figure 4.10: Architecture Feature Model.

Figure 4.11 summarizes the processes of (1) expressing the variability in our case study SPL and (2) configuring a Smart-Home system, by using only metamodels and feature models. First, a *building architect* creates a `Domain Model` based on the `Domain Metamodel`. Then, a *facilities designer* creates a feature configuration based on the facilities feature model. The facilities feature model affects the transformation of the `Domain Model` into the `Facilities Model`. According to selected facilities features, particular transformation rules must be executed to transform domain models into facilities models. For instance, if the feature `Automatic Windows` is selected, a particular transformation rule is executed to transform `Window` elements into `Automatic Windows` elements. If the feature `Automatic Windows` is not selected, another different transformation rule is executed to transform `Window` elements into `Manual Window` elements. The `Facilities Model` is transformed into a `Components model` and then a *software architect* creates another feature configuration based on the architecture feature model. The `Architecture Feature Model Configuration` affects the transformation of the `Components Model` into the `Architectural Model`. Finally, the `Architectural Model` and the `Facilities Model` are used to generate the

final `Java Source Code`. The next section details the process of deriving products using the described model transformation stages.

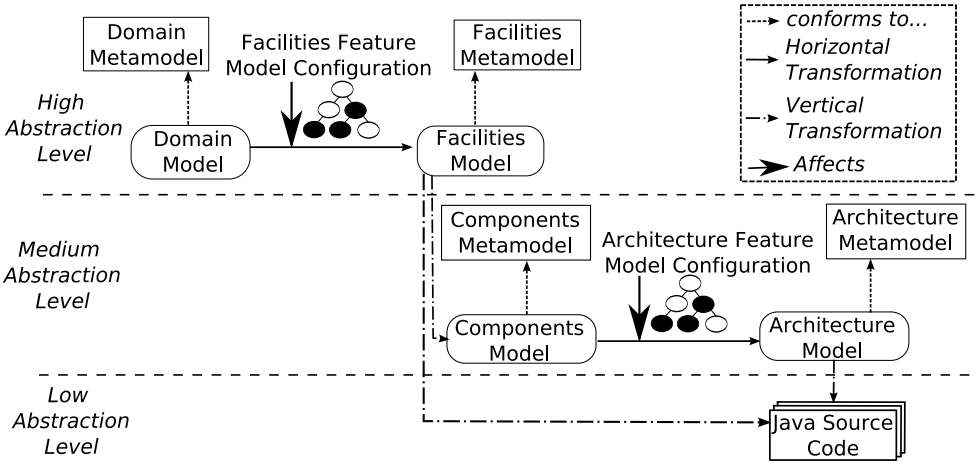

Figure 4.11: Summary of the Smart-Home Systems' Configuration Process.

Figure 4.12 presents an example of the staged configuration of two different Smart-Home systems. In the example we only present two stages. In the first stage a building architect configures the architectural structure of a building. In the second one a facilities designer creates two configurations to derive two different Smart-Home systems from the same building: on the left the configuration indicates that the Smart-Home system will have keypad as lock door control in all the doors, on the right the configuration indicates that the Smart-Home system will have automatic windows as environmental control, which implies that all the windows will be automatic windows.

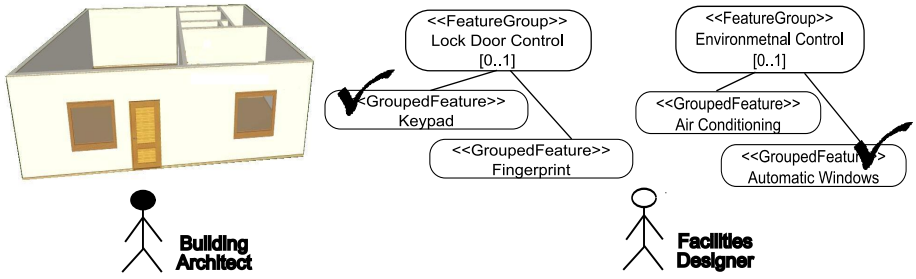

Figure 4.12: Example of Configuration with Variability Models.

4.4 Binding Models and Constraint Models

The base mechanisms we have introduced until now in this chapter allow product line architects to capture and express the possible variations between members of a product line by separately creating metamodels and feature models. This allows us to capture and express *coarse-grained variations* between products. For example, a first Smart-Home system has a coarse-grained variation in relation with a second Smart-Home system if all the `Windows` in the first Smart-Home system are `Automatic Windows`, and none of the `Windows` in the second Smart-Home system are `Automatic Windows`.

We obtain coarse-grained variations between members of our product line example by creating *coarse-grained configurations*. A *coarse-grained configuration* consists of models that conform to metamodels, and instances of feature models. Thus, for instance, a first Smart-Home system can be coarse-grained configured by creating a domain model and selecting the feature `Fingerprint`. A second Smart-Home system can be coarse-grained configured by using the same domain model, and selecting the feature `Keypad`. When Smart-Home systems are derived, a coarse-grained variation between them appears: all the `Doors` in the first Smart-Home system have `Fingerprint` as lock door control mechanism, and all the `Doors` in the second Smart-Home system have `Keypad` as lock door control mechanism.

We propose to improve the power of expression of variability providing a mechanism to capture and express that we have named *fine-grained variations* between products of a MD-SPL. For instance, a first Smart-Home system has a fine-grained variation in relation with a second Smart-Home system if both systems have automatic windows, but they differ in the particular windows which are automatic.

Additionally, we propose a mechanism to create *fine-grained* configurations, which allows us to configure model elements individually based on features. For example, by creating a fine-grained configuration we could configure the `mainRoom` to manage `Air Conditioning` as environmental control and the `livingRoom` to manage `Automatic Windows` as environmental control [ACR09].

The mechanisms we propose are based on that we have named *constraint models* and *binding models*. To facilitate the understanding of our proposal,

first we introduce our mechanism for configuring products by using binding models, after we present our approach to improve the power of expression of variability in MD-SPLs by using constraint models.

4.4.1 Binding Models

We call a *binding* the relationship between a model element and a feature. For example, to express that the `leavingRoom` (see Figure 4.3) has `Air Conditioning` as environmental control mechanism (see Figure 4.9). A binding B is a pair composed by a model element E and a feature F, $B = [E, F]$, where F is either a `SolitaryFeature` or a `GroupedFeature`. For example, a product designer can create a binding relating the `livingRoom` and the `Automatic Windows` feature, $B =$ [`livingRoom`, `Automatic Windows`].

We define a *binding model* as the set of bindings defined by a product designer between a model that conforms to a metamodel and a feature model, which conforms to a feature metamodel. Figure 4.13 presents a binding model example for our case study. This binding model is created between the domain model from Figure 4.3 and the facilities feature model from Figure 4.9. `binding1` configures the `livingRoomD1` to have `Keypad` as `Lock Door Control`. `binding2` denotes the designer selection of `Air Conditioning` in the `livingRoom` as environmental control system. Finally, the `binding3` defines that the `mainRoomW1` is configured to be an `Automatic Windows`. We include in Chapter 5 a complete example of the configuration and derivation of a Smart-Home system of our case study. The example also includes a binding model between a component model and the architecture feature model.

4.4.2 Constraint Models

Product line architects must use *constraint models* to restrict the bindings among model elements and features. For example, to express that only domain models can be bound to facilities models, or that maximum three `Room` elements can be bound to the feature `Air Conditioning`.

A constraint model is a set of *constraints*. A constraint $C = [M, F, A, D]$ is a tuple composed of a metaconcept M, a feature F, and two properties A

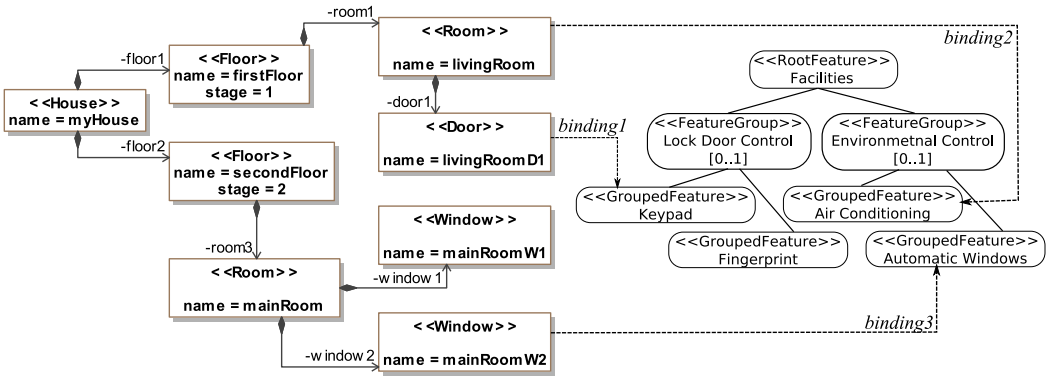

Figure 4.13: Binding Model Example.

and D. A constraint C expresses the fact that model elements that conform to the metaconcept M can be bound to the feature F. Each constraint is unique in a constraint model; this means, only one constraint includes a pair $[M, F]$.

Our constraints serve to avoid inconsistencies during the configuration and derivation processes. Constraints must prevent the following:

- Any model is bound to any feature model. For example in our case study, only domain models can be bound to the facilities feature models and only components models can be bound to the architecture feature model.
- Model elements that conform to any metaconcept are bound to any feature. For example, for a requirement of the product line ($R1$) specifying that only windows can be automatic, a constraint must prevent that, *e.g.* `Door` elements are associated to the `Automatic Windows` feature.
- Any number of model elements that conform to a metaconcept is bound to any number of features. For example, since the installation of automatic windows could be expensive, in a product line for economical Smart-Homes a product line architect may deal with a requirement ($R2$) which specifies that only (maximum) one window can be automatic. Thus, a constraint must prevent more than one `Window` element being bound to the `Automatic Windows` feature.

- Model elements and features are bound without taking into account properties inherent in functional requirements. For example, for a requirement of the product line ($R3$) which specifies that automatic windows must have sensors, a constraint must prevent `Window` elements without an associated `Sensor` element being configured as `Automatic Windows`.
- Model elements and features are bound without taking into account configuration's prerequisites. For example, for a requirement of the product line ($R4$) which specifies that automatic windows only can be selected from rooms which are not configured to have air conditioning. A constraint must prevent `Window` elements with their rooms associated to the `Air Conditioning` feature being configured as `Automatic Windows`.

Therefore for our case study, regarding the requirement ($R1$), a product line architect could define a constraint between the `Window` metaconcept and the `Automatic Windows` feature, `constraint1`= [`Window`, `Automatic Windows`, A, D]. The constraint describes that, during the configuration of a product, product designers can bind `Window` elements, for example the `mainRoomW2` with the feature `Automatic Windows` (see Figure 4.13). Another constraint can be created between the `Door` metaconcept and the `Lock Door Control` feature, `constraint2`= [`Door`, `Lock Door Control`, A, D]. The constraint describes that product designers can bind `Door` elements with either the feature `Keypad` or the feature `Fingerprint`. Figure 4.14 presents these constraints. In Chapter 5 we present the constraint models we created for our Smart-Home systems' SPL.

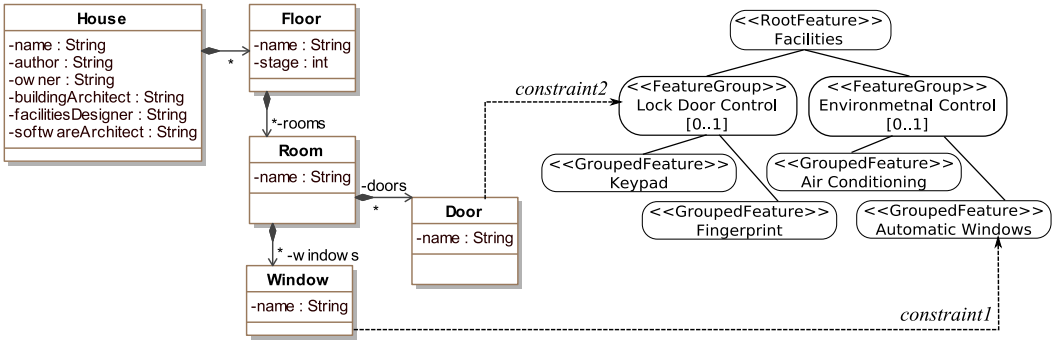

Figure 4.14: Constraint Model Example.

Semantics of constraints can be different. For example, a product line architect could also define a constraint between the `Room` metaconcept and the `Automatic Windows` feature, `constraint3`. The constraint describes that product designers may bind `Room` elements with the feature `Automatic Windows` to indicate that all the windows in the bound room are automatic windows. Product line architects can add descriptions to constraints to help product designers during the creation of bindings. Thus, for `constraint1` we added the description: A window may be an automatic window; for `constraint3`: All the windows in a room may be automatic windows.

4.4.3 The Cardinality Property

To fulfill the requirement $R2$ presented before, which specifies that only (maximum) one window can be automatic, our approach includes the definition of the cardinality property (A).

The cardinality property has a similar form to feature cardinality. We define the cardinality as a UML-like cardinality $A = [i..j]$, where $i <= j$, i and j are natural numbers, and j can be denoted by $*$ to express an unbounded number. Cardinality (A) adds semantics to a constraint $C = [M, F, A, D]$ by expressing the fact that the designer can create a restricted number of bindings between model elements that conform to M and the feature F (a number between i and j).

The requirement ($R2$) is an example where cardinality is required to limit the number of bindings among model elements and features. This indicates that only (maximum) one window can be automatic.

The next two sub-sections present the semantics of the cardinality property in a constraint. The semantics depend on the type of feature included in the constraint, *i.e.* group, grouped or solitary. The introduction of the cardinality property particularizes the cardinality of the original features in the feature model.

Cardinality on Solitary and Grouped Features.

In a constraint $C = [M, F, A = [i..j], D]$ where F is a solitary or grouped feature, the meanings of i and j are respectively the minimum and maximum number of model elements that conform to M that can be bound to F.

For example, if a product line architect wants to restrict to 0 or 1 the number of automatic windows, s/he must add the cardinality $A = [0..1]$ to the `constraint1` presented in Figure 4.14. Thus, maximum one window could be automatic, *e.g.* the `mainRoomW2` (see Figure 4.13).

Cardinality on Group Features.

In a constraint $C = [M, F, A = [i..j], D]$ where F is a group feature, the meanings of i and j are respectively the minimum and maximum number of features grouped by F that can be bound to a particular model element that conforms to M.

For example, for a requirement of the product line specifying that lock doors control can be managed by using either keypad or fingerprint, the product line architect creates a constraint using the `Room` metaconcept and the `Lock Door Control` feature, `constraint2`= [`Door`, `Lock Door Control`, A, D] (see Figure 4.14). The architect sets the cardinality $A = [0..1]$, constraining to zero or one the number of grouped features (`Fingerprint, Keypad`) that can be bound to a `Door` element. Thus a door, *e.g.* the `livingRoomD1` (Figure 4.13), can be bound to only one of the features `Keypad` or `Fingerprint`.

When a group feature F has the cardinality $[n..m]$, the cardinality of a constraint $C = [M, F, A, D]$ has to be inside the limits of the cardinality of F. It implies that for $A = [i..j]$, $i \geq n$ and $j \leq m$. This ensures that constraint models are consistent with feature models used for their construction.

4.4.4 The Structural Dependency Property

The structural dependency property D in a $C = [M, F, A, D]$, denotes conditions that model elements have to satisfy in order to be bound to specific features. An example from requirement $R3$ presented before is: automatic windows must have sensors. In this case, the model elements we identified in the conditions are `Window` and `Sensor`, and the feature is `Automatic Windows`. Thus, to bind a `Window` element to the `Automatic Windows` feature, allowed by the `constraint1`, the `Window` element should have a `Sensor` element.

Another example is a requirement specifying that only one room can have automatic windows. This requirement particularizes the requirement ($R1$),

specifying that windows must be localized in the same room. Then, only `Window` elements from the same `Room` element can be bound to the `Automatic Windows` feature.

We also use the dependency property to describe dependencies between bindings. For example, the requirement ($R4$), which specifies that automatic windows only can be selected from rooms which are no configured to have air conditioning, implies that a `Window` element can be bound to the `Automatic Windows` feature only if the `Room` element where the window is located is not bound to the `Air Conditioning` feature.

We express the value of the property D as a set of OCL sentences. For example, for the requirement ($R3$) a product line architect must set the structural dependency property of the `constraint1` to $D = \{$`sensor` $\rightarrow$ `notEmpty()`$\}$.

4.4.5 The Constraint Metamodel and The Binding Metamodel

The Constraint Metamodel.

We have created a *constraint metamodel* to facilitate the creation of constraint models. Our constraint metamodel is based on our feature metamodel (see Figure 4.8). We extended its semantics to include the constraints for managing binding models.

Figure 4.15 presents our constraint metamodel. The main new concepts and attributes added to our feature metamodel are the following:

- `GroupConstraint`. It allows us to create constraints including group features.
- `Constraint`. It allows us to create constraints including solitary or grouped features.
- `fineMin` and `fineMax` attributes. They allow us to relate the cardinality property to constraints.
- `OCLExpression`. It represents the structural dependency property of constraints.

- `Metaconcept`. It represents metaconcepts related to constraints.

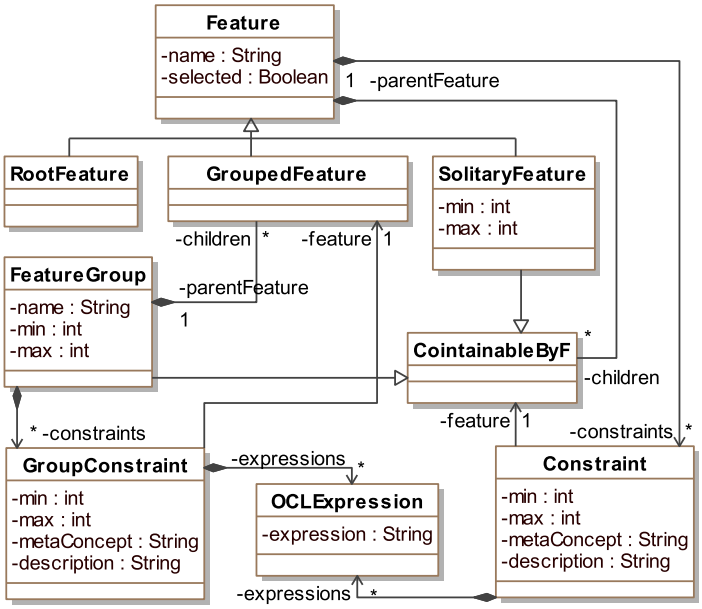

Figure 4.15: Constraint Metamodel.

Thus, in a constraint $C = [M, F, A, D]$, C conforms either to `GroupConstraint` or `Constraint`, M conforms to `MetaConcept`; F conforms to either `Grouped` or `CointainableByF` (`Group` or `Solitary`), i and j (from $A = [i, j]$) conforms to `fineMin` and `fineMax`, and D conforms to `OCLExpression`.

The Binding Metamodel.

To introduce the concept of binding, we create a binding metamodel, Figure 4.16. This metamodel extends our constraint metamodel with concepts for binding model elements to features. Thus, associated to a `RootFeature`, a set of `Configurations` can be created. A `Configuration` groups a set of bindings between `Features` and model elements. We maintain the information of model elements as properties of the `Binding` metaconcept, `metaconceptName` and `elementName`.

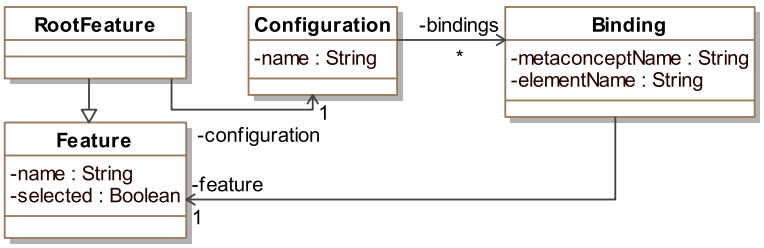

Figure 4.16: Binding Metamodel.

4.4.6 Validating Binding Models against Constraint Models

We say a binding $B = [E, F_1]$ *satisfies* a constraint $C = [M, F_2, A, D]$ when E conforms to M, $F_1 = F_2$ and B satisfies the restrictions defined by the properties A and D. We note this relationship $B \xrightarrow{s} C$. For example, `binding3` from Figure 4.13 satisfies the `constraint1` from Figure 4.14 because `mainRoomW2` conforms to `Window` and B satisfies the restrictions defined by the properties A and D of the `constraint1`. The validation of a binding model against a constraint model implies that every existing binding *satisfies* one constraint in the constraint model.

We validate existing bindings in a binding model automatically against a set of OCL-type sentences that we generate from each constraint in a constraint model. For example, if the feature involved in the constraint C = [`$metaConcept`, `$feature`, [`$fineMin`, `$fineMax`], D] is a grouped or solitary feature, we generate the sentence in Listing 4.1. The dollar symbol $ denotes variables and the operator `aCollection.between(a,b)` is equivalent to the expression `(aCollection.size`$\geq$`a)` && `(aCollection.size`$\leq$`b)`. Listing 4.2 presents the particular sentence generated for the `constraint`= [`Window`,`Automatic Windows`, [0..1],D], where D = bindings.select(b|b.elementName=="mainRoomW1").between(0,0). In this case D specifies that cannot exists any binding where the `mainRoomW1` is involved.

For the generation of OCL-type sentences we have created model-to-text transformation rules. These transformation rules generate *Check* expressions. *Check* is a language included in the oAW framework which allows us to validate models against OCL-type expressions [OAW09a]. We generate

```
1
2 Context Configuration inv:
3   bindings.select(b|b.feature.name==$feature and
4     b.metaConceptName==$metaConcept).between($fineMin,$fineMax) and $D;
```

Listing 4.1: Example of a Generated OCL-Type Sentence.

```
1
2 Context Configuration inv:
3   bindings.select(b|b.feature.name=="Automatic_Windows"
4     and b.metaConceptName=="Window").between(0,1)
5     and bindings.select(b|b.elementName=="mainRoomW1").between(0,0);
```

Listing 4.2: Example of a Generated OCL-Type Sentence.

Check expressions from the constraint models we create using the constraint models creator that we present in Chapter 5. Therefore, product designers are able to validate binding models against the generated Check expressions. We present details of the model-to-text transformation rules in charge of creating Check expression in Appendix A. In Chapter 5 we present a complete example of the staged configuration and derivation of Smart-Home systems of our MD-SPL. This example includes examples of the generated Check expressions for our constraint models.

4.5 Core Assets Development and Product Derivation

We have introduced metamodels and feature models as the core assets to express variability and configure products. Similar to many other MD-SPL approaches, we use model transformation rules as the main core assets to derive product line members.

In the next two subsections we present (1) the transformation rules we have created for our case study and (2) the mechanism we use to create decision models, *i.e.*, models where we relate the created transformation rules to features configurations and we define the required execution ordering of such

transformation rules to derive configured products.

4.5.1 Rule Transformations in the Smart-Home systems' SPL

As we introduced before (see Figure 4.1 and Figure 4.11), the rule transformations we have created for our case study are used in four stages. The first set of rules is defined from the domain metamodel to the facilities metamodel. They are created taking into account the facilities feature model. The second set is defined from the facilities metamodel to the components metamodel. The third set is defined from the components metamodel to the architecture metamodel. These, in turn, are created taking into account the architecture feature model. Finally, the fourth set of rule transformations includes model-to-text transformations which produce the source code of product line members.

First Stage: Domain-to-Facilities Transformation Rules.

The purpose of these transformation rules is adding to domain models information about Smart-Homes' facilities. These are horizontal model-to-model transformations. It means, they transform models inside the same abstraction level, the application domain abstraction level, but adding concerns related to Smart-Homes' facilities.

In this stage we create two sets of transformation rules: the *base* and the *specific* ones. On the one hand, base transformation rules do not depend of any variant of the product line. Thus, they are always executed during the transformation process. For instance, we create a base transformation rule to transform `DomainMetamodel::House` elements into `FacilitiesMetamodel::House` elements. Similarly, we create a base transformation rule to transform `DomainMetamodel::Floor` elements into `FacilitiesMetamodel::Floor` elements.

On the other hand, we create specific transformation rules taking into account the possible features which can affect the transformation process. In this case, those are features from the facilities feature model. For instance, we create two transformation rules to transform `DomainMetamodel::Window` elements. The first one, taking into account the `Automatic Windows` fea-

ture, creates `FacilitiesMetamodel::Automatic` (`Window`) elements and one `FacilitiesMetamodel::WindowsController` element for each created `Room` element. The second one, taking into account the `Air Conditioning` feature, creates `FacilitiesMetamodel::Manual` (`Window`) elements and one `FacilitiesMetamodel::AirConditioning` element for each created `Room` element. Therefore, if the feature `Automatic Windows` is selected the first transformation rule must be executed; if the feature `Air Conditioning` is selected the second transformation rule must be executed.

Similarly, we create two different transformation rules to transform `DomainMetamodel::Door` elements. The first one creates `FacilitiesMetamodel::Door` elements containing each one a `DomainMetamodel::Fingerprint` element; the second one creates `FacilitiesMetamodel::Door` elements containing each one a `DomainMetamodel::Keypad` element. The model-to-model and the model-to-text transformation rules we have created for our case study are available in the web-site of our research group under the link MD-SPL Engineering [Sof09].

Second Stage: Facilities-to-Components Transformation Rules.

The second set of transformation rules are defined from the facilities metamodel to the components metamodel. These are vertical model-to-model transformations since they transform models between different abstraction levels. The source abstraction level is the application domain abstraction level, the target one is the abstraction level including concerns related to software components.

We create only base transformation rules given that there are no feature models affecting this transformation stage. However, in this particular case, not all the base transformation rules are always executed. Their execution depends of the facility models to be transformed. For instance, only if exists at least one `FacilitiesMetamodel::WindowsController` element, then a base transformation rule in charge of creating a component which serves as controller for the automatic windows is executed.

Figure 4.17 presents an example of a derived component model. This model is presented using the UML2 syntax. `Periodic` components in this model can (or cannot) become `Periodic` components after the next transformation stage, which creates architecture models. The components inside dashed

squares are not always created. The conditions to create such components are the following:

- The rule to create the `WindowController` component, its ports and interfaces, is only executed if exists at least one `FacilitiesMetamodel::WindowsController` element in the source model.
- The rule to create the `AirConditioningController` component, its ports and interfaces, is only executed if exist at least one `FacilitiesMetamodel::AirConditioning` element in the source model.

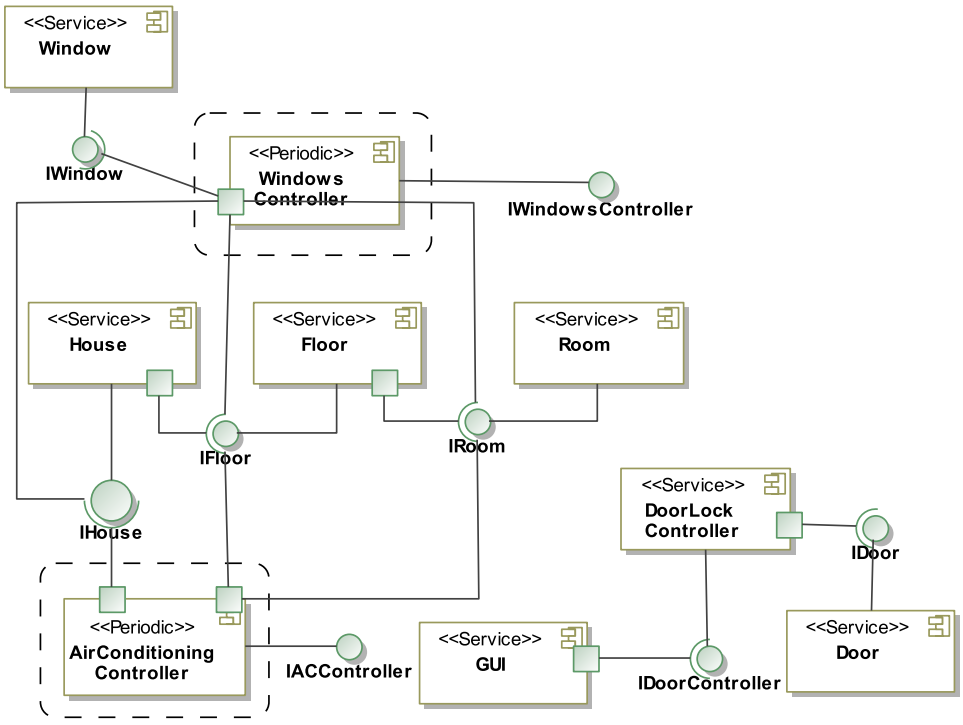

Figure 4.17: Example of a Smart-Home Systems' Components Model.

The `GUI` component corresponds to the Graphical User Interface (GUI) of the Smart-Home systems. This component requires services of all the other components. In this figure we only include one of its `Required` interfaces, `IDoorController`. In Chapter 5 we present a complete example of the staged

configuration and derivation of Smart-Home systems of our MD-SPL. This example includes the description of a particular component model derived for a particular configuration.

Third Stage: Components-to-Architecture Transformation Rules.

The purpose of these transformation rules is adding to component models information about the type of the components, *periodic* or *service*, and their instantiation mode, *on invocation* or *on deployment*. These are horizontal model-to-model transformations given that models are transformed inside the same abstraction level.

In this stage we create base and specific transformation rules. For instance, we create a base transformation rule to transform `ComponentMetamodel::Interface` elements into `ArchitectureMetamodel::Interface` elements.

We create specific transformation rules having into account the possible features which can affect the transformation process. In this case, those are features from the architecture feature model. For instance, we create two transformation rules to transform `ComponentMetamodel::Component` elements. The first one, having into account the `Service` feature, creates `ArchitectureMetamodel::Service` (Component) elements. The second one, having into account the `Periodic` feature, creates `ArchitectureMetamodel::Periodic` (Component) elements from `ComponentMetamodel::Periodic` elements. Therefore, if the feature `Service` is selected the first transformation rule must be executed; if the feature `Periodic` is selected the second transformation rule must be executed.

Similarly, we create two different specific transformation rules to transform `DomainMetamodel::Door` elements. The first one creates `FacilitiesMetamodel::Door` elements containing each one a `DomainMetamodel::Fingerprint` element; the second one creates `FacilitiesMetamodel::Door` elements containing each one a `DomainMetamodel::Keypad` element.

Fourth Stage: Model-to-Text Transformation Rules.

The model-to-text transformation rules produce the source code of prod-

uct line members. These transformation rules have as input an architecture model and a facilities model. On the one hand, the architecture model is transformed into the source code of OSGi components (Bundles) as the presented in Figure 4.17. For this transformation we reuse pieces of code already written. Thus, the transformation rules are only in charge of connecting the already created pieces of code representing components.

On the other hand, the facilities model is transformed into an extra OSGi component, *HouseStructure*, which manages the structural design of the configured Smart-Home. Thus, if the Smart-Home has been configured to have one floor and two rooms, the *HouseStructure* component maintains this structure to provide the required services to the configured structural element. These model-to-text transformation rules are available along with the model-to-model transformation rules in the web-site of our research group under the link MD-SPL Engineering [Sof09].

Figure 4.18 presents an example of the Graphical User Interface corresponding to one configured Smart-Home System. The Smart-Home system was configured to have one floor with one room, the `Main Room`. This room has `Automatic Windows` as `Environmental Control`. The only door in the `Main Room` has `Fingerprint` as `Door Lock Control`.

4.5.2 Creating and Using Decision Models

In the previous section we explained that we create *specific* transformation rules having into account the possible features which can affect a model transformation stage. For instance, in the first transformation stage those are features from the facilities feature model.

Given that one objective of our MD-SPL approach is to automate completely the process of transforming models, we have the need of using a mechanism which allows selecting and executing automatically the base transformation rules and only some specific transformation rules. These are specific rules related to selected features in feature configurations. This mechanism must also ensure the *correct* execution ordering, also called execution scheduling, of the selected transformation rules. By *correct* we mean an execution ordering which allows deriving the desired configured product.

We propose the use of explicit decision models in the context of MDD as

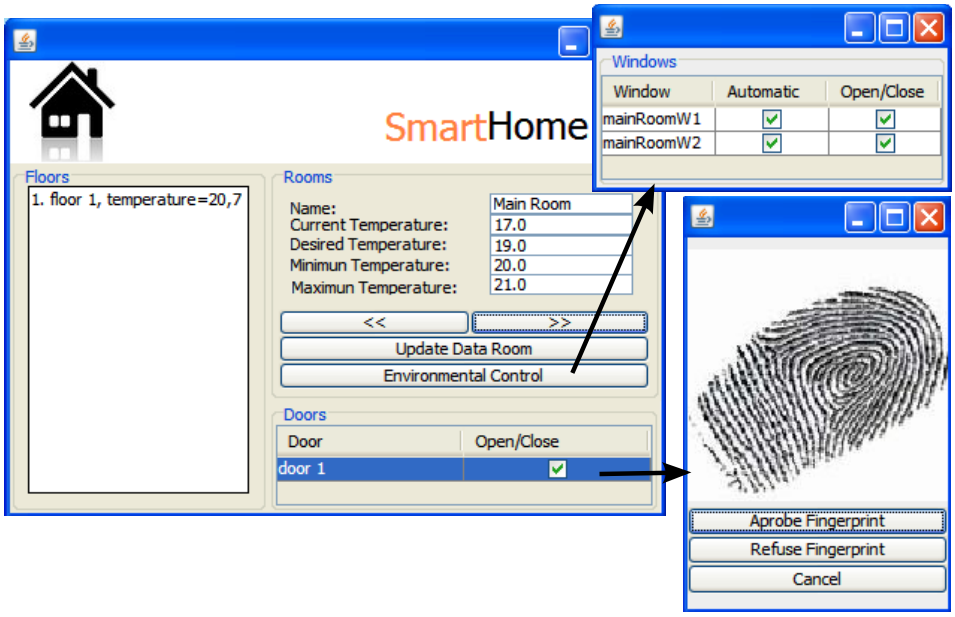

Figure 4.18: Example of a Smart-Home System.

a mechanism for composition of transformation rules based on feature configurations [ARCR09]. This mechanism can be used in conjunction with transformation languages which provide facilities for composition of transformation rules. In particular we used the oAW modeling framework and the Xtend and Xpand model transformation languages, which provide a mechanism based on Aspect-Oriented Programming (AOP) for composition of transformation rules (see Section 2.5.2).

Our decision models are useful to capture (1) the relationships between features and specific transformation rules, and (2) the required execution ordering of transformation rules to create products based on feature configurations. Our basic idea to obtain a final execution scheduling is to construct a base-line ordering, which is modified according to valid feature configurations. A base-line ordering describes a sequence of calls to base transformation rules. Our mechanism to adapt the base-line ordering is supported by AOP ideas. We capture in decision models information about *aspects* that must be woven with a base-line ordering to adapt it. Aspects maintain the information of what base transformation rules must be intercepted (*joinpoints*) and what specific transformation rules must then be executed (*advices*) according to

Table 4.1: Examples of Conditions on Feature Configurations which Imply to Adapt a Base-Line (Transformation Rules') Ordering

Condition	**Joinpoint**	**Advice**
Feature One Selected	Rule A	Rule A'
Feature Two Unselected	Rule B	Rule B'
Feature One Unselected and Feature Three Selected	Rule A	Rule C

defined *conditions* on feature configurations.

Table 4.1 presents examples of conditions on feature configurations that we can capture in our decision models. These conditions imply modifying a base-line ordering. In the first column we present examples of conditions, in the second column we present the name of the base rule in the base-line ordering to be intercepted (joinpoint), in the third column we present the name of the specific rule (advice) to be executed if the condition appears in a feature configuration. Thus, if the `Feature One` appears `Selected` in a feature configuration, no matter the other features, the `Rule A` must be intercepted and the `Rule A'` must be executed instead. If the `Feature Two` appears `Unselected` in a feature configuration, no matter the other features, the `Rule B` must be intercepted and the `Rule B'` must be executed instead. We can also capture more complex conditions. For instance, in row three, we express that if the `Feature One` appears `Unselected` and the `Feature Three` appears `Selected` in a feature configuration, no matter the other features, the `Rule A` must be intercepted and the `Rule C` must be executed instead.

For example in the context of our case study, during the derivation of a Smart-Home system, if the feature `Automatic Windows` is selected in a feature configuration, the base sequence to transform domain models into facilities models must be modified. This modification is done in a defined point to include an alternative step where the transformation rule in charge of creating automatic windows is called. Figure 4.19 presents a small part of our decision model to transform domain models into facilities models. We first define a base-line ordering which includes the execution of the transformation rules `domainFloorsToFacilitiesFloors`

and `domainWindowsToFacilitiesWindows`. We after create an aspect which indicates that if the feature `Automatic Windows` is selected in a feature configuration, the execution of the base transformation rule `domainWindowsToFacilitiesWindows` must be intercepted and the specific transformation rule `windowsToAutomaticWindows` must be then executed.

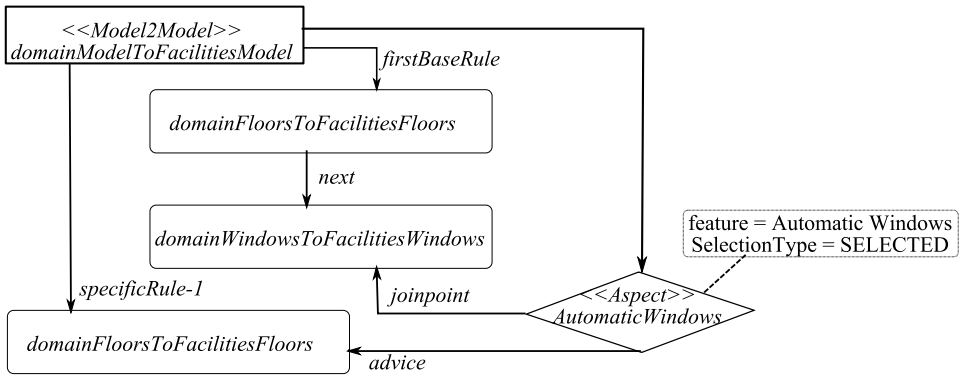

Figure 4.19: Example of a Decision Model to Create Smart-Home Systems.

Our decision models also allow us to capture the different transformation stages included in a product line derivation process. For our case study these are four transformation stages, from domain models until to obtain Java source code. This type of composition, which compose transformation rules using the output model of a rule as the input model of another rule is called external composition [Wag08a]. Figure 4.20 presents the part of our decision model capturing the external composition required for deriving Smart-Home systems given our four transformation stages. We create this model using the decision model editor we present in Chapter 5. In Section 4.7 we discuss limitations of our mechanism to derive products based on decision models.

The Decision Metamodel.

Figure 4.21 presents the decision metamodel we have created to create decision models. A model transformation `Workflow` contains a sequence of `TransformationPrograms`. A `TransformationProgram` is either a `Model2Model` or a `Model2Text` transformation. Each `TransformationProgram` uses a set of `TransformationRules` and a set of `Aspects` to perform its process of transformation. As introduced before, we classify `TransformationRules` in `Base` and `Specific` ones. An `Aspect` spec-

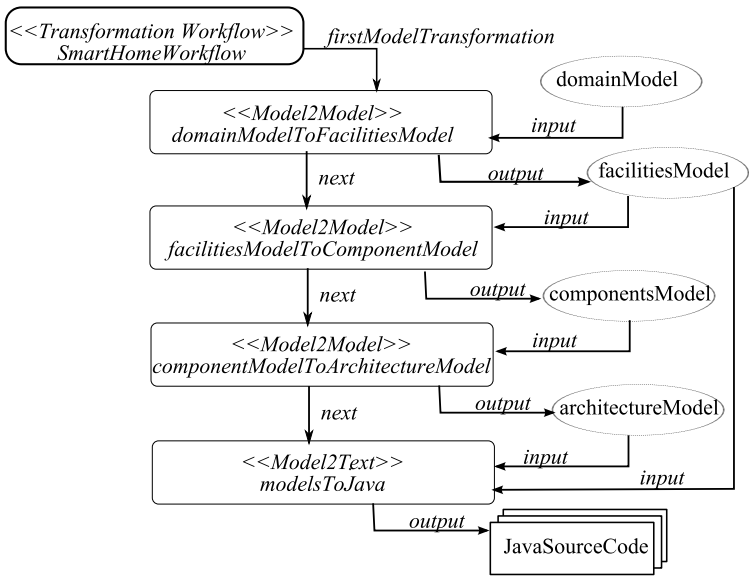

Figure 4.20: Decision Model including External Composition.

ifies its `advice`, which is a `Specific` transformation rule, and its joinpoint, which in turn is a `Base` transformation rule. A `Workflow` must have into account a set of `ExecutionConditions`, which depends of a set of `Features` with a particular `SelectionType`, `SELECTED` or `NOT_ SELECTED`. Finally, an `Aspect` must be woven if its `executionCondition` appears in a feature configuration.

Creating Executable Model Transformation Workflows from Decision Models.

As we mentioned before, we use the oAW modelling framework and the Xtend and Xpand model transformation languages to implement our approach and case study. We then had the need of transforming our decision models into oAW workflows which include the required instructions (1) to execute model transformations in different stages (external composition), and modifying a base-line ordering of a set of transformation rules (internal composition [Wag08a]). For transforming our decision models into oAW workflows we created a model-to-text transformation. This model-to-text transformation is presented in Appendix A. We use this model-to-text transformation in the decision models editor (Chapter 5) to provide the facility of

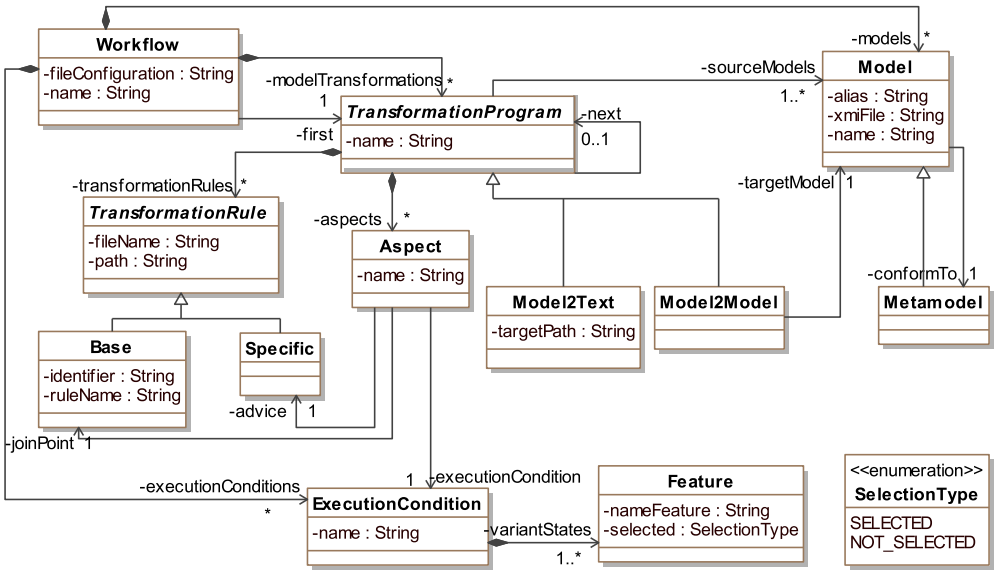

Figure 4.21: Decision Metamodel.

transforming decision models into executable oAW workflows.

Listing 4.3 presents a part of a sample generated oAW workflow. This workflow specifies that the transformation rule `domainWindowsToFacilitiesWindows` is intercepted (line 2-3 and line 9) and the transformation rule `adviceWindowsToAutomaticWindows` is executed (line 5) if the feature `Automatic Windows` is selected in a feature configuration (line 1).

4.6 Deriving Products based on Constraint Models and Binding Models

In Section 4.5.2 we introduced decision models in the context of MDD as our mechanism for composition of transformation rules based on feature configurations. We discussed how our decision models are useful to capture (1) the relationships between features and specific transformation rules, and (2) the required execution ordering of transformation rules to create products based

```
1
2  <Feature selected="Automatic Windows">
3    <transformationAspect adviceTarget=
4                          "domainWindowsToFacilitiesWindows">
5      <extensionAdvice
6        value="adviceWindowsToAutomaticWindows" />
7    </transformationAspect>
8  </Feature>
9
10 <transform id="domainWindowsToFacilitiesWindows">
11   <invoke value="domainWindowsToFacilitiesWindows" />
12 </transform>
```

Listing 4.3: Example of a Generated oAW Workflow.

on feature configurations.

Our basic idea to obtain a final execution scheduling was to construct a baseline ordering, which is modified according to valid feature configurations. Thus for example, during the derivation of a Smart-Home system, if the feature `Automatic Windows` was selected in a feature configuration, the base sequence to transform domain models into facilities models was modified to replace the rule `domainWindowsToFacilitiesWindows` by the rule `windowsToAutomaticWindows`.

Binding models imply to modify a base line ordering having into account not only features from feature configurations, but also bindings from binding models. Thus for example, if any `Window` element is bound to the feature `Automatic Windows` in a binding model, the base sequence to transform domain models into facilities models must be modified. This modification implies to replace the rule `domainWindowsToFacilitiesWindows` by the rule `particularWindowsToAutomaticWindows`. This rule must transform only the `DomainMetamodel::Window` elements, which are bound to the `Automatic Windows` feature, into `FacilitiesMetamodel:Automatic` window elements. For instance, from the binding model presented in Figure 4.13, given that the `mainRoomW2` is the only window bound to the feature `Automatic Windows`, this is the only window that must be transformed into an automatic window.

Table 4.2 presents examples of conditions on binding models that we can

Table 4.2: Examples of Fine-Grained Conditions on Feature Configurations which Imply to Adapt a Base-Line (Transformation Rules') Ordering

Condition	**Joinpoint**	**Advice**
Exists at least one binding $B_1 = [E_1, F_1]$ that satisfies the constraint $C_1 = [M_1, F_1, A, D]$	Rule A	Rule A'(E_1)
Feature Two Unselected and exists at least one binding $B_2 = [E_2, F_2]$ that satisfies the constraint $C_2 = [M_2, F_2, A, D]$	Rule B	Rule B'(E_2)

capture in our extended decision models. These conditions imply modifying a base-line ordering. In the first column we present examples of conditions, in the second column we present the name of the base rule in the base-line ordering to be intercepted (joinpoint), in the third column we present the name of the specific rule (advice) to be executed if the condition appears in a binding model. We express conditions in terms of bindings that satisfy constraints. Thus, row one in Table 4.2 expresses that if exists at least one binding $B_1 = [E_1, F_1]$ that satisfies the constraint $C_1 = [M_1, F_1, A, D]$ in a binding model, the `Rule A` must be intercepted and the `Rule A'` must be executed instead using E_1 as parameter. We can also capture conditions which have into account not only bindings but also selection of features. For instance, in row two, we express that if the `Feature Two` appears `Unselected` and exists at least one binding $B_2 = [E_2, F_2]$ that satisfies the constraint $C_2 = [M_2, F_2, A, D]$ in a binding model, the `Rule B` must be intercepted and the `Rule B'` must be executed instead using E_2 as parameter.

Figure 4.22 presents a small part of our decision model to transform domain models into facilities models having into account binding models. Similarly as presented before in Section 4.5.2, we first define a base-line ordering which includes the execution of the transformation rules `domainFloorsToFacilitiesFloors` and `domainWindowsToFacilitiesWindows`. We after create an aspect indicating that if exist bindings satisfying the `constraint1`, which describes that product designers can bind `Window` elements with the feature `Automatic Windows`, the execution of the base transformation rule `domainWindowsToFacilitiesWindows` must be inter-

cepted. After the interception is done, the specific transformation rule `particularWindowsToAutomaticWindows` must be then executed. This rule queries the binding model used to configure the product which is derived, and transforms only the `Window` elements bound to the `Automatic Windows` feature. In Section 4.7 we discuss limitations of our mechanism to derive products based on decision models.

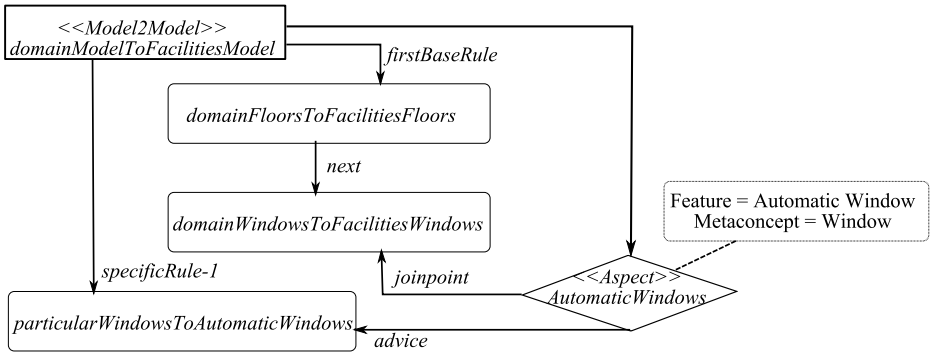

Figure 4.22: Example of a Decision Model to create Smart-Home Systems Having into Account Binding Models.

4.6.1 The Extended Decision Metamodel.

We extended the decision metamodel that we presented before in Figure 4.21. Figure 4.23 presents our decision metamodel which allows us for deriving products having into account binding models.

We still include the concepts of `Workflow`, `TransformationProgram`, `TransformationRule` and `Aspect`. We modify, however, the concept of `ExecutionCondition`. In this extended decision metamodel an `ExecutionCondition` depends of a set of `Variants`, which we specialized in `CoarseCondition` and `FineCondition`. A `CoarseCondition` represents a feature that can be `SELECTED/NOT_SELECTED`. A `FineCondition` represents a constraint.

Thus, based on a binding model, we can indicate that a `Specific` transformation rule must be woven with a `Base` transformation rule when exist bindings that satisfy the constraint denoted by a `FineCondition` element. For instance, we can indicate that a specific transformation rule must be

woven with a base transformation rule when the feature `Air Conditioning` appears bound to a `Window` element.

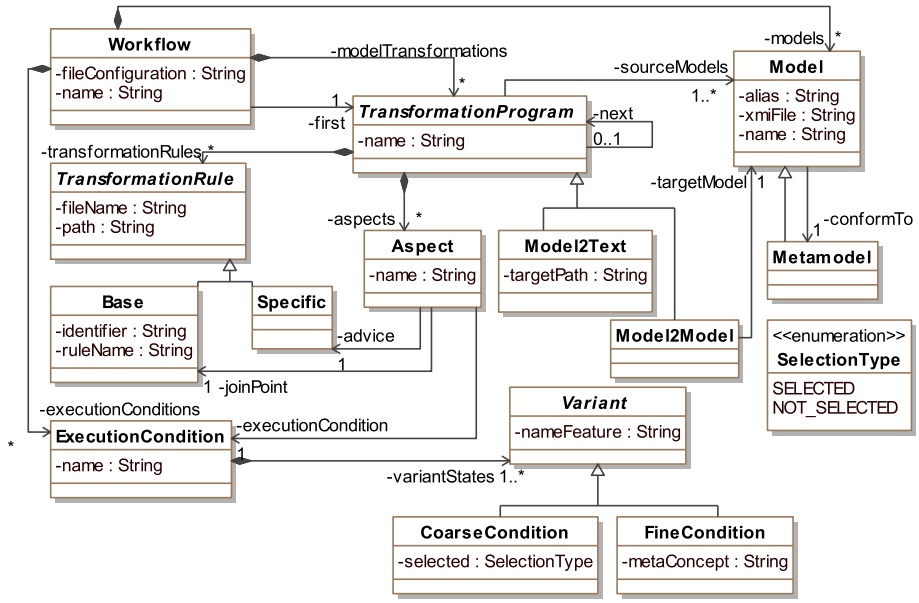

Figure 4.23: Decision Metamodel Having into Account Binding Models.

4.6.2 Creating Executable Model Transformation Workflows from Decision Models and Constraint Models.

As we presented before, we transform our decision models into oAW workflows which include the required instructions to (1) execute model transformations in different stages (external composition), and modifying a base-line ordering of a set of transformation rules (internal composition).

For transforming our decision models into oAW workflows having into account constraint models and binding models, we modified the model-to-text transformation we introduce before in Section 4.5.2. This model-to-text transformation allows us for generating executable oAW from decision models where we indicate that binding models must by query locking for

```
1
2  <fineFeature toFeature="Automatic" boundMetaconcept="Window">
   <transformationAspect adviceTarget=
3                          "domainWindowsToFacilitiesWindows">
4      <extensionAdvice
5        value="particularWindowsToAutomaticWindows" />
6    </transformationAspect>
7  </fineFeature>
8
9  <transform id="domainWindowsToFacilitiesWindows">
10   <invoke value="domainWindowsToFacilitiesWindows" />
11 </transform>
```

Listing 4.4: Example of a Generated oAW Workflow.

bindings that satisfy particular constraints. Listing 4.4 presents a part of a generated oAW workflow. This workflow specifies that the transformation rule `domainWindowsToFacilitiesWindows` is intercepted (line 2-3 and line 9) and the transformation rule `particularWindowsToAutomaticWindows` is executed (line 5) if exist bindings that satisfy the constraint created between the `Automatic Windows` feature and the `Window` metaconcept (line 1).

We created the oAW component which allows us for querying binding models locking for bindings that satisfy a particular constraint. The line 1 from Listing 4.4 shows a call to our oAW component. The component queries a binding model which has been previously loaded in the execution context of an oAW workflow. The model-to-text transformation we created to generate oAW workflows from decision models is available in Appendix A.

4.7 Identified Limitations

In previous sections we discussed how our decision models are useful to capture (1) the relationships between features and/or bindings, and specific transnformation rules, and (2) the required execution ordering of transformation rules to create products based on feature configurations and/or binding models. Our idea to obtain a final execution scheduling was to construct a baseline ordering, which is modified according to execution conditions defined

in terms of feature configurations and/or binding models.

We have identified at least three limitations in our strategy of relating execution conditions to specific transformation rules. Two of them occur when conditions only take into account feature configurations (see Table 4.1). The other one occurs when conditions take into account not only feature configurations but also binding models (see Table 4.2).

Features Combinatory.

The first limitation of our approximation lies in the fact that the number of valid feature configurations that can be created based on one feature model is big.

In our current approach we do not include mechanisms to guarantee neither that for all possible valid feature configurations there is a set of transformation rules in charge of generating a runnable product, nor that in our decision models we include execution conditions that take into account each valid feature configuration. Currently, this is a responsibility of product line architects.

Features Interaction.

A feature interaction occurs when a feature modifies or influences another feature in defining overall system behaviour [CKMM03]. For example, assume a feature model including three features, A, B and C. If the feature A interacts with the features B and C, the selection in a feature configuration of A along with B will imply to adapt a base line ordering of transformation rules, the selection of A along with C will imply a different adaptation, and, it will be required another adaptation when only the feature A is selected.

The problem of dealing with feature interactions is an important problem which currently deserves special attention in the field of feature modeling [Rei09].

In our current approach, we take into account that the presence of one particular feature in different valid feature configurations may imply different adaptations of a baseline ordering of transformation rules. For instance, in Table 4.1 the presence of the `Feature One` in two different possible feature configurations implies a different adaptation. In row one we specify that if the `Feature One` appears `Selected` in a feature configuration, no matter

the other features, the `Rule A` must be intercepted and the `Rule A'` must be executed instead. In row three, we express that if the `Feature One` appears `Unselected` and the `Feature Three` appears `Selected` in a feature configuration, no matter the other features, the `Rule A` must be intercepted and the `Rule C` must be executed instead.

Nevertheless, it is responsibility of product line architects (1) to identify feature interactions, (2) to define transformation rules for the different scenarios derived from feature interactions, (3) to define execution conditions for such scenarios and (4) to create and relate transformations rules to the defined execution conditions. Our approach does not provide mechanisms to validate that all possible feature interactions are taken into account.

Bindings Interaction.

When conditions take into account binding models (see Table 4.2), our approach allows product line architects to create decision models where decisions consider bindings satisfying only one constraint. For instance, row one in Table 4.2 expresses that if exists at least one binding $B_1 = [E_1, F_1]$ that satisfies the constraint $C_1 = [M_1, F_1, A, D]$ in a binding model, the `Rule A` must be intercepted and the `Rule A'` must be executed instead using E_1 as parameter. In this case we only consider bindings satisfying one constraint, C_1.

To understand why we cannot consider bindings that satisfy more than one constraint, please assume the following scenario. Suppose we have a condition expressing that if exists in a binding model at least one binding $B_1 = [E_1, F_1]$ that satisfies the constraint $C_1 = [M_1, F_1, A, D]$ and at least one binding $B_2 = [E_2, F_2]$ that satisfies the constraint $C_2 = [M_2, F_2, A, D]$, then the `Rule B` must be intercepted and the `Rule B'` must be executed instead using E_1 and E_2 as parameter. Now suppose we have a binding model with two bindings that satisfy C_1, $B_1 = [E_1, F_1]$ and $B_{1'} = [E_{1'}, F_1]$, and two bindings that satisfy C_2, $B_2 = [E_2, F_2]$ and $B_{2'} = [E_{2'}, F_2]$. In this case, it is not possible to know the ordering of the parameters to execute the `Rule B'`. It means, we are not able to know if we must invoke `Rule B'`(E_1, E_2), `Rule B'`$(E_{1'}, E_2)$ or `Rule B'`$(E_1, E_{2'})$.

Therefore, if for each condition we consider bindings satisfying several constraints, we cannot guarantee that the specific rules (advices) will be executed with the suitable parameters.

4.8 Summary

In this chapter we first introduced a case study of a Smart-Home systems' SPL. We used this case study through the whole chapter for explaining the mechanism we used to create MD-SPLs. We presented the base mechanisms we use in the processes of (1) expressing variability and configuring products, and (2) deriving configured products. These mechanisms included the use of metamodels and feature models for expressing variability and configuring products, and decision models to derive product line members. We presented the MDD mechanisms we propose in this thesis to improve the creation of MD-SPLs. These mechanisms include the use of constraint models which include the cardinality and structural properties, binding models and more expressive decision models. We have also described the metamodels we created to support the creation of constraint, binding and decision models. We discussed our general strategy for validating binding models against constraint models and for generating executable model transformation workflows from decision models, which allow us to derive product line members. Finally, we presented limitations of our approach for deriving products based on decision models.

Chapter 5

Validation and Tool Support

5.1 Introduction

Section 5.2 presents a validation of FieSta, our MD-SPL approach, using examples of products that we are able to derive by means of our MD-SPL mechanisms. We present results of configuring and deriving products of two MD-SPLs.

Then, we present our implementation strategy for FieSta. The implementation strategy defines the general process for the implementation of our MD-SPL engineering mechanisms for creating product lines. Our implementation strategy includes (1) the required activities to create products, and (2) the tools we have created to support these activities. We present the tool support for expressing variability and configuring products, Section 5.3, and the tool support for deriving configured products, Section 5.4.

Our tool support assists product line architects and product designers during the whole development lifecycle of MD-SPLs. We provide Eclipse plug-ins to create MD-SPL projects, feature models, constraint models, binding models, OCL-type expressions to validate binding models against constraint models, and decision models. We also provide openArchitectureWare (oAW) components to facilitate the processing of binding models and decision models to derive products. The entire FieSta toolkit, the instructions for installing it and two case studies, can be found in the web-site of our research group

under the link MD-SPL Engineering [Sof09].

For the implementation of our tool support we chose EMF as modeling framework, which means we express all our metamodels based on the Ecore metametamodel (see Section 2). We opted to use oAW as our model transformation engine. We selected oAW because, as presented before in Section 2, this is a complete MDD framework integrated with Eclipse that makes the reading, instantiation, checking, and transformation of models possible. oAW has been used successfully to create SPLs, and there is an active community of SPL and MDD developers using and improving it.

The UML activity diagram in Figure 5.1 presents the general overview of the implementation strategy for FieSta. *Domain engineering* and *application engineering* organize the activities. For domain engineering, we built tools to support product line architects in the creation of a special type of Eclipse projects, MD-SPL projects. An `MD-SPL project` includes the required oAW and EMF dependencies to create MD-SPLs and predefine a hierarchical folders structure to manage and centralize the core assets used to derive products. Then, architects can create and manage in a common repository domain `metamodels`, `feature models` and `constraint models`, which capture and express the possible fine-grained variations affecting the product line. Product line architects also create `transformations rules` and `decision models`, which are transformed into (executable) `model transformation workflows`.

During `Application Engineering` product designers use the variability identified and the core assets created during `Domain Engineering`, `metamodels`, `feature models`, `models` and `model transformation workflows`, to ensure the correct derivation of desired products. Product designers create domain `models` and `binding models`, which must satisfy the `constraint models` created before, to configure and derive products. Finally, designers execute the generated `model transformation workflows` using domain `models` and `binding models` as inputs, and `transformation` rules for processing the inputs.

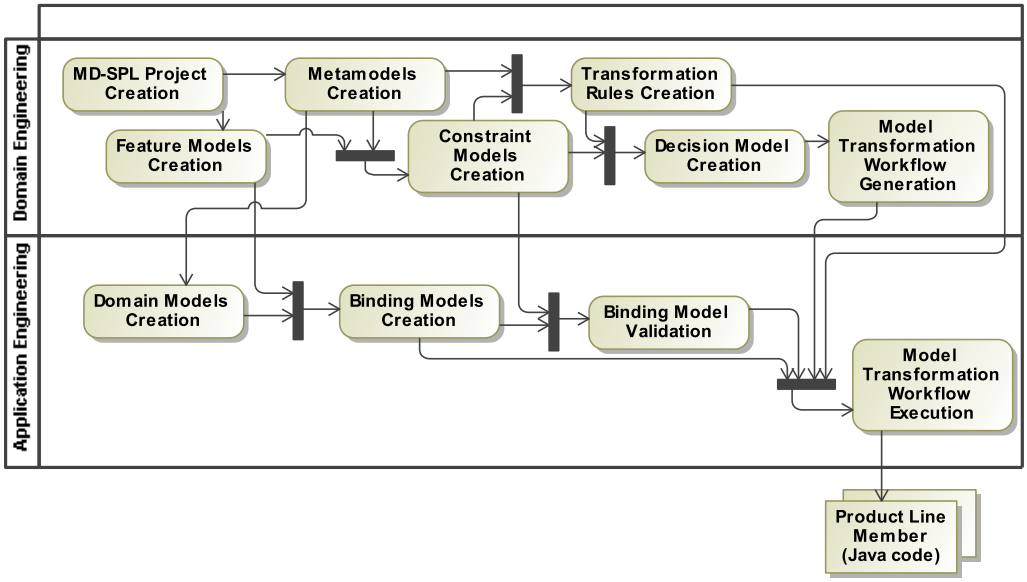

Figure 5.1: Overview of Our Implementation Strategy to Create MD-SPLs.

5.2 Running MD-SPLs

5.2.1 The Smart-Home Systems' SPL

The SPL we use through this chapter is the SPL of Smart-Home systems that was introduced in Chapter 4. We present examples of diverse Smart-Home systems which can be derived from the `Small Building` in Figure 5.2. To derive such Smart-Home systems we reuse a common set of base and specific transformation rules that we developed as product line architects.

Figure 5.3 presents the stages to configure and derive products. To configure diverse Smart-Home systems, on the one hand, *facilities designers* have three features from the facilities feature model (see Figure 4.9): `Fingerprint`, `Keypad` and `Automatic Windows`. On the other hand, software architects two features from the architecture feature model (see Figure 4.10): `Periodic` and `Service`. We have created specific transformation rules for deriving products taking into account possible configurations the designers can create. For instance, we created one specific transformation rule for creating automatic

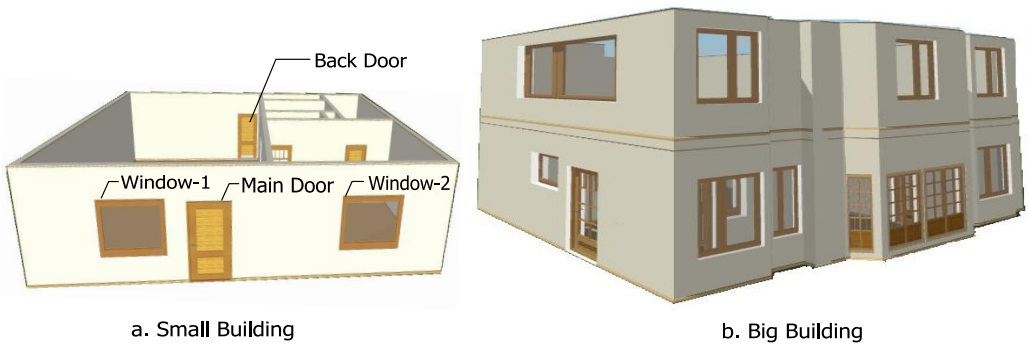

Figure 5.2: Examples of Buildings Created by Building Architects.

windows. This transformation rule is reused each time an automatic window must be created.

In the second configuration stage (see Figure 5.3), facilities designers relate facilities to structural elements of buildings. For example, the `Small Building` can be configured to use `Fingerprint` in the `Main Door` as lock door control and `Keypad` in the `Back Door`. Similarly, each window can be individually configured as `Automatic` or `Manual Window`. Table 5.1 presents the possible fine-grained configurations a facilities designer can create from the `Small Building` taking into account the variants `Fingerprint`, `Keypad` and `Automatic Windows`. These are sixteen (16) possible Smart-Home systems.

Table 5.2 presents the possible configurations a designer can create taking into account only coarse-grained variations, such as related approaches propose (see Section 3.6). In this case only four (4) possible Smart-Home systems can be configured.

In the third configuration stage (see Figure 5.3), software architects relate software architecture variants to model elements representing software components. Table 5.3 presents the possible fine-grained configurations a software architect can create for the Smart-Home system from row one in Table 5.1 (`SH-1`), taking into account the variants `Periodic` and `Service`. There are four possible Smart-Home systems that can be configured. Thus, from the `Small Building`, taking into account the variants `Fingerprint`, `Keypad` and `Automatic Windows`, and the variants `Periodic` and `Service`, product designers are able to configure sixty four (64) Smart-Home systems.

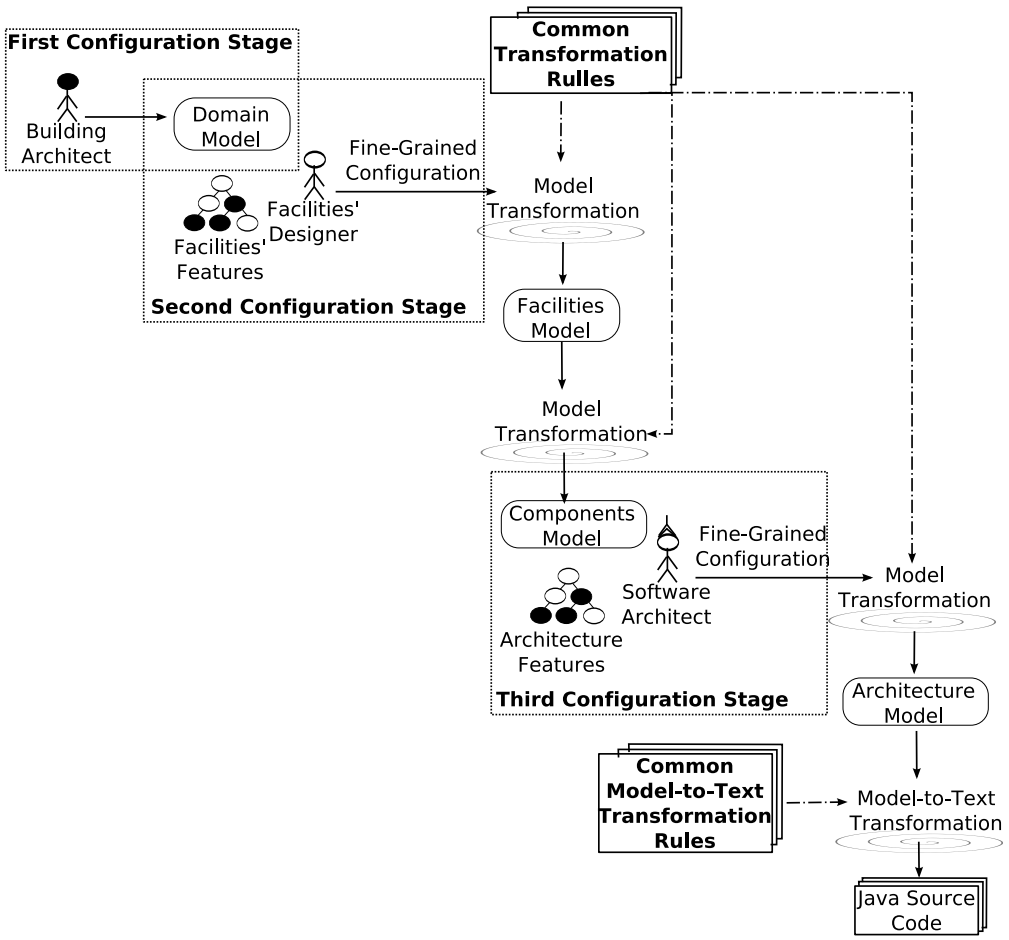

Figure 5.3: Stages to Configure and Derive Products.

Table 5.4 presents the possible configurations a software architect can create for the Smart-Home system from row one in Table 5.1 (`SH-1`), taking into account the variants `Periodic` and `Service`, but taking into account only coarse-grained variations. In this case, product designers only can configure two (2) different Smart-Home systems. Therefore, from the `Small Building`, taking into account the variants `Fingerprint`, `Keypad`, `Automatic Windows`, `Periodic` and `Service`, but considering only coarse-grained variations, product designers can configure only eight (8) Smart-Home systems.

Using this small example, we showed how the concept of fine-grained configuration allows product designers to extend the scope of MD-SPLs. From

Table 5.1: Example of a Fine-Grained Configuration for a Smart-Home System including Smart-Homes' Facilities

Smart-Home	Window-1	Window-2	Main Door	Back Door
SH-1	Automatic	Automatic	Keypad	Keypad
SH-2	Automatic	Automatic	Fingerprint	Fingerprint
SH-3	Automatic	Automatic	Keypad	Fingerprint
SH-4	Automatic	Automatic	Fingerprint	Keypad
SH-5	manual	manual	Keypad	Keypad
SH-6	manual	manual	Fingerprint	Fingerprint
SH-7	manual	manual	Keypad	Fingerprint
SH-8	manual	manual	Fingerprint	Keypad
SH-9	Automatic	manual	Keypad	Keypad
SH-10	Automatic	manual	Fingerprint	Fingerprint
SH-11	Automatic	manual	Keypad	Fingerprint
SH-12	Automatic	manual	Fingerprint	Keypad
SH-13	manual	Automatic	Keypad	Keypad
SH-14	manual	Automatic	Fingerprint	Fingerprint
SH-15	manual	Automatic	Keypad	Fingerprint
SH-16	manual	Automatic	Fingerprint	Keypad

Table 5.2: Example of a Coarse-Grained Configuration for a Smart-Home System including Smart-Homes' Facilities

Smart-Home	Window-1	Window-2	Main Door	Back Door
SH-1	Automatic	Automatic	Keypad	Keypad
SH-2	Automatic	Automatic	Fingerprint	Fingerprint
SH-3	manual	manual	Keypad	Keypad
SH-4	manual	manual	Fingerprint	Fingerprint

eight (8) Smart-Home systems that can be configured using coarse-grained configurations, we have shown how we can configure sixty four (64) Smart-Home systems using the concept of fine-grained configuration. These fine-grained configurations satisfy the constraints defined in the constraint models of our case study, which capture the possible variability of the MD-SPL.

Regarding the derivation of the configured products, we created transformation rules that guarantee we can generate valid products from the fine-grained configurations. We define a valid product as a runnable system that accomplish the requirements that product designer specify by means of fine-feature configurations, or binding models, which satisfy constraint models. How-

Table 5.3: Example of a Fine-Grained Configuration for a Smart-Home System including Software Components' Variants

Smart-Home	Window-1	Window-2	Main Door	Back Door	Windows Controller Component	Doors Lock Controller Component
SH-1.1	Automatic	Automatic	Keypad	Keypad	Periodic	Periodic
SH-1.2	Automatic	Automatic	Keypad	Keypad	Service	Service
SH-1.3	Automatic	Automatic	Keypad	Keypad	Periodic	Service
SH-1.4	Automatic	Automatic	Keypad	Keypad	Service	Periodic

Table 5.4: Example of a Coarse-Grained Configuration for a Smart-Home System including Software Components' Variants

Smart-Home	Window-1	Window-2	Main Door	Back Door	Windows Controller Component	Doors Lock Controller Component
SH-1.1	Automatic	Automatic	Keypad	Keypad	Periodic	Periodic
SH-1.2	Automatic	Automatic	Keypad	Keypad	Service	Service

ever, taking into account the limitations presented in Section 4.7, it was our responsibility as product line architects to create the transformations rules. Our approach does not provide mechanisms to validate that exist transformation rules to guarantee that valid products are derived from fine-grained configurations.

Figure 5.4 and Figure 5.5 are examples of the Graphical User Interfaces corresponding to one (fine-grained) configured Smart-Home Systems we derived. The Smart-Home system was configured to have one floor with two rooms, the `Main Room` and the `Living Room`. Figure 5.4 presents the `Main Room` which has `Air Conditioning` as `Environmental Control`, and its door has `Fingerprint` as `Door Lock Control`. In this case the product was configured to have the `Air Conditioning Controller` (software) component as a `Service` component. That is the reason why the air conditioning must be turned on/off manually.

Figure 5.5 presents the `Living Room` which has `Automatic Windows` as `Environmental Control`, and its door has `Keypad` as `Door Lock Control`. The `Living Room` has three windows, two of them were (fine-grained) configured as `Automatic Windows`.

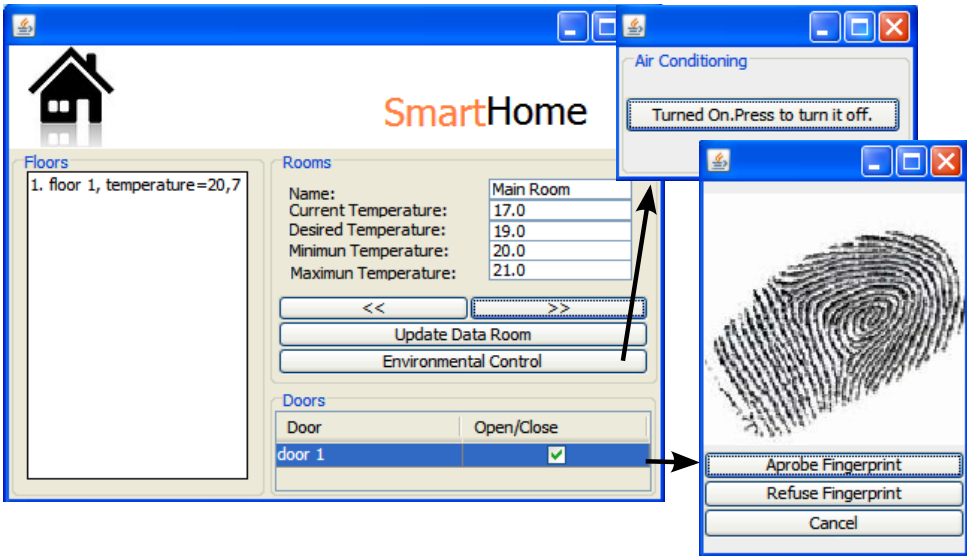

Figure 5.4: Example One of the GUI of a Fine-Grained Configured Smart-Home System.

Regarding the cost of production, the highest cost of producing members of the MD-SPL of Smart-Home systems is concentrated in the activities of core assets development (metamodels, feature models, transformation rules and decision models development), which are responsibility of product line architects who must be also MDD developers. However, we achieve a good return of investment since we obtain high quality in derived products and product designers invest few time configuring products. Given that the activities of products configuration are responsibility of several (specialized) product designers, *e.g.* building architects, facilities designers and software architects, designers are focus on particular concerns.

5.2.2 An MD-SPL of Stand Alone Applications to Manage Data Collections

Besides our Smart-Home systems' MD-SPL, we have also created a product line of stand-alone applications to manage data collections. We call *collection manager system* a product line member of this MD-SPL. For example, a collection manager system manages students from a school, and their personal

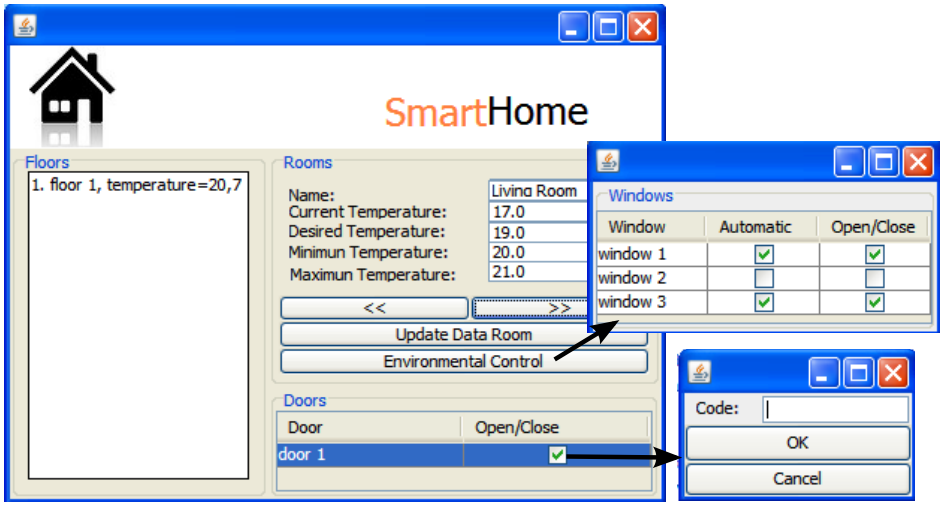

Figure 5.5: Example Two of the GUI of a Fine-Grained Configured Smart-Home System.

information: name, address, e-mail, etc. Another product manages discs in a music store, and their related information: name, artist, price, etc. At the architecture level, products are structured in two tiers: the kernel and the Graphical User Interface (GUI). The kernel tier implements functional requirements to add elements into the collection and to order the collection. The GUI tier implements visualization and interaction with the final users and the kernel component.

Kernel Commonalities. The kernel manages data associated to instances of a business logic concept such as *student* or *music store*. We use an aggregation structure to represent the business concept and its related attributes. For example, a *student* assembles the set of attributes *code*, *name*, *address*, and *e-mail*. Any modeled business concept has a *name* attribute and all the products of the product line have functionality for adding data.

GUI Commonalities. Graphical User Interfaces use elements like panels, lists, labels, and images, among others. All the GUI elements are grouped by different types of views. There are five types of views that are mandatory for any product: (1) *main*, (2) *list*, (3) *information*, (4) *order*, and (5) *creation* view. The *main* view is in charge of communicating the kernel and the GUI by grouping all the other views. The *list* view displays data related

to the *name* attribute of created instances of the business logic concept. The *information* view is used to show the data related to all the attributes of created instances of the business logic concept. The *order* view is used to select an attribute that will be used as reference for ordering the data displayed in the *information* and *list* views. The *creation* view is used to enter data for new instances of the business logic concept. Figure 5.6 presents the GUI of a product managing information of students.

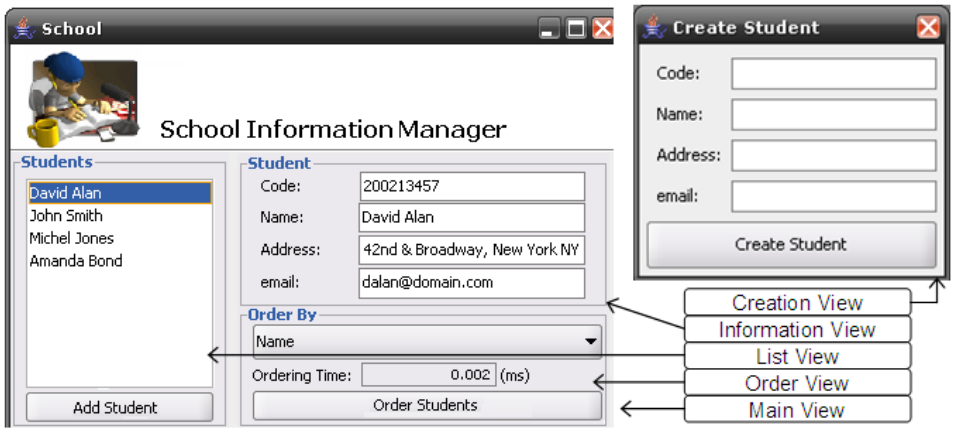

Figure 5.6: Graphical User Interface of a Collection Manager System.

Kernel Variability.

The most evident source of variation is the business concept and its attributes. As we presented before, products can be created to manage data such as students, music stores, or address books. A product may (or may not) provide functionality for ordering data. If it does, data can be ordered using either the *bubble* or *insertion* algorithms.

GUI Variability.

The user can select two different alternative views to present the data in the *information* view. The first one is a *simple* view with labels and text fields for each attribute related to the problem space concept managed by the product. Instances are displayed one-by-one such as presented in Figure 5.6. The second one uses a *grid* component. Grid component facilitates the display of many instances of the problem space concept at the same time.

Configuring and Deriving a Collection Manager System's Example.

Such as for the Smart-Home systems' MD-SPL, for the product line of stand-alone applications to manage data collections we use a staged process for capturing variability, configuring and deriving products.

In this MD-SPL the most evident source of variation is the business concept and its attributes. We have created a `Problem Space Metamodel` to capture this structural variability. Figure 5.7 presents this metamodel (left) and an example of a problem space model (right).

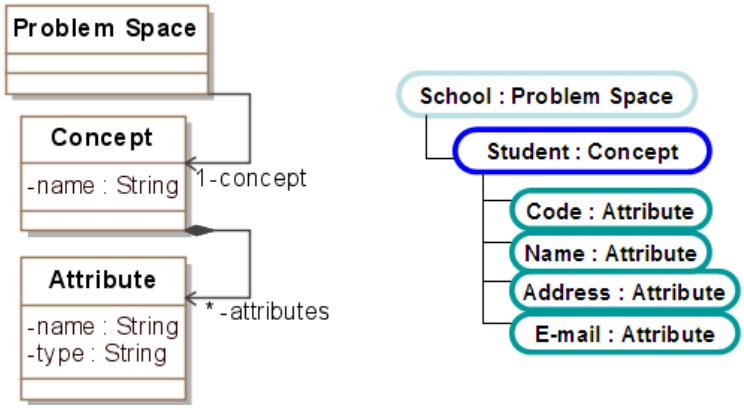

Figure 5.7: Problem Space Metamodel and Problem Space Model.

We capture in a feature model the variations regarding the type of algorithms that can be used to order data of our collections. Figure 5.8 presents part of the feature model including two variants, `Bubble` and `Insertion`.

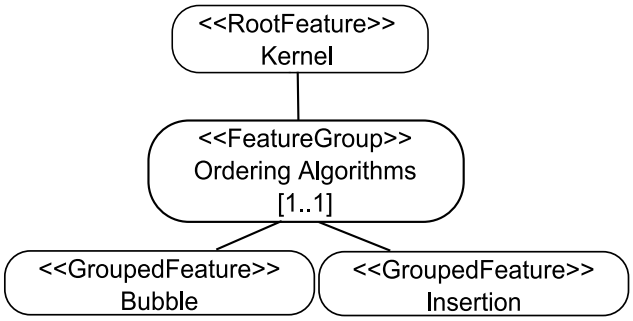

Figure 5.8: Feature Model for The Product Line of Stand-Alone Applications to Manage Data Collections.

Thus, using the problem space model from Figure 5.7 and the feature model from Figure 5.8, product designers are able to create fine-grained configurations where `Attributes` are related to the features `Bubble` or `Insertion`. For instance, if the attribute `Code` is related to the feature `Insertion`, data will be ordered by using the `Insertion` algorithm when the criteria for ordering is the attribute `Code`. Figure 5.9 presents a collection manager system that was configured to order data using the `Bubble` algorithm when the ordering criteria is the attribute `Name`.

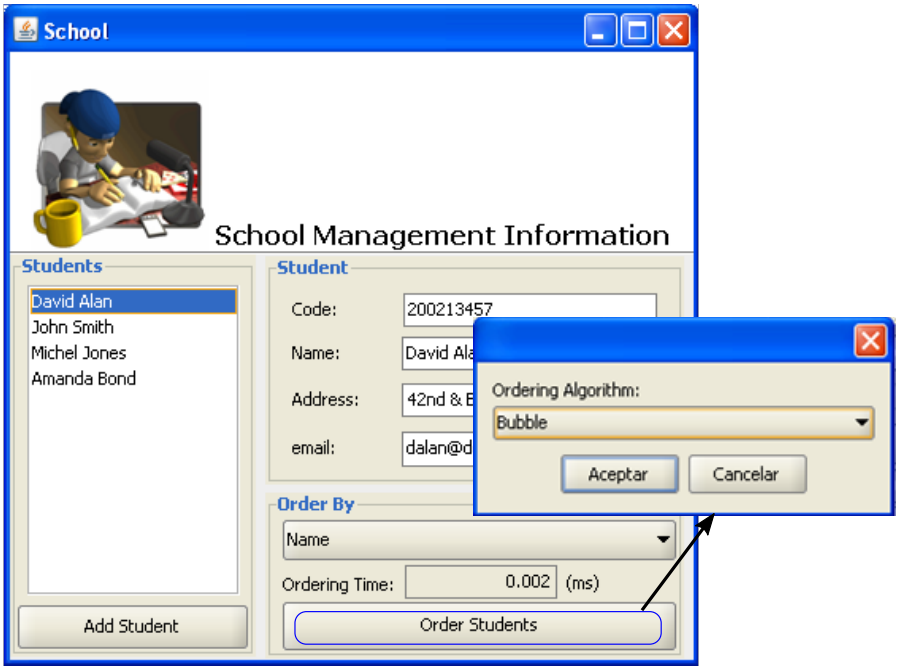

Figure 5.9: Graphical User Interface of a Fine-Grained Configured Collection Manager System.

This case study, including detailed documentation of metamodels, constraint models, decision models, and other core assets, is available along with the case study of the product line of Smart-Home systems at `http://qualdev.uniandes.edu.co/wikiMain/doku.php?id=projects:md-slp_engineering`.

5.3 Variability Expression and Product Configuration

5.3.1 MD-SPL Project Creation

We built an Eclipse plug-in that allows product line architects creating a particular type of Eclipse projects. This type of projects includes the required oAW and EMF dependencies to create MD-SPLs and predefine a hierarchical folders' structure to manage and centralize the core assets associated to an MD-SPL project. We named this plug-in the *(MD-SPL) Project Creator*.

Figure 5.10 presents on the left a screenshot of the Eclipse menu including the option to create MD-SPL projects. On the right Figure 5.10 presents the folders' structure of an empty MD-SPL project.

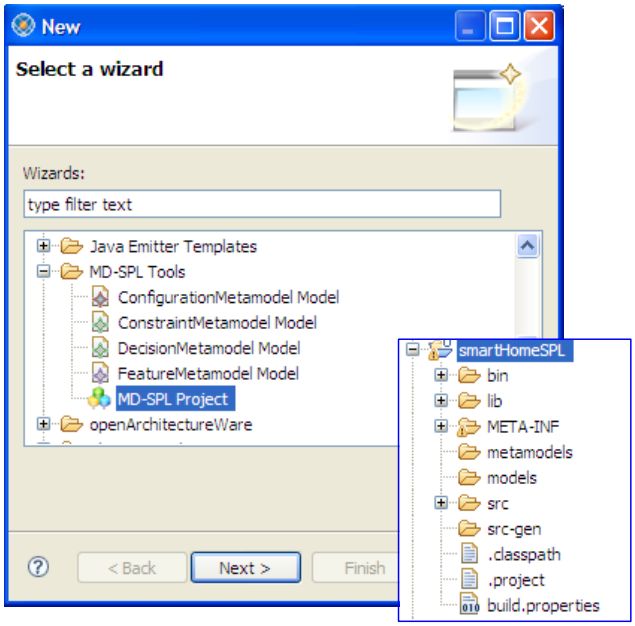

Figure 5.10: Screenshot of the Project Creator Plug-In.

5.3.2 Metamodels and Feature Models Creation

Metamodels Creation.

Once an MD-SPL project has been created product line architects can create metamodels and feature models.

Product line architects create metamodels by using MagicDraw [No 09], which is a UML2 modeling tool that allows us for creating UML Class Models and exporting them into UML2 XMI files. Thus, from the UML2 XMI files, product line architects generate Ecore models by using a component provided by oAW to transforms UML2 class models into Ecore models.

The MD-SPL projects we create by using our Project Creator plug-in include an oAW workflow file which invokes the oAW component in charge of transforming UML2 XMI files into Ecore models. To generate Ecore models, product line architects parameterize this oAW workflow file and then execute it to obtain the Ecore model. Therefore, we allow product line architects to create metamodels from a classic UML perspective, which facilitates the creation of domain metamodels.

Listing 5.1 presents an example of a parameterized oAW workflow file to generate Ecore models from UML2 XMI files. In line 3 we define the location of the UML2 model to be transformed. In line 4 we specify the target location of the resultant Ecore model. Line 5 to line 8 describe some additional properties required to perform the transformation.

The Feature Models Creator.

To create feature models we provide the *Feature Models Creator*, which is an Eclipse plug-in. We decided to create our own Feature Models Creator instead of using commercial tools such as pure::variants [Pur09] or tools which are under development and do not provide mature APIs such as fmp [AC04].

Our Feature Models Creator includes a facility for validation of feature models. This plug-in validates that (1) the lower bound of features' cardinality is minor or equal than the upper bound of features' cardinality and (2) solitary features have cardinality between zero and one, this is, the cardinality is *[0..1]* or *[1..1]*.

```
1
2  <cartridge
3     file="org/openarchitectureware/util/uml2ecore/
4  ____uml2ecoreWorkflow.oaw"
5     uml2ModelFile="..\uml2Models\domainMetamodel.uml2"
6        outputPath="..\ecoreModels\domainMetamodel.ecore"
7        nsUriPrefix="http://domainModel"
8        includedPackages="Data"
9        addNameAttribute="false"
10       resourcePerToplevelPackage="false"/>
```

Listing 5.1: Example of an oAW Workflow to Generate Metamodels from UML2 XMI files.

To perform the validation of a feature model, we modified the Eclipse contextual menu that is related to files with extension *.featuremetamodel*, which is the extension that the Feature Models Creator associates to feature models. Thus, we provide the option to *Validate Feature Models Structure*, and we are able to present messages to inform if any inconsistency was found in a feature model.

Metamodels and Feature Models for The Smart-Home Systems' SPL.

In Section 4.3 we introduced in detail the metamodels and feature models for our case study of the Smart-Home systems' SPL (see from Figure 4.2 to Figure 4.10). Figure 5.11 presents the feature models created with the Feature Models Creator for this case study. Figure on the left presents the facilities feature model and the one on the right presents the architecture feature model.

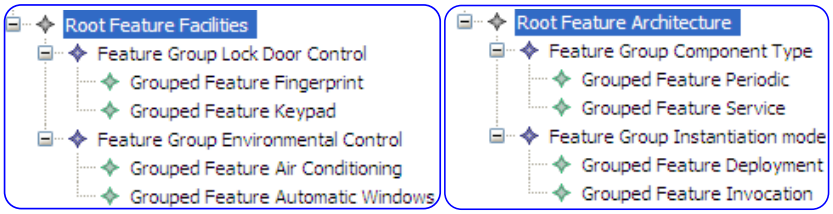

Figure 5.11: Feature Models for the Smart-Home systems' SPL.

5.3.3 Constraint Models Creation

The Constraint Models Creator.

We built an Eclipse plug-in to create constraint models, the *Constraint Models Creator*. Figure 5.12 presents the view associated to the Constraint Models Creator. The figure shows the creation of constraints between the domain metamodel and the facilities feature model from our Smart-Home systems' SPL. Using our Constraint Models Creator product line architects can load a metamodel and a feature model, create and delete constraints, cleane the works' areas and then reloade a new metamodel and a new feature model, and save a constraint model. The Constraint Models Creator allows for capturing the minimum and maximum cardinality that defines the constraint's cardinality property, and a description associated to the constraint.

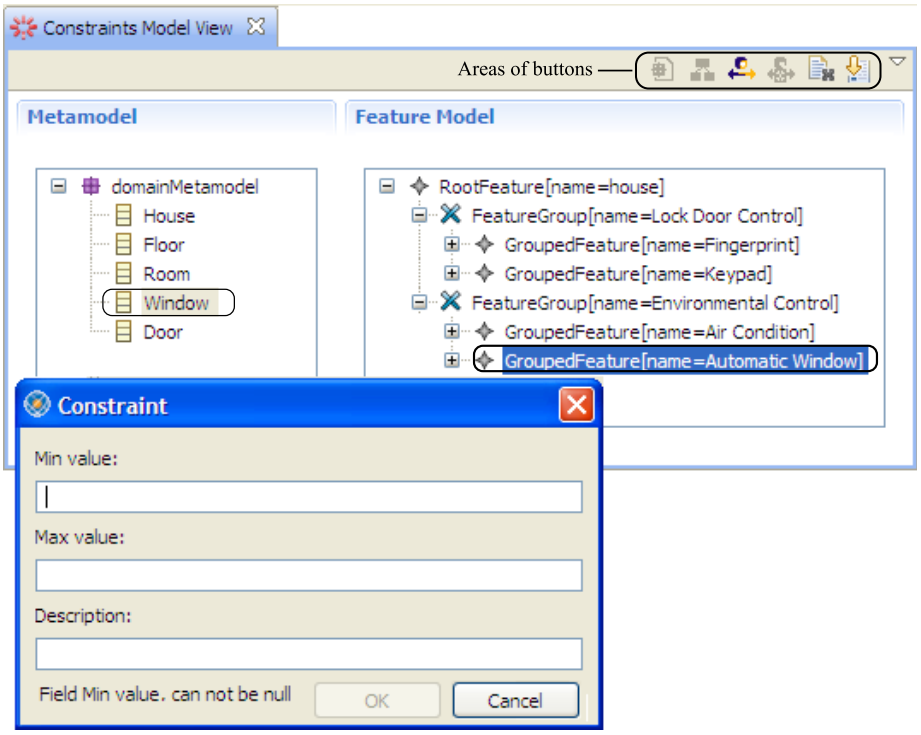

Figure 5.12: Eclipse View of The Constraint Models Creator.

When a product line architect selects to save a constraint model, the plug-in performs two activities. First, it saves a file with extension *.constraintmeta-*

```
1
2  context Binding ERROR loc() +
3    "There are less than 1 Door element bound"+
4    "to the feature Lock Door Control":
5
6    (this.metaConcept == 'Door' &&
7    this.feature.parentFeature.name == 'Lock Door Control'
8    && ((Configuration)this.eContainer).binding.
9      select(b|b.name==this.name && this.metaConcept=='Door'
10   && this.feature.parentFeature.name=='Lock Door Control').
11     size >= 1);
```

Listing 5.2: Example of a Check file generated by The Constraint Models Creator.

model containing the constraint model. Second, it saves a file with extension *.chk* which contains the Check expressions to validate binding models against the constraint model. Listing 5.2 presents an example of a Check expression generated by the Constraint Models Creator. The expression is generated from a constraint specifying that it has to exist at least one `Door` element bound to the feature `Lock Door Control` in the binding model that is being validated.

Our current implementation of the Constraint Models Creator allows product line architects to create the *constraint properties* associated to constraints. This implementation does not allow, however, product line architects to create the *structural properties* associated to constraints. Therefore, the *structural properties* must be written directly on the Check files.

Listing 5.3 presents an example of a Check expression for a structural property. This is related to a constraint between the `Component` metaconcept and the feature `Periodic`. The structural property defines that a `Component` element only can be bound to the feature `Periodic` if the `Component` element is also bound to the feature `On Invocation`. As part of our future work, we will allow product line architects to create the *structural properties* associated to constraints directly on the Constraint Models Creator.

Summarizing, our Constraint Models Creator allows product line architects to capture and express the variability described by possible fine-grained con-

```
1
2  context Binding ERROR loc() +
3    "The Component"+ this.elementName +"must be"+
4    "also bound to the *On Invocation* feature":
5
6    (this.metaConcept == 'Component' &&
7    this.feature.name == 'Periodic'
8    && ((Configuration)this.eContainer).binding.
9       select(b|b.name==this.name && this.feature.name==
10      'On Invocation').size == 1);
```

Listing 5.3: Example of a Check Expression for a Structural Property.

figurations, which we represent by using binding models, taking into account that fine-grained configurations have to be also restricted to represent valid products.

Constraint Models for The Smart-Home Systems' SPL.

We create two constraint models for our Smart-Home Systems' SPL. The first one is created between the domain metamodel and the facilities feature model. Table 5.5 presents these constraints which allow product line architects to capture and express the possible fine-grained variations between Smart-Home systems regarding domain and facilities' concepts.

For example, product line architects can express that between one and two `Doors` can have `Fingerprint` as `Lock Door Control` in Smart-Home systems. As a result, product designers will be able to configure a Smart-Home system with one particular door having `Fingerprint` as `Lock door control` and another Smart-Home system with two selected doors having `Fingerprint` as `Lock door control`.

The second constraint model is created between the components metamodel and the architecture feature model. Table 5.6 presents these constraints which allow product line architects to capture and express the possible fine-grained variations between Smart-Home systems regarding software components and software architecture concepts.

As a result, product line architects can express that in Smart-Home systems, for example, a component for managing `Automatic Windows` could be either

Table 5.5: Constraints Between the Domain Metamodel and the Facilities Feature Model

Metaconcept	Feature	Cardinality	Description
Door	Lock Door Control	[0..1]	Doors can have either Fingerprint or Keypad or none of them as Lock Door Control
Door	Fingerprint	[1..2]	Between one and two Doors can have Fingerprint as Lock doorcontrol
Door	Keypad	[0..1]	Between zero and one Doors can have Keypad as Lock Door Control
Room	Environmental Control	[0..1]	Rooms can have either Automatic Windows or Air Conditioning or none of them as Environmental Control
Room	Automatic Windows	[1..1]	Only one Room can have Automatic Windows as Environmental Control
Room	Air Conditioning	[1..3]	Between one and three Rooms can have Air Conditioning as Environmental Control
Window	Automatic Windows	[0..4]	Between zero and four Windows can be Automatic Windows

Table 5.6: Constraints Between the Components' Metamodel and the Architecture Feature Model

Metaconcept	Feature	Cardinality	Description
`Periodic`	`Component Type`	[0..1]	Components classified as `Periodic` can be either `Service` or `Periodic Components` in the final software architecture
`Component`	`Instantiation Mode`	[1..2]	`Components` can be instantiated either `On Deployment` or `On Invocation`

a `Service Component` or a `Periodic Component`. Product designers will be able to configure a Smart-Home system with the component for managing `Automatic Windows` as a `Periodic Component`. This component will check automatically the temperature of the room where the automatic windows are used to open or close the windows. Another product designers will be able to configure a Smart-Home system with the component for managing `Automatic Windows` as a `Service Component`. In this case, the inhabitants must manually select checking the temperature of the room where the automatic windows are. The inhabitants must also manually open or close the windows.

5.3.4 Domain Models and Binding Models Creation

Domain Models Creation.

We built an Eclipse plug-in to create domain models using the facility provided by Eclipse to generate model editors from Ecore models. We named this plug-in the *Smart-Homes' Domain Models Creator*. Product line architects have to create new domain metamodels and new domain models' editors for producing new MD-SPLs.

Figure 5.13 presents a domain model created with our Smart-Homes' Domain Models Creator. The model, which is created by a *building architect*, defines `firstFloor` and `secondFloor`. In the `firstFloor` there are two

rooms, `livingRoom` and `kitchen`. In the `secondFloor` there is another room, `mainRoom`, which has two windows, `mainRoomW1` and `mainRoomW2`. There are also two doors. The first door, `livingRoomD1`, is in the `livingRoom`. The second door, `mainRoomD2`, is in the `mainRoom`.

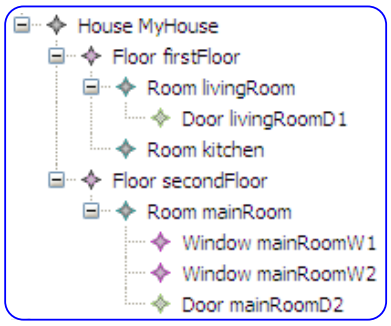

Figure 5.13: Example of a Domain Model Created with our Smart-Homes' Domain Models Creator.

The Binding Models Creator.

We developed an Eclipse plug-in named the *Binding Models Creator* to create binding models. Figure 5.14 presents the view associated to the Binding Models Creator. In the figure we present the creation of bindings between the domain model and the facilities feature model from our Smart-Home systems' SPL.

Using the Binding Models Creator, product designers can load a feature model, a domain model, and a constraint model, which will be used to validate the created binding model. Designers can create and delete bindings, or select a feature. The facility to select features is useful when coarse-grained configurations are required. Therefore, we can select for example the automatic windows for all the windows in the house only by selecting the `Automatic Windows` feature.

When a product designer selects to save a binding model, the plug-in performs two activities. First, it saves a file with extension *.configurationmetamodel* containing the binding model. Second, the binding model is validated against the constraint model loaded before. What really occurs is that the Check expressions generated from the constraint model are used to check the binding model to know if it satisfies the constraints. After the validation, the product designer obtains messages informing the state of the validation.

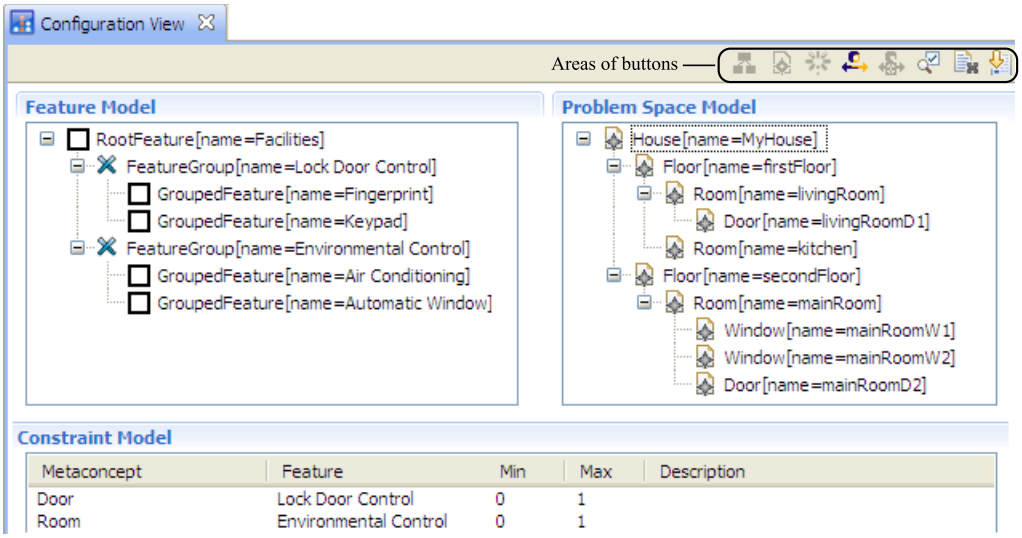

Figure 5.14: Eclipse View of The Constraint Models Creator.

Summarizing, our Binding Models Creator allows product designers to create fine-grained configurations by means of binding models. Our Binding Models Creator also allows product designers to validate the configurations against constraints expressing the valid fine-grained variations between products of the MD-SPL. This guarantees the configuration and subsequent derivation of valid products.

Binding Models for The Smart-Home Systems' SPL.

Product designers can create several binding models, as well as domain models, to configure diverse Smart-Home systems of our MD-SPL. In the following, we will present the process of configuring one particular Smart-Home system by creating the required binding models, which must satisfy the constraints presented before in Table 5.5 and Table 5.5. The result will be a complete fine-grained configuration of a particular Smart-Home system of our MD-SPL.

Table 5.7 presents a set of bindings between the domain model from Figure 5.13 and our facilities feature model. These bindings are created by a *facilities designer*, and along with the domain model are part of the fine-grained configuration of the particular Smart-Home system we are configuring. They must satisfy the constraints presented in Table 5.6.

Table 5.7: Bindings Between the Domain Model from Figure 5.13 and Our Facilities Feature Model

Element	Feature	Description
livingRoom	Air Conditioning	The livingRoom will manage Air Conditioning as Environmental Control
livingRoomD1	Fingerprint	The livingRoomD1 will manage Fingerprint as Lock Door Control System
mainRoomW1	Automatic Windows	The mainRoomW1 will be an Automatic Window
mainRoomD2	Keypad	The mainRoomD2 will manage Keypad as lock Door Control system

Accordingly to this configuration, after the execution of the model transformation process, the product designer will obtain a particular Smart-Home system which GUI is presented in Figure 5.15 and Figure 5.16. Figure 5.15 shows the view associated to the mainRoom, which has one Automatic Windows, mainRoomW1, and its door, mainRoomD2, has Keypad as Lock Door Control mechanism.

Figure 5.16 presents the view associated to the livingRoom. In this case the Air Conditioning is managed by a Periodic software component. That is the reason why the system automatically turns it on/off according to the desired temperature of the room. In this case the Desired Temperature of the Living Room is 19 degrees and the Current Temperature is 17 degrees, then the Air Conditioning is turned off. The door, livingRoomD1, has Fingerprint as Lock Door Control mechanism.

Figure 5.17 presents the component's model derived from the domain model in Figure 5.13 given the bindings from Table 5.7. Product designers, who are *software architects*, have to create a binding model between this generated components' model and the architecture feature model. This binding model corresponds to the fine-grained configuration of the software components included in the Smart-Home system, and these bindings have to satisfy the constraints presented in Table 5.6.

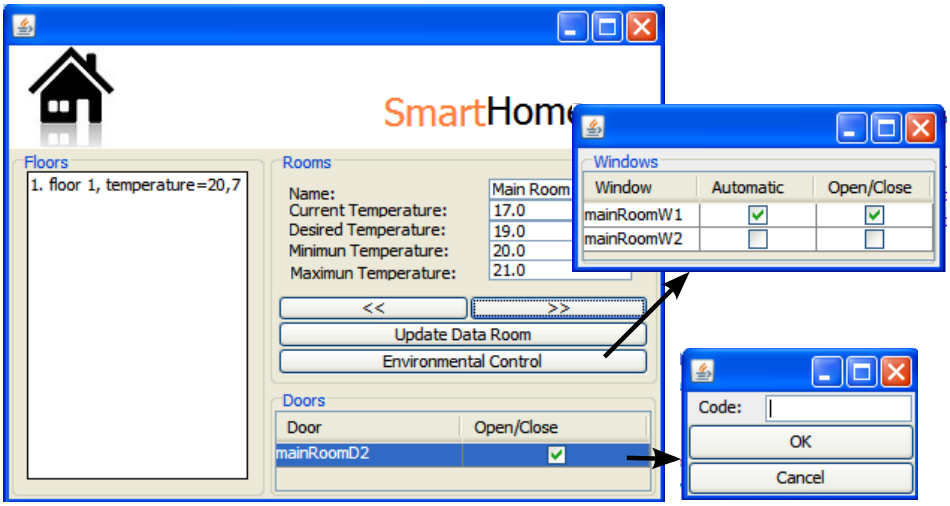

Figure 5.15: View of the `Main Room` of the Configured Smart-Home System.

Table 5.8 presents a set of bindings between the components' model from Figure 5.17 and our architecture feature model. These bindings complete the required configuration to derive the Smart-Home system we are configuring. According to these bindings, the final architecture model for the Smart-Home system presented in Figure 5.15 and Figure 5.16 will have only one `Periodic Component`, the `Air Conditioning Controller Component`. Furthermore, the `House` and `Floor Components` will be instantiated on `Invocation`. The other components will be instantiated on `Deployment`.

5.4 Core Assets Development and Product Derivation

5.4.1 Transformation Rules Creation

In Section 4.5 we introduced the several stages of model-to-model and model-to-text transformation rules for deriving configured Smart-Home systems.

Figure 5.18 presents a screenshot of the folders' structure to maintain our model transformation rules. We use the Xpand and Xtend languages to create our transformation rules. These languages create files with extensions

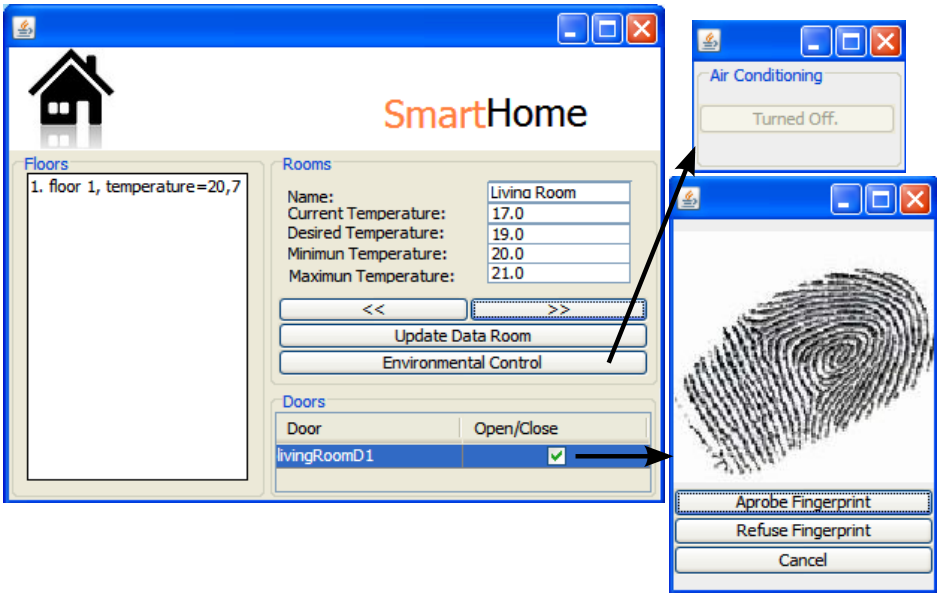

Figure 5.16: View of the `Living Room` of the Configured Smart-Home System.

`.xpt` and `.ext` respectively. We create two sets of transformation rules: the `base` and the `specific` ones. On the one hand, base transformation rules do not depend of any variant of the product line. Thus, they are always executed during the transformation process. On the other hand, we create specific transformation rules having into account features that can affect the transformation process. Our transformation rules are organized in folders created for each transformation step.

Listing 5.4 presents an example of part of the model-to-text transformation rule to transform `Component` elements into Java source code. As we introduced in Chapter 4, we reuse pieces of code which have been previously tested to build complete OSGi implementations. As a result we guarantee the quality of derived Smart-Home systems. The source code in Listing 5.4 correspond to the method we created to turn on the air conditioning located in a particular room.

Table 5.8: Bindings Between the Components' Model From Figure 5.17 and Our Architecture Feature Model.

Element	Feature	Description
`Windows Controller`	`Service`	The `Windows Controller` component will be a `Service Component`
`Air Conditioning Controller`	`Periodic`	The `Air Conditioning Controller` component will be a `Periodic Component`
`House`	`Invocation`	The `House Component` will be instantiated on `Invocation`
`Floor`	`Invocation`	The `Floor Component` will be instantiated on `Invocation`

```
1  "DEFINE implementation FOR componentsMetamodel::Component "
2  public void start(Integer floorId, Integer roomId)
3    throws Exception{
4
5    Room room = getRoom(floorId, roomId);
6    if(room != null && room.getEnvironmentalControl() ==
7      TypeEnvironmentalControl.AIRCONDITIONING){
8      room.setAirConditionStatus(true);
9    }
10 }
11 "ENDDEFINE"
```

Listing 5.4: Model-to-Text Transformation Rule to Transform `Component` Elements into Java Source Code.

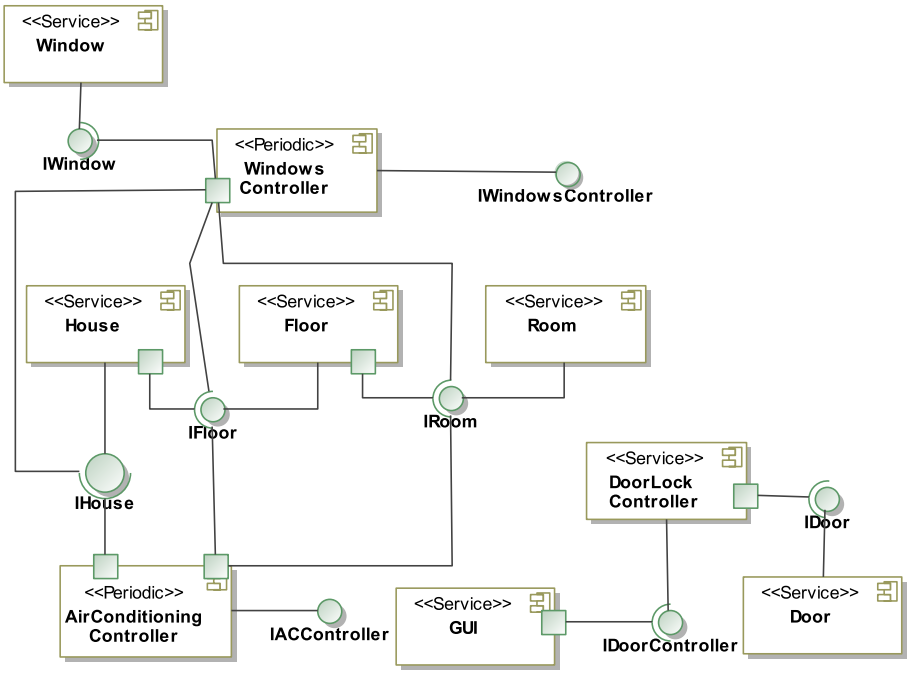

Figure 5.17: Components' Model Derived from a Domain Model.

5.4.2 Decision Models Creation

The Decision Models Editor.

We built an Eclipse plug-in to create decision models, the *Decision Models Editor*. This editor was developed by using the Topcased's facility to create model editors (see Section 2.4), and is part of the contributions of the Master thesis of Andrés Romero [Rom09].

Figure 5.19 presents the GUI of our Decision Models Editor. On the left, we present the palette of options to create `Model-to-Model` and `Model-to-Text` transformations, `Base` and `Specific` transformation rules, `Aspects`, `Execution Conditions`, `CoarseConditions` and `FineConditions`. Options also include to define the `Source` and `Target` models of the model transformations. On the right, we present part of the decision model created for our Smart-Home systems' SPL.

Our Decision Models Editor allows product line architects to maintain uncou-

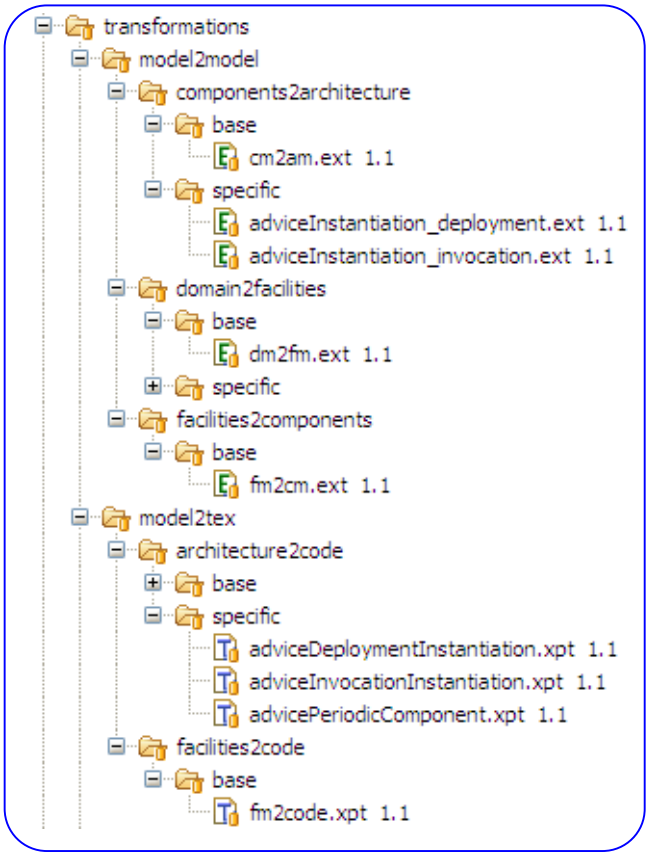

Figure 5.18: Folders' Structure for Transformation Rules Files.

pled (1) the information of features, (2) the transformation rules, and (3) the possible execution's conditions of transformation rules that particular feature configurations imply. Furthermore, our Decision Models Editor allows product line architects to capture as independent `Aspects` the information of how transformation rules must be composed to derive configured products. This is a high-level mechanism which is independent of the technology used to implement our approach. Finally, our plug-in can capture execution's conditions of transformation rules in order to derive products based on binding models, which represent fine-grained configurations.

Decision Models for The Smart-Home Systems' SPL.

The decision models of our case study facilitates to derive any product which

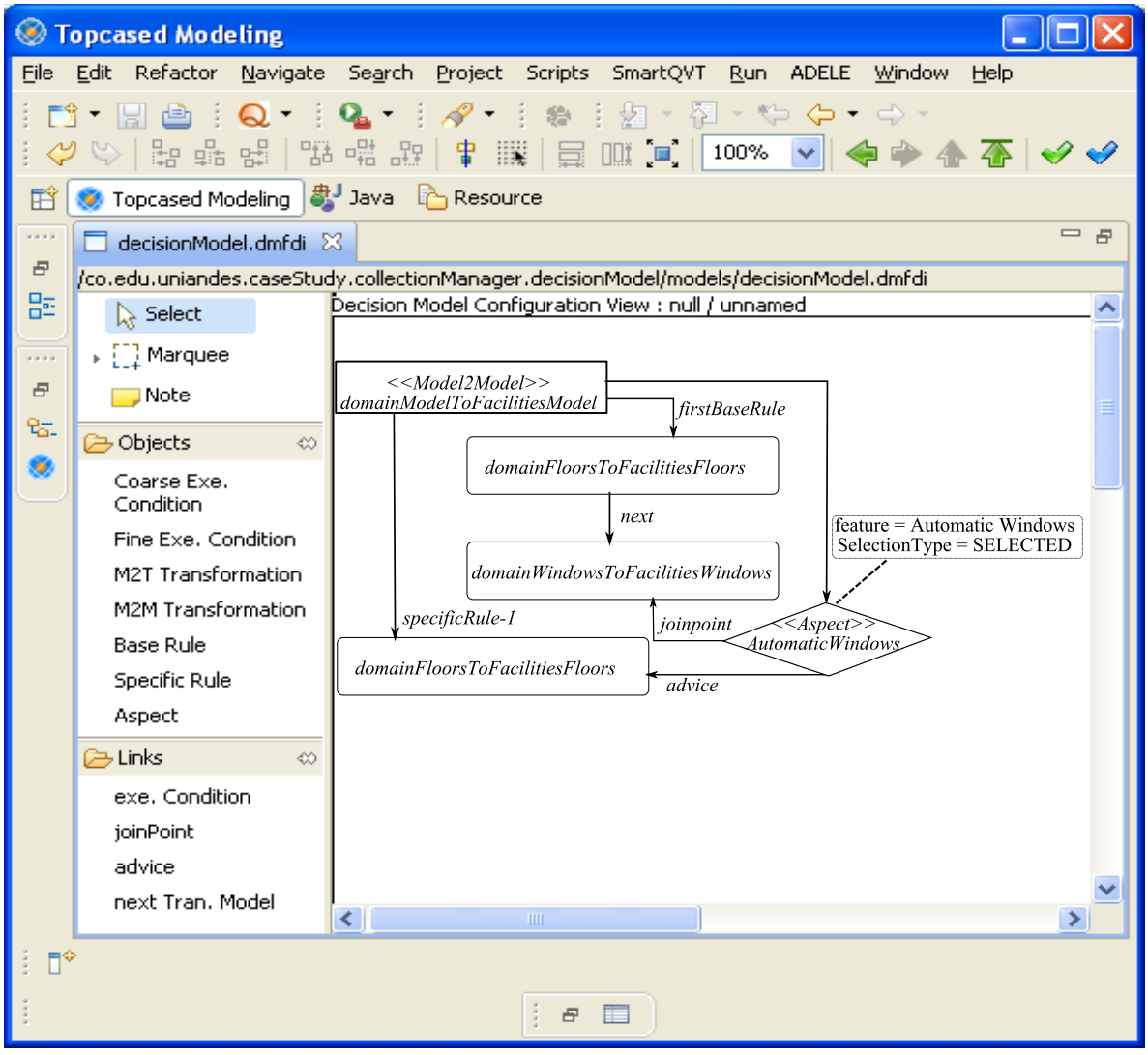

Figure 5.19: Graphical User Interface of our Decision Models Editor.

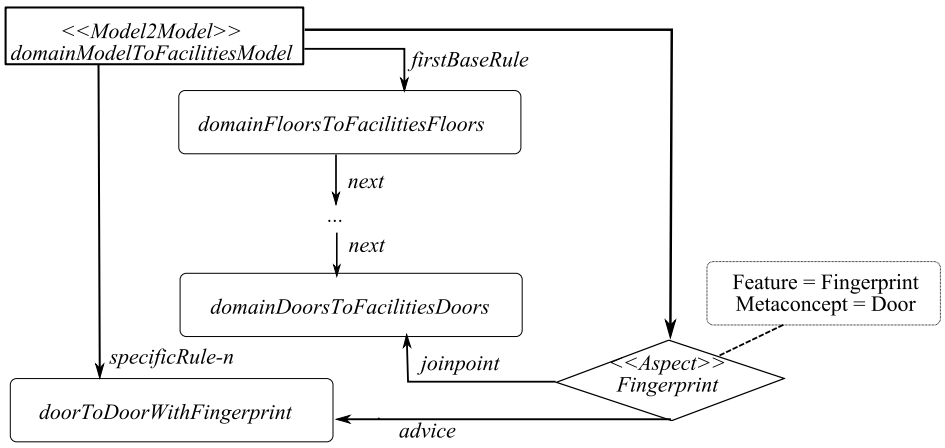

Figure 5.20: Decision Model Including an Aspect to Derive Doors with Fingerprint as Lock Door Control Mechanism.

has been configured by creating (1) a domain model, (2) a valid binding model between the facilities metamodel and the domain model, and (3) a valid binding model between the architecture metamodel and the components' model derived from the domain model.

In Section 4.5 we introduced part of the decision model we created for deriving configured Smart-Home systems. Similarly to Figure 4.22, we defined an `Aspect` element related to each constraint in the two constraint models. As a result, we can guarantee that any binding satisfying a constraint will be taken into account during the derivation process. The model element involved in the binding will be transformed using a specific transformation rule in charge of transforming it according to the feature involved in the binding.

For instance, Figure 5.20 presents another part of the decision model for deriving configured Smart-Home systems. In this case, we present the `Aspect` we created for the constraint between the `Door` metaconcept and the `Fingerprint` feature. This `Aspect` specifies that any `Door` element in a binding model will be transformed by using the specific transformation rule `doorToDoorWithFingerprint`. As a result, we can guarantee that any binding satisfying the constraint between the `Door` metaconcept and the `Fingerprint` feature will be taken into account to derive a Smart-Home system. The doors involved in the bindings will have fingerprint as lock door control mechanism.

5.4.3 Generation and Execution of Model Transformation Workflows

As we explained in Chapter 4, to execute our decision models we need to transform them into executable oAW workflows by using a model-to-text transformation. This transformation is achieved using a model-to-text transformation, which we include in Appendix A. As a result, we can execute the generated model transformation workflows on the model transformation engine of oAW. Thus, we derive any (fine-grained) configured product.

Figure 5.21 presents the final result of executing the sequence of model transformations we defined to generate a Smart-Home system of our product line. The files correspond to Java (OSGi) source code and XML descriptors which have been generated departing from the domain model in Figure 5.13 and the binding models in Table 5.8 and Table 5.8.

We include in the web-site of our research group, under the link MD-SPL Engineering [Sof09], details about our entire tool support and the instructions for installing it. We also included all the core assets to create MD-SPLs of Smart-Home systems such as the one we have used through this document to illustrate our work. Additionally, we present another MD-SPL of standalone systems for managing collections, including all the required core assets to derive its product line members.

5.5 Summary

In this chapter we presented the FieSta toolkit, which includes the tools we developed for supporting our MD-SPL Engineering mechanisms to create SPLs. We also presented the results obtained of using these mechanisms and tool support. We introduced each tool we developed integrated into our toolkit. Among these tools, we presented our Constraint Models Creator, our Binding Models Creator and our Decision Models Editor. Through this chapter we have presented several examples of Smart-Home systems we derived using our MD-SPL engineering mechanisms and tool support. We also presented some particular (fine-grained) configurations we created to derive these Smart-Home systems.

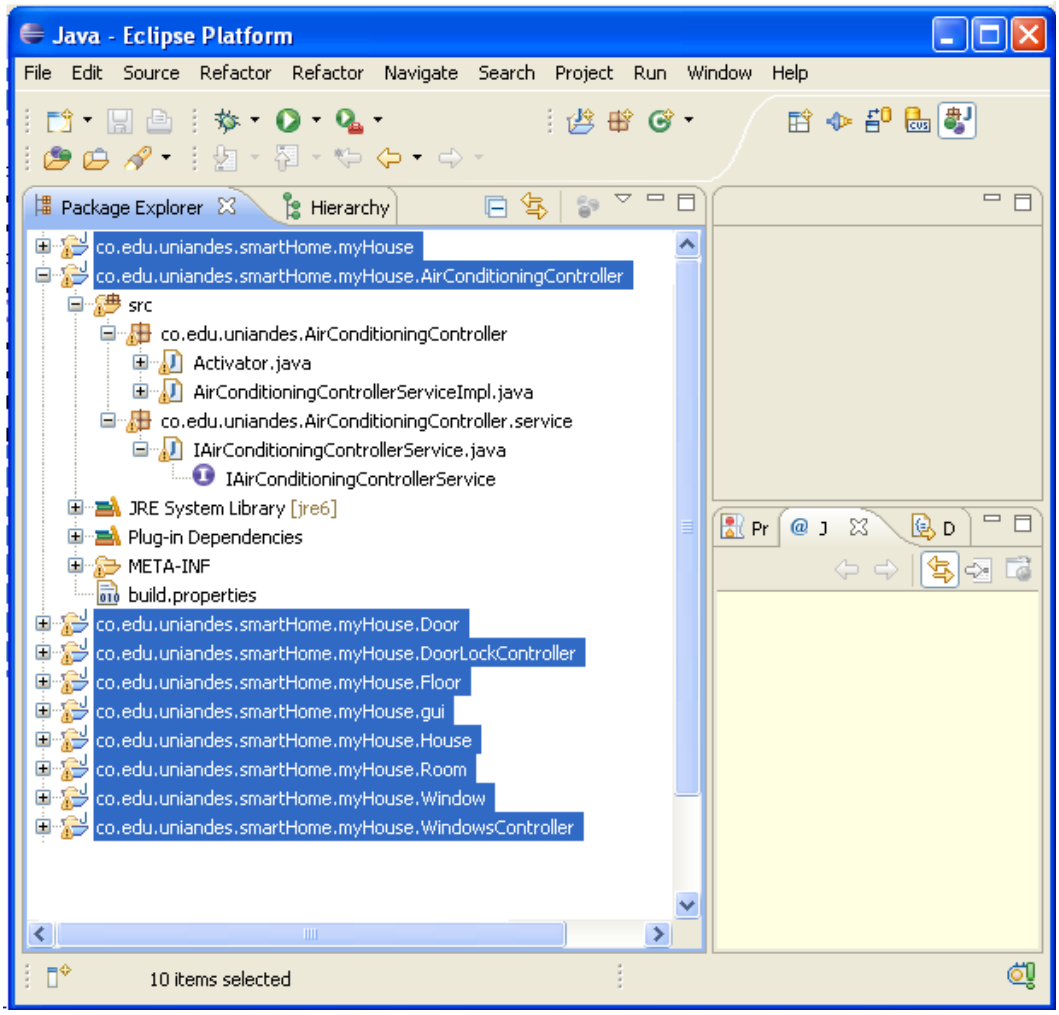

Figure 5.21: Source Code of a Generated Smart-Home system.

Part IV. Conclusion

Chapter 6

Conclusion

6.1 Introduction

The main motivation for this thesis was to propose Model-Based mechanisms to assist product line architects and product designers during the creation of Software Product Lines. We were interested in (1) to extend the power of expression of variability, and consequently to extend the scope of products that can be fine-grained configured, and (2) to facilitate the maintenance, reuse and evolution of the core assets developed to derive MD-SPLs.

We designed and implemented three domain-specific modeling languages to support the definition of constraint models, binding models and decision models. We also provided the model transformations required to (1) validate binding models against constraint models and (2) execute decision models into a workflow engine to automatically derive products. This chapter presents a summary of this work, its main contributions, our conclusions and potential future research directions.

6.2 Thesis Summary

In the first part of this thesis, we introduced our context of work, research problems and research objectives.

In the second part, we introduced the main concepts regarding Model-Driven Development and Software Product Line Engineering, and we introduced and analyzed several Model-Driven Software Product Line approaches. We explained the concept of *separation of concerns* of a system in different models and we discussed the concept of *level of abstraction* of models. We introduced the concept of Domain Specific Modeling, the relation of conformity, and the four-layer metamodeling framework. We defined the concept of model transformations and transformation rules, and we classified them into four major categories. We introduced some modeling frameworks and model transformation languages such as Xpand and Xtend. We presented the basis of Software Product Line Engineering, discussing the main processes involved in the creation of Software Product Lines: the *domain engineering process* and the *application engineering process*. Finally, we introduced the MD-SPL Engineering paradigm and we presented a State-of-the-Art of it, presenting a discussion emphasizing on the advantages and drawbacks of representative MD-SPL approaches.

In the third part of this document we presented FieSta, our Model-Driven Software Product Line approach, and a case study we used to validate it. Constraint models were presented and their use was illustrated in the context of our case study. Binding models were explained and also illustrated with our case study. We then showed how we derived products based on binding models and decision models. We also presented limitations of our approach for deriving products based on decision models. Finally, we validated our approach presenting examples of products that we are able to derive using our Model-Based mechanisms. We present results of configuring and deriving products of two product lines. We also presented the implementation strategy for our approach, including (1) the required activities to create products, and (2) the tools we created to support these activities.

6.3 Results and Contributions

In this section, we analyze the advantages and drawbacks of FieSta regarding the MD-SPL engineering mechanisms we propose (1) for expressing variability and configuring products in MD-SPLs, and (2) for deriving configured products. We remark, by comparison with others MD-SPL approaches,

where our work presents a contribution to the MD-SPL engineering domain. This allows us to emphasize on the significance of our results in terms of the research objectives of this thesis.

6.3.1 Metamodeling and Feature Modeling

We use Metamodeling and feature modeling for capturing and expressing variability. Metamodels facilitate modeling variations at language level. Product designers, for instance *building architects*, are capable of configuring different products by creating diverse building's models. Feature modeling allows us configuring products by selecting features. Therefore, for instance *facilities designers* and *software architects* can configure products without the need of creating complex models.

Using feature modeling and metamodeling separately gives us the flexibility and power of expression of metamodels, and the simplicity of feature models. We have also proposed to relate metamodels and feature models to create what we named constraint models. Constraint models allow us expressing fine-grained variations between products of MD-SPLs. We have shown how to express the possible fine-grained variations between products of an MD-SPL by creating relationships between metamodels and feature models. For example to express that two Smart-Home systems could be different by the location of their automatic windows.

Such as we demonstrated in Chapter 5, our mechanism for expressing fine-grained variations between products of an MD-SPL using constraint models extends the power of expression of variability in MD-SPLs, and consequently it extends the scope of products that can be fine-grained configured. This satisfies our research objective **RO1** presented in Chapter 1. We presented this mechanism in [ACR09].

6.3.2 Multi-Staged Configuration of Products

Our approach supports the modeling of variability in several stages. we allow product line architects, at different (staged) times, to express and capture coarse- and fine-grained variations between members of product lines. This

facilitates to product line architects with different skills focusing on particular concerns at different moments.

At configuration time, we allow product designers configuring products at different binding times where s/he can chose at each stage specific variants to create domain models and binding models. Thus, we postponed the binding time of variations facilitating the intervention of stakeholders with different profiles in the configuration process. For instance, regarding Smart-Homes' facilities and software architecture, *facilities designers* and *software architects* can provide their choices at different time.

6.3.3 Coarse- and Fine-Grained Variations and Configurations

As far as we know, our approach is the only MD-SPL approach allowing for creating fine-grained configurations and deriving products based on such configurations. We have presented the way as we represent fine-grained configurations between product line members by means of binding models. A binding model allows us configuring model elements individually based on features. For example, we have created a binding to indicate that the feature `Periodic Component` affects individually the component `Air Conditioning Controller`, and the feature `Keypad` affects individually the door `mainDoorD2`.

We first introduce our mechanism for creating fine-grained configurations in [GPA+07], then we used it in [ACR09, ACR07b, AGGa+08, ACR07a]. This mechanism contributes to satisfy our research objective **RO1** presented in Chapter 1.

6.3.4 Core Assets Development and Decision Models

We introduced the use of explicit decision models in MD-SPL engineering. Our decision models allow us to capture separately (1) the base and specific model transformation rules used to derive product line members, (2) the variants represented in feature models, and (3) the relationships between model transformations and variants. Decision models are the key of our

mechanism to compose model transformations and adapt their execution ordering according to particular product configurations.

Other approaches such as Loughran et al.'s approach [LSGF, SLFG08] and Voelter and Groher's approach [VG07b] have proposed the use of decision models. Our approach, however, is concerned about both (1) the problem of transformation rules composition based on product configurations, which is a complex problem in MD-SPL Engineering, and (2) the independency from model transformation languages to create decision models. As we have presented before in Section 3.6, the Loughran and Colleagues' approach in only concerned about the composition of software components, and the Voelter and Groher's approach is restricted to use a platform-dependent language, Xtend, to create decision models. Furthermore, our mechanism based on decision models to derive products has into account that several features selected together may imply different adaptation than the required one when features are selected separately. This is not taken into account by the Loughran and Colleagues' approach.

As we have presented in this thesis, our decision models also capture the required information about how transformation rules must be composed to derive fine-grained configured products. Given that our approach takes into account fine-grained variations and fine-grained configurations, this is also worry about how to derive fine-grained configured products. Our decision models has been presented in [ACR09, ARCR09, ACR08], and contributes to satisfy our research objective **RO2**.

6.3.5 Product Derivation

Based on our decision models, we propose a mechanism for selecting transformation rules and modifying their execution ordering according to selected variants. In our current implementation we have used the model transformation engine of oAW to execute model transformation workflows derived from our decision models. Our decision models, however, are independent of model transformation languages and can be used to support product derivation in contexts different to the oAW context. For instance, currently we explore how our decision models can be used to derive products by using the ATL language and its facilities for transformation rules composition [RA09]. Further work on this field is part of our future work.

Our mechanism for product derivation has been presented in [ACR09, ACR08], and contributes to satisfy our research objective **RO2** presented.

6.3.6 Summary

Table 6.1 presents a summary of this section having into account our approach and the related approaches.

Regarding the scalability of our MDD mechanisms to traditional SPL engineering, where models are used only as artifacts for documentation, we believe this is easily reachable. Currently MDD is being used not only in academic exercises but also in real industry. Several international events, journals and research projects are concern about the subject. Thus, body of knowledge including tool support is being created to support MDD. We have shown through this document how our MDD mechanisms contribute to make SPL engineering more feasible and profitable, and consequently more interesting for SPL developers to adopt it.

6.4 Future Work

In this thesis we integrated Model-Driven Development, Software Product Line Engineering, and Aspect-Oriented Programming. We proposed a coordinated use of these paradigms to solve the research problems we described in Chapter 1. The following sections present some logical continuations of this work.

6.4.1 Dealing with Current Limitations: Features Combinatory, Features Interaction and Bindings Interaction

We discussed in Section 4.7 some limitations of our approach. We consider important to improve our approach overcoming such limitations. First, it is required a mechanism to validate that, for each possible feature configuration, product line architects provide the required transformation rules to

Table 6.1: Summary of the Discussion Regarding our Contribution to the MD-SPL Engineering Domain

	Our Work	Czarnecki and Antkiewicz	Wagelaar	Loughran et al.	Voelter and Groher
Metamodeling for expressing variability and modeling for configuring products	Yes	No	Yes	No	Yes
Multi-staged configuration of products	Yes	No	No	No	Yes
Expression of *fine-grained* variations and creation of *fine-grained* configurations	Yes	No	No	No	No
Creation of explicit decision models	Yes	No	No	Yes	Yes
Decision models take into account the effects that possible feature combinations may have in final products	Yes	n/a	n/a	No	Yes
Decision models independent of particular implementation languages	Yes	n/a	n/a	Yes	No
Selection of transformation rules according to selected variants	Yes	No	Yes	Yes	Yes
Modification of transformation rules' execution ordering according to selected variants	Yes	No	Yes	No	Yes
Mechanisms for modifying execution ordering of transformation rules independent of particular model transformation languages	Yes	n/a	No	n/a	Yes

derive valid products. Second, it is required another mechanism that allows product line architects to capture possible feature interactions by means of *scenarios*, and thus, to create and relate transformations rules to such scenarios. Finally, it is required to consider bindings that satisfy several constraints when execution ordering of transformation rules is modified according to binding models.

6.4.2 Using Complementary Variability Models

We focused on the use of feature models as variability models. However, other variability models, such as Ontology models or the one presented by Bayer et al. [Bay06], involve other relevant concepts different that only `Group`, `Grouped` or `Solitary Feature`. These variability models deserve special attention for the rich semantics they provide to express variability in product lines. We consider important to integrate variability models such as Ontology models into our approach. This will complement feature models improving the power of expression of variability and allowing to extend the scope of MD-SPLs.

6.4.3 Integrating Architectural Description Languages

Our approach supports staged capture of variability, and also staged configuration of products. We showed, using our case study, how one of these stages involve concerns about software architecture based on components. For this, we created one specialized metamodel capturing concepts of component-based software development. There are, however, several Architectural Description Languages (ADL) such as [AMS07, DvdHT05], which are based on metamodels that include very complete information about architectural concerns. We consider important to include the use of these ADLs into our approach to extend the scope of variations we are able to manage regarding software architecture, no matter the domain of the MD-SPL we are interested in developing.

6.4.4 Incorporating Aspect Oriented Modeling

We considered Aspect Oriented Programming as the indicated paradigm to tackle the problem of adapting the execution ordering of transformation rules. Recent work (see [onl08]), have shown also how Aspect Oriented Modeling (AOM) is a valuable paradigm to be incorporated in Model-Driven Development. AOM allows product line architects to create reusable models, which during the derivation of product line members can be woven with other models according to variability choices performed by product designers. We believe that AOM enables the explicit expression and modularization of variability on model level, and facilitates the maintainability and reuse of models as core assets. Thus, we consider important to integrate AOM in our approach.

6.4.5 Using Declarative Programming to Create Transformation Rules

Declarative programming minimize side effects by describing *what* the program should accomplish, rather than describing *how* to go about accomplishing it [Llo94]. Declarative programming in MDD has a number of advantages. Declarative transformation rules are based on specifying relations between source and target patterns, hiding the details related to selection of source elements, rule triggering and ordering [JK05]. We believe that the integration of declarative transformation rules may help to deal with the problem of adapting the execution ordering of transformations rules given different product configurations.

Furthermore, currently we explore constraint programming, which is a type of declarative programming, to tackle the problem of relating transformation rules to sets of variants with particular interactions stated in the form of constraints. The idea of using constraints to define suitable configurations can be extended to the design and implementation of a Constraint System specialized in that kind of constraints, in which all the power of specialized solvers can be used to validate if proposed configurations are feasible or not. Likewise, it could be important to explore if modelling the transformation rules scheduling problem using constraints can be a more efficient way to solve it instead of using aspect oriented programming.

6.4.6 Formalizing the Approach

We have started a work tending to formalize our approach using basic set theory. Our aim is to generalize our MD-SPL approach making it extensible and independent of specific platform modelling frameworks and/or model transformation languages.

Bibliography

[ABM00] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component-based product line development: The kobra approach. In P. Donohoe, editor, *Proceedings of 1st Software Product Line Conference*, pages 289–309, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[AC04] Michal Antkiewicz and Krzysztof Czarnecki. Featureplugin: Feature modeling plug-in for eclipse. In *Proceedings of the Workshop on Eclipse Technology eXchange at OOPSLA'04*, pages 67–72, 2004.

[ACR07a] H. Arboleda, R. Cassallas, and J. C. Royer. Implementing an MDA Approach for Managing Variability in Product Line Construction Using the GMF and GME Frameworks. In *Proceedings of the 5th Nordic Workshop on Model Driven Software Engineering (NW-MoDE'07)*, pages 67–82, Ronneby, Sweden, August 2007.

[ACR07b] Hugo Arboleda, Rubby Casallas, and Jean-Claude Royer. Dealing with constraints during a feature configuration process in a model-driven software product line. In *Proceedings of the 7th Workshop on Domain-Specific Modeling at the 22th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 178–183, Montreal, Canada, 2007.

[ACR08] Hugo Arboleda, Rubby Casallas, and Jean-Claude Royer. Using transformation-aspects in model-driven software product lines. In *Proceedings of the 3th International Workshop on Aspects, De-*

pendencies, and Interactions at 22nd European Conference on Object-Oriented Programming (ECOOP'08), pages 46–56, Paphos, Cyprus, July 2008.

[ACR09] Hugo Arboleda, Rubby Casallas, and Jean-Claude Royer. Dealing with fine-grained configurations in model-driven spls. In *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, San Francisco, US, August 2009.

[AGGa+08] N. Anquetil, B. Grammel, I. Galvão, J. Noppen, S. Shakil, H. Arboleda, A. Rashid, and A. Garcia. Traceability for model driven, software product line engineering. In *Proceedings of the 4th Workshop on Traceability at the 4th European Conference on Model Driven Architecture (ECMDA'08)*, Berlin, Germany, June 2008.

[AMP09] AMPLE project. European Commission STREP Project AMPLE IST-033710, last visited in June 2009.

[AMS07] Timo Asikainen, Tomi Männistö, and Timo Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Adv. Eng. Inform.*, 21(1):23–40, 2007.

[ARCR09] Hugo Arboleda, Andres Romero, Rubby Casallas, and Jean-Claude Royer. Product derivation in a model-driven software product line using decision models. In *Proceedings of the 12th Iberoamerican Conference on Requirements Engineering and Software Environments (IDEAS'09)*, pages 59–72, Medellin, Colombia, April 2009.

[BÓ5] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, May 2005.

[Bay06] *Consolidated Product Line Variability Modeling*, pages 195–241. Springer Berlin Heidelberg, 2006.

[BBM03] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework (EMF).* Pearson Education, 2003.

[BFG00] Joachim Bayer, Oliver Flege, and Cristina Gacek. Creating product line architectures. In *IW-SAPF-3: Proceedings of the International Workshop on Software Architectures for Product Families*, pages 210–216, London, UK, 2000. Springer-Verlag.

[BFG+02] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, Henk J. Obbink, and Klaus Pohl. Variability issues in software product lines. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 13–21, London, UK, 2002. Springer-Verlag.

[BGJ+03] Felix Bachmann, Michael Goedicke, Julio, Robert L. Nord, Klaus Pohl, Balasubramaniam Ramesh, and Alexander Vilbig. A meta-model for representing variability in product family development. In *Proceedings of the 5th International Workshop on Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 66–80, Siena, Italy, November 2003. Springer.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach*. Addison-Wesley, Boston, MA, USA, 2000.

[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.

[Car09] Carnegie Mellon University. The Software Engineering Institute, last visited in June 2009.

[CBA09] Lianping Chen, Muhammad A. Babar, and Nour Ali. Variability management in software product lines: A systematic review. In *Proceedings of the XIII Software Product Line International Conference*, San Francisco, CA, USA, August 2009.

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative programming for embedded software: An industrial experience report. In *Proceedings of the 1st Conference on Generative Programming and Component Engineering*, pages 156–172. Springer-Verlag, 2002.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Oct 2003.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[CHE04] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *Proceedings of the 3th Software Product Line Conference 2004*, pages 266–282. Springer, LNCS 3154, 2004.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[CKMM03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan R. Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Netw.*, 41(1):115–141, 2003.

[Cle02] Paul C. Clements. On the importance of product line scope. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 70–78, London, UK, 2002. Springer-Verlag.

[CNN01] Paul Clements, Linda Northrop, and Linda M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, August 2001.

[DGR08] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. Decisionking: A flexible and extensible tool for integrated variability modeling. In *Proceedings of the 2nd Int. Workshop on Variability Modelling of Software-intensive Systems*, Essen, Germany, January 2008.

[DvdHT05] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Methodol.*, 14(2):199–245, 2005.

[EFG+08] Christoph Elsner, Ludger Fiege, Iris Groher, Michael Jäger, Christa Schwanninger, and Markus Völter. Ample project. deliverable d5.3 - implementation of first case study: Smart home. Technical report, December 2008.

[ESB04] Matthew Emerson, Janos Sztipanovits, and Ted Bapty. A mof-based metamodeling environment. *Journal of Universal Computer Science*, 10:1357–1382, October 2004.

[FB01] R. France and J. Bieman. Multi-view software evolution: A uml-based framework for evolving object-oriented software. *Software Maintenance, 17th IEEE International Conference on Software Maintenance (ICSM'01)*, 0:386, 2001.

[FMP08] Thomas Forster, Dirk Muthig, and Daniel Pech. Understanding decision models. visualization and complexity reduction of software variability. In *Proceedings of the 2nd Int. Workshop on Variability Modeling of Software-intensive Systems*, Essen, Germany, January 2008.

[FS04] Frédéric Fondement and Raul Silaghi. Defining model driven engineering processes. In *Proceedings of the 3rd Workshop in Software Model Engineering. WiSME*, 2004.

[FSJ99] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[Goe09] 1st international workshop on model-driven approaches in software product line engineering (maple 2009). San Francisco, US, August 2009.

[GPA+07] Kelly Garces, Carlos Parra, Hugo Arboleda, Andrés Yie, and Rubby Casallas. Variability management in a model-driven software product line. *Avances en Sistemas e Informática*, 4(2):3–12, 2007.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, Indianapolis, US, 2004.

[GSV08] Iris Groher, Christa Schwanninger, and Markus Voelter. An integrated aspect-oriented model-driven software product line tool suite. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 939–940, New York, NY, USA, 2008. ACM.

[JB06] Frédéric Jouault and Jean Bézivin. Km3: A dsl for metamodel specification. In Roberto Gorrieri and Heike Wehrheim, editors, *FMOODS*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2006.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[JK06] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195, New York, NY, USA, 2006. ACM.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. *Technical Report CMU/SEI-90-TR-21*, 1990.

[KKL+98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[LH05] Shourong Lu and Wolfgang A. Halang. Platform-independent specification of component architectures for embedded real-time systems based on an extended uml. In C. Atkinson, C. Bunse, H. G. Gross, and C. Peper, editors, *Component-Based Software Development for Embedded Systems*, volume 3778 of *Lecture Notes in Computer Science*, pages 123–142. Springer-Verlag, Berlin Heidelberg, 2005.

[Llo94] J. W. Lloyd. Practical Advantages of Declarative Programming. In *Joint Conference on Declarative Programming*, 1994.

[LR07] Michael Lawley and Kerry Raymond. Implementing a practical declarative logic-based model transformation engine. In *SAC '07:*

Proceedings of the 2007 ACM symposium on Applied computing, pages 971–977, New York, NY, USA, 2007. ACM.

[LSGF] N. Loughran, P. Sanchez, A. Garcia, and L. Fuentes. Language support for managing variability in architectural models. In *Proceeding of the 7th International Symposium on Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 36–51, Budapest, Hungary, March. Springer.

[Mez09] 1st international workshop on model-driven product line engineering (mdple'2009). Enschede, The Netherlands, June 2009.

[MFJ05] Pierre A. Muller, Franck Fleurey, and Jean M. Jézéquel. Weaving executability into object-oriented meta-languages. In Lionel C. Briand and Clay Williams, editors, *Proceedings of the 8th International on Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[Mic04] Michael K. Smith and Chris Welty and Deborah L. McGuinness. Owl web ontology language guide. world wide web consortium. Technical report, February W3C Recommendation 10 February 2004.

[MO04] Mira Mezini and Klaus Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *SIGSOFT Softw. Eng. Notes*, 29(6):127–136, 2004.

[MRV08] Nathalie Moreno, José R. Romero, and Antonio Vallecillo. *An Overview Of Model-Driven Web Engineering and the Mda*, chapter 12, pages 353–382. Springer, 2008.

[No 09] No Magic, Inc. Magicdraw, last visited in June 2009.

[OAW09a] OAW. openArchitectureWare 4.3 User Guide., last visited in June 2009.

[OAW09b] OAW. The Openarchitectureware Framework, last visited in June 2009.

[OMG03] OMG. Object Management Group. Model driven architecture, mda guide version 1.0.1. Technical report, June 2003.

[OMG06a] OMG. Object Management Group. Meta object facility (mof) 2.0. query/view/transformation specification. Technical report, January 2006.

[OMG06b] OMG. Object Management Group. Meta object facility, mof specification version 2.0. Technical report, January 2006.

[onl08] Aspect-oriented modelling workshops website, 2008.

[OSG09] OSGi Alliance. Osgi framework., last visited in June 2009.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer, Berlin, 2005.

[PM06] Klaus Pohl and Andreas Metzger. Variability management in software product line engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 1049–1050, New York, NY, USA, 2006. ACM.

[Pttt07] Marc Pantel, ACADIE team, OLC team, and TOPCASED team. The topcased project - a toolkit in open source for critical applications and systems design. In *TOOLS EUROPE / Model-Driven Development Tool Implementers Forum (MDDTIF)*, Zurich, Switzerland, 2007.

[Pur09] Pure Systems. Pure::variants, last visited in June 2009.

[RA09] Andres Romero and Hugo Arboleda. Modelos de decisión como mecanismo decomposición de reglas de transformación. *Paradigma*, 3(2), August 2009.

[RBSP02] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending feature diagrams with uml multiplicities. In *Proceedings of the 6th World Conference on Integrated Design & Process Technology (IDPT2002)*, Pasadena, California, 2002.

[Rei09] 10th international conference on feature interactions (icfi 2009). Lisbon, Portugal, June 2009.

[Rom09] Andres Romero. *Derivación en Lineas de Productos de Software Dirigidas por Modelos Usando Modelos de Decision.* PhD thesis, Universidad de Los Andes, Bogotá, Colombia, July 2009.

[Ryd79] B. G. Ryder. Constructing the call graph of a program. *IEEE Trans. Softw. Eng.*, 5(3):216–226, 1979.

[SD07] Marco Sinnema and Sybren Deelstra. Classifying variability modeling techniques. *Inf. Softw. Technol.*, 49(7):717–739, 2007.

[SJ04] Klaus Schmid and Isabel John. A customizable approach to full lifecycle variability management. *Science of Computer Programming*, 53(3):259–284, 2004.

[SLFG08] P. Sanchez, N. Loughran, L. Fuentes, and A. Garcia. Engineering languages for specifyingproduct-derivation processes in softwareproduct lines. In *Proceedings of the First International Conference in Software Language Engineering (SLE'08)*, Toulouse, France, September 2008.

[Sof09] Software Construction Group, University of Los Andes. Model-Driven Software Product Line Engineering: Tool Support and Case Studies, last visited in June 2009.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[Tel09] France Telecom. SmartQVT: An open source model transformation tool implementing the MOF 2.0 QVT-Operational language. Web Site, last visited in June 2009.

[vdL02] Frank van der Linden. Software product families in europe: The esaps & caféé projects. *IEEE Softw.*, 19(4):41–49, 2002.

[VG07a] M. Voelter and I. Groher. Handling variability in model transformations and generators. In *Proceedings of the 7th Workshop on Domain-Specific Modeling (DSM'07) at OOPSLA '07*, 2007.

[VG07b] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *Proceedings of the 11th International Software Product Line Conference*, pages 233–242, 2007.

[vGB02] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

[vO02] Rob van Ommering. Building product populations with software components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 255–265, New York, NY, USA, 2002. ACM.

[Voe05] Markus Voelter. Patterns for handling cross-cutting concerns in model-driven software development. In *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP)*, Irsee, Bavaria, Germany, July 2005.

[Wag05] Dennis Wagelaar. Context-driven model refinement. In *MDAFA*, volume 3599 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2005.

[Wag08a] 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.

[Wag08b] Dennis Wagelaar. *Platform Ontologies for the Model-Driven Architecture*. PhD thesis, April 2008.

[WL99] David M. Weiss and Chi T. R. Lay. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

Appendix A

Model Transformation Rules

In this appendix we present the following:

1. The model-to-text transformation rules to generate executable oAW workflows from decision models.
2. The model-to-text transformation rules to generate Check expressions from constraint models.

A.1 Model-to-Text Transformation Rules to Generate Executable oAW Workflows From Decision Models

---

```
«REM»
/**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * $Id: appendixA.tex,v 1.13 2009/08/11 23:40:04 hf.
    arboleda34 Exp $
 * $Responsible: Hugo Arboleda$
 * $Plug-in: Decision models to workflow
 * $Rule Type: Model2Text$
 * Universidad de los Andes (Bogotá - Colombia)
 * Departamento de Ingeniería de Sistemas y Computación
 * Software Construction Group
```

```
 *  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 */

 /**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * Transformation Rules Begin
 *  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 */
«ENDREM»

«DEFINE Root(String nameFile) FOR decisionModel::Workflow»

 «FILE nameFile -»
<?xml version="1.0"?>
<workflow>

 <component class="co.features.ConfigureKeywords" />
 <property file="workflow.properties" />

 <readFineConfig uri="«fileConfiguration»" />

 «EXPAND loadConfiguration FOR this -»
 «EXPAND transformationProgram(this) FOR first -»

</workflow>
 «ENDFILE-»
«ENDDEFINE»

«DEFINE loadConfiguration FOR decisionModel::Workflow -»
 <component class="org.eclipse.mwe.emf.Reader">
  <useSingleGlobalResourceSet value="true" />
  <uri value="«fileConfiguration»" />
  <modelSlot value="configurationModel" />
    </component>
«ENDDEFINE»

«DEFINE transformationProgram(decisionModel::Workflow wf)
   FOR decisionModel::TransformationProgram -»
 «FOREACH sourceModels AS sModel-»
  «IF !wf.modelTransformations.select(e|e.metaType==
      decisionModel::Model2Model).exists(e|((decisionModel::
```

```
      Model2Model)e).targetModel.alias == sModel.alias) -»
    «EXPAND loadModel FOR sModel -»
   «ENDIF -»
  «ENDFOREACH -»

  «IF this.metaType == decisionModel::Model2Model -»
   «EXPAND transformationModel2Model FOR (decisionModel::
      Model2Model)this -»
  «ELSEIF this.metaType == decisionModel::Model2Text -»
   «EXPAND transformationModel2Text FOR (decisionModel::
      Model2Text)this -»
  «ENDIF»

  «IF next != null -»
   «EXPAND transformationProgram(wf) FOR next -»
  «ENDIF -»

«ENDDEFINE»

«DEFINE loadModel FOR decisionModel::Model -»
  «IF this != null-»
  <component class="org.eclipse.mwe.emf.Reader">
   <useSingleGlobalResourceSet value="true" />
   <uri value="«xmiFile»" />
   <modelSlot value="«alias»" />
     </component>
  «ENDIF -»
«ENDDEFINE»

«DEFINE transformationModel2Model FOR decisionModel::
    Model2Model -»
  «FOREACH aspects AS aspect»
   «EXPAND featureOpen('M2M') FOR aspect -»
   «EXPAND aspectM2M(this) FOR aspect -»
   «EXPAND featureClose FOR aspect -»
  «ENDFOREACH »
  «FOREACH transformationRules.select(e|e.metaType ==
     decisionModel::Base) AS rule»
   «EXPAND baseRuleM2M(this) FOR (decisionModel::Base)rule -
      »
```

```
 «ENDFOREACH »
 «EXPAND saveModel FOR targetModel -»
«ENDDEFINE»

«DEFINE transformationModel2Text FOR decisionModel::
    Model2Text -»
 «FOREACH aspects AS aspect»
  «EXPAND featureOpen('M2T') FOR aspect -»
  «EXPAND aspectM2T(this) FOR aspect -»
  «EXPAND featureClose FOR aspect -»
 «ENDFOREACH »
 «FOREACH transformationRules.select(e|e.metaType ==
     decisionModel::Base) AS rule»
  «EXPAND baseRuleM2T(this) FOR (decisionModel::Base)rule -
      »
 «ENDFOREACH »
«ENDDEFINE»

«DEFINE baseRuleM2M(decisionModel::Model2Model model) FOR
    decisionModel::Base -»
 <transform id="«identifier»">
  <globalVarDef name="configurationModel" value="
      configurationModel.configuration"/>
   «EXPAND metamodels FOR (decisionModel::
       TransformationProgram)model -»
   «EXPAND invokeRule(model) FOR this -»
  <outputSlot value="«model.targetModel.alias»" />
 </transform>
«ENDDEFINE»

«DEFINE baseRuleM2T(decisionModel::Model2Text model) FOR
    decisionModel::Base -»
 <component id="«identifier»" class="org.
     openarchitectureware.xpand2.Generator">
 <globalVarDef name="configurationModel" value="
     configurationModel.configuration"/>
 <metaModel class="org.openarchitectureware.type.emf.
     EmfMetaModel"><metaModelFile value="
     configurationMetamodel.ecore" /></metaModel>
```

```
  «EXPAND metamodels FOR (decisionModel::
      TransformationProgram)model -»
  «EXPAND expandRule(model) FOR this -»
  <outlet path="«model.targetPath»">
   <postprocessor class="org.openarchitectureware.xpand2.
       output.JavaBeautifier" />
  </outlet>
 </component>
«ENDDEFINE»

«DEFINE invokeRule(decisionModel::TransformationProgram
    transformation) FOR decisionModel::Base-»
 «IF this != null -»
  <invoke value="«path»::«fileName»::«ruleName»(«EXPAND␣
      modelsParameters␣FOR␣transformation␣-»)" />
 «ENDIF -»
«ENDDEFINE»

«DEFINE expandRule(decisionModel::TransformationProgram
    transformation) FOR decisionModel::Base-»
 «IF this != null -»
  <expand value="«path»::«fileName»::«ruleName»␣FOR␣«EXPAND
      ␣modelsParameters␣FOR␣transformation␣-»" />
 «ENDIF -»
«ENDDEFINE»

«DEFINE modelsParameters FOR decisionModel::
    TransformationProgram -»
«FOREACH sourceModels AS sModel SEPARATOR "," -»
«sModel.alias -»
«ENDFOREACH -»
«ENDDEFINE»

«DEFINE aspectM2M(decisionModel::Model2Model model) FOR
    decisionModel::Aspect -»
  <transformationAspect adviceTarget="«joinPoint.
      identifier»">
   <extensionAdvice value="«advice.path»::«advice.fileName»
       " />
  </transformationAspect>
```

```
«ENDDEFINE»

«DEFINE aspectM2T(decisionModel::Model2Text model) FOR
    decisionModel::Aspect -»
  <generatorAspect adviceTarget="«joinPoint.identifier»">
   <Advice value="«advice.path»::«advice.fileName»" />
  </generatorAspect>
«ENDDEFINE»

«DEFINE metamodels FOR decisionModel::TransformationProgram
    -»
  «FOREACH sourceModels AS sModel-»
   <metaModel class="org.openarchitectureware.type.emf.
       EmfMetaModel"><metaModelFile value="«sModel.conformTo
       .xmiFile»" /></metaModel>
  «ENDFOREACH -»
  «IF this.metaType == decisionModel::Model2Model  -»
  <metaModel class="org.openarchitectureware.type.emf.
      EmfMetaModel"><metaModelFile value="«((decisionModel::
      Model2Model)this).targetModel.conformTo.xmiFile-»"
      /></metaModel>
  «ENDIF -»

«ENDDEFINE»

«DEFINE saveModel FOR decisionModel::Model-»
 <component class="org.openarchitectureware.emf.XmiWriter">
  <inputSlot value="«alias»" />
  <modelFile value="«xmiFile»" />
 </component>
«ENDDEFINE»

«DEFINE featureOpen(String typeTransformation) FOR
    decisionModel::Aspect -»
 «FOREACH executionCondition.variantStates.select(e|e.
     metaType==decisionModel::FineCondition) AS selection -»
  «EXPAND fineFeatureOpen FOR (decisionModel::FineCondition
      )selection -»
  «EXPAND fineFeatureEcho(typeTransformation) FOR (
      decisionModel::FineCondition)selection -»
```

```
 «ENDFOREACH-»
 «IF executionCondition.variantStates.exists(e|e.metaType
    == decisionModel::CoarseCondition) -»
  «EXPAND coarseFeatureOpen FOR this -»
  «EXPAND coarseFeatureEcho(typeTransformation) FOR this -»
 «ENDIF -»
«ENDDEFINE»

«DEFINE featureClose FOR decisionModel::Aspect -»
 «IF executionCondition.variantStates.exists(e|e.metaType
    == decisionModel::CoarseCondition) -»
  «EXPAND coarseFeatureClose FOR this -»
 «ENDIF -»
 «FOREACH executionCondition.variantStates.select(e|e.
    metaType == decisionModel::FineCondition) AS
    variantStateConfiguration -»
  «EXPAND fineFeatureClose FOR this -»
 «ENDFOREACH-»
«ENDDEFINE»

«DEFINE coarseFeatureOpen FOR decisionModel::Aspect -»
 <coarseFeature «EXPAND isSelectedProperty FOR this -»
    «EXPAND isNotSelectedProperty FOR this -»>
«ENDDEFINE»

«DEFINE isSelectedProperty FOR decisionModel::Aspect -»
«IF executionCondition != null -»
«IF  executionCondition.variantStates.select(e|e.metaType==
   decisionModel::CoarseCondition).exists(e|((decisionModel
   ::CoarseCondition)e).selected==decisionModel::
   SelectionType::SELECTED) -»isSelected="«EXPAND␣
   getSelectedFeatures␣FOR␣this.executionCondition␣-»"
   «ENDIF-»
«ENDIF-»
«ENDDEFINE»

«DEFINE isNotSelectedProperty FOR decisionModel::Aspect -»
«IF executionCondition != null-»
«IF  executionCondition.variantStates.select(e|e.metaType==
   decisionModel::CoarseCondition).exists(e|((decisionModel
```

```
    ::CoarseCondition)e).selected==decisionModel::
    SelectionType::NOT_SELECTED) -»isNotSelected="«EXPAND␣
    getNotSelectedFeatures␣FOR␣this.executionCondition␣-»"
    «ENDIF-»
«ENDIF-»
«ENDDEFINE»

«DEFINE getSelectedFeatures FOR decisionModel::
    ExecutionCondition -»
«FOREACH variantStates.select(e|e.metaType==decisionModel::
    CoarseCondition).select(e|((decisionModel::
    CoarseCondition)e).selected == decisionModel::
    SelectionType::SELECTED)  AS variantState SEPARATOR ','
    -»«variantState.nameFeature -»«ENDFOREACH -»
«ENDDEFINE»

«DEFINE getNotSelectedFeatures FOR decisionModel::
    ExecutionCondition -»
«FOREACH variantStates.select(e|e.metaType==decisionModel::
    CoarseCondition).select(e|((decisionModel::
    CoarseCondition)e).selected == decisionModel::
    SelectionType::NOT_SELECTED)  AS variantState SEPARATOR
    ',' -»«variantState.nameFeature -»«ENDFOREACH -»
«ENDDEFINE»

«DEFINE coarseFeatureClose FOR decisionModel::Aspect -»
 </coarseFeature>
«ENDDEFINE»

«DEFINE coarseFeatureEcho(String typeTransformation) FOR
    decisionModel::Aspect -»
  <echo>
   <message value="executing␣coarse␣aspect␣
       «typeTransformation»␣isSelected=[«EXPAND␣
       getSelectedFeatures␣FOR␣this.executionCondition␣-»]␣
       isNotSelected=[«EXPAND␣getNotSelectedFeatures␣FOR␣
       this.executionCondition␣-»]" />
  </echo>
«ENDDEFINE»
```

```
«DEFINE fineFeatureOpen FOR decisionModel::FineCondition -»
 <fineFeature toFeature="«nameFeature»" boundMetaconcept="
     «metaConcept»">
«ENDDEFINE»

«DEFINE fineFeatureClose FOR decisionModel::Aspect -»
 </fineFeature>
«ENDDEFINE»

«DEFINE fineFeatureEcho(String typeTransformation) FOR
    decisionModel::FineCondition -»
  <echo>
   <message value="executing fine aspect
       «typeTransformation» toFeature=[«nameFeature»]
       boundMetaconcept=[«metaConcept»]" />
  </echo>
«ENDDEFINE»
```

Listing A.1: Model-to-Text Transformation Rules to Generate Executable oAW Workflows From Decision Models.

A.2 Model-to-Text Transformation Rules to Generate Check Expressions From Constraint Models

```
«REM»
/**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * $Id: appendixA.tex,v 1.13 2009/08/11 23:40:04 hf.
    arboleda34 Exp $
 * $Responsible: Hugo Arboleda$
 * $Plug-in: Check expressions generator$
 * $Rule Type: Model2Text$
 * Universidad de los Andes (Bogotá - Colombia)
 * Departamento de Ingeniería de Sistemas y Computación
 * Software Construction Group
 * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
 */
«ENDREM»

«EXTENSION templates::chk::chkGenerator»

«REM»
 /**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * Transformation Rules Begin
 * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 */
«ENDREM»

«REM»
 Root transformation, generates the check file. This
    transformation creates restrictions to validate
 bindings between features and elements. Moreover, invokes
    transformations to create others restrictions.
«ENDREM»
«DEFINE main(String nameFile) FOR constraintMetamodel::
   RootFeature»

 «FILE nameFile+".chk" -»
import configurationMetamodel;

extension org::openarchitectureware::util::stdlib::naming;


«IF eAllContents.exists(e|e.metaType == constraintMetamodel
   ::Constraint) -»
 context Binding ERROR loc()+"You can not create a binding
    between element ("+metaConcept+") and feature ("+
    feature.name+")":
 «FOREACH eAllContents.select(e|e.metaType ==
    constraintMetamodel::Constraint) AS const SEPARATOR
    '||' -»
  «EXPAND constraintText FOR (constraintMetamodel::
     Constraint)const -»
 «ENDFOREACH -»;
«ENDIF -»
```

```
 «FOREACH  eAllContents AS eobject -»
  «IF  eobject.metaType == constraintMetamodel::Constraint -
     »
   «EXPAND constraintValidation FOR (constraintMetamodel::
      Constraint)eobject -»
  «ELSEIF eobject.metaType == constraintMetamodel::
     GroupConstraint -»
   «EXPAND groupConstraintValidation FOR (
      constraintMetamodel::GroupConstraint)eobject -»
  «ENDIF -»
 «ENDFOREACH  -»

 «ENDFILE-»
«ENDDEFINE»

«REM»
  Invokes transformations to create restrictions for
     constraintMetamodel::GroupConstraint elements
«ENDREM»
«DEFINE groupConstraintValidation FOR constraintMetamodel::
   GroupConstraint -»
 «EXPAND constraint1(max, min,feature.getFeatureName(),
    metaConcept) FOR this -»
«ENDDEFINE»

«REM»
  Invokes transformations to create restrictions for
     constraintMetamodel::Constraint elements
«ENDREM»
«DEFINE constraintValidation FOR constraintMetamodel::
   Constraint -»
 «IF feature.metaType ==  constraintMetamodel::
    SolitaryFeature -»
  «EXPAND constraint1(max, min,feature.getFeatureName(),
     metaConcept) FOR this -»
 «ELSEIF feature.metaType ==  constraintMetamodel::
    FeatureGroup -»
  «EXPAND constraint2(max, min,feature.getFeatureName(),
     metaConcept) FOR this -»
```

```
 «ENDIF -»
«ENDDEFINE»

«REM»
   Constraint to specify the min and max number of bindings
      between features and elements for elements conform
   to metaconcept constraintMetamodel::SolitaryFeature and
      constraintMetamodel::GroupConstraint
«ENDREM»
«DEFINE constraint1(Integer max, Integer min, String name,
    String metaConcept) FOR emf::EObject»
context Configuration ERROR loc() + "there are less than
    «min» «metaConcept» element bound to the feature «name»"
    :
 this.binding.select(b|b.feature.name=='«name»' && b.
     metaConcept=='«metaConcept»' ).size >= «min»;

context Configuration ERROR loc() + "there are more than
    «max» «metaConcept» element bound to the feature «name»"
    :
 this.binding.select(b|b.feature.name=='«name»' && b.
     metaConcept=='«metaConcept»' ).size <= «max»;
«ENDDEFINE»

«REM»
   Constraint to specify the min and max number of bindings
      between features and elements for elements conform
   to metaconcept constraintMetamodel::FeatureGroup
«ENDREM»
«DEFINE constraint2(Integer max, Integer min, String name,
    String metaConcept) FOR constraintMetamodel::Constraint»
context Binding ERROR loc() + "there are less than «min»
    «metaConcept» element bound to the feature «name»":
(this.metaConcept == '«metaConcept»' &&
this.feature.metaType == configurationMetamodel::
    GroupedFeature &&
((configurationMetamodel::GroupedFeature)this.feature).
    parentFeature.name == '«name»' &&
((Configuration)this.eContainer).binding.select(b|b.name ==
     this.name && this.metaConcept == '«metaConcept»' && ((
```

```
    configurationMetamodel::GroupedFeature)this.feature).
    parentFeature.name == '«name»').size >= «min») ||
 (this.metaConcept != '«metaConcept»') ||
(this.metaConcept == '«metaConcept»' &&
 this.feature.metaType == configurationMetamodel::
     GroupedFeature &&
((configurationMetamodel::GroupedFeature)this.feature).
    parentFeature.name != '«name»');

context Binding ERROR loc() + "there are more than «max»
    «metaConcept» element bound to the feature «name»":
(this.metaConcept == '«metaConcept»' &&
this.feature.metaType == configurationMetamodel::
    GroupedFeature &&
((configurationMetamodel::GroupedFeature)this.feature).
    parentFeature.name == '«name»' &&
((Configuration)this.eContainer).binding.select(b|b.name ==
     this.name && this.metaConcept == '«metaConcept»' && ((
    configurationMetamodel::GroupedFeature)this.feature).
    parentFeature.name == '«name»').size <= «max») ||
 (this.metaConcept != '«metaConcept»') ||
(this.metaConcept == '«metaConcept»' &&
 this.feature.metaType == configurationMetamodel::
     GroupedFeature &&
((configurationMetamodel::GroupedFeature)this.feature).
    parentFeature.name != '«name»');
«ENDDEFINE»

«REM»
 /**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * Utilities
 * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *
 */
«ENDREM»

«REM»
  Prints the feature name for constraintMetamodel::
     CointainableByF elements
«ENDREM»
```

```
«DEFINE featureName FOR constraintMetamodel::
    CointainableByF -»
«IF this.metaType == constraintMetamodel::SolitaryFeature -
    »«((constraintMetamodel::SolitaryFeature)this).name -»
«ELSEIF this.metaType == constraintMetamodel::FeatureGroup
    -»«((constraintMetamodel::FeatureGroup)this).name -»
«ENDIF-»
«ENDDEFINE»

«DEFINE constraintText  FOR constraintMetamodel::Constraint
     -»
«IF feature.metaType ==  constraintMetamodel::
    SolitaryFeature -»
«EXPAND constraintTextSolitaryFeature(metaConcept) FOR (
    constraintMetamodel::SolitaryFeature)feature -»
«ELSEIF feature.metaType ==  constraintMetamodel::
    FeatureGroup -»
«EXPAND constraintTextGroupFeature(metaConcept) FOR (
    constraintMetamodel::FeatureGroup)feature -»
«ENDIF -»
«ENDDEFINE»


«DEFINE constraintTextGroupFeature(String nameMetaconcept)
     FOR constraintMetamodel::FeatureGroup -»
«FOREACH children AS gruopedfeature SEPARATOR '||'-»
(metaConcept == '«nameMetaconcept»' && feature.name == "
    «EXPAND␣featureName␣FOR␣gruopedfeature»")
«ENDFOREACH -»
«ENDDEFINE»

«DEFINE constraintTextSolitaryFeature(String
    nameMetaconcept)  FOR constraintMetamodel::
    SolitaryFeature -»
(metaConcept == '«nameMetaconcept»' && feature.name == "
    «EXPAND␣featureName␣FOR␣this»")
«ENDDEFINE»

«REM»
```

```
  Prints the feature name for constraintMetamodel::
     GroupedFeature elements
«ENDREM»
«DEFINE featureName  FOR constraintMetamodel::
    GroupedFeature -»«name»«ENDDEFINE»
```

Listing A.2: Model-to-Text Transformation Rules to Generate Check Expressions From Constraint Models.