# A Generic Approach for Automated Verification of Product Line Models

Raul Mazo

# A Generic Approach for Automated Verification of Product Line Models

A thesis submitted by

Raúl Mazo Peña

in partial satisfaction of the requirements for the degree of

**Doctor of Philosophy in Computer Science**

Centre de Recherche en Informatique (CRI)

Paris 1 Panthéon – Sorbonne University, Paris, France.

Defended the 24 November 2011.

**Members of the jury:**

| | |
|---|---|
| Dr. Patrick Heymans. Professor at University of Namur, Belgium. | President |
| Dr. Camille Salinesi. Professor at Paris 1 Panthéon – Sorbonne University, France. | Advisor/ Supervisor |
| Dr. Colette Rolland. Professor at Paris 1 Panthéon – Sorbonne University, France. | Co-Advisor/ Supervisor |
| Dr. Régine Laleau. Professor at Paris-Est Créteil University, France. | Reviewer |
| Dr. Peter Sawyer. Professor at the Lancaster University, United Kingdom. | Reviewer |
| Dr. Daniel Diaz. Associate Professor at Paris 1 Panthéon – Sorbonne University, France. | Examiner |

This page intentionally left blank

# Thèse de Doctorat de l'Université Paris 1 – Panthéon Sorbonne

Spécialité

**Informatique**

Présentée par

Raúl Mazo Peña

Pour l'obtention du diplôme de Docteur de
l'Université Paris 1 Panthéon – Sorbonne

# Méthode Générique pour la Vérification Automatique de Modèles de Lignes de Produits

Soutenue publiquement le 24 novembre 2011 devant la commission d'examen composée de :

| | |
|---|---|
| Dr. Patrick Heymans. Professeur à l'Université de Namur, Belgique. | Président |
| Dr. Camille Salinesi. Professeur à l'Université Paris 1 Panthéon – Sorbonne, France. | Directeur de thèse |
| Dr. Colette Rolland. Professeur à l'Université Paris 1 Panthéon – Sorbonne, France. | Co-directeur de thèse |
| Dr. Régine Laleau. Professeur à l'Université Paris-Est Créteil, France. | Rapporteur |
| Dr. Peter Sawyer. Professeur à l'Université de Lancaster, Royaume Uni. | Rapporteur |
| Dr. Daniel Diaz. Maître de conférences à l'Université Paris 1 Panthéon – Sorbonne, France. | Examinateur |

This page intentionally left blank

# A Generic Approach for Automated Verification of Product Line Models

This thesis explores the subject of automatic verification of product line models. This approach is based on the hypothesis that to automatically verify product line models, they should first be transformed into a language that makes them computable. In this thesis, product line models are transformed into constraint (logic) programs, then verified against a typology of verification criteria. The typology enumerates, classifies and formalizes a collection of generic verification criteria, i.e. criteria that can be applied (with or without adaptation) to any product line formalism. The typology makes the distinction between two categories of criteria: criteria that deal with the formalism in which models are represented, and the formalism-independent criteria. To identify defects in the first category, the thesis proposes a conformance checking approach directly related with verification of the abstract syntactic aspects of a model. To identify defects in the second category, the thesis proposes a domain-specific verification approach. An optimal algorithm is specified and implemented in constraint logic program for each criterion in the typology. These can be used independently - or in combination- to verify individual product line models. The thesis offers to support the verification of multiple product line models using an integration approach. Besides, this thesis proposes a series of integration strategies that can be used before applying the verification as for individual models. The product line verification approach proposed in this thesis is generic in the sense that it can be reused for any kind of product line model that instantiates the generic meta model based on which it was developed. It is general in the sense that it supports the verification of a comprehensive collection of criteria defined in the typology. This approach was implemented in a prototype tool that supports the specification, transformation, integration, configuration, analysis and verification of product line models via constraints (logic) programming. A benchmark gathering a corpus of 54 product line models was developed, then used in a series of experiments. The experiments showed that (i) the implementation of the domain-specific verification approach is fast and scalable to product line models up-to 2000 artefacts; (ii) the implementation of the conformance checking approach is fast and scalable to product line models up-to 10000 artefacts; and (iii) both approaches are correct and useful for industrial-size models.

# Declaration

I declare that this thesis was written by myself and presents my own work developed in the context of my PhD in the Centre de Recherche en Informatique at Paris 1 Panthéon-Sorbonne University and a PhD internship in the Institute for Software Engineering and Automation (SEA) at the Johannes Kepler University, Austria. The work reported in this thesis has not been previously submitted for a degree, in this or any form.

This research was fully funded by the "Allocataire de recherche" fellowship of the French Minister of Education and Research. The internship at the SEA laboratory was fully funded by the “Bourse de Mobilité d’Île de France” and the “Programme pour la mobilité des doctorants: Aires culturelles” programs.

# Remerciements/Agradecimientos

Je voudrais tout d'abord exprimer mes vifs remerciements à Monsieur Camille Salinesi et à Madame Colette Rolland, Professeurs à l'Université Paris 1 Panthéon – Sorbonne, pour la confiance qu'ils m'ont témoignée en m'accueillant dans le Centre de Recherche en Informatique et en acceptant la direction scientifique de mes travaux. Je leurs suis reconnaissant aussi de m'avoir encadré tout au long de ce travail avec une grande compétence, efficacité et rigueur scientifique. Enfin, je les remercie aussi pour leurs précieux conseils, leur soutien et de leur patience pendant ces trois années.

Je remercie sincèrement Madame Régine Laleau, Professeur à l'Université Paris–Est Créteil, et Monsieur Peter Sawyer, Professeur à l'Université de Lancaster (UK), qui ont eu la gentillesse d'accepter les rôles de rapporteurs.

Je remercie également Monsieur Patrick Heymans, Professeur à l'Université de Namur (Belgique) et Monsieur Daniel Diaz, Maître de conférences à l'Université Paris 1 Panthéon – Sorbonne pour avoir accepté de faire partie du jury de cette thèse.

Je remercie tous les (ex-)membres de l'équipe du Centre de Recherche en Informatique, Mme. Gire, Mme. Souveyet, Mme. Nurcan, Rébecca, Manuele, Irina, Charlotte, Said, Farida, Christophe, Yves-Royer, Islem, Adrian, Bruno, Assia, Salma, Sana, Elena, Olfa, Yves, Kadan, Amina, Ramzy, Kahina, Cosmin, Hela, Hicham, Oumaima, Manuelle, Astrid et Stéphane, ainsi que mes collègues de l'AFIS, pour leur amitié, collaboration, gentillesse et soutien.

A mis amigos: Alberto, Santiago, Padre Samuel, Germán, Judith, Nelly, Margarita, Michelle, Serge, Laura y Aldrin. Y por supuesto a las familias amigas: De los Ríos, Marín, Gualteros, Tabares, Tamayo, Giraldo, Mesa, López-Herrejón, Boulanger, Matos, Kutra, Moreau, Douthe, Garcìa y Velandia. A todos mil gracias por la ayuda que me brindaron durante estos años, por los consejos, y por la amistad que me han permitido construir con ustedes.

A mi familia en Colombia y a la familia de mi esposa acá en Francia (Morizot y Chaise), mis agradecimientos más sinceros por su ayuda, por sus ánimos y por su paciencia.

This page intentionally left blank

I would like to dedicate this thesis to my wife Marielle and my mother Alicia.

Je voudrais dédier cette thèse à mon épouse Marielle et à ma mère Alicia.

A mi esposa Marielle y mi mamá Alicia.

This page intentionally left blank

# Contents

# List of Figures

# List of Tables

This page intentionally left blank

# Chapitre 1 en Français

# Introduction

La production de masse n'est pas un phénomène nouveau: depuis le 12ème siècle à Venise, jusqu'à nos jours, les industries du navire, du coton et de la voiture utilisent ce concept pour accroître l'efficacité et réduire les pertes, deux facteurs clés pour améliorer les bénéfices. Cependant, le consumérisme de nos jours rend la production de masse insuffisante pour satisfaire les nouvelles exigences où la personnalisation est essentielle. Un nouveau paradigme de production était nécessaire pour soutenir ces nouvelles exigences, en gardant des coûts de production raisonnables, des besoins de main d'œuvre et des délais de commercialisation. Comme réponse à ce besoin, l'ingénierie des lignes de produits surgit comme un nouveau paradigme de développement conduit par la réutilisation qui permet la gestion de composants réutilisables. Dans cette thèse, une ligne de produits est définie comme un groupe d'applications similaires au sein d'un secteur de marché et qui partage un ensemble commun d'exigences, mais aussi présente une variabilité importante des exigences (Bosch 2000, Clements & Northrop 2001).

Le concept central pour traiter la réutilisation dans l'ingénierie des lignes de produits est la définition de composants communs et variables dans un modèle de domaine ou Modèle de Lignes de Produits (MLPs). Un MLP définit l'ensemble des combinaisons correctes de composants réutilisables de la ligne de produits (Pohl et al. 2005) par le biais des relations qu'il y a entre eux. Les composants communs font référence à des parties, des aspects, des exigences (Sommerville & Sawyer 1997, Sawyer 2005) ou n'importe quel type de caractéristiques de la ligne de produits qui font partie de tous les produits de la ligne de produits. Les composants variables font référence aux éléments réutilisables qui font partie de certains produits (mais pas tous) pouvant être construits à partir de la ligne de produits.

**Pourquoi les lignes de produits sont-elles importantes?**

Comme nous venons de le voir, la stratégie de production orientée lignes de produits a plusieurs avantages. Selon l'étude réalisée par Clements & Northrop (2001) l'approche de production orientée lignes de produits diminue non seulement le coût par produit (jusqu'à

60%), mais aussi le temps de mise sur le marché (jusqu'à 98%), le besoin en main d'œuvre (jusqu'à 60%) et améliore la productivité (jusqu'à 10 fois), la qualité de chaque produit dérivé (jusqu'à 10 fois) et augmente la taille du portefeuille, et ainsi, la possibilité de gagner de nouveaux marchés.

Cependant, il peut aussi avoir des inconvénients. Par exemple, l'assurance qualité dans le contexte des lignes de produits consistant à traiter les problèmes de qualité de chaque produit est très coûteuse, sujette à l'erreur et irréalisable pour des très grandes lignes de produits (Von der Massen & Lichter 2004, Benavides 2007). La contrepartie est qu'un défaut dans un composant du domaine ou dans le MLP peut affecter de nombreux produits de la ligne de produits et donc peut devenir coûteux à supprimer, puisque tous ces produits devraient être corrigés (Lauenroth et al. 2010). Pour cette raison, assurer la qualité au tout début de l'approche de production orientée ligne de produits doit être en soi un processus de haute qualité afin de profiter des avantages qu'elle est sensée fournir.

**Pourquoi les modèles des lignes de produits sont-ils importants?**

L'histoire du développement de logiciels et de systèmes montre que l'abstraction joue un rôle majeur dans la maitrise de la complexité (Bosch, 2000). Ainsi, abstraire des composants communs et variables d'une collection indéfinie de produits et les organiser dans un modèle peut être une bonne option pour gérer la complexité de la ligne de produits. Les modèles des lignes de produits améliorent les processus de prise de décisions. En outre, la représentation de MLPs dans plusieurs vues améliore la communication des acteurs participant à la gestion des lignes de produits (Finkelstein et al. 1992). Nuseibeh et al. (1994) décrivent les vues comme des représentations partielles d'un système et de son domaine.

**Pourquoi l'assurance qualité des modèles de la ligne de produits est-elle importante?**

L'Ingénierie des Lignes de Produits (ILP) est un nouveau paradigme de développement conduit par la réutilisation qui a été appliqué avec succès dans l'ingénierie de systèmes (Bass et al. 2000, Bosch 2000, Clements & Northrop 2001), dans l'ingénierie des processus métier (Rolland et al. 2007, Rolland & Nurcan 2010) et dans d'autres domaines (Pohl et al.2005). Cependant, le succès de ce nouveau paradigme de développement dépend fortement de la qualité des MLPs. Bien qu'il ne soit pas possible de garantir la qualité totale de MLPs (Batory 2005), ni de prouver qu'un modèle est correct, la qualité peut être améliorée au moyen d'un processus de vérification.

## F.1.1 Problématique

Malgré le succès relatif des approches existants de vérification de MLPs (Von der Maßen & Lichter 2004, Zhang et al. 2004, Batory 2005, Czarnecki & Pietroszek 2006, Benavides 2007, Janota & Kiniry 2007, Lauenroth & Pohl 2007, Trinidad et al. 2008, Van den Broek & Galvão 2009, Kim et al. 2011, Liu et al. 2011), il ya encore un certain nombre de questions qui restent ouvertes et qui ont motivé la recherche présentée dans cette thèse:

**a.** Les techniques d'assurance de la qualité du développement des systèmes simples ne peuvent pas être appliquées directement aux spécifications des lignes de produits car ces spécifications contiennent de la variabilité. Comme l'exemple de Lauenroth et al. (2010) le montre, une ligne de produit peut contenir des exigences E et ¬E en même temps. L'utilisation d'une technique traditionnelle pour vérifier cette spécification soulèvera une contradiction puisque les exigences E et ¬E ne peuvent pas être incluses dans le même produit. Par conséquent, il est nécessaire de prendre en compte la variabilité de la ligne de produits afin de vérifier que les exigences contradictoires ne peuvent pas faire partie d'un même produit.

**b.** L'état de l'art sur la vérification spécifique au domaine des LPs est principalement axé sur les modèles de caractéristiques (Kang et al. 1990). Seules les propriétés qui peuvent être évaluées par rapport aux modèles de caractéristiques représentés comme expressions booléennes sont pris en compte dans ces travaux. Ceci écarte les éléments non-booléens des formalismes de spécification de lignes de produits les plus sophistiqués (par exemple, cardinalités sur le domaine des entiers, attributs et contraintes complexes). La raison sous-jacente est que la plupart des approches actuelles restreignent les opérations de vérification à celles qui peuvent être résolues par des solveurs booléens. La vérification est donc guidée par la technologie présélectionnée et non par les exigences de vérification elles-mêmes. En conséquence, les techniques de vérification sont conçues pour un nombre limité de formalismes. Ces techniques de vérification sont inadaptées pour la plupart des formalismes existants, certains de ces formalismes sont déjà utilisés dans l'industrie (Djebbi et al. 2007, Dhungana et al. 2010).

**c.** L'état de l'art du développement de lignes de produits montre un support inadéquat pour la vérification de multi-modèles. La taille et la complexité de MLPs industriels motive le développement des modèles par des équipes hétérogènes (Dhungana et al. 2006, Segura et al. 2008). Néanmoins, les outils existants fournissent peu de support pour l'intégration des modèles développés par différentes équipes, pour la vérification ultérieure du modèle

global et pour des configurations de produits à partir de ces modèles. Par exemple, un modèle global qui intègre deux modèles doit lui-même ne pas présenter de défauts résultant de l'intégration.

## F.1.2 Questions de recherche

Cette thèse porte sur les trois problèmes précédents en proposant une approche qui guide la vérification des modèles de lignes de produits indépendamment du langage et du nombre de modèles dans lesquels la LP est spécifiée. De cette façon, l'approche proposée dans cette thèse peut être réutilisée indépendamment du langage et du nombre de modèles de la ligne de produits. Par conséquent, l'approche proposée est générique, ce qui permet une application directe ou une adaptation sur les MLPs Ainsi, l'objectif principal de cette thèse est de répondre à la question de recherche suivante:

*Principale question de recherche: Comment des-modèles de ligne de produits peuvent-ils être automatiquement vérifiés d'une manière générique et réutilisable?*

## F.1.3 Hypothèses de recherche

Les hypothèses de recherche sont les suivantes:

a. Une approche générique permettra de vérifier les propriétés des spécifications de lignes de produits, indépendamment du langage dans laquelle ces spécifications sont modélisées;
b. Une approche adaptable permettra de vérifier les propriétés structurelles des modèles de lignes de produits par l'adaptation de l'approche de vérification d'origine au langage particulier dans lequel les modèles sont définis;
c. Une implémentation correcte et scalable des approches susmentionnées est possible et utile pour vérifier les modèles de lignes de produits.

## F.1.4 Méthode de recherche

Adoptant la stratégie Design Science, cette méthode de recherche est en accord avec le modèle de processus de conception des sciences proposé par Peffers et al. (2007). La Figure F.1.1 présente, en gris, le modèle de processus design science pour la recherche en systèmes d'information, et l'application de ce procéssus aux recherches menées dans cette thèse.

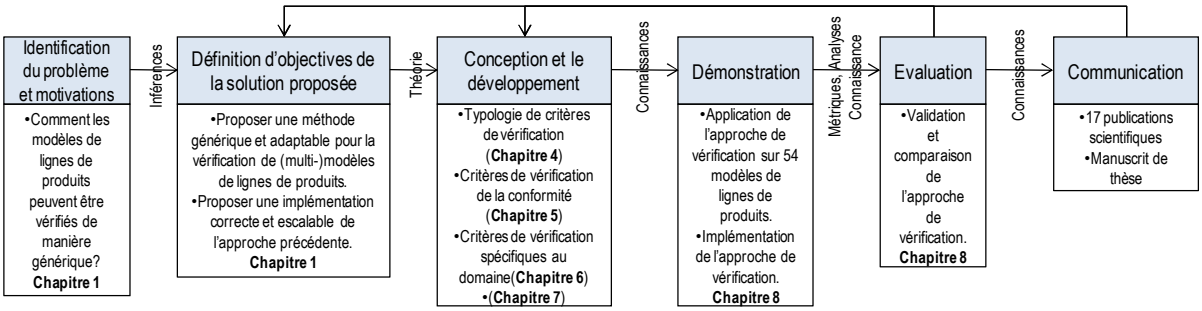



**Figure F. 1.1.** Instanciation de la méthodologie de recherche proposée par (Peffers et al. 2007).

# F.1.5 Contributions

Pour surmonter les limites présentées dans la Section 1, cette thèse propose une approche indépendante du langage et entièrement automatisée pour la vérification de modèles de lignes de produits. En particulier, les principales contributions de cette thèse sont les suivantes:

a. **Spécification de modèles de lignes de produits dans programmes logiques de contraintes**. Dans cette approche, nous transformons (i) la structure des MLPs et ses métamodèles associés par des faits de programmation logique par contraintes, et (ii) la sémantique des MLPs dans des programmes par contraintes. Cette thèse propose une collection de règles de transformation (Salinesi et al. 2011, Mazo et al. 2011E) et deux stratégies (Mazo et al. 2011e) pour transformer les MLPs en programmes de contraintes.
b. Une fois que le modèle est représenté comme un programme logique de contraintes, la vérification est guidée par une **typologie de critères de vérification** (Salinesi et al. 2010a, Salinesi & Mazo 2012). Cette typologie de critères de vérification permet de trouver des défauts spécifiques au domaine et de conformité sur les modèles de ligne de produits. Cette typologie n'est pas une contribution *per se*, bien au contraire, cette typologie contient la collection des critères de vérification trouvée dans la littérature (parfois avec des noms différents), classifie ces critères en fonction de leur nature et leur impact dans un processus de vérification, et organise ces critères dans un ordre approprié pour leur utilisation et leur implémentation.
c. Une **approche générique pour vérifier les propriétés spécifiques au domaine des LPs**. Ces propriétés spécifiques au domaine sont associées à l'expressivité et l'absence d'erreurs, d'incohérences et de redondances dans les MLPs. Cette approche est générique car les MLPs sont représentés comme des programmes de contraintes et ensuite vérifiés contre un ensemble de critères de vérification spécifique au domaine des LPs que tout MLP devra respecter.

**d.** Une **approche adaptable pour vérifier la conformité des MLPs** avec leur métamodèle correspondant. Dans le cas de la vérification de conformité, l'approche consiste à vérifier si la syntaxe abstraite (Harel & Rumpe 2000, 2004) de MLPs est correcte par rapport au métamodèle correspondant. L'approche de la vérification de la conformité proposée dans cette thèse est basée sur une collection de contraintes prises à partir d'un métamodèle générique. Le métamodèle générique concerne les concepts communs trouvés dans les formalismes de LPs que nous avons étudié.

**e.** Une **amélioration de la scalabilité des algorithmes de vérification de MLPs existants**. La validation de la démarche de vérification présentée dans cette thèse a été réalisée au moyen de deux outils. Les deux implémentations ont été testées à partir d'un benchmark constitué à partir de cas industriels et académiques. L'exactitude et la performance de l'implémentation de la vérification spécifique au domaine ont été comparées à deux outils connus (i.e. Fama et SPLOT). Les résultats sont prometteurs et l'implémentation est exploitable pour des MLPs qui ont jusqu'à 2000 composants. L'implémentation de la vérification de la conformité n'a pas été comparée avec d'autres implémentations, car aucun autre outil pour vérifier la conformité des modèles de ligne de produits n'a été trouvé dans la littérature; pourtant l'exactitude des résultats a été vérifiée manuellement. La performance de l'implémentation proposée est prometteuse et exploitable pour les modèles qui ont jusqu'à 10000 artefacts.

# Chapter 1

# Introduction

Mass-production is not new, from the 12th century in Venice to our days, ship, gold extraction, cotton and car industries used this concept to increase efficiency and less waste, two key factors to improve profit. However, the consumerism of our days makes mass-production insufficient to satisfy the new requirements in which customization is essential. A new production paradigm was needed to support these new requirements, keeping reasonable production costs, labour needs and time to market. As an answer to this need, product line engineering arises as a new reuse-driven development paradigm that permits the management of reusable artefacts. In this thesis, a product line is defined as a group of similar applications within a market segment and that shares a common set of requirements, but also exhibits significant variability in requirements (Bosch 2000, Clements & Northrop 2001). According to Clements & Northrop (2007) product line engineering differences from single-system development with reuse in two aspects: First, building a product line implies the development of a family of product with often "choices and options that are optimized from the beginning" and not just one that evolves over time. And second, it implies a preplanned reuse strategy that applies across the entire set of products rather than ad hoc or one-time-only reuse. Two examples of product lines are (i) "the software for commercial avionics and the software for military avionics" (Clements & Northrop 2007), each one serving different market segments but being developed as a single product line by a software group; and (ii) the vehicle product line of the French manufacturer Renault that can lead to $10^{21}$ configurations for the van family "Traffic" (Dauron & Astesana 2010).

Product line engineering explicitly addresses reuse by differentiating between two kinds of development processes (Pohl *et al.* 2005): domain engineering and application engineering.

***Definition 1.1: Domain Engineering.*** *During domain engineering the requirements, specifications, artefacts, domain tests and evolution of the product line are managed in a coherent process.*

The aim of the domain engineering process is to manage the reusable artefacts participating in the PL and the dependencies among them (Stropky & Laforme 1995). The reusable artefacts, called domain artefacts, are for instance: requirements, architectural components, pieces of processes, methods, tests, etc.

***Definition 1.2: Application Engineering.*** *During application engineering the requirements, architectures, specifications, tests and evolution of each application (or product) of the product line are managed in a coherent process.*

The aim of the application engineering process is to exploit the variability of the PL in order to derive specific applications by reusing the domain artefacts.

The central concept for addressing reuse in product line engineering is the definition of common and variable artefacts on the product line model. A Product Line Model (PLM) defines all the legal combinations of reusable artefacts of the product line (Pohl *et al*. 2005) by means of relationships among them. *Common artefacts* refer to parts, aspects, requirements (Sommerville & Sawyer 1997, Sawyer 2005) or any kind of features of the product line that are part of all the products of the product line. *Variable artefacts* refer to the possible variations of the product line. In other words, variable artefacts refer to reusable elements that are part of some, but not all, products that can be build from the product line.

**Why are product lines important?**

As discussed above, there are several advantages to the product line production strategy. According to the study realized by Clements & Northrop (2001) the product line production approach decreases not only the cost per product (by as much as 60%), but also the time to market (by as much as 98%), the labour needs (by as much as 60%) and improves the productivity (by as much as 10x), the quality of each derived product (by as much as 10x) and increases the portfolio size and therefore the possibility to gain new markets.

However, there are also drawbacks. For example, quality assurance in the product line context, which consists of assuring the quality of the domain artefacts instead of treating quality issues in each product, is very expensive, error-prone and computationally infeasible in very large product lines (Von der Maßen & Lichter 2004, Benavides 2007). The counter part is that a defect in a domain artefact can affect many products of the product line and thus can become costly to remove, as all those products might have to be corrected (Lauenroth *et al.* 2010). For this reason, assuring quality from the very beginning of the product line

production approach must be itself a process of high quality in order to take advantage of the benefits that it is expected to provide.

**Why are product lines models important?**

The history of software and system development shows that abstraction plays a major role in making complexity manageable (Bosch 2000). Thus, abstracting the common and variable artefacts of an undefined collection of products and organising them into a model may be a good option to manage the complexity of the product line. Product line models improve decision-making processes. In addition, the representation of PLMs in different views improves communication of the actors participating in the product line management (Finkelstein *et al.* 1992). Nuseibeh *et al.* (1994) describe views as partial representations of a system and its domain.

Several approaches have been found in literature to represent commonality and variability of a product line. Most of the approaches use features (Kang *et al*. 1990) as the central concept of product line models. However, other modelling approaches exist like Orthogonal Variability Models (OVM, cf. Pohl *et al.* 2005), Dopler variability models (Dhungana *et al.* 2010), Textual Variability Language (TVL, cf. Boucher *et al.* 2010 and Classen *et al.* 2011), Extended KAOS (Semmak *et al.* 2009, 2010) and constraint-based product line language (Salinesi *et al.* 2010b, Salinesi *et al.* 2011).

**Why is quality-assurance of product line models important?**

Product Line Engineering (PLE) is a reuse-driven development paradigm that has been applied successfully in systems engineering (Bass *et al.* 2000, Bosch 2000, Clements & Northrop 2001), business process engineering (Rolland *et al.* 2007, Rolland & Nurcan 2010) and other domains (Pohl *et al.* 2005). However, the success of this development paradigm highly depends on the quality of the PLMs. Although it is not possible to guarantee the total quality of PLMs (Batory 2005), neither to prove that a model is correct, the quality can be improved by means of a verification process.

***Definition 1.3: PLM Verification.*** *Verification of product line models, at the domain engineering level, consists of finding defects in the product line model itself.*

The verification process can be considered from two points of view: verification of semantic-related criteria and verification of syntax-related criteria. This thesis refers to the first category as *domain-specific verification* and to the second one as *conformance checking*.

***Definition 1.4: Domain-specific verification*** *is about the identification of non-structural defects on product line models.*

Domain-specific verification is directly related with aspects of the domain of product lines. Some of these aspects are common with other than PLMs, like the verification of redundancies. Other aspects are specific to product lines domain, like the verification that a model should permit several configurations.

***Definition 1.5: Conformance Checking*** *consists of verifying that a model satisfies the constraints captured in the meta-model.*

According to this definition of conformance checking, taken from Paige *et al*. (2007), it is verified that the model is indeed a valid instance of its meta-model. Conformance checking is directly related with the syntactic properties that a model should respect according to the constraints defined in the corresponding metamodel. Some of these syntactic aspects are generic to every PLM and other aspects are particular to each PLM metamodel. An example of generic conformance criteria is that every PLM should be composed of at least one dependency and at least two artefacts; i.e., there is no PLM with only one artefact since a single artefact does not guarantee the minimal variability needed in a PLM. In this thesis, each conformance criterion is automated be means of a conformance rule. The same reasoning can be used for domain-specific verification, and then for verification in general (cf. Definiton 1.6). Conformance rules can be compared to the negation of well-formedness rules of Spanoudakis & Zisman (2001) and Heymans *et al.* (2008), structural rules of Van Der Straeten *et al*. (2003), and syntactic rules of Elaasar & Brian (2004).

***Definition 1.6: A verification rule*** *is the automation of a verification criterion*

In this thesis, to automate verification criteria, the model to be verified must be executed in a solver. Then, one or several queries to the solver must be executed in order to gather the information needed for the verification process.

***Definition 1.7: A solver*** *is a generic term indicating a piece of mathematical software that 'solves' a mathematical problem. A solver takes problem descriptions in some sort of generic form and calculates their solution.*

Quality assurance of PLMs has recently been a prominent topic for researchers and practitioners in the context of product lines. As aforementioned, identification and correction of PLMs defects, is vital for efficient management and exploitation of the product line. Defects that are not identified or not corrected will inevitably spread to the products created from the product line or affect the evolution of the product line, which can drastically diminish the benefits of the product line strategy (Von der Maßen & Lichter 2004, Benavides 2007). Besides, product line modeling is an error-prone activity. Indeed, a product line specification represents not one, but a collection of products that are defined implicitely and that may even include contradictory requirements (Lauenroth *et al.* 2010). The aforementioned problems enforce the urgent need of early identification and correction of defects in the context of product lines.

## 1.1 Problem Statement

Product line model quality has been an intensive research topic over the last decade (Von der Maßen & Lichter 2004, Zhang *et al.* 2004, Batory 2005, Czarnecki & Pietroszek 2006, Benavides 2007, Janota & Kiniry 2007, Lauenroth & Pohl 2007, Trinidad *et al.* 2008, Van den Broek & Galvão 2009, Kim *et al.* 2011, Liu *et al.* 2011). Usually, to guarantee a certain level of quality of a model, this one must be verified against a collection of criteria and then, defects must be corrected. Verifying PLMs entails finding undesirable properties, such as redundancies, anomalies or inconsistencies (Von der Maßen *et al.* 2004). It is widely accepted that manual verification of single products is already tedious and error-prone (Benavides *et al.* 2005). This is even worst when several (up to millions) products are represented altogether in a single specification. Several approaches to automate PLM verification have been proposed in order to overcome this limitation. However, despite the relative success of these approaches, there are still a number of pending issues that have motivated the research presented in this thesis:

a. Quality assurance techniques from the development of single systems cannot be directly applied to product line specifications because these specifications contain variability. As Lauenroth's *et al.* (2010) example shows it, a product line may contain requirements `R`

and ¬R at the same time. Using a traditional technique for verifying this specification will raise a contradiction since requirements R and ¬R cannot be fulfilled together due to the fact that those requirements are not supposed to be included in the same product. Therefore, it is necessary to take into account the variability of the product line to check whether contradictory requirements can really be part of the same product.

b. The current state of the art on domain specific verification is mainly focused on feature models (Kang *et al.* 1990). Only properties that can be evaluated over feature models represented as Boolean expressions are considered in these works. This brushes aside the non-Boolean elements of the more sophisticated product line specification formalisms (e.g., Integer cardinalities, attributes and complex constraints). The underlying reason is that most of current approaches restrict the verification operations to those that can be solved by Boolean solvers. The verification is thus guided by the pre-selected technology and not by the verification requirements themselves. As a result, verification techniques are designed for a limited number of formalisms. These verification techniques are inadequate for many of the existing formalisms, some of these formalisms are already used in industry (Djebbi *et al.* 2007, Dhungana *et al.* 2010).

c. The current state of the art of product line development shows an inadequate support for the verification of PLs specified with several models. The size and complexity of industrial PLMs motivates the development of the product line by heterogeneous teams (Dhungana *et al.* 2006, Segura *et al.* 2008). Nevertheless, existing tools only provide little support for integrating the models developed by different teams and the subsequent verification of the global model and configurations of products from that model. For instance, a global model that integrates two models must itself have no defects resulting from the integration.

## 1.2 Research Questions

The thesis addresses the three aforementioned problems by proposing an approach that guides the verification of product line models independently of the language and the number of models in which the PL is specified. In that way, the approach proposed in this thesis can be reused independently of the language and the number of models of the product line. Consequently, the proposed approach is generic, permitting a direct application or an

adaptation over PLMs. Thus, the main objective of this thesis is to answer the following research question:

*Main RQ: How can product line models be automatically verified in a generic and reusable way?*

To answer this main research question, several sub-problems must be solved too. Resolution of each of the following four research questions is necessary to solve the main research question of the thesis.

*RQ1: How should product line models be formally represented?*

To answer this research question this thesis proposes a language that permits the representation of any product line model. There are two aspects of a PLM that can be represented: its semantics and its structure. On the one hand, the semantics of a PLM is the set of products that can be configured from the PLM. Thus, the semantic representation of a PLM permits configuring, without ambiguity, the same products that can be configured from the PLM itself. The semantic representation of PLMs will be used to verify the domain-specific verification criteria that PLMs must respect. On the other hand, the representation of the structure of a PLM permits representing the elements (or entities on the corresponding metamodel) that constitute the model, the dependencies among them and the order in which these elements are related in the PLM. The structure of PLMs will be used to verify the criteria assuring the respect of the PLM with its corresponding language; i.e., the conformance of the model with the corresponding meta-model. Consequently, both representations are necessary to achieve verification of product line models from the semantic and structural points of view.

*RQ2: How should verification criteria be classified?*

Some properties of PLMs are independent of the language while other ones are particular to each language. This shows that not all criteria are equivalent and therefore several types of defects can be checked in a verification process. Thus, one can be interested in executing one or another verification criterion according to the impact of these criteria, or the expected level of quality of a particular PLM. In addition, this question is about the order in which the verification criteria should be executed in order to improve the performance and the quality of the verification process.

*RQ3: How should different models of a product line system be integrated?*

An important challenge in PL domain engineering and application engineering is that product lines are often, in practice, specified using several models at the same time (Djebbi *et al*. 2007, Segura *et al*. 2008, Rosenmüller *et al*. 2011). This is due to the fact that size and complexity of industrial product lines constrain the specification of PL models by heterogeneous teams (Dhungana *et al*. 2006, Segura *et al*. 2008). In addition, different aspects of the product line will be specified with different models, each one appropriated to the kind of aspect to model. Besides, it is a fact of industrial life that product line models evolve over time, for instance to reflect new marketing requirements, product level innovations that should be capitalized at the PL level, or new design decisions about the PL architecture. The problem is that any change in a model can impact other models too. For example, changes in the architecture can make the corresponding model inconsistent with the technical solution models, or with the PL models that represent the sales and marketing models. Thus, in the absence of a global model, (i) requirements can get missed or misunderstood (Finkelstein *et al*. 1992) both during domain and application engineering activities. Indeed, a particular product line model can be correct when taken standalone and be incorrect when it is integrated with other ones. (ii) Configuration, analysis and verification of the entire product line will be unfeasible.

*RQ4: Which kind of support can be offered to system engineers for improving quality of product line models?*

This question addresses the need of tool support for automatic, efficient and scalable verification of product line models. It is well known that developing high quality systems depends on developing high quality models (Paige *et al.* 2007). Verifying the quality of models has recently been a prominent topic for many researchers in the community. However, the literature review carried out in this thesis shows that scalable methods, techniques and tools are needed to deal with this important issue (cf. Chapter 2). In that way the answer of this question contributes to solve the main concern of this thesis: propose a generic and reusable approach to automatically verify product line models.

## 1.3 Research Hypotheses

Two approaches can be adopted to accomplish the aforementioned objectives. The first approach proposes a collection of generic verification criteria that will be applied on product

line models previously represented with a unique formalism. This approach has the advantage that it makes verification independent of the original language in which the model is represented. The second approach proposes a collection of generic verification criteria that will be adapted to each formalism according to the language in which the model is represented. This second approach is also somehow independent of the language in which the product line specification is modeled. The idea is that verification criteria are adapted, to the formalism at hand, from the original definition. Thus, we have chosen the first strategy for the domain-specific verification criteria and the adaptable strategy to check structural properties. To summarize, research hypotheses are the following:

a. A generic approach will allow verifying domain-specific properties of product line specifications independently of the language in which these specifications are modeled;
b. An adaptable approach will allow verifying structural properties of product line models by adaptation of the original verification approach to the particular language in which the models are defined;
c. A correct and scalable implementation of the aforementioned approaches is possible and useful to verify product line models.

## 1.4 Research Method

The research presented in this thesis, as most of the researches in computer science, is *design oriented*. As defined by March & Smith (1995) and Hevner *et al*. (2004) design science is about design and validation of solution proposals to practical problems. Hevner *et al*. (2004) suggest that design science differs in two aspects from other branches of science: (a) it is concerned with artefacts rather than facts of nature or social structure, and (b) it is concerned with a search for prescriptive rules for design, rather than a search for descriptions, explanations and predictions, as other branches of science are. Simons (1981) also proposes a differentiation between natural science and design science. For him, natural science is about the way things are and design science is concerned with how things ought to be. Being design oriented, the research method used in this thesis intends to validate the research hypotheses presented above by means of prototypes and several case studies. Karl Popper stated that "[a] theory which is not refutable by any conceivable event is non-scientific" (Popper 1974). To test the research hypotheses of this thesis, the following research strategy was implemented:

a. I conducted an investigation of the product line engineering production strategy, its benefits, drawbacks and the modus operandi of this strategy from the point of view of requirements engineering.
b. I conducted a survey of the state of the art in product line engineering. In particular on the techniques, methods and tools for verification of product line specifications.
c. I conducted a state of the art in verification of product line specifications. In particular, these verification techniques, methods and tools were classified according to the kind of verification, the verification criteria proposed in each approach, the kind of specifications in which the approach is applied and the technology used to implement the approach.
d. I identified a collection of gaps and drawbacks of the existing approaches with regards to the research question of this thesis. In particular, to examine how these solutions could be used together to address the problem tackled by this thesis.
e. I proposed a language-independent and fully-automated approach to verify PLMs;
f. I evaluated the correctness of the proposed approach through three case studies. The results of these case studies were intended to support or refute the hypothesis proposed in this thesis (cf. Popperian falsification, Popper (1974));
g. I improved the verification approach initially proposed and at the same time I identified new research directions. The results of the case studies, a follow-up the current verification approaches and the feedback from the computer science community were taken into account to improve the initial approach.

From the design science point of view, this research methodology matches perfectly with the design science process model proposed by Peffers *et al*. (2007). Figure 1.1 presents, in shadow, the design science process model for information system research, and the application of this process to the research carried out in this thesis.

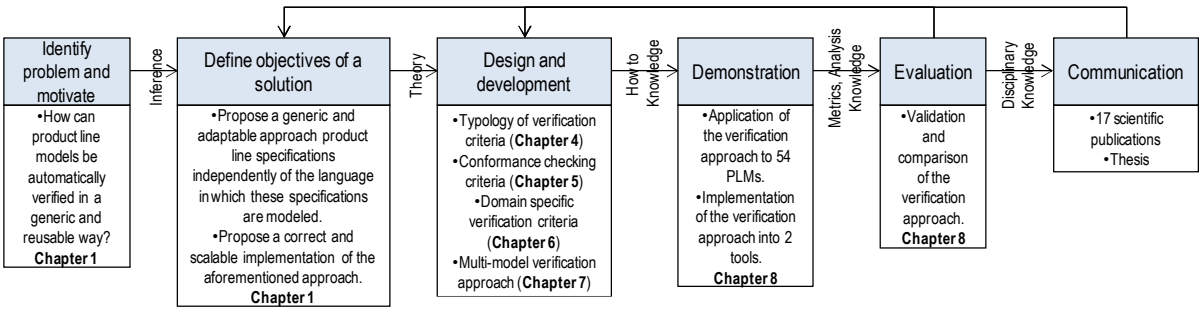



**Figure 1.1.** Application of the design science process model for information system research (Peffers *et al*. 2007) to the research carried out in this thesis.

From an epistemological point of view, this research method reflects Popper's view, in which the advance of scientific knowledge is an evolutionary process characterized by the formula (Popper 1994):

$PS_1 \rightarrow TT_1 \rightarrow EE_1 \rightarrow PS_2$

Following Popper's formula, this thesis takes as a basis a problem situation ($PS_1$). As an attempt to solve this problem, a tentative theory ($TT_1$) with conjectures and limitations is proposed. The approach proposed in this thesis was then systematically subjected to the most rigorous attempts at falsification possible during a limited period of time. However, it does not mean that our approach is true. Nevertheless, as Popper holds, it is more applicable to the problem situation at hand ($PS_1$). Consequently, neither does rigorous testing protect a scientific theory from refutation in the future. Continuing our research method, a particular process of error elimination ($EE_1$) will permit the improvement of our approach and will permit the identification of more interesting problems ($PS_2$).

## 1.5 Contributions

To overcome the limitations presented in Section 1, this thesis proposes a language-independent and fully-automated approach to verify product line models. In particular, the main contributions of this thesis are the following:

a. **Specification of product line models into constraint logic programs**. In this approach, we translate (i) the structure of PLMs and its associated metamodels into constraint logic programming facts, and (ii) the semantics of PLMs into constraint programs. This thesis proposes a collection of transformation rules (Salinesi *et al.* 2011, Mazo *et al*. 2011e) and two strategies (Mazo *et al.* 2011e) to transform product line models into constraint programs.
b. Once the model is represented as a constraint logic program, the user's verification is guided by a **typology of verification criteria** (Salinesi *et al.* 2010a, Salinesi & Mazo 2012). This typology of verification criteria permits finding domain-specific and conformance defects on product line models. This typology is not a contribution *per se*; on the contrary, this typology contains the collection verification criteria found in literature (sometimes with different names), classifies these criteria according to their nature and impact in a verification process, and arranges these criteria in a convenient

order for their use and implementation. In that way, the results obtained from the execution of a verification criterion can be reused in the succeeding executions to save time and computational recourses, at a time that users choose the criteria to verify according to the nature and the intended quality of the model at hand.

c. A generic **approach to verify domain-specific properties in PLMs**. These domain-specific properties are associated with the expressiveness and the absence of errors, inconsistencies and redundancies in PLMs. This approach is generic since PLMs are represented as constraint programs and then verified against a collection of domain-specific verification criteria that any PLM should respect. However, certain verification criteria cannot be used in certain PLMs since the models do not contain the concepts intended to be verified with these particular criteria. For instance, not all PLMs contain the notion of optional artefacts; therefore, verification of false optional artefacts in these models simply has no sense. This approach can also be used to verify domain-specific properties on product lines specified by means of several models, even when models are specified in different notations. To do that, this thesis uses the fact that all PLMs we used can be represented as variables and constraints among these variables (Salinesi *et al*. 2011b, Mazo *et al*. 2011c, 2011d), which allows the definition of a pivot language (i.e., constraint programming) that allows the integration of the PL into a single model. Once the models are integrated into a constraint program, the *modus operandi* to verify multi-models product lines is similar to the one proposed for single-model product lines.

d. An adaptable **approach to check conformance of PLMs** with their corresponding metamodel. In the case of conformance checking, the approach consists of verifying whether the abstract syntax (Harel & Rumpe 2000, 2004) of PLMs is correct with regards to the corresponding metamodel. The conformance checking approach proposed in this thesis is based in a collection of constraints taken from a generic metamodel. The generic metamodel relates the common concepts found in the PL formalisms that we sensed. In that way, even if some verification criteria to check conformance of PLMs are shared for several models, each one of the generic conformance rules should be adapted to each particular formalism. Of course, since the generic metamodel proposed in this thesis only relates common concepts of several PL formalisms, conformance rules corresponding to concepts not present in our generic metamodel should be generated according to the particular metamodel. However, the constraint logic programming-based approach proposed in this thesis will remain being an option to implement the new conformance criteria.

e. **An improvement of the existing PLMs verification algorithms' scalability**. The validation of the verification approach presented in this thesis was carried out by means of two tools. Both implementations were tested with several industrial and academic benchmarks and the correctness and performance of the domain-specific verification implementation was compared with two popular tools (i.e. FaMa and SPLOT). The results are promising and the implementation is scalable to PLMs up-to 2000 artefacts, in the worst-case scenario. The conformance checking implementation was not be compared with others implementation because no other implementation to check conformance of product line models was found in literature; nevertheless the correctness of the results were verified manually. The performance of the implementation proposed is promising and scalable to models up-to 10000 artefacts.

## 1.6 Thesis Organization

This thesis is organized as follows.

**Chapter 2** reviews related work presented in the literature and classifies them according the product line modeling language in which each verification approach is applied. This chapter presents five research questions about the state of the art on verification of product line models (and even in other kind of models), the advancements, gaps and challenges found in literature in this topic. These questions will be systematically answered throughout the chapter.

**Chapter 3** provides an overview of the verification approach presented in this thesis. In addition, this chapter provides the background information necessary for reading this thesis including a transformation and integration approaches, previous stages before verifying product line models. This chapter we introduce also the motivating example that will be used in the rest of the thesis to develop our approach.

**Chapter 4** presents the first contribution of this thesis: a typology of verification criteria developed from our experience with a large number of product line models and the cooperation with industries and other research laboratories. This typology classifies the PLMs verification criteria according to its nature (domain-specific and conformance checking criteria) and its execution order in a verification process. Each criterion is introduced, then formalized using first order logic, then illustrated through our running example.

**Chapter 5** presents the conformance checking approach proposed in this thesis to verify the abstract syntax of product line models. This approach is developed in a running example

and presented from two points of view: generic and metamodel-dependent, for single-models product lines. In addition, generic algorithms and their implementations are also presented in this chapter. References to algorithms found in literature to implement the criteria and a discussion about the performance and scalability of these algorithms are provided.

**Chapter 6** presents the constraint-based approach to automatically verify domain specific criteria of product line models. This approach is centred on standalone product lines models and developed in our running example. In addition, this chapter also presents generic algorithms and their implementations.

**Chapter 7** presents a multi-model verification approach based in the transformation and the integration approaches presented in chapter 3. In addition, this chapter shows how the approaches to verify stand alone models can be also used to verify multi-model product lines.

**Chapter 8** provides details about the running environment we build in order to implement and evaluate our verification approach and the empirical results obtained from this evaluation. We discuss and compare the experimental results, its quality and scalability, against one of the approaches existing in literature.

Finally, **Chapter 9** concludes the thesis and proposes future research directions.

# Chapter 2

# State of the Art

This chapter carries out a literature review in order to examine studies proposing PLMs verification approaches. This review follows the systematic method proposed by Kitchenham (2004) and Webster & Watson (2002). The main aspects regarding the review process are presented as follows.

**Research questions**

The aim of this review is to answer the following questions:

- *Q1: What kind of product line modelling notations have been the subject of verification techniques?*
- *Q2: What verification criteria on product line models have been proposed?*
- *Q3: What kind of automated support has been proposed?*
- *Q4: What kind of validation was made and what have been the results?*
- *Q5: What are the gaps and challenges to be faced in the future?*

Question Q1 gives the structure to this chapter. Each section of this chapter tackles with a product line modelling notation for which at least one verification technique has been applied. There is also a section for the verification approaches independent of the product line modelling notation. The product line modeling notations considered in this state of the art are: FM, OVM, Dopler and Latice Structure.

Questions Q2, Q3 and Q4 have driven the analysis during the literature review presented in this chapter. Some of the aspects discussed in order to solve questions Q2, Q3 and Q4 refer to (a) the verification criteria; (b) scalability; and (c) applicability to large models. Question Q5 is discussed in the conclusion section, based on these aspects and a recapitulation table.

**Source material**

As recommended by Webster & Watson (2002), we used both manual and automated methods to make a selection of candidate papers in leading journals and conferences and other related events. This was augmented with a number of papers, reports and books that relate to product

line engineering. This state of the art presents the results of 40 research works. These 40 works are referred as *primary studies* (Kitchenham 2004).

In the following, each section deals with a formalism. Then, the presentation is done approach by approach. For each approach, (a) the list of criteria handled; (b) verification criteria; (c) algorithms and implementations; (d) details about validation of the approach and; (e) a critical analysis of each approach are presented.

This chapter is structured as follows:

**Section 2.1** presents the state of the art related with verification of Feature Models.

**Section 2.2** presents the state of the art related with verification of Orthogonal Variability Models.

**Section 2.3** presents the state of the art related with verification of Dopler Variability Models.

**Section 2.4** presents the state of the art related with verification of Latice Structure Models.

**Section 2.5** presents the state of the art related with verification approaches that are not entangled with a product line modelling language.

Finally, **Section 2.6** reports a systematic analysis of the related works discussed all along this chapter, in the light of the aforementioned research questions.

# 2.1. Verification of Feature Models

FMs were first introduced in 1990 as a part of the **Feature-Oriented Domain Analysis (FODA)** method (Kang *et al*. 1990). Since then, feature modeling has become a de facto standard adopted by the software product line community to model product lines. A feature model is a compact representation of all the product of a product line in terms of features (requirement, quality, or characteristic of a software system) and dependencies among them. Since the appearance of FODA, several extensions have been proposed to improve and enrich their expressiveness; for instance, cardinalities (Riebisch *et al.* 2002, Czarnecki *et al.* 2005), and attributes (Streitferdt *et al.* 2003, Benavides *et al.* 2005c, White *et al.* 2009). Feature Models (FMs) with these two extensions are called **extended feature models.** The reader can refer to (Schobbens *et al*. 2007) for a detailed survey on the different feature modelling dialects, and to Section 3.4 for the formal definition that is adopted in this thesis to handle the feature modeling language.

### 2.1.1. Verification of FODA-like Feature Models

Seven approaches have been found in the literature to verify FODA-like models: Von der Maßen & Lichter (2004), Van der Storm (2004), Batory (2005), Hemakumar (2008), Broek & Galvão (2009), Salinesi *et al.* (2010a) and Mendonca *et al.* (2009).

**A.** Von der Maßen & Lichter (2004) present an approach to identify *redundancies*, *anomalies* and *inconsistencies*. According to these authors, a feature model contains a redundancy "if at least one semantic information is modeled in a multiple way"; contains anomalies "if potential configurations are being lost, though these configurations should be possible"; and contains inconsistencies "if the model includes contradictory information". (**Verification criteria)**

*Redundancies* identified in the approach are: (i) mandatory and requires relationships between two features; (ii) exclusion of two features related in an alternative relationship; (iii) a feature is required by multiple features *F1,...,Fn* whereas *F1* is a parent of *F2,...,Fn*; (iv) a feature excludes multiple features *F1,...,Fn* whereas *F1* is a parent of *F2,...,Fn*; and (v) transitive relationships among several features.

*Anomalies* identified in the approach are: (i) optional features required by full-mandatory features; (ii) alternative-child features required by full-mandatory features; (iii) or-child features required by full-mandatory features; (iv) optional features mutually exclusive with full-mandatory features; (v) alternative-child features mutually exclusive with full-mandatory features; and (vi) or-child features mutually exclusive with full-mandatory features.

*Inconsistencies* identified in the approach are: (i) exclusion between full-mandatory features; (ii) exclusion between relative-full mandatory features; (iii) requirement between alternative child features; and (iv) mutual exclusion and requirement between two features.

**(Implementation)** Authors use RequiLine (Von der Maßen & Lichter 2003) to validate the approach. Requiline is a tool that allows the detection of inconsistencies on the domain level and on the application level.

**(Validation)** The approach was evaluated in "a small local software company" (Von der Maßen & Lichter 2004) and "in a global player of the automotive industry" (Von der Maßen & Lichter 2004). According to the authors, RequiLine helps to detect inconsistencies in the domain model and in product models.

**(Results)** No information is given about how the automatic detection of redundancies and anomalies is achieved. Neither author provides details about the size of the models or about the technology used to automate the approach. The lack of results about the evaluation experiment makes it difficult to compare or evaluate the approach according to its performance, scalability or usability properties in large models.

**B.** Van der Storm (2004, 2007) proposes an approach to check consistency of feature diagrams and dependency graphs connected with each other by requires-like dependencies. Since graphical formalisms are not practical to perform the verification tasks in an automated way, the author uses a textual version of feature diagrams, called Feature Description Language (FDL) (Van Deursen & Klint 2002). FDL is used to represent the hierarchical structure of feature diagram and cross-tree constraints between features.

**(Implementation)** On the technical level, Van der Storm (2004) proposes the use of Binary Decision Diagram (BDD) solvers to make automatic consistency checking of feature configurations.

**(Verification criteria)** The approach is able to check (i) if feature diagrams are consistent (i.e., feature diagrams permit the generation of one or more products); and (ii) if a configuration is consistent with the feature diagram.

**(Validation)** There is no case-study that shows how the approach works in practice.

**(Results)** Van der Storm does not discuss the conformance checking of feature diagrams regarding his metamodel, or about the application of his work on anything other than feature diagrams—all the consistency checking work is focused on the feature diagrams.

**C.** The proposal of Batory (2005) is to use grammars and propositional formulas to represent basic FMs. Proposition formulas enable the verification process of FMs using truth maintenance systems and SAT solvers.

**(Verification criteria)** Batory's verification proposal identifies contradictory (or inconsistency) predicates and verifying that a given combination of features effectively defines a product.

**(Implementation)** Propositional formulae, in Conjunctive Normal Form (CNF), plus a collection of constraints are derived from FMs represented as grammars. Formulae are not directly derived from the FM. Indeed, the author holds that exclusion and inclusion

constraints of FMs are too simplistic and do not permit the specification of more complex constraints, such as for example: *F implies A or B or C*.

**(Validation)** Once FMs are represented as CNF formulae, they are executed in a SAT solver. By means of a query to the solver it is possible to determine whether there is a valid solution (product) and whether sets of variable assignments satisfy the propositional formulae.

**(Results)** There is no discussion about the application of this approach on anything other than feature models, or about the performance and scalability of this approach on large models.

**D.** As a continuation of the work of Batory, Hemakumar (2008) proposed a dynamic solution to find contradictions on FMs. In this approach, errors can be detected while using FMs, and then reported to domain designers.

**(Verification criteria)** The author proposes an incremental consistency algorithm that verifies if FMs are contradiction-free or not. A FM is contradiction-free if it is k-contradiction free for all `k` where $0 < k \leq n$. A FM is k-contradiction free if no selection of `k` features exposes a contradiction. For example: dead features can be identified when `k=1`. When `k=n`, where `n` is the number of user selectable features, the model is proven to be contradiction-free.

**(Implementation)** Hemakumar holds that this approach, automated with a SAT solver, is at least an order of magnitude faster than model checking.

**(Validation)** The incremental consistency algorithm has important practical limits due to its poor scalability. Indeed, Hemakumar (2008) claims that his approach "can verify contradiction freedom of models with about 20 or fewer features".

**(Results)** Hemakumar (2008) claims that "static analysis to find contradictions in feature models with large number of features may be very difficult". This seems to confirm that the proposed approach is not scalable.

**E.** Broek & Galvão (2009) analyze FODA models specified as generalized feature trees. Their approach transforms FMs into feature trees together with additional constraints specified in the Miranda language (Turner 1985).

**(Verification criteria)** Once FMs are represented in the functional programming language Miranda, the translated FMs are verified against the following criteria: ability to configure

several products, no dead features, and absence of conflicting constraints. This last criterion is used to provide an explanation to "dead features". A dead feature is a feature that does not appear in any product.

**(Implementation)** The approach, fully implemented in Miranda, shows that in cases where there are no cross-tree constraints, the function to detect the existence of products (by generating one of them) has a *O(1)* complexity and the function to find the number of products *O(N)*. If there are cross-tree constraints, the complexity of the function to find the number of products is $O(N*2^M)$, where *N* is the number of features and *M* is the number of cross-tree constraints.

**(Validation)** Unfortunately, these calculations of efficiency are purely theoretical. No systemic empirical evaluation is reported in the paper.

**(Results)** Broek & Galvão (2009) claim that their approach is more efficient than other approaches that require a transformation of the feature tree into another data structure like BDDs. However, Broek & Galvão omit to count the time needed to transform features trees into the Miranda language. The approach was validated with a feature tree of 13 features and two cross-tree constraints, which is not enough to demonstrate its scalability and usability on industrial models.

**F.** Salinesi *et al.* (2009b) present a tool for the automatic verification of structural correctness of feature models supporting group-cardinalities.

**(Verification criteria)** The verification operations implemented in this tool were the identification of redundant features, inconsistent constraints, cyclic relationships, and poorly defined cardinalities.

**(Implementation)** The approach uses graph navigation algorithms, implemented in C#, to evaluate each verification criterion.

**(Validation)** A case study based in two FMs that contain 21 and 49 features was achieved to validate the approach.

**(Results)** This preliminary experiment showed that the approach is effective. However, the approach, proposed for FODA-like models supporting group-cardinalities, presents major scalability issues related with the graph-based algorithms used to implement the approach.

**G.** SPLOT (Mendonca *et al.* 2009) is a Web-based reasoning and configuration system for feature models supporting group-cardinalities instead of alternative and or-relations.

**(Implementation)** The system maps feature models into propositional logic formulas and uses Boolean-based techniques, such as binary decision diagrams and SAT solvers, to reason on feature models.

**(Verification criteria)** SPLOT supports two verification operations: detection of void models and dead features.

**(Validation and results)** The approach presents promising results even with very large models. However, the tool does not support conformance checking, and it only supports feature models.

### 2.1.2. Verification of Extended Feature Models

Six approaches have been found in the literature to verify extended feature models: Zhang *et al*. (2004), Benavides *et al.* (2005a), Benavides *et al.* (2005b, 2006), Janota & Kiniry (2007), Trinidad *et al*. (2008) and Yan *et al*. (2009).

**A.** Zhang *et al*. (2004) propose an approach based on propositional logical expressions to verify FMs.

**(Verification criteria)** The approach covers three criteria: (i) "consistency"; the model is consistent (or not void) if there exists at least one collection of features that does not violate any constraint in the feature model; (ii) "no dead features"; this occurs when each feature in a feature model can be selected without violating any constraint in the feature model; and (iii) each optional feature in a feature model can be removed without violating any constraint in the feature model. The authors hold that by using feature sets, they can reduce the computational complexity of the verification operations.

**(Implementation)** Zang *et al*. argue that these verification criteria can be automated by using model checking techniques such as SMV[1].

**(Validation and results)** However, they do not provide any evaluation to substantiate this claim. No detail about the approach validation and its results are provided.

**B.** Benavides *et al.* (2005a) propose an approach to analyse FMs. Their approach consists of a collection of analysis operations on feature models with attributes and arithmetic relations among these attributes.

**(Implementation)** All these analysis operations are executed using OPL Studio, a commercial Constraint Satisfaction Problem (CSP) solver.

[1] http://www.cs.cmu.edu/~modelcheck/smv.html

**(Verification criteria)** Benavides *et al*'s approach computes the number of products that can be configured from FMs. This operation can be used to verify if a given FM is void, and to verify if the model is rich enough to be considered a product line model, as opposed to just a product model.

**(Validation)** Authors have experimentally inferred that the implementation of the operation to compute the number of products that can be configured from a FM has an exponential behaviour with respect to the number of features. Benavides *et al*. claims that their approach "has a good performance up to 25 features".

**(Results)** The approach is not scalable to large models. The main problem is that real life feature models are usually much larger than the models used to validate this approach.

**C.** Benavides *et al.* (2005b, 2006) present an approach for reasoning on FMs with individual cardinalities and group cardinalities and with complex constraints on attributes.

**(Verification criteria)** Feature models are considered valid if at least one product can be configured from it. Valid configurations are collections of features and attributes that satisfy all the constraints of the corresponding FM.

**(Implementation)** The approach transforms FMs into constraint programs and then, uses CPL Studio, a commercial Constraint Satisfaction Problem (CSP) solver, to check if a given configuration is valid with regard to the FM from which it was configured. In this approach each feature is represented as a CSP variable. The domain of these variables depends on the cardinality associated to each variable. By default the domain is `{0,1}`. The domain of variables (features) with individual cardinalities corresponds to the range of values of the individual cardinality. As a consequence, it does not consider the possibility to clone these features as determined by their individual cardinality. The relationships in the FM are represented as `ifThenElse` CSP constrains, plus a constraint to express the selection of the root feature (i.e., `root = 1`). The overall CSP that corresponds to an entire FM is the conjunction of all the constraints.

**(Validation)** Authors performed a comparative test between two off the shelf Java constraint solvers: JaCoP (Kuchcinski 2003) and Choco (Laburthe & Jussien 2005). The tests show that JaCoP is faster than Choco except in finding the number of solutions. The experiment was executed on five FMs with up to 52 features.

**(Results)** The time to get one solution seems to be linear and the time to g*et al*l solutions seemed to be exponential. These results show that the approach suffers extensibility and scalability issues. Besides, the problem with this approach is that the constraint representing an individual cardinality `(m,n)` between the father feature `A` and its child `B` (`ifThenElse(A=0;B=0;B in {n,m})`) does not consider the case when feature `A` has itself a cardinality. Therefore, the semantics of individual cardinalities is not well represented in the CSP.

**D.** Janota & Kiniry (2007) have formalized in higher-order logic (HOL, cf. Gordon & Melham 1993) a feature based meta-model that integrates properties of several feature modeling approaches such as attributes and cardinalities.

**(Verification criteria)** Once the model represented in HOL, expressions can be used to evaluate root selectivity, the existence of a path from the root to a given feature, and group cardinality satisfaction. Group cardinality satisfaction consists of verifying that the boundaries of the group cardinality are correct with reference to the number of features that that can be selected from the bundle of features grouped in the cardinality.

**(Implementation)** The approach has been implemented in the Mobius program verification environment (Barthe *et al.* 2007), an Eclipse-based platform for designing, testing, performing various kinds of static analyses, that was designed to automatically and interactively formally verify Java programs and bytecode.

**(Validation and results)** The paper does not provide evidence about the efficiency of the approach, its scalability, or its applicability to real life cases.

**E.** Trinidad *et al*. (2008) propose a CSP based approach to verify and diagnose FMs.

**(Verification criteria)** Trinidad *et al*'s approach handles three verification criteria: (i) "dead features"; (ii) "false optional features", i.e., features that in spite of being modeled as optional, are always chosen whenever their parents are chosen; and (iii) "void models", i.e., models from which no product can be configured. The goal of Trinidad *et al*. is not just to detect the above three errors but also to provide explanations for the cause of these errors.

**(Implementation)** In order to achieve the first goal, the approach transforms the FM into a CSP expression, then queries the Choco solver (by means of the FaMa tool) to find the errors.

**(Validation)** The approach has been evaluated on five FMs up to 86 features.

**(Results)** Unfortunately, no details about the scalability and the efficiency of the approach are provided.

**F.** Recently, Yan *et al*. (2009) proposed a method to **(Verification criteria)** find redundant constraints and features in FMs. A redundant constraint in a constraint that does not modify the semantics of the product line model and a redundant feature is a repeated feature. This approach is motivated by the fact that the problem size of feature model verification is exponential to the number of features and constraints in the model. Therefore, eliminating verification-irrelevant features and constraints from FMs should reduce the problem size of verification, and alleviates the state-space explosion problem.

**(Implementation)** The approach eliminates verification-irrelevant features and constraints from feature models. The authors use a BDD solver to execute the non-optimized and the optimized feature models in order to compute the difference of time executing both groups of models.

**(Validation)** The authors carried out an experiment in which they generated three groups of in-house random FMs. The first group had 9 FMs, all of them with 500 features and 50 explicit cross-tree constraints. In this first experiment, authors verified the consistency of FMs without eliminating redundant features and constraints in 62.7 sec. The same operation took 6.0 sec after 80% of the redundant features were eliminated. The second group contained 7 FMs with 100 to 700 features, and from 10 to 70 cross-tree constraints. In this second experiment, authors verified the consistency of the model with 600 features in 60.5 sec (without eliminating verification-irrelevant features and constraints), and in 43.9 sec (after eliminating verification-irrelevant features and constraints). The third group had 19 FMs with 100 to 1900 features and from 20 to 56 cross-tree constraints. The authors verified the consistency of a model with 1200 features and 42 cross-tree constraints without eliminating redundant features and constraints in 64 sec. The same operation took 3.0 sec when the redundant features had been eliminated. Once redundant features were eliminated the Yan *et al*'s approach allows the verification of consistency on models with 1900 features in 64 sec.

**(Results)** These experiments show that the approach proposed by Yan *et al*. improves the efficiency and the capability of the approach to FMs' consistency verification when the models contain a large number of redundancies. The problem with this approach is that it only considers as redundant constraints these that contain redundant features. Typical

redundancies such as domain overlapping, or cyclic relationships (Salinesi *et al*. 2010a, Mazo *et al.* 2011a) are therefore overlooked. Besides, the validation of the approach was done with in-house and random build feature models. There is no guarantee that it works with real world feature models. In particular, one can wonder how many redundancies a real model typically contains, and what their severity is. Last, no detail is provided about the formalisation and implementation of the approach.

## 2.2. Verification of Orthogonal Variability Models

In Orthogonal Variability Models (OVMs, cf. Pohl *et al.* 2005), a variation point describes what varies between the products of a software product line. For each variation point, a collection of variants is defined. Configuration consists of selecting among variants associated with each variation point. Pohl *et al.* (2005) propose three types of dependencies to specify configuration constraints:

(1) a mandatory variability dependency between a variation point and a variant indicates that this variant must always be selected when the variation point is considered for the product at hand. A mandatory variability dependency is drawn as a continuous line;

(2) an optional variability dependency between a variation point and a variant describes that this variant can be selected but it does not need to. An optional variability dependency is drawn as a dashed line;

(3) an alternative choice is a specialization of optional variability dependencies. An alternative choice group comprises at least two variants which are related to a variation point by optional variability dependencies. Min, max bounds define how many variants of the alternative choice group must be selected at least (min) and how many variants can be selected at most (max).

In addition to variability dependencies, the OVMs permit the definition of constraint dependencies to document additional dependencies between variation points and variants, e.g. to enforce that two variants of different variation points cannot be selected together.

Three approaches have been found in the literature to verify OVMs: Metzger *et al*. (2007), Roos-Frantz *et al.* (2008) and Lauenroth *et al.* (2010).

**A.** Metzger *et al*. (2007) introduce a formalization of OVMs and propose to use a SAT solver to automate verification of OVMs. In this approach, automated reasoning on OVMs is supported using the VFD (Varied Feature Diagram) semantics. VFD is based on FFD

(Free Feature Diagrams) which is a parametric construct designed to provide the syntax and semantics of FODA-like dialects in a generic way (Schobbens *et al.* 2006). Metzger *et al*. propose to reuse this formalization of feature diagrams, to introduce a formalization of OVMs. They introduce a formal version of OVM's abstract syntax and describe a translation from OVM to VFD, thereby they give OVM a formal semantics.

**(Verification criteria)** The approach deals with three verification criteria:

- Valid model: to check whether a VFD is consistent, i.e., whether it permits at least one configuration.
- Product checking: to verify that a given product is a valid configuration of the VFD.
- Dead variables: those that do not appear in any product.

**(Implementation)** The approach was not implemented; however, Metzger *et al*. propose to use SAT solvers to automate their verification approach.

**(Validation and results)** The approach was not validated. No details about its applicability or scalability are provided.

**B.** Roos-Frantz *et al.* (2008) propose a tool to verify OVMs; however the development of the tool is still future work.

**(Verification criteria)** The approach deals with:

- Valid product. Check whether a given product belongs to the set of products represented by the OVM or not.
- Void OVM. Check whether an OVM is void or not, i.e. if it represents at least one product.
- Dead nodes. To identify nodes that do not appear in any product. Dead nodes are caused by a wrong usage of constraint dependencies and are the responsible for void OVMs.

**(Implementation)** The approach proposes to transform OVMs into feature models, then to use the FaMa tool to verify the models. Roos-Frantz *et al.* also proposes other alternatives to verify OVMs: for instance, using a formal specification language like Z or B. However, no details are provided about the alternative selected by the authors in order to implement their approach. Besides, the approach that proposes the aforementioned verification criteria was not implemented.

**(Validation and results)** Unfortunately, no detail is provided about the implementation of these criteria, or about its validation, or about its scalability and performance.

**C.** Lauenroth *et al.* (2010) present a quality assurance approach that applies model checking (Clarke *et al.* 1999) at the level of the PLM itself, and not product by product (authors call this approach: comprehensive strategy) as presented by Metzger *et al.* (2007).

**(Verification criteria)** The approach considers the variability model to ensure that the state space of individual products is valid with respect to the variability model.

**(Implementation)** Lauenroth *et al.* (2010) focus on the next-time-operator (`EX f1`) (Clarke *et al.* 1999), which can be verified for single systems and can be adapted for the verification of PLMs. The next-time-operator over a variable `f1` (`EX f1`) evaluates to `true`, if there is one path starting at the initial state on which `f1` holds on the next state. The main idea of the Lauenroth *et al.*'s approach is to include the variability information specified in the variability model, as Boolean variables, in the model checking algorithms.

**(Validation and results)** No detail about the implementation or evaluation of the approach is presented. The study of the applicability of the approach is presented as future work.

## 2.3. Verification of Dopler Variability Models

In Decision-oriented (Dopler) variability models (Dhungana *et al.* 2010), the problem space is defined using *decision models* whereas the solution space is specified using *asset models*. A *decision model* consists of a set of decisions and dependencies between them. *Assets* provide an abstract view of the solution space to the degree of detail needed for subsequent product derivation. Decisions and assets are linked with *inclusion conditions* defining traceability from the solution space to the problem space.

Vierhauser *et al*. (2010) proposes a framework to incrementally detect inconsistencies in DOPLER models based in the approach presented by Egyed (2006) for UML models.

**(Verification criteria)** Finding inconsistencies like "assets on an asset model calling a decision that is not defined in the decision model" are the scope of this framework.

**(Implementation)** In this approach inconsistency criteria are specified with OCL. Each criterion is implemented by a rule that starts by identifying the model elements to analyze. Then, all the model elements for which an inconsistency is detected are inserted in a "rule scope" in order to keep track of them. The rule scope consists of a relation between an inconsistency detection rule and the collection of model elements that need to be re-analyzed after they have been corrected. Next time the rule is executed, the check is only made over the

elements in the "rule scope", and not over the complete model which avoids repeating the same verification over and over again.

**(Validation and results)** The approach reduces the execution time after the first checking. Egyed presents very efficient performance charts for his approach even in UML models with 10000 classes. Vierhauser *et al*. (2010) applied the approach over Dopler models with up-to 121 reusable elements. However, they also observe that this approach may not be efficient for all kinds of consistency rules due to the limitations of OCL constraints to compute certain verification functions (e.g., verification functions that need computation efforts or that involve multi-context data).

## 2.4. Verification of Latice Structure Models

In a product line model, the links between requirements are parent-child links so that requirements can be modelled hierarchically in a lattice (Mannion 2002). In this Latice, a requirement can have zero to many children and zero to many parents.

**(Verification criteria)** In this approach, PLMs are entirely represented as logical expressions that can be tested to verify the following aspects.

- Validity of the PLM: A valid PLM is one in which it is possible to select at least one set of requirements that satisfy the relationships between them in the model.
- A selected combination of requirements can also be tested using this expression in order to know if it forms a valid product.
- Richness of the PLM: this operation computes the number of valid products that can be built using a PLM. The result of this operation can be used to determine the flexibility level of the model. A small number may mean that there is insufficient resilience in the system for future markets. A large number may mean that there is unnecessary resilience and that the model should be further constrained.

**(Implementation)** Mannion (2002) and Mannion & Kaindl (2007) use first order logic to represent PLMs as logic expressions with the aim of verify them. In order to do that, Mannion & Kaindl consider each requirement of the PLM as a Boolean variable and each dependency between requirements as a logical expression. Indeed, `true` is assigned to those requirements that are selected, and `false` is assigned to those not selected. These selection values are substituted into the product line logical expression. A valid product is one for which the product line logical expression evaluates to `true`. To implement this operation it

is enough to find the first product that causes the PLM logical expression to evaluate to `true`.

Mannion & Kaindl (2007) represent the graph corresponding to the PLM as a Prolog programme, which is used to verify the validity and richness of the PLM and the validity of a configuration.

**(Validation and results)** No validation is provided to test the applicability, precision, scalability and usability of the approach, which makes it difficult to compare this approach with other ones according to these criteria. In addition, the proposal is not generic. Indeed, it was proposed for the lattice notation of PLMs and therefore it does not consider cross-tree constraints, or even more complex constraints such as for example constrains over attributes.

## 2.5. Formalism-independent Approaches to Verify Product Line Models

Three formalism-independent approaches have been found in the literature to verify product line models: Lauenroth & Pohl (2007), Kästner & Apel (2008) and Bruns *et al* (2011).

**A.** Lauenroth & Pohl (2007) propose a formal definition of the properties that a PLM must offer in order to support contradiction checks in domain engineering, a formal definition of contradiction, and an algorithm to detect possible contradictions on a PLM.

**(Verification criteria)** To know if the reusable components (`S`) of the PLM expressed in a language `L` contradict each other, the approach defines contradiction as a function $\texttt{contradiction}: S \rightarrow \wp(S)$ with the following properties:

- `contradiction(S) = C` if the set `S` contains contradicting requirements. The resulting set $C = \{C_1, \ldots, C_x\}$ contains subsets of requirements ($C_i \subseteq S$), where each subset $C_i$ contains a set of contradicting requirements.
- `contradiction(S) =` $\emptyset$ if the set `S` of requirements is free of contradictions.

**(Implementation)** Two assumptions are made in order to implement their approach: each requirement that is related to a variant is considered as a variable requirement and each requirement that is not related to a variant is considered as a common requirement. Next, the approach considers a PLM as a set of `n` variants $V = \{v_1, \ldots, v_n\}$, where each $v_i$ is represented by a Boolean variable. $v_i$ evaluates to

`true` if the variant $v_i$ is selected in a configuration and $v_i$ evaluates to `false` otherwise. The dependencies of the PLM are codified as a Boolean function over the variants `V`.

Lauenroth & Pohl (2007) use the law of contraposition ($A \Rightarrow B \Leftrightarrow \neg B \Rightarrow \neg A$) to find contradictions in PLMs. The central idea is that a contradiction in a PLM does not matter as long as it is not possible that the contradicting requirements become part of one single product. This requires a function that calculates whether a PLM satisfies—with the help of a SAT solver—a given set of `x` preselected variants.

**(Validation)** Lauenroth & Pohl (2007) hold that the performance of the algorithm to find contradictions among requirements in a PLM is superior to the brute force approach because the search for possible contradictions in the variable requirements is more efficient than the search for contradictions in all possible product models. The main drawback of this approach is that the set of potential contradictions must be known before the algorithm is executed. Thus, the algorithm cannot be used to systematically identify the contradictions in a PLM, but only to check if given contradictory requirements can be configurated in valid products of the PLM.

**(Results)** To the best of our knowledge, this approach was neither implemented, nor evaluated with real models. Even if Lauenroth & Pohl claim that complexity of their approach is NP-complete, they do not provide any detail about its scalability or usability in real cases, which makes it difficult to compare this approach with other ones according to non-subjective criteria.

**B.** Kästner & Apel (2008) extend the Featherweight Java (FJ, cf. Igarashi *et al.* 2001) calculus with annotations to prove that a software product line is well typed at the level of source code fragments. They have shown that this extension can be modeled on top of FJ, extending only the typing rules and auxiliary functions with implications on pairs of annotations.

**(Verification criteria)** The approach starts with an informal list of criteria specified as annotation rules, then modeled formally in the extended FJ. It provides typing rules such as: "a class *L* can only extend a class that is present" (Kästner & Apel 2008) or even rules that deal with the removal process of children from their parent element: "if a class is removed then also all methods therein must be removed, if a method is removed then also its parameters and term must be removed" (Kästner & Apel 2008).

**(Implementation)** The rules are automated by propagating colours from parent to child structures (e.g., if a class is 'green', then all its methods are automatically 'green' too). Some rules (like: "a method is only present when the enclosing class is present" and "a constructor parameter is only present when the enclosing class is present" (Kästner & Apel 2008)) are automated by means of a function that descends recursively through the product line and checks all code fragments that can be annotated. For those code fragments the annotations that evaluate to false should be removed. Thus, the remaining code fragments are stripped off their annotations. The approach is implemented in an extension of the CIDE tool, an annotation checker for Java.

**(Validation and results)** Unfortunately, no details about the evaluation or the implementation of the approach are provided.

**C.** Bruns *et al.* (2011) present delta-oriented slicing, an approach to reduce the deductive verification effort on product lines where individual products are Java programs and their relations are described by deltas.

**(Verification criteria)** The verification approach answers the question of which proofs are influenced by a delta module. Proofs considered in the work look like: (i) for each `adds(C; I)` prove that the invariant `I` is fulfilled by all relevant implementations; and (ii) for each `removes(C; I)` invalidate all pre-existing proofs that assume the invariant `I`.

**(Implementation)** In their approach, Bruns *et al.* (2011) analyze the product line model to determine which parts of the original product change in the new product and do not have to verify these parts again. When a new product is derived by delta application, the implementation and the specification of the product change. However, from the structural information available in the used delta modules, authors are able to infer which specifications of the new product remain valid (i.e., the proofs done for the old products are not affected by the change) and which parts have to be (re-)proven in order to establish the specified properties. Authors call the latter delta-oriented slice. The technologies used to implement the approach are the Java language for programming single products, the JML language (Leavens *et al.* 2006) for formal specifications and the KeY system (Beckert *et al.* 2007) for deductive verification.

**(Validation and results)** No details about the implementation of the verification algorithm or even about its evaluation are provided by Bruns *et al.* (2011).

## 2.6. Conclusions

This chapter reported the collection of a series of works related with the verification of PLMs. A first comment is that only some of these verification approaches (Van der Storm 2004, Zhang *et al.* 2004, Batory 2005, Sun *et al.* 2005, Benavides *et al.* 2005a, Benavides *et al.* 2005b, Benavides *et al.* 2006, Benavides *et al.* 2007, Egyed 2006, Metzger *et al.* 2007, Janota & Kiniry 2007, Van der Storm 2007, Lauenroth & Pohl 2007, Mannion & Kaindl 2007, Trinidad *et al.* 2008, Broek & Galvão 2009, Lauenroth *et al.* 2009, Lauenroth *et al.* 2010, Vierhauser *et al.* 2010, Kim *et al.* 2011, Liu *et al.* 2011) are formally described in the papers in which they were presented. Other approaches such in (Von der Maßen & Lichter 2004, Hemakumar 2008, Roos-Frantz *et al.* 2008), were not formally described. Formally described or not, each of these verification approaches proposes a collection of verification operations over a particular PLM formalism. The state of the art on PLM verification of PLMs is summarized in Table 2.1. The columns of the table correspond to the product lime modelling languages for which verification criteria have been found in literature. Each verification criterion is presented in a different row. Cells indentify which approach handles the corresponding criterion for the corresponding formalism. As a consequence, generic approaches are not fixed to a particular column and comprehensive approaches are presented over several rows. Empty cells in the table do not mean that there is no approach to handle the criterion for the corresponding formalism, but that was not found in our literature review. The question whether or not this is possible is of course an open issue.

**Table 2.1.** Literature review of product line models verification

| *Language* / Ver. criteria | *Independent of the language* | *Latice Structure Models* | *Feature Models* | *OVM* | *Dopler variability language* | *UML and other languages* |
|---|---|---|---|---|---|---|
| Absence of contradictions on products. | (Lauenroth & Pohl 2007) | | | | | |
| Validity, consistency or satisfiability of the product line model: the PLM allows generate at leas one product. | | (Mannion 2002) | (Zang *et al.* 2004;Benavides *et al.* 2005b; Benavides *et al.* 2006; Van der Storm 2007; Trinidad *et al.* 2008;Hemakumar 2008; Broek & Galvão 2009; Mendonca *et al.* 2009) | (Metzger *et al.* 2007; Roos-Frantz *et al.* 2008) | | |
| Consistent or satisfiable configuration: a | | (Mannion 2002) | (Van der Storm 2004; Batory 2005; Benavides | (Metzger *et al.* 2007; Roos-Frantz | | (Bruns *et al.* 2011) |

| | | | | | | |
|---|---|---|---|---|---|---|
| configuration forms a valid product or a collection of them. | | | *et al*. 2005b; Van der Storm 2007) | *et al*. 2008; Lauenroth *et al*. 2009; Lauenroth *et al*. 2010) | | |
| Richness or flexibility of the product line model. | | (Mannion 2002) | (Benavides *et al*. 2006; Broek & Galvão 2009; Mendonca *et al*. 2009) | | | |
| Dead artefacts. | | | (Von der Maßen & Lichter 2004), (Zang *et al*. 2004; Trinidad *et al*. 2008;Hemakumar 2008; Broek & Galvão 2009; Mendonca *et al*. 2009; Kim *et al*. 2011) | (Metzger *et al*. 2007; Roos- Frantz *et al*. 2008) | | |
| False optional artefacts. | | | (Von der Maßen & Lichter 2004), (Trinidad *et al*. 2008), | | | |
| Non-removable artefacts. | | | (Zang *et al*. 2004 | | | (Schaefer *et al*. 2010) |
| Redundant constraint and artefacts. | | | (Von der Maßen & Lichter 2004), (Yan *et al*. 2009) | | | |
| Consistency checking: absence of contradictions or non-existent elements in PLMs. | | | | | (Bruns *et al*. 2011) | (Egyed 2006; Kästner & Apel 2008, Vierhauser *et al*. 2010) |
| Root selectivity. | | | (Janota & Kiniry 2007, Salinesi *et al*. 2009b) | | | |
| Existence of a path from a selected artefact to the root. | | | (Janota & Kiniry 2007) | | | |
| Cardinality satisfaction | | | (Janota & Kiniry 2007) | | | |
| Circular dependencies (between the features of a given configuration). | | | (Liu *et al*. 2011) | | | |
| Conformance checking with the corresponding metamodel. | | | | | | |

The literature review carried out in this chapter permits answering the questions presented at the beginning of the chapter as follows:

- *Q1: What kind of product line modelling notations have been the subjects of verification?*

  As our literature review shows, the formalisms handled by verification approaches are: Feature-based models, OVMs, Latice Structure Models, Dopler variability models and UML-based models. However, most of the approaches existing in the literature focused in verification of feature models, as presented in Table 2.1. Of course, it is not impossible that other works have been presented on other formalisms. Another observation is that, in our literature review, there is no approach to verify multi-model product lines.

- *Q2: What verification criteria on product line models have been proposed?*

  *Answer:* (1) Consistency checking among the artefacts of the PLM. (2) Validity or satisfiability of the product line model. (3) Consistent or satisfiable configuration: A configuration forms a valid product or a collection of them. (4) Richness or flexibility of the product line model. (5) Identification of dead artefacts. (6) Identification of false optional artefacts. (7) Identification of non-removable artefacts (or core artefacts). (8) Identification of redundant constraint and artefacts. (11) Root selectivity. (12) Existence of a path from a selected artefact to the root. (13) Group cardinality satisfaction. (14) Find circular dependencies between the features of a given configuration.

  It is worth noting that most of these criteria are overlapped. Sometimes, different names are used to refer to the same criterion; for instance, the criteria 6 and 7 are complementary, the first four criteria refer to the same thing. Criterion 5 contains 11, and 8 contains 12. Besides, conformance checking (to verify if a PLM respects the language in which the model is specified) was never handled so far in the context of product lines, at least to the best of our knowledge.

  Another observation is that there is no comprehensive approach, i.e., an approach that handles all the criteria (or allows to do it) in a consistent way.

- *Q3: What kind of automated support has been proposed?*

  All the techniques we found in the literature represent PLMs in another formalism: sometimes a conjunctive normal form formula, at other times in an `if-then-else` structure (i.e., BDD), constraint satisfaction problem (CSP), OCL and in-house

representations. The goal of these representations is to automate verification using the supporting tools: Prolog solver (Mannion & Kaindl 2007), BDD solver (Van der Storm 2004, 2007; Trinidad *et al*. 2008) SAT solver (Batory 2005; Metzger *et al.* 2007, Trinidad *et al*. 2008), Miranda (Broek & Galvão 2009), model checkers such as SMV (Zhang *et al.* 2004), higher-order logic solvers (Janota & Kiniry 2007), CSP solvers (Benavides *et al.* 2005a, 2005b, 2006, 2007, Tinidad *et al*. 2008) and OCL interpreters (Egyed 2006, Vierhauser *et al*. 2010).

- *Q4: What kind of validation was made and what have been the results?*

  Most of the approaches we found in literature were not evaluated. Other approaches like (Kim *et al.* 2011, Benavides *et al.* 2005a, 2005b, 2006, 2007, Trinidad *et al.* 2008, Lauenroth *et al.* 2009, Sun *et al.* 2005) have been evaluated against few and small models with promising results. Authors like Yan *et al*. (2009) have evaluated the scalability of their approach, by applying them against a large number of large models generated at random. Egyed (2006) and (Vierhauser *et al*. 2010) validate the correctness, scalability and usability of their approaches on real, large models. Our literature review reveals that approaches' validation is commonly handled in three ways (i) by calculating theoretical complexity, (ii) by application to small life-like models, (iii) by application to large collection of randomles generated models of various sizes, and (iv) by application to large real life models.

- *Q5: What are the gaps and challenges to be faced in the future?*

  Our analysis is as follows:

  - Two interesting questions arise from the answer corresponding to Q1: can the verification approaches originally created for FMs also be used on other notations? And if it is possible, then, how to do that? To the best of our knowledge, there is not yet an answer to these questions.
  - This brings us to the question of generality: how to verify a PLM independently of the language in which the model at hand is represented and for any verification criteria? There is virtually generic approach to verify product lines models, and generality is not actually demonstrated in a systematic way.
  - A complementary question if that of comprehensiveness. Even if some approaches try to deal with this by implementing several verification criteria, there is not yet an approach covering all the verification criteria sensed from literature and from

industrial needs. This claim is supported by the last literature review on analysis and verification of feature models presented by Benavides *et al.* (2010).

- How to verify PLMs in a scalable way? The state of the art presented in this chapter shows that none of the techniques that we found in literature scales up to large models (e.g., 10000 artefacts in less than one second). This last point is important from an industrial point of view. A verification approach that does not allow the verification of large models in acceptable times is useless for industrial practitioners.

# Chapter 3

# Overview

This chapter provides the overview of the verification approach proposed in this thesis. The chapter does not focus on verification itself, but on the process that we propose to make product line models verifiable. Overall, the proposed approach is to transform PLMs, usually represented graphically, into executable code in order to verify them. In the case of multi-model product lines, the models in which the product line is represented should be integrated in a rich-enough formalism that allows representing the input models and the relationships among them in a homogeneous way. In addition, this pivot language should allow the automatic verification of the product line model(s).

The **transformation** stage is a preparation step that must be applied to the input PLM in order to make it verifiable by means of automatic tools. The automatic **verification of PLMs** entails finding several undesirable properties, such as redundant or contradictory information, or cases where the model does not respect the language in which it is specified. Automating PLM verification has been the subject of intensive research in recent years. Each verification approach usually focuses on one or two verification operations applied over a particular product line modelling language. The focus has mostly been on properties that strictly map to Boolean expressions. This thesis addresses these limitations by relying on **Constraint Logic Programming** (CLP) and **Constraint Programming** (CP) over finite domains as a pivot language to represent PLMs. Once PLMs are represented as constraint logic programs and constraint programs, they can be verified in a generic way against a collection of verification criteria. Generality is obtained by the transformation rules from different formalisms into CP over finite domains. Comprehensiveness is obtained by exploiting a **typology of verification criteria.** That is further explained and detailed in Chapter 4. This typology of verification criteria takes into account the fact that a PLM, independently of the language used to express it, must respect (i) certain properties associated with the domain of product lines; and (ii) certain properties are associated with the fact that each PLM respects the syntax rules of the language in which it is expressed. This typology has several advantages. First, from a pragmatic point of view, it can be used to select the criteria against which one wants to verify

a PLM, e.g., according to the impact that these criteria have or the expected level of quality of a particular PLM. Indeed not all the verification criteria have the same impact on the quality level of the PLM. For instance, having a product line model that does not allow to configure any product is much more critical than having a PLM with a redundant dependency. The aforementioned PLM transformation and verification approach is graphically represented in Figure 3.1 for the case of a stand-alone PLM like the feature models.

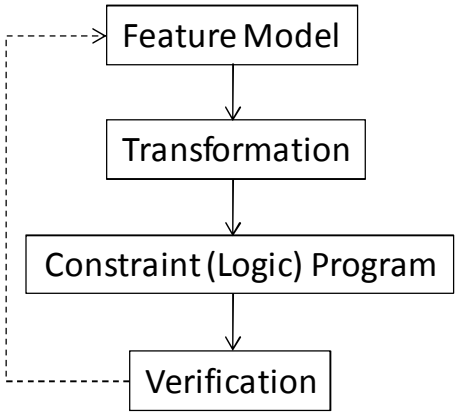



**Figure 3.1.** Overview of the verification process of product line models: the case of FMs.

However, the product line can be represented by means of several models. Thus is the case of Dopler models and product lines represented with multiple feature models. When the product line is represented by means of different models, their semantics must be transformed into CP as a previous stage to integrate and then verify them against the two categories of verification criteria presented in this thesis. This process is graphically presented in Figure 3.2 for the case of Dopler models.

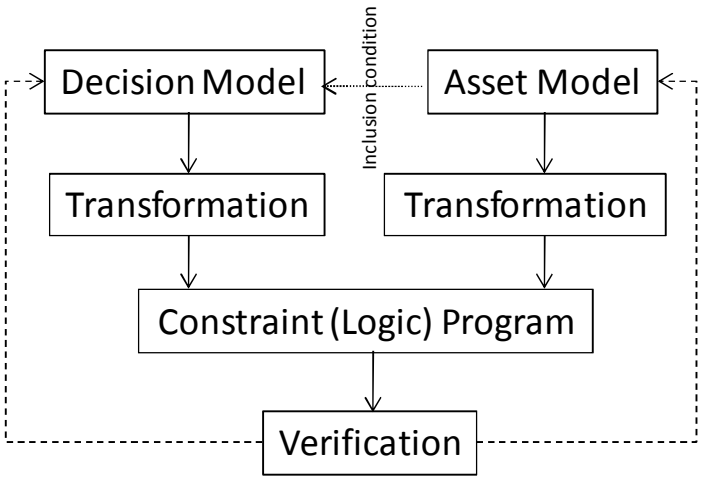



**Figure 3.2.** Verification scenario for multi-model product lines: the case of Dopler models.

It is worth noting that certain multi-model formalisms consider in the models themselves the integration mechanisms and therefore no new integration strategies are necessary. For instance, an integration step is not necessary for the case of Dopler models. That is due to the fact that the integration constraints are already considered in the Dopler's asset model. On the contrary, if the modelling formalisms do not consider any integration mechanism in the formalisms themselves, the **integration** step is necessary. That is the case of PLMs that are represented by means of several FMs, as in the running example presented in the next section, for instance. The verification process of multiple FMs is presented in Figure 3.3.

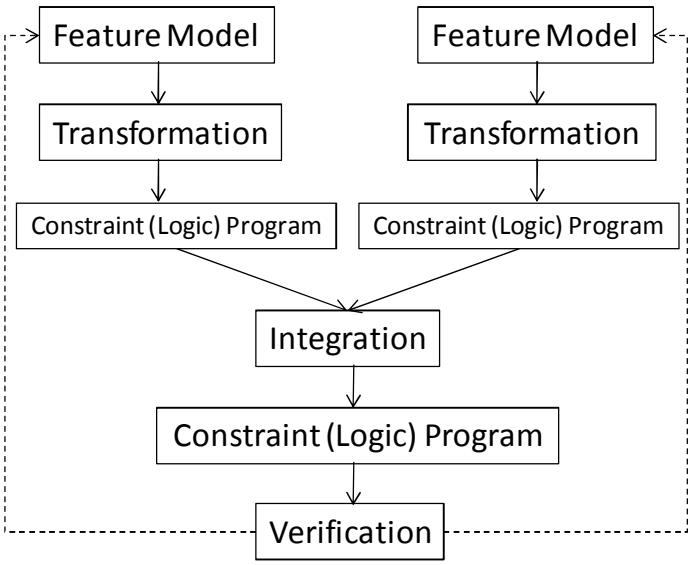



**Figure 3.3.** Verification scenario for multi-model product lines: the case of FMs.

This chapter is organized as follows. Section 3.1 presents the running example that will be used in the rest of the thesis to illustrate the proposed approach. The example refers to the UNIX product line presented in (Mazo *et al.* 2011c). This product line is specified with two FMs using the feature notation presented in Section 3.1.1, and a Dopler model (Dhungana *et al.* 2010) presented in Section 3.1.2, each one representing a particular view of the product line. Next, the chapter presents the background information necessary to read this thesis. This background includes: (i) the notions of FMs; (ii) the notion of Dopler models; (iii) the transformation process of FMs and Dopler models into constraint programs (cf. Section 3.2) and constraint logic programs (cf. Section 3.3); and (iv) the integration process of FMs and Dopler models (cf. Section 3.4). Section 3.5 presents a discussion about the issues of transformation and integration of product line models, which are fundamental aspects of our approach to handle generality, comprehensibility, ability to deal with multiple models, automation and scalability. Finally, Section 3.6 concludes the chapter.

## 3.1. Running Example

The example taken in this thesis is the one of the UNIX operating system. UNIX was first developed in the 1960s, and has been under constant development ever since. As other operating systems, UNIX is a suite of programs that makes computers work. In particular, UNIX is a stable, multi-user and multi-tasking system for many different types of computing devices such as servers, desktops, laptops, down to embedded calculators, routers, or even mobile phones. There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, Berkeley (BSD), GNU/Linux, and MacOS X.

The UNIX operating system is made up of three parts: the kernel, the shell and the programs; and two constituent elements: files and processes. These three parts consist of a collection of files and processes allowing interaction among the parts. The UNIX kernel is the hub of the operating system: it allocates time and memory to programs and handles the file-store and communications in response to system calls. The shell acts as an interface between the user and the kernel, interprets the commands (programs) typed in by users, and arranges for them to be carried out. As an illustration of the way the shell, the programs and the kernel work together, suppose a user types *rm myfile* (which has the effect of removing the file *myfile*). The shell searches the file-store for the file containing the program *rm*, and then requests the kernel, through system calls, to execute the program *rm* on *myfile*. The process *rm* removes *myfile* using a specific system-call. When the process *rm myfile* has finished running, the shell gives the user the possibility to execute further commands.

As for any product line, our example emphasizes the common and variable elements of the UNIX family and the constraints among these elements. This example is built from our experience with UNIX operating systems and it does not pretend to be exhaustive, neither on the constituent elements nor on the constraints among these elements (the purpose is to have a realistic, easy to understand example to illustrate our approach). The example is presented with two models. The first model deals with the technical aspects of UNIX; for instance, the technical specification of the screen resolution according to the available types of interface. The second view is the one of final users; for instance, it looks at which utility programs or what kinds of interfaces are available for a particular user.

We have chosen a series of 12 important characteristics of the UNIX product line and sorted them out in these two views.

**Technical view:**

Characteristic 1. UNIX has one KERNEL.

Characteristic 2. Some of the mandatory functions of the KERNEL are:

- ALLOCATING THE MACHINE'S MEMORY to each PROCESS
- SCHEDULING the PROCESSES
- ACCOMPLISHING THE TRANSFER OF DATA from one part of the machine to another

Characteristic 3. UNIX can have several PROCESSES (or none) for each user. The collection of PROCESSES varies even when the UNIX product is full-configured. For the sake of presentation, this thesis will consider only five processes.

Characteristic 4. UNIX offers a logical view of the FILE SYSTEM. A FILE SYSTEM is a logical method for organising and storing large amounts of information in a way that makes its management easy.

Characteristic 5. The KERNEL is composed of static or dynamic software modules. If the kernel was compiled for a specific hardware platform and cannot be changed, it is called a static Kernel. If the Kernel has the ability to dynamically load modules so that it can 'adapt' to a platform, it is called a dynamic Kernel. For instance, the modules SUPPORT_USB, CDROM_ATECH, and PCMCIA_SUPPORT cannot be charged, can be charged in a static way or can be charged in a dynamic way.

Characteristic 6. The SHELL is a command interpreter; it takes each command and passes it to the KERNEL to be acted upon.

Characteristic 7. The GRAPHICAL interface is characterized by a WIDTH RESOLUTION and a HEIGHT RESOLUTION that can have the following couples of values `[800,600]`, `[1024,768]` and `[1366,768]`.

**User view:**

Characteristic 8. UNIX can be installed or not and the installation can be from a CDROM, a USB device or from the NET.

Characteristic 9. UNIX provides several hundred UTILITY PROGRAMS for each user. The collection of UTILITY PROGRAMS varies even when the UNIX product is full-configured.

Characteristic 10. The SHELL is a kind of UTILITY PROGRAM. Different USERS may use different SHELLS. Initially, each USER has a default shell, which can be overridden or changed by users. Some common SHELLS are:

- Bourne shell (SH)
- TC Shell (TCSH)
- Bourne Again Shell (BASH)

For the sake of simplicity this thesis will consider only two users in this running example: ROOT_USER and GUEST_USER.

Characteristic 11. Some functions accomplished by the UTILITY PROGRAMS are:

- EDITING (mandatory and requires USER INTERFACE)

- FILE MAINTENANCE (mandatory and requires USER INTERFACE)
- PROGRAMMING SUPPORT (optional and requires USER INTERFACE)
- ONLINE INFO (optional and requires USER INTERFACE)

Characteristic 12. The USER INTERFACE can be GRAPHICAL and/or TEXTUAL.

### 3.1.1 Representation of the Running Example with a Feature Notation

A FM defines the valid combinations of features in a PL, and is depicted as a graph-like structure in which nodes represent features, and edges the relationships between them (Kang *et al*. 2002). We use extended feature models, i.e., feature models with individual cardinality (cf. `Process` in Figure 3.5), group cardinalities for bundles of features (cf. `Cdrom`, `Usb` and `Net` in Figure 3.6) and attributes (cf. `WithResolution` in Figure 3.5). We use the semantics of (Schobbens *et al.* 2007) combined with that of cardinality-based feature models as proposed by (Michel *et al.* 2011). The resulting metamodel used in this thesis is depicted in Figure 3.4 using the UML notation. According to this metamodel, a feature model is composed of at least two features, one of them must be the root feature, and one or more dependencies that relates two features in a given order (i.e., *from* to indicate the beginning of the dependency and *to* to indicate the end of the dependency).

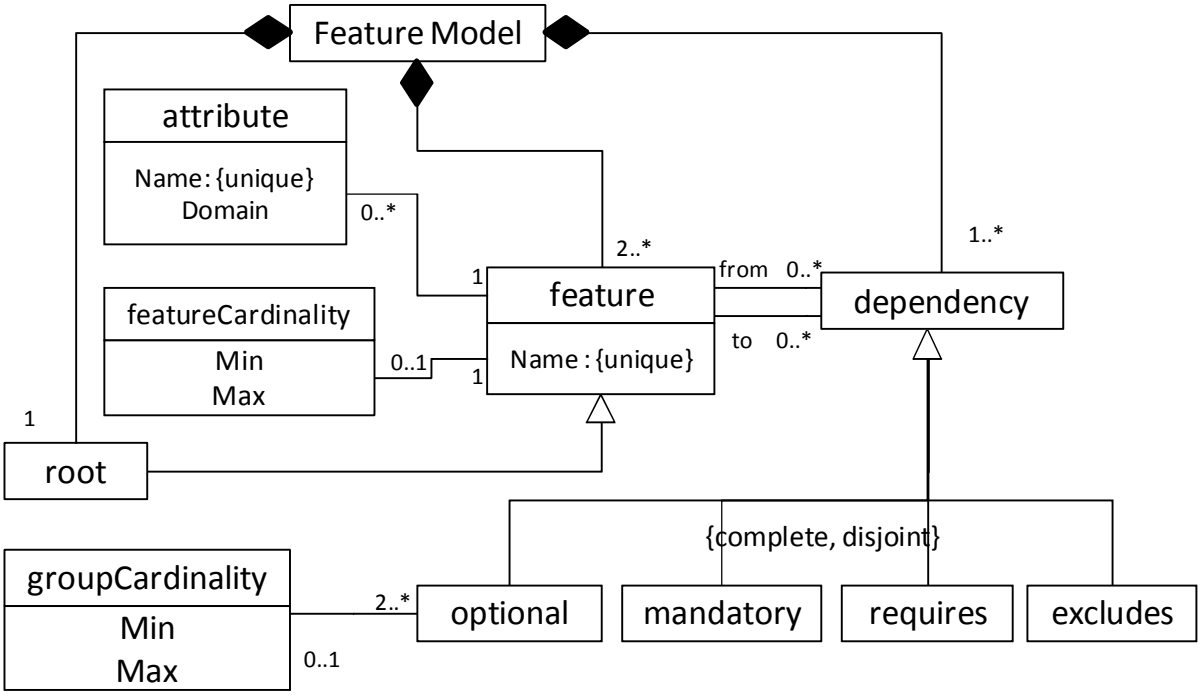



**Figure 3.4.** Cardinality and attribute-based feature model metamodel.

Two instances of this metamodel are presented in Figures 3.5 and 3.6, which correspond to the models of our UNIX running example presented above. The elements of the FM metamodel of Figure 3.4 are presented and exemplified by means of the following definitions:

***Definition 3.1: Feature***

A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system (Kang *et al.* 1990) and has a name (`Name`). For the sake of simplicity features are usually identified in FMs, e.g. their name of the feature; for instance `Kernel` in Figure 3.5. Every FM must have one root, which is called *root feature* and identifies the product line; for example `UNIX` in Figures 3.5 and 3.6. Feature names are unique in each model. However, two models can have two different features with the same name.

***Definition 3.2: Feature cardinality***

Usually, a feature cardinality is represented as an interval `[Min..Max]`, with `Min` as lower bound and `Max` as upper bound limiting the number of instances of a particular feature that can be part of a product. Each instance is called a *clone*. For instance in Figure 3.5, feature `Process` is constrained by a `[0..*]` cardinality where `*` is an undefined Integer number greater or equal than `Min`.

***Definition 3.3: Attribute in feature models***

Attributes in feature models are specific measurable characteristics of a feature. Although there is no consensus on a notation to define attributes, most proposals agree that an attribute is a variable with a name (`Name`), a domain (`Domain`), and a value (consistent with the domain) at a given configuration time. For instance in Figure 3.5, `WidthResolution` and `HeightResolution` are two attributes with a domain determined by the constraint at the bottom of the model.

***Definition 3.4: Mandatory dependency in feature models***

Given two features `F1` and `F2`, `F1` father of `F2`, a mandatory relationship from `F1` to `F2` means that if the `F1` is selected, then `F2` must be selected too, and vice versa. For instance in Figure 3.5, features `UNIX` and `Kernel` are related by a mandatory relationship.

***Definition 3.5: Optional dependency in feature models***

Given two features `F1` and `F2`, `F1` father of `F2`, an optional relationship from `F1` to `F2` means that if `F1` is selected, then `F2` can be selected or not. However, if `F2` is selected, then `F1` must also be selected. For instance in Figure 3.5, features `UNIX` and `UserInterface` are related by an optional relationship.

***Definition 3.6: Requires dependency in feature models***

Given two features F1 and F2, F1 requires F2 means that if F1 is selected in product, then F2 has to be selected too. Additionally, it means that F2 can be selected even when F1 is not. For instance, Shell requires ExecutingInstructions (cf. Figure 3.5) and Editing requires UserInterface (cf. Figure 3.6). The difference between a requires and an optional dependency is that in the requirement, if F2 is selected, F1 must be selected too, which is not the case in requirements.

***Definition 3.7: Exclusion dependency in feature models***

Given two features F1 and F2, F1 excludes F2 means that if F1 is selected then F2 cannot be selected in the same product. This relationship is bi-directional: if F2 is selected, then F1 cannot be selected in the same product.

***Definition 3.8: Group cardinality in feature models***

Usually, a group cardinality is an interval denoted <n..m>, with n as lower bound and m as upper bound and is associated with a collection of optional dependencies that originate from the same features. Group cardinalities help limiting the number of child features that can be part of a product when their common parent feature is selected. For instance in Figure 3.6, Cdrom, Usb and Net are related in a <1..1> group cardinality, which means that only one of these options can be selected in a UNIX configuration.

Figure 3.5 depicts the model of some technical aspects of a UNIX operating system family. This model represents a UNIX product line in which each derived operating system must have one kernel, one or several shell applications, one file system, a certain number of processes and, optionally, one graphical user interface with a width and height resolution respecting the constraint at the bottom of the model. The kernel must ensure certain operations related with the machine's processor scheduling, the interpretation and execution of instructions coming from the shell, accomplishing the transfer or data and allocating the machine's memory. The objective of this model is not to be exhaustive in the reusable elements of an UNIX system, but to provide a real and easy running example to develop the concepts of this thesis.

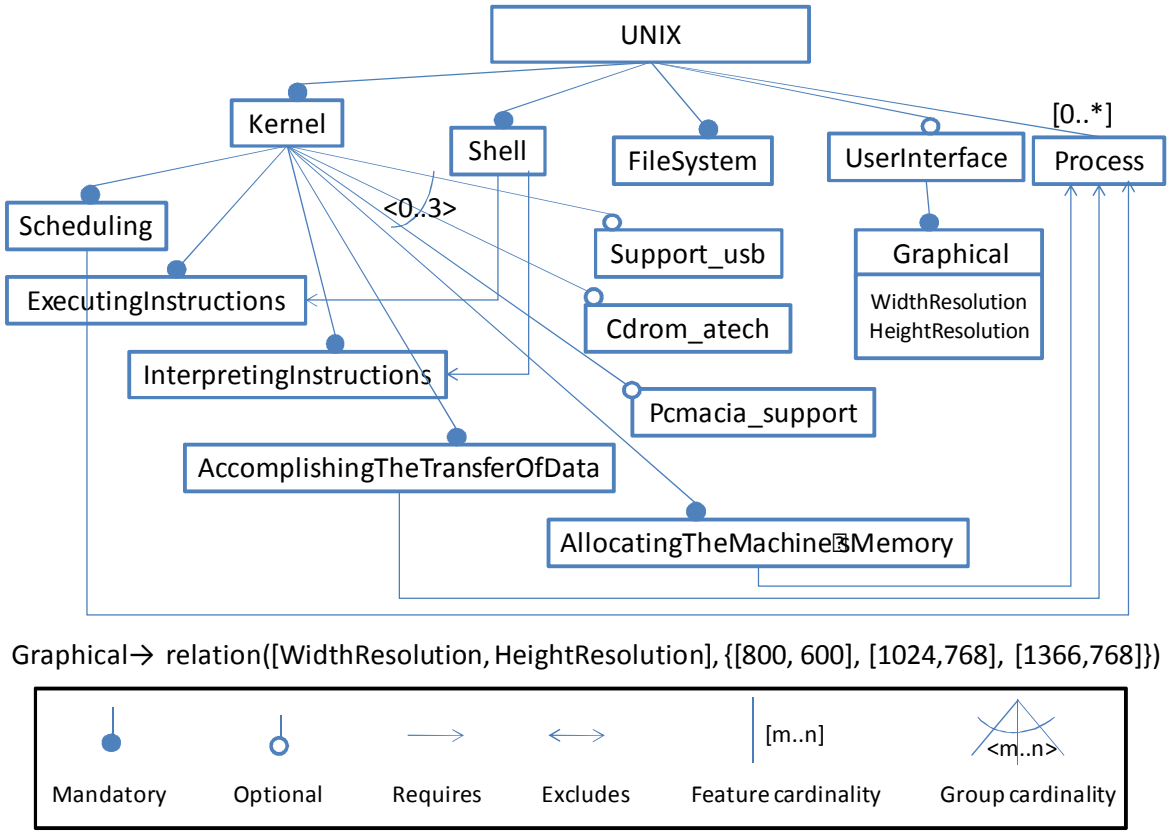



**Figure 3.5.** Technical model of the UNIX operating system family of our running example

Figure 3.6 provides the feature model that specifies the characteristics of our running example that are related with the final-user view of a UNIX product line.

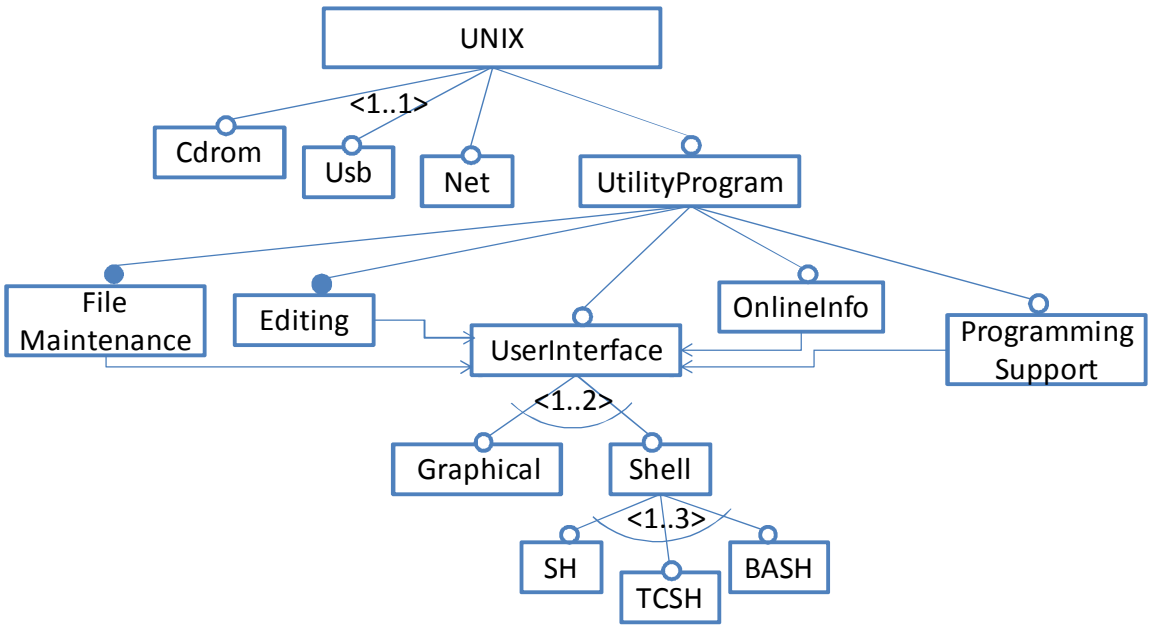



**Figure 3.6.** User model of the UNIX operating system family of our running example

As shown in Figure 3.6, a user has the possibility to install a UNIX system from one of the following supports: a CD ROM, a USB device or a network. In addition, users have the possibility to install or not utility programs for file maintenance, editing, online access, and user interface. The user interface may be graphical or command-line (`Shell`) based; there are three options for the command-line interface: `SH`, `TCSH` and `BASH`. The utility programs for the user interface, online information and programming support are specified with optional features.

### 3.1.2 Representation of the Running Example with the Dopler Language

The Decision-oriented (Dopler) variability modeling language focuses on product derivation and aims at supporting users configuring products. In Dopler variability models (Dhungana *et al.* 2010a; 2010b), the product line's problem space is defined using *decision models* whereas the solution space is specified using *asset models*. Decisions can be of four types: Boolean, Integer, String or Enumeration. Decisions (from the decision model) and assets (from the asset model) are related by means of inclusion conditions. Decisions are related by means of hierarchical and logical dependencies, and assets are related by means of functional and structural dependencies. The most important concepts in the Dopler modeling language are presented in (Dhungana *et al.* 2010b) and reproduced in Figure 3.7.

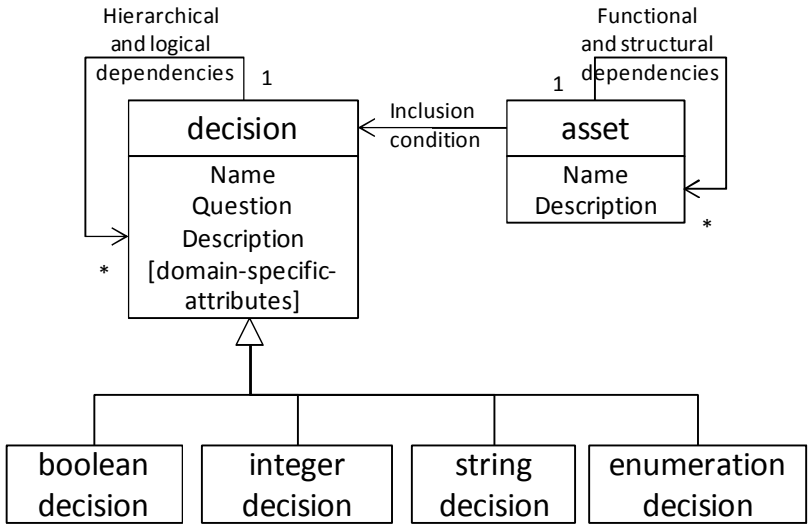



**Figure 3.7.** The core meta-model of Dopler modeling language, taken from (Dhungana *et al.* 2010b).

An example of Dopler model is presented in Figure 3.8. This figure depicts the installation of a UNIX operating system (decision model) and the associated packages (asset model) that can be selected if the UNIX system is installed with a graphical interface. The decision model

is composed of four decisions. The first decision proposes to choose the three ways to install a UNIX operating system (from a CD ROM, from a USB or from the Net). This decision impacts the second decision, in which the user must select the utility programs to be installed in the particular UNIX system. In that regard, five utility programs are proposed: editing tool, file maintenance tool, programming tool, online information tool and shell. If the choice contains the utility program for online information, the user must decide what kind of graphical resolution will be configured and several choices are proposed: `800x600`, `1024x768`, `1366x768`. The choice of width and height resolution has several decision effects. For instance, Figure 3.8 indicates that `if(GraphicalResolution==800x600) then Width=800`. To finish, the allocation of values for the width and height resolution must respect a certain number of conditions, such as: `Width` $\geq$ `800` and `Width` $\leq$ `1366`. The asset model is composed of seven graphical user interfaces and libraries that can be used in a UNIX graphical interface. The `Tab Window Manager` asset is available for all UNIX implementations with a graphical interface and requires the asset `Motif`; the other assets are optional. The `IRIS 4d window manager` is based on `Mwm` and `Motif` and therefore requires all of them in order to work in the same way as the `KDE` asset requires the `Qt widget toolkit` to work.

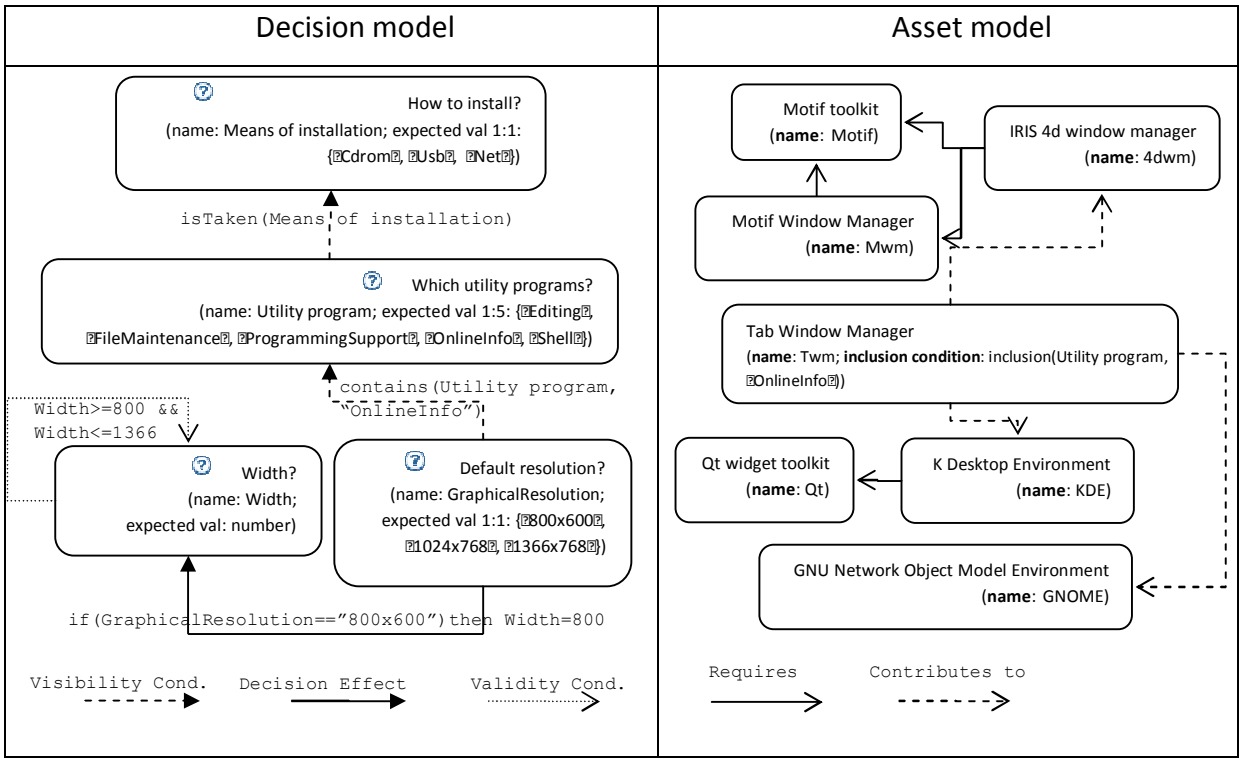



**Figure 3.8.** Example of Dopler Model: Installation of a UNIX System

In Dopler, a **decision model** consists of a set of decisions of utility programs (e.g., `Which utility programs?` with attributes, e.g. **name** and **expected values**) and dependencies among them (the **Visibility condition** `isTaken(Means of installation)` forces to make the decision `Utility program` if the decision `Means of installation` is taken). The **Assets** model permit the definition of an abstract view of the solution space, to the degree of details needed for subsequent product derivation. Decisions and assets are linked with **inclusion conditions** defining traceability from the solution space to the problem space (e.g., the asset `Tab Window Manager` must be included in the solution space if the option `OnlineInfo` of the decision `Utility program` is selected in a particular configuration). In our example, these inclusion conditions are specified as constraints that are added to the collection of constraints representing the decision and asset models. Adding these constraints integrates, both viewpoints of the PL, and the model is ready to be verified against the typology of verification criteria presented in this chapter.

## 3.2. Transforming the Semantics of PLMs into Constraint Programs

Constraint Programming (CP) emerged in the 1990's as a successful paradigm to tackle complex combinatorial problems in a declarative manner (Van Hentenryck 1989). It is at the crossroads of combinatorial optimization (operations research), constraint satisfaction problems (artificial intelligence), declarative programming language (logic and concurrent programming) and satisfiability (SAT) problems (Boolean constraint solvers). CP extends programming languages with the ability to deal with logical variables of different domains (e.g. Integers, Reals, Booleans, ...) and specific declarative relations between these variables called constraints (e.g. arithmetic constraints, symbolic constraints, …). A constraint is a logical relationship among several variables, each one taking a value in a given domain of possible values. A constraint thus restricts the possible values that variables can take. A Constraint Satisfaction Problem (CSP) is defined as a triple (`X, D, C`), where `X` is a set of variables, `D` is a set of domains, i.e. finite sets of possible values (one domain for each variable), and `C` a set of constraints restricting the values that the variables can simultaneously take. In modern CP languages (Diaz & Codognet 2001, Schulte & Stuckey 2008), many different types of constraints exist and are used to represent real-life problems: arithmetic constraints (e.g. `X + Y < Z`), symbolic constraints e.g.

- `atmost(N, [X1,X2,X3],V)`, meaning that at most `N` variables among `[X1, X2, X3]` can take the value `V`),
- global constraints (e.g. `all_different(X1, X2, ··· , Xn)`, meaning that all variables should have different values),
- reified constraints allowing the user to reason about the truth-value of a constraint.

Solving constraints consists of *reducing* the variable domains by propagation techniques (Bessiere 2006) and then *finding values* for each constrained variable in a *labelling* phase. This is achieved by iteratively grounding variables (fixing a value for a variable) and propagating its effect onto other variable domains (by applying again the same propagation-based techniques).

This thesis uses the notion of CP in order to represent the semantics of product line models with the purpose of verifying them by using an existing constraint solver or integrate different models into a single program. The transformation of feature models and Dopler models into constraint programs are presented in the following sections. Both cases are illustrated with the corresponding models of our running example.

### 3.2.1 Transforming Feature Models into Constraint Programs

It has been shown that Feature-Oriented Domain Analysis (FODA, cf. Kang *et al.* 1990) models can be represented as Boolean constraint programs through a series of Boolean variables, where each variable corresponds to a feature (Benavides *et al.* 2005), (White *et al.* 2008). Literature review shows that most existing approaches transform existing PLMs into Boolean constraint program. We believe that this approach hinders the full exploitation of the versatility of CP; for instance, the possibility to specify more complex requirements than select/de-select a feature, or to make more complex analyses and verification operations (Salinesi *et al.* 2010a, 2010b, 2011, Mazo *et al.* 2011a). We recall that this thesis uses the semantics of cardinality-based FMs proposed by Michel *et al*. (2011).

To fill that gap, our thesis is that the following rules can be used to transform FMs into constraint programs over finite domains:

- Each feature is represented as a Boolean (`0,1`) CP variable.
- Each attribute is represented as a CP variable, the domain of the attribute belongs to the domain of the CP variable.
- Each feature cardinality `[m..n]` determines (i) a collection of `n` variables associated to the feature of which this cardinality belongs; and (ii) a constraint restricting the

minimum (m) and the maximum (n) number of variables that can belong to a product in a certain moment.

- The domain of a variable is a finite collection of Integer values. When a variable takes the value of zero, it means that the variable is not selected. When a variable takes another value of its domain (different to zero), the variable is considered as selected.
- Every relationship is implemented as a constraint.

And the components of the FMs can be transformed by means of the following transformation rules:

- *Feature cardinality*: let P be a feature with a feature cardinality [m, n], then we create a CP variable P, a collection of n CP variables, one for each possible clone of P and an association between P and each of its clones. It is: $P \in \{0,1\} \wedge \forall i \in [1..n] \cdot P_i \in \{0,1\} \wedge (P_i \Rightarrow P)$. Which comes down to: $m * P \leq \sum_{i=1}^{n} P_i \leq n * P$
- *Mandatory:* let P and C be two features, where P is the father of C in a mandatory dependency. This constraint can be represented in a generic way (independently of the domain of P and C) as follows:

  P ⇔ C which, for P and C Boolean features, is equivalent to C = P
- *Optional:* let P and C be two features, where P is the father of C in an optional dependency. This constraint can be represented in a generic way (independently of the domain of P and C) as follows:

  C ⇒ P which, for P and C Boolean features, is equivalent to $C \leq P$
- *Requires:* let P and C be two features, where P requires C. If P has a feature cardinality [m..n] with $\forall_{i=1}^{n} P_i \in \{0,1\}$ clones of P, the constraint is: $\wedge_{i=1}^{n} P_i \Rightarrow C$. If P does not have feature cardinality, the equivalent constraint is: P ⇒ C, which means that if P is selected, C has to be selected as well, but not vice versa.
- *Exclusion:* let P and C be two features, where P excludes C. If P has a feature cardinality [m..n] with $\forall_{i=1}^{n} P_i \in \{0,1\}$ clones of P, the constraint is: $\wedge_{i=1}^{n} P_i * C = 0$. If P does not have feature cardinality, the equivalent constraint is: P * C = 0. If P and C are Boolean features, the equivalent constraint is: $\wedge_{i=1}^{n} P_i + C \leq 1$. This means that both P and C cannot be selected simultaneously.
- *Group cardinality:* let $C_1, C_2, \ldots, C_k$ be features with a non-negative integer domain, with the same parent P, and <m, n> the group cardinality boundaries. The equivalent constraint implies that each feature $C_1, C_2, \ldots, C_k$ set to one a Boolean

variable $BoolC_i \in \{0,1\}$ each time that the feature takes a value different to 0 (i.e. the feature is selected) : $\bigwedge_{i=1}^{k} C_i \Leftrightarrow BoolC_i \wedge m * P \leq \sum_{i=1}^{k} BoolC_i \leq n * P$.

When $P, C_1, C_2, \ldots, C_k$ are features with domain $\{0,1\}$, the CP representation of the group cardinality dependency can be optimized as follows: $m * P \leq \sum_{i=1}^{k} C_i \leq n * P$

It means that at least $m$ and at most $n$ children features must be selected. Note that the dependencies of $C_1, C_2, \ldots, C_k$ with their parent are constrained by means of the optional dependency, or by the feature cardinality constraint in cases where a child feature has an individual cardinality.

The following CP corresponds to the first FM of our running example (cf. Figure 3.5). This CP was obtained by means of the aforementioned transformation rules applied over the model of Figure 3.5. A complete transformation of our running example of extended feature models into CP is presented in Mazo *et al.* (2011e).

```
[UNIX, Kernel, Scheduling, ExecutingInstructions,
InterpretingInstructions, AccomplishingTheTransferOfData,
AllocatingTheMachinesMemory, Shell, FileSystem,
UserInterface, Graphical, Process] ∈ {0, 1} ∧
[WidthResolution] ∈ {0, 800, 1024, 1366} ∧
[HeightResolution] ∈ {0, 600, 768} ∧
[Support_usb, Cdrom_atech, Pcmacia_support] ∈ {0,1,2} ∧
[BoolSupport_usb, BoolCdrom_atech, BoolPcmacia_support] ∈
{0, 1} ∧

UNIX = 1 ∧
UNIX = Kernel ∧

Kernel = AllocatingTheMachinesMemory ∧
AllocatingTheMachinesMemory ⇒ Process ∧

Kernel = Scheduling ∧
Scheduling ⇒ Process ∧
Kernel = AccomplishingTheTransferOfData ∧
AccomplishingTheTransferOfData ⇒ Process ∧

Shell ⇒ InterpretingInstructions ∧
Kernel = InterpretingInstructions ∧
Shell ⇒ ExecutingInstructions ∧
Kernel = ExecutingInstructions ∧

Support_usb ⇔ BoolSupport_usb ∧
Cdrom_atech ⇔ BoolCdrom_atech ∧
Pcmacia_support ⇔ BoolPcmacia_support ∧
0 ≤ BoolSupport_usb + BoolCdrom_atech +
BoolPcmacia_support ≤ 3*Kernel ∧

UNIX = Shell ∧
UNIX = FileSystem ∧
UNIX ≥ UserInterface ∧
```

```
UserInterface = Graphical ∧
Graphical=1 ⇔ (WidthResolution=W1 ∧ HeightResolution=H1)∧
Graphical = 0 ⇔ (WidthResolution=0 ∧ HeightResolution=0)∧
fd_relation([[800,600], [1024,768], [1366,768]], [W1,H1])∧

UNIX ≥ Process ∧
R1 = Process1 + Process2 + Process3 + Process4 + Process5∧
Process ≤ R1 ≤ Process*5
```

The second FM of the running example, cf. Figure 3.6, was transformed into the following CP by means of the aforementioned transformation rules applied over the model of Figure 3.6.

```
[UNIX, Cdrom, Usb, Net, UtilityProgram, FileMaintenance,
Editing, OnlineInfo, ProgrammingSupport, UserInterface,
Shell, SH, TCSH, BASH] ∈ {0, 1} ∧

UNIX = 1 ∧
UNIX = Cdrom + Usb + Net ∧
UNIX ≥ UtilityProgram ∧
UtilityProgram = FileMaintenance ∧
FileMaintenance ⇒ UserInterface ∧
UtilityProgram = Editing ∧
Editing ⇒ UserInterface ∧
UtilityProgram ≥ UserInterface ∧
UtilityProgram ≥ OnlineInfo ∧
OnlineInfo ⇒ UserInterface ∧
UtilityProgram ≥ ProgrammingSupport ∧
ProgrammingSupport ⇒ UserInterface ∧
UserInterface ≤ Graphical + Shell ≤ UserInterface*2 ∧
R2 = SH + TCSH + BASH ∧
Shell ≤ R2 ≤ Shell*3
```

### 3.2.2 Transforming Dopler Models into Constraint Programs

To represent Dopler models as constraint programs, we first need to identify the Dopler model elements defining the variability of a product line. For instance, attributes (such as the description attribute of an asset or a decision) do not affect variability and can thus be ignored in the constraint representation. The representation of Dopler models as constraint programs hence has the following properties (Mazo *et al*. 2011a):

- Each decision is represented as a CP variable
- Each asset is represented as a CP variable.

The domain and semantics of variables that represent decisions is as follows:

- Let *D* be a decision with a visibility condition. If the visibility condition indicates that the decision is not visible, the corresponding variable is assigned with zero (0). If the visibility condition is a formula, the variable representing the decision is assigned with that particular formula. If the visibility condition indicates that the decision is always

visible, the variable representing the decision is affected with one (1). If the visibility condition of the decision *D* is not defined, its domain is {0,1}.

- For Number and String decisions the validity condition becomes the domain of variables representing these decisions. The domains of all variables are finite and must be composed of Integer values.
- The domain of Boolean and Enumeration decisions is mapped to a {0,1} domain. Zero indicates that nothing is selected and 1 indicates the selection of the associated variable.
- The domain of assets is mapped to a {0, 1} domain. If the variable representing an asset takes the value *0* in a configuration process it means that the asset is not included. If it takes the value *1*, the asset will be included in a derived product.
- Asset dependencies are described as constraints.
- Decisions, assets, and dependencies among them can be mapped into CPs by using the following rules.
    - **Decision type and validity condition**: Let *D* be a decision, *type* be its type and *valc* its validity condition. If *D.type = Boolean* or *Enumeration* then the equivalent constraint is $D \in \{0, 1\}$. If *D.type = Number* or *String* then the equivalent constraint is $D \in valc$. Note that the validity condition of String decisions must be previously represented as Integer values. For example, a String decision with validity condition *valc* = {Sunday, Monday, Tuesday} can be represented as *valc*={1, 2, 3}, where 1 means Sunday, etc. If *D.type = Enumeration*, let $<m, n>$ be its cardinality and $DOpt_1$, $DOpt_2$, ..., $DOpt_i$, a set of *i* decision options grouped in cardinality $<m, n>$. Then the corresponding constraint is: $DOpt_1 \in \{0, 1\} \bigwedge DOpt_2 \in \{0, 1\} \bigwedge$, ..., $DOpt_i \in \{0, 1\} \bigwedge D \Leftrightarrow m \leq DOpt_1 + DOpt_2 + ... + DOpt_i \leq n$. Which is equivalent to $m*D \leq \sum DOpt_i \leq n*D$
    - **Visibility condition**: Let *D* be a decision and *visc* its visibility condition. If *visc = false* then $D = 0$. If *visc = true* then $D = 1$. If *visc* is a different expression, then the corresponding constraint is: $D \Rightarrow visc$. Note that a visibility condition (i.e., *visc*) can be *true, false* or depending on one or more decisions and their values (e.g., *scope=="assemble yourself"* or *isTaken(scope)*).
    - **Decision Effects**: Let *D* be a decision and *df* its decision effect. The corresponding constraint is: $D \Rightarrow df$.
    - **Asset Inclusion Conditions**: Let *A* be an asset and *ic* its inclusion condition. The corresponding constraint is: $A \Rightarrow ic$.

- **Asset Dependencies**: Let *A* be an asset, *ad* its dependency and *type* its type. If *type* is "requires", the corresponding constraint is: $A \Rightarrow ad$. If *type* is "excludes", the corresponding constraint is: $A * ad = 0$. This means that if *A* is selected (equal to 1), *ad* must not be selected (must be equal to 0) and vice-versa. Currently, we do not take into account other types of asset dependencies (like parent or child).

The Dopler model of the running example, cf. Figure 3.8, is formed into the following CP by means of the aforementioned transformation rules.

**Decision Model:**

```
[MeansOfInstallation, Cdrom, Usb, Net, UtilityProgram,
FileMaintenance, Editing, OnlineInfo, ProgrammingSupport,
Shell, GraphicalResolution] ∈{0, 1} ∧
[Width] ∈ {0, 800, 1024, 1366} ∧
[Height] ∈ {0, 600, 768} ∧

MeansOfInstallation = Cdrom + Usb + Net ∧
MeansOfInstallation ⇒ UtilityProgram ∧

R1 = Editing + FileMaintenance + ProgrammingSupport +
OnlineInfo + Shell ∧
UtilityProgram ≤ R1 ≤ UtilityProgram*5 ∧

OnlineInfo ⇒ GraphicalResolution ∧
GraphicalResolution = 1 ⇔ (Width=W1 ∧ Height=H1) ∧
GraphicalResolution = 0 ⇔ (Width=0 ∧ Height=0) ∧
fd_relation([[800,600], [1024,768], [1366,768]], [W1,H1])∧
```

**Asset Model:**

```
[OnlineInfo, ATwm, A4dwm, AMwm, AMotif, AKDE, AQt, AGNOME] ∈
{0, 1} ∧

OnlineInfo ⇒ Twm ∧
A4dwm ⇒ ATwm ∧
A4dwm ⇒ AMotif ∧
A4dwm ⇒ AMwm ∧
AMwm ⇒ AMotif ∧
AKDE ⇒ ATwm ∧
AKDE ⇒ AQt ∧
AGNOME ⇒ ATwm
```

## 3.3. Implementing the Structure of PLMs into Constraint Logic Programs by Transformation

Constraint Logic Programming (CLP, cf. Apt & Wallace 2006) represents a successful attempt to merge the best features of *logic programming* and *constraint programming*. On the one hand, *logic programming* is based on the idea that (a subset of) first order logic can be used

for computing. Logic programming is concerned with the correct statement of a problem, by this reason the program should state what is true about the problem, not how to solve it procedurally. On the other hand, constraint programming is a programming paradigm wherein relations between variables are stated in the form of constraints. Constraint programming differs from the imperative programming languages in that it does not specify a step or sequence of steps to execute, but rather the properties of a solution to be found.

These two concepts allow the development of the CLP paradigm, where constraints are embedded in the logic programming paradigm. The main goal is to maintain a declarative programming paradigm while increasing expressivity and efficiency via the use of specific constraint sorts and algorithms. In other words, a CLP clause is just like a logic programming clause, except that its body may also contain constraints of the considered sort. For example, if one can use linear inequations over Reals, a CLP clause could be:

```
p(X,Y) :-
        X < Y+1,
        q(X),
        r(X,Y,Z).
```

Logically speaking, this clause states that `p(X,Y)` is true if `q(X)` and `r(X,Y,Z)` are true, and if the value of `X` is smaller than that of `Y + 1`.

Several notations exist to represent CLPs (Colmerauer 1982, Jaffar & Lassez 1987). In this thesis, we use the CLP language GNU Prolog (Diaz & Codognet 2001) and its associated solver, in order to represent and solve the constraint logic programs of this thesis, and then reason over it.

Now, let us return to the subject of this thesis: verification or product line models, and specifically to the transformation of PLMs into CLP, which is the subject of this section. It is well known that a metamodel defines the abstract syntax of a language, i.e. concepts and the nature of their relationships (constraints on the structure of its instances), therefore the structure of a PLM is represented in its metamodel. Thus, from a syntax point of view, a PLM is a set of variables and dependencies among them, which are each one instance of an element defined in the corresponding metamodel (OMG 2003).

The particular aim of the abstract syntax transformation consists in verifying the conformance of the transformed model with its corresponding metamodel. This process is presented in the next section for the case of feature models.

### 3.3.1 Transforming Feature Models into Constraint Logic Programs

The FM metamodel that we use in this thesis is presented in Figure 3.4 as a UML class diagram. Adopting this metamodel rather than other ones allowed us to validate our approach against the FMs from the SPLOT benchmark (Mendonça *et al.* 2009a), which use the concepts of cardinalities (Czarnecki *et al.* 2005) and attributes (Benavides *et al*. 2005b). The former adaptation was also necessary to deal with FMs developed from our experience with industrial partners. Due to the fact that a metamodel represents the abstract syntax of a language; the instantiation process of models from the metamodel can be used to keep the syntax of the language in each model. To automate this process we propose to represent the components of the FM metamodel (cf Figure 3.4) as meta-facts as follows:

```
(1) feature(IdFeature, Name, IdAttributes).
(2) root(IdFeature).
(3) attribute( IdAttribute, Name, Domain).
(4) dependency(IdDependency, IdFeature1, IdFeature2).
(5) optional (IdDependency).
(6) mandatory (IdDependency).
(7) requires(IdDependency).
(8) excludes(IdDependency).
(9) groupCardinality(IdDependencies, Min, Max).
(10) featureCardinality(IdFeature, Min, Max).
```

In the metamodel depicted in Figure 3.4 FM's elements are modeled by meta-classes, and relationships between these elements are modelled by meta-associations. In CLP, FM's elements and its links are called meta-facts and are implemented as CLP facts. In other words, a meta-fact is the CLP structure that represents a fact. In order to define a meta-fact; it is necessary to define its name, its parameters and its arity (in case of equal names, the number of parameters make two meta-facts different). For instance, the metafact of the line 1 has the name: `feature`, and three parameters: `IdFeature, Name, IdAttributes`. The mapping between the FM metamodel and the aforepresented meta-facts (lines 1 to 10) are explained in the rest of this section. It is worth noting that each meta-fact has an identifier, even if they make not part of the metamodel. This decision is purely technical; the aim is to improve the implementation efficiency due to the fact that identifiers are numbers, which allows more performant implementations (e.g., in Prolog, sorting a list of numbers is less time consuming than sorting a list of words).

Each meta-fact has a parameter that uniquely identifies each instance of the meta-fact. Identifiers are represented as strings (Prolog's atoms) and the references to other FM's entities are represented as lists of identifiers; in both cases, the name of the corresponding variable is preceded by the label `Id`.

**Meta-fact 1:** `feature(IdFeature, Name, IdAttributes).`

Name is a string representing the feature's name and `IdAttributes` is a list of attribute identifiers `[IdAtt1,...,IdAttN]`, where `[]` represents an empty list.

**Meta-fact 2:** `root(IdFeature).`

The root feature (i.e., UNIX in Figures 3.5 and 3.6) identifies the product line. In this meta-fact the attribute `IdFeature` references to the root feature.

**Meta-fact 3:** `attribute(IdAttribute, Name, Domain).`

An attribute has an identifier, a name and a domain. Name is a string representing the name of the attribute instantiated with this meta-fact. Domain is a collection of values that the attribute can take. For example `[read]` means that the value of the corresponding attribute can only be `read`; `[1..5]` means that the value of the corresponding attribute can be an Integer between `1` and `5`; `[integer]` means that the value of the corresponding attribute must be an `integer`.

**Meta-fact 4:** `dependency(IdDependency,IdFeature1,IdFeature2).`
**Meta-fact 5:** `optional(IdDependency).`
**Meta-fact 6:** `mandatory(IdDependency).`
**Meta-fact 7:** `requires(IdDependency).`
**Meta-fact 8:** `excludes(IdDependency).`

Dependencies between two features are represented by the meta-fact 4. In this meta-fact, `IdFeature1` and `IdFeature2` respectively represent the identifiers of the initial and target features involved in the dependency. Dependencies can be of four types: `mandatory`, `optional`, `requires`, or `excludes`, respectively represented by meta-facts 5, 6, 7 and 8. Each meta-fact from 5 to 8 references the corresponding dependency. For example, an optional dependency references the corresponding dependency having the identifiers of the parent and child features (`IdFeature1` and `IdFeature2` respectively) intervening in the optional dependency. In *requires* dependencies `IdFeature1` is the requiring feature and `IdFeature2` represents the required feature.

**Meta-fact 9:** `groupCardinality(IdDependencies, Min, Max).`

Group cardinality is a relationship between several features constrained by a `Min` and a `Max` value. Group cardinalities can be represented by instantiation of meta-fact 9, where `IdDependencies` is a list of dependency's identifiers related in the group cardinality.

**Meta-fact 10:** `featureCardinality(IdFeature, Min, Max).`

Feature cardinality is represented as a sequence of intervals `[Min..Max]` determining the lower (`Min`) and upper (`Max`) number of instances of a particular feature that can be part of a

product. In meta-fact 10, `IdFeature` is the identifier of the feature to which the individual cardinality belongs.

The relationship between the meta-fact and the derived facts respects the basic principle of meta-modeling. In our case, the instantiation of a meta-fact consists of giving constant values to the parameters of this meta-fact. We show this instantiation with the user model of the UNIX operating system family (cf. Figure 3.6). Note that in the following representation of the UNIX systems as CLP facts, each feature, attribute and dependency, is identified by a sequential number preceded by the prefix `fea`, `att` and `dep`, respectively.

```
(1) root(fea1).
(2) feature(fea1, 'UNIX',[]).
(3) feature(fea2, 'Cdrom', []).
(4) feature(fea3, 'Usb', []).
(5) feature(fea4, 'Net', []).
(6) feature(fea5, 'UtilityProgram', []).
(7) feature(fea6, 'FileMaintenance', []).
(8) feature(fea7, 'Editing', []).
(9) feature(fea8,  'UserInterface', []).
(10) feature(fea9, 'Graphical', []).
(11) feature(fea10, 'Shell', []).
(12) feature(fea11, 'SH', []).
(13) feature(fea12, 'TCSH', []).
(14) feature(fea13, 'BASH', []).
(15) feature(fea14, 'OnlineInfo', []).
(16) feature(fea15, 'ProgrammingSupport', []).
(17) dependency(dep1, fea1, fea2).
(18) dependency(dep2, fea1, fea3).
(19) dependency(dep3, fea1, fea4).
(20) dependency(dep4, fea1, fea5).
(21) dependency(dep5, fea5, fea6).
(22) dependency(dep6, fea5, fea7).
(23) dependency(dep7, fea5, fea8).
(24) dependency(dep8, fea8, fea9).
(25) dependency(dep9, fea8, fea10).
(26) dependency(dep10, fea10, fea11).
(27) dependency(dep11, fea10, fea12).
(28) dependency(dep12, fea10, fea13).
(29) dependency(dep13, fea5, fea14).
(30) dependency(dep14, fea5, fea15).
(31) dependency(dep15, fea8, fea16).
(32) dependency(dep16, fea8, fea17).
(33) dependency(dep17, fea8, fea18).
(34) dependency(dep18, fea8, fea8).
(35) optional(dep1).
(36) optional(dep2).
(37) optional(dep3).
(38) mandatory(dep4).
(39) mandatory(dep5).
(40) mandatory(dep6).
(41) optional(dep7).
(42) optional(dep8).
(43) optional(dep9).
(44) optional(dep10).
(45) optional(dep11).
(46) optional(dep12).
(47) optional(dep13).
```

```
(48) optional(dep14).
(49) requires(dep15)
(50) requires(dep16).
(51) requires(dep17).
(52) requires(dep18).
(53) groupCardinality([dep1,dep2, dep3], 1, 1).
(54) groupCardinality([dep8,dep9], 1, 2).
(55) groupCardinality([dep10,dep11, dep12], 1, 3).
```

Lines 1 and 2 define the root feature `UNIX` with no attributes. Lines 3 to 16 define the rest of the features with no attributes. Lines 17 to 34 define the collection of dependencies on the model. The nature of these dependencies is declared from lines 35 to 52. Line 53 defines the group cardinality `<1..1>` for dependencies among the root feature `UNIX` and features `Cdrom`, `Usb` and `Net`. Line 54 defines the group cardinality between the relationships identified by the atoms `dep8` and `dep9`, corresponding to the father-child pairs `UserInterface–Gaphical` and `UserInterface–Shell`. Finally, line 55 corresponds to the group cardinality `<1..3>` defining the number of shells that can have a system configured from the model of Figure 3.6.

## 3.4. Multi-model Verification

An important challenge in PL domain engineering and application engineering is that product lines are, in practice, often specified using several models at the same time (Dhungana *et al*. 2006, Djebbi *et al*. 2007, Segura *et al*. 2008, Rosenmüller *et al*. 2011). This permits dealing with various facets of the PL and products, and representing the viewpoints of various stakeholders such as executives, developers, distributors, marketing, architects, testers, etc. (Nuseibeh *et al*. 1994). For example, analysts may deliver a requirements model that specifies user-oriented system functionality, while architects may deliver a feature-based model focusing on the system structure from a more technical design-oriented point of view. In the absence of a global model, and given the number of models in which the PL can be specified, requirements can get missed or misunderstood (Finkelstein *et al*. 1992) both during domain and application engineering activities, e.g., when the selection of an artefact in one model of the product line depends of the selection of another artefact in another model of the product line (Zhao *et al*. 2008, Hubaux *et al*. 2009, 2010).

Integration of PLMs is not new. Authors like Alves *et al.* (2006), Schobbens *et al.* (2006), Liu *et al.* (2006), Dhungana *et al.* (2006), Fleurey *et al.* (2007b), Apel *et al.* (2007), Jayaraman *et al.* (2007), Segura *et al.* (2008), Acher *et al.* (2010) and Rosenmüller *et al.* (2011) propose different approaches to integrate PLMs. It is worth noting that our thesis to verify multi-model product lines is to use an integration approach, in contrast to the priorisation of models

proposed by Zhao *et al*. (2008), or the constraint-based approach to define the order in which models are configured (Hubaux *et al*. 2009, 2010).

CP can be exploited in the context of multi-model PL engineering to capture in a unified way the various models, and to arrange them into a unique specification. As a result, domain and application engineering activities such as PL analysis or product configuration are facilitated. Indeed, the unique representation facilitates the propagation of constraints between variables that belong to the different models. When configuration entails a variable in a model, it entails the variable in all the other models to which the variable belongs. Another considerable advantage is that having all the models of the PL integrated in a single CP permits the specification of constraints between different variables that belong to different models. Our literature survey did not reveal any interoperability meta-model that would have permitted to define relationships among several PL models specified in different languages.

The constraint-based integration approach proposed in this thesis will be developed in two cases: the Dopler models and the FMs. As explained in the next sections, the application of the constraint-based integration approach is much simpler in Dopler models than in FMs due to the fact that Dopler formalism is itself conceived as a multi-view product line formalist and therefore the integration strategy is already defined in the formalism itself. On the contrary, FMs are intended to be developed as standalone models, and to the best of our knowledge, there are no formal integration mechanisms in this notation itself. Therefore, to integrate FMs, this thesis proposes five different ways or *integration strategies* to do that.

### 3.4.1 Integration: the Case of Dopler Models

Decisions and assets are linked with *inclusion conditions* defining traceability from the solution space to the problem space (e.g., the asset `Tab Window Manager` must be included in the solution space if the option `OnlineInfo` of the decision `Utility program` is selected in a particular configuration). In our integration approach, these inclusion conditions are constraints that will be added to the collection of constraints representing the decision and asset model. Once these constraints are added, both models of the PL are integrated in a global program. This program is presented as follows:

```
[MeansOfInstallation, Cdrom, Usb, Net, UtilityProgram, FileMaintenance,
Editing, OnlineInfo, ProgrammingSupport, Shell, GraphicalResolution,
ATwm, A4dwm, AMwm, AMotif, AKDE, AQt, AGNOME]∈{0, 1} ∧
fd_domain([Width]∈{0, 800, 1024, 1366} ∧
fd_domain([Height]∈{0, 600, 768} ∧

MeansOfInstallation = Cdrom + Usb + Net ∧
```

```
MeansOfInstallation ⇒ UtilityProgram ∧

R1 = Editing + FileMaintenance + ProgrammingSupport + OnlineInfo +
Shell ∧
UtilityProgram ≤ R1 ≤ UtilityProgram * 5 ∧

OnlineInfo ⇒ GraphicalResolution ∧
GraphicalResolution = 1 ⇔ (Width=W1 ∧ Height=H1) ∧
GraphicalResolution = 0 ⇔ (Width=0 ∧ Height=0) ∧
fd_relation([[800,600], [1024,768], [1366,768]], [W1,H1])∧

OnlineInfo ⇒ Twm ∧

A4dwm ⇒ ATwm ∧
A4dwm ⇒ AMotif ∧
A4dwm ⇒ AMwm ∧
AMwm ⇒ AMotif ∧
AKDE ⇒ ATwm ∧
AKDE ⇒ AQt ∧
AGNOME ⇒ ATwm
```

### 3.4.2 Integration: the Case of Feature Models

In our process, integrating two PLMs consists of (i) integrating the variables that correspond to reusable elements; (ii) integrating attributes and their domains; and (iii) integrating the relationships among reusable elements. Integrating two models can be done in two steps: matching and merging (Finkelstein *et al*. 1992, Fleurey *et al*. 2007b). The matching step specifies which elements can match and how they can match. The merge step defines how two model elements that match are merged, as well as a mechanism to handle the non-matching elements of the input models. For example, if two feature models (Kang *et al*. 1990) that specify a single PL own the same feature `A`, which is required by another feature in the first model, and which is excluded by another feature in the second model, then the situation matches because of the feature `A`. However, the decision to include or not feature `A` in the resulting model depends on the integration strategy. In particular, one has to reason on the dependencies between feature `A` and the other features in both models.

Integration strategies are about the ways in which models are merged. Indeed, there are different ways to merge models, depending on how the models match and depending on the expected outcome. Take the following example scenario: a company decides to lengthen the production spectrum of the PL. Therefore, it integrates the FMs of two headquarters, and keeps in the resulting FM the reusable elements and the production capacity of both headquarters. This thesis exploits five different strategies that may be used to integrate FMs: two restrictive strategies, two conservative strategies, and one disjunctive strategy.

We believe that handling multi-model verification calls for a rich integration approach, as offered by the following five strategies:

Strategy N° 1 is restrictive in the sense that it allows representing in the resulting FM the common products represented in both input models that can be configured with the common features and attributes.

Strategy N° 2 is also restrictive but unlike the first strategy, the products can be configured with all features and attributes available on both input models (Acher *et al.* 2010).

Strategy N° 3 is conservative in the sense that it is possible to configure from the resulting FM the products represented in both input models by using only the common features and attributes.

Strategy N° 4 is also conservative, but this time it is possible to configure products with all features and attributes available in both input models (Segura *et al.* 2008, Acher *et al.* 2010).

Strategy N° 5 is disjunctive in the sense that the resulting model allows configuring the products presented in one of the input models by using the features and attributes of this particular model without considering the features and attributes of the other one.

Each one of the aforementioned integration strategies are further explained and exemplified in Chapter 7.

## 3.5. Discussion

The main working hypothesis in this thesis consists of choosing a constraint language that can be handled by a solver in order to execute PLMs and supports the verification approach proposed in this thesis. The rest of the section presents some related works about the transformation of PLM into other languages and some discussions and challenges in this topic.

Van Deursen & Klint (2002) proposes to reason about FODA models by translating them into a logic program using predicates such as `all`, `one-of`, or `more-of`, that respectively specify mandatory, mutually exclusive, and alternative features. For instance constraints: `F1 = all(F2, F3, F4), F4 = one-of(F5, F6)` specify that if `F1` is included in a configuration, then `F2`, `F3`, and `F4`, and therefore either `F5` or `F6` should be included too. The use of CP to reason about feature model was extended by Batory (2005), who proposes an approach to transform a feature model into propositional formula using the $\wedge$, $\vee$, $\neg$, $\Rightarrow$ and $\Leftrightarrow$ operations of propositional logic. The advantage of using CP is that it enables, for example, constraints of the form $F \Rightarrow A \vee B \vee (C \wedge D)$, meaning that feature `F` needs features `A`, or `B`, or `C` and `D`, which cannot be specified with FODA without creating an extra feature to

represent C ⋀ D. As in (Van Deursen & Klint 2002). In these constraints, features are Boolean variables (either they are included or not in a configuration). The transformation approach presented in this thesis goes a step further as (a) it does not consider Boolean models only, but also models with arithmetic constraints, symbolic constraints and reified constraints over finite-domain variables and (b) it supports the specification of constraints directly in CP, and not just as the result of a transformation from another model.

In (Benavides *et al*. 2005a, Benavides *et al*. 2005b, Benavides *et al*. 2007) and (Trinidad *et al*. 2008), the authors transform FODA models with and without attributes into Boolean expressions. These expressions are executed on Constraint Satisfaction Problem (CSP), Satisfiability (SAT) and Binary Decision Diagrams (BDD) solvers in order to execute analysis and verification operations over feature models. In (Benavides *et al*. 2006), the authors show how to transform a FM with feature and group cardinalities into a CSP. The approach consists of representing each feature as a CSP variable. The domain of each variable depends on the cardinality associated to each variable. By default the domain is {0,1}. If a feature has a feature cardinality, then the domain of the variable is changed by the cardinality, disregarding the possibility of this feature to be cloned in the number of features determined by the feature cardinality. The relationships of the FM are represented as *ifThenElse* constrains on CPS and the final CSP for a FM is the conjunction of all the constraints. Despite the originality of this proposal, the constraint representing a feature cardinality (*m,n*) between the father feature *A* and its child *B* (according to their notation: *ifThenElse(A=0;B=0;B in {n,m})*) does not represent the fact that feature *B* can be cloned at last *n* and at most *m* times, neither consider that the feature *A* can itself have a feature cardinality, and in this case the semantics of feature cardinalities is not well represented in the constraint. The transformation approach of FMs into CP proposed in this thesis (cf. Section 3.2.1) corrects the defects of the Benavides's approach.

Recent work by Karataş *et al*. (2010) proposes a transformation from extended feature models to CP. This work does not consider the real semantics of features' attributes, considering them as sub-features that can be selected or not. Additionally, the transformation patterns used by Karataş *et al*. consider only Boolean formulas to represent extended feature models, which reduce the richness of the constraint programming paradigm, a richness necessary to represent complex feature models or to execute certain reasoning operations (e.g. to detect the optimal product according to a cost criterion). Besides, Karataş *et al*.'s representation of optional features permits the selection of child features without constraining the selection of the father feature.

From the point of view of the optimal representation of constraints, we exploit the fact that a PLMs can be represented in different CPs. For example: instead of representing the <1..1> group cardinality of Figure 3.6 as `UNIX * 1 ≤ Cdrom + Usb + Net ⋀ Cdrom + Usb + Net ≤ UNIX * 3` we use the representation `UNIX = Cdrom + Usb + Net`, which has the same semantics, but is more compact and less expensive from a computational point of view. With this kind of optimization, verification operations proposed in this thesis can be executed up to 25% faster. However further work is necessary in that direction to fully understand all the possible optimizations. Additionally, further work is necessary in multidirectional transformation, since the approach used in this thesis only considers unidirectional transformations, i.e., transformation from different product line modelling notations to CP but not inversely.

## 3.6. Conclusion

This chapter has presented an overview of the verification framework proposed in this thesis. In particular, it focused in the initial stages before the verification of PLM against the typology of verification criteria proposed in this thesis. Indeed, this chapter presents the following three key concepts, vital for reading and understanding of this thesis:

1. A UNIX product line, the running example used in this thesis to explain our verification approach. This running example is formulated as a collection of characteristics described in natural language. These characteristics depict the commonality and the variability of a UNIX product line from two points of view: technical point of view and end-user's point of view. From the technical point of view, they represent the components and the dependencies among them, that a UNIX PL should respect. From the end-user's point of view, the constraints represent the choice that an end-user has when he/she is installing a UNIX operating system. Both, the technical and the end-user characteristics are modeled with the feature formalism adopted in this thesis (cf. Section 3.1.1). The end-user view is modeled in the Dopler notation. The technical view is not modeled in the Dopler notation due to the fact that this notation is decision-oriented and therefore less adequate for users' views. Our thesis is that models specified with classical formalisms (FODA, OVM, Dopler, etc) cannot be verified as such, and therefore, they should first be transformed into an equivalent formalism that is more adequate for automating operations.
2. In our transformation approach, the PLM's semantics is represented with a collection of constraints and the PLM's abstract syntax is represented with a collection of facts. The

transformation approach presented in this chapter was tested against a large collection of product line models (cf. Chapter 8). The experiments showed that the approach is viable, scalable to very large PLMs and correct – according to the comparison with the tools FaMa (Trinidad *et al.* 2008b) and SPLOT (Mendonça *et al*. 2009).

3. The aim of the transformation and integration is to get product line models that are verifiable in an automated way. In order to achieve this, we propose a typology of verification criteria. This typology of verification criteria proposes two categories of verification: the first one associated with the verification of the abstract syntax of the model and other associated with the criteria that product line models should respect. This typology is presented in depth in the next chapters.

The integration approach exploits the fact that the PLMs that we verify are first represented as constraint programs. Once the models are represented as CPs, they can be integrated by means of a number of integration rules that can be chosen depending on the situation (e.g. integrate two views of a Dopler model *vs*. integrate two views specified with different formalisms), and the intention (conservative, restrictive, etc).

# Chapter 4

# Typology of Verification Criteria

Verifying PLMs entails several aspects. Certain properties associated with the domain of product lines and other properties are associated with the fact that each PLM respects the syntactic rules of the language in which it is expressed. Therefore, some properties of PLMs are associated with the language in which the model is represented, while other properties are associated to the domain of product lines. Thus, product line models seem to be verifiable from two different points of view. The first point of view is associated with the formalism. The other one is independent of the formalism in which models are represented. At first glance, it seems that the first category is specific to the formalism at hand, while the second one is applicable to every PLM, in other words, it should be generic. Our experience with various formalisms such as Dopler and several dialects of feature models showed us that the PL meta-models share some common concepts. For instance, all the metamodels have one or several start points from which the model should be navigated. They all provide concepts to specify reusable artefacts, dependencies among them to specify the variability, and in some cases properties to characterize these artefacts. Our thesis is that a generic approach can be taken. The proposal is to group them in a metamodel and then defined as a collection of generic criteria (Salinesi *et al*. 2004) that can be adapted to any PL meta-model in a fully automatic way. To identify defects in the first category, we propose a **conformance checking** approach directly related with verification of the abstract syntactic aspects of a model (cf. Definition 1.5). To identify defects in the second category, our approach uses a **domain-specific verification** approach (cf. Definition 1.4). Both categories of verification exploit verification criteria classified in the typology of verification criteria depicted in Figure 4.1. Each verification category is referenced under a unique number indicating the order in which the verification criteria should be considered. The typology has the form of a tree in which the nodes are categories of criteria, and edges generalization structures. The leaves of the typology represent individual criteria that can be used to perform verification. Intermediate nodes represent the category to which each criterion or sub-category belongs. For instance, in order to verify the expressiveness of a PLM, two criteria should be evaluated: *Non-void* and

*Non-false*. In the same way, the expressiveness criteria are used in order to verify the correctness of PLMs from a domain-specific point of view.

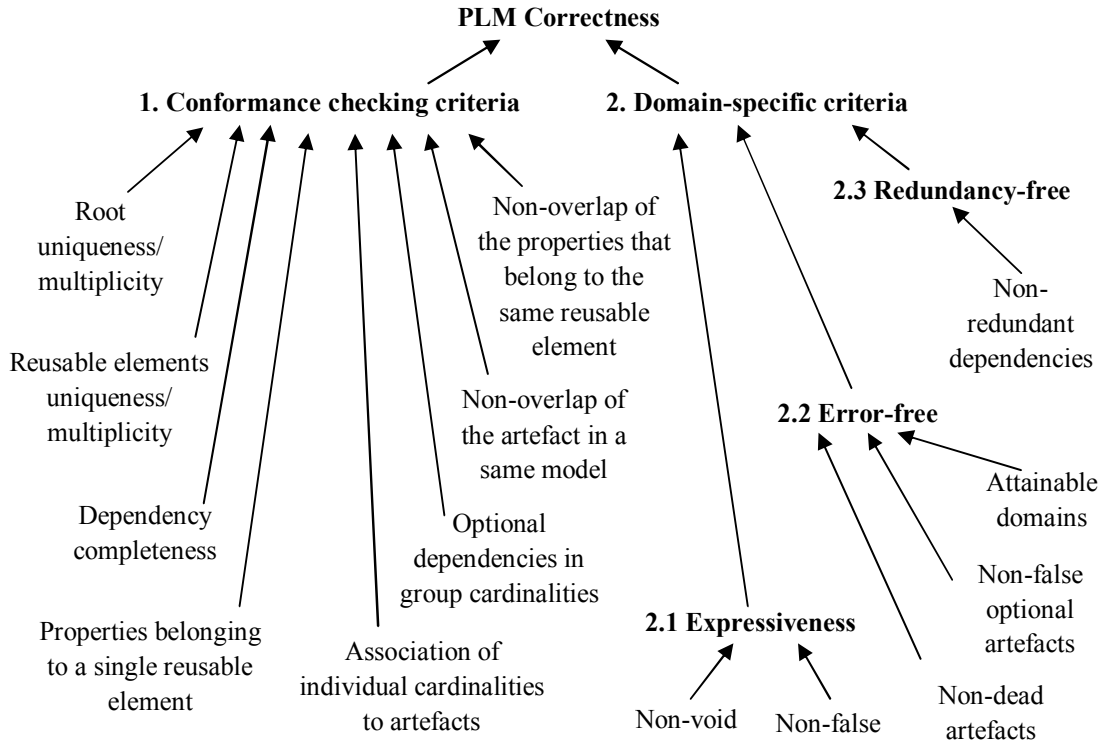



**Figure 4.1.** Typology of verification criteria on PLMs

The outcomes of the typology are multiple:

(a) the typology classifies the criteria from a semantic perspective, allowing the identification of similarities and differences among the criteria;

(b) the typology makes easier the identification of some defects for which no verification criterion is available in the literature. Redundancy of relationships among reusable elements is an example of defect for which no verification criterion has been defined in the literature (at least to our knowledge).

(c) the classification behind the typology produces a standard and reusable approach to verify PLMs; and

(d) the typology can be used to select the criteria that one wants to use to verify a PLM according to the impact that these criteria have or the expected quality level of a particular PLM.

(e) due to the fact that not all the verification criteria have the same impact, they have neither the same priority and consequently the same execution order. For instance, to check if

a PLM is non-false (i.e., rich enough) it is preferable to verify before that the model is non-void beforehand.

It is worth noting that the collection of verification criteria presented in Figure 4.1 could be extended as far as other PL modelling languages are considered and other verification criteria are identified. To guarantee that a PLM is defects-free, the typology of verification criteria considered must be as exhaustive as possible. Nevertheless, even in such a situation, (i) verification alone does not guarantee elimination of defects, and (ii) the correctness of a model can only be guaranteed with regard to the criteria used to evaluate the model (Finkelstein *et al*. 1996, Nuseibeh *et al*. 2000, Spanoudakis & Zisman 2001).

The following sections use the typology of verification criteria presented in Figure 4.1 to develop the verification approach proposed in this thesis.

# 4.1 Conformance Checking Criteria

From the point of view of conformance checking, the aim of this thesis is to verify if PLMs satisfy constraints captured from their respective metamodels. For the sake of generality, this thesis abstracts in a UML model the most important elements of several product line metamodels. This abstraction contains the common concepts of several product line metamodels existing in the literature such as FMs (Kang *et al*. 1990, 1998, 2002) and Dopler models (Dhungana *et al*. 2010). We decided to use these two formalisms to validate the verification approach proposed in this thesis, nevertheless we believe that our approach is applicable to other formalisms such as FODA, feature trees, OVMs, TVL, etc. We choose these two formalisms due to the fact that the first formalism is a common way to represent standalone PLMs and the second formalism is a common way to represent multi-model product lines; other formalisms were not used due to time constraints of the thesis' schedule. This abstraction is represented in Figure 4.2 and can be used to infer a collection of eight generic conformance criteria. For instance, the fact that every PLM should have one or several start points (or "roots"), from which the product line model must start in a configuration process. Of course, some aspects that directly relate with particular metamodels will remain untreated by the generic conformance checking approach. This chapter focuses on the generic criteria to verify PLMs; the particular criteria are then discussed in Chapter 5.

As Figure 4.2 shows, our view is that a PLM is composed of the description of at least one reusable element and at least one root artefact. Artefacts and dependencies are considered as reusable elements due to the fact that in a multi-stage configuration approach (Czarnecki *et*

*al*. 2004), not only artefacts are reused but also the dependencies among them. A PLM with only one dependency, should have at least two artefacts (at least one of them a root) due to the fact that a dependency relates to two or more artefacts. A PLM with only one artefact is not permitted. Besides, artefacts are related among them by means of one or several dependencies and each dependency relates at least two artefacts. Each artefact has a unique name and a domain of values. A domain is represented as a list of values. Each value of the domain has a specific meanning (e.g., the artefact PCMCIA_SUPPORT of our running example, where the value 0 means that the artefact is not charged (selected) in a particular UNIX system, 1 means that the artefact is charged in a static way, and 2 means that the artefact is charged in a dynamic way). An artefact can have an individual cardinality (at most one). In addition, optional dependencies can be grouped in a group cardinality (by two or more). Both, individual and group cardinalities have two attributes; `Min` and `Max`, that represent the minimum and maximum values of the cardinality. A reusable element can have several properties, and the other way round a property belongs to one and only one reusable element. Examples of properties of reusable elements are: the attribute of an artefact, or the type of a dependency. An attribute such as for instance the price of an artefact, should have a unique name, a type and the possible values that the attribute can take (its domain). Dependencies are also reusable elements, and therefore they can have properties. For instance, dependency properties should indicate the type of dependency, e.g requires or excludes (which are usually represented by a special arrow), and the unique name or identifier used to uniquely identify the dependency in the model. It is worth noting that in Dopler and feature models, the name of dependencies is not explicitly visible in the models. However, they are necessary when the models are merged, for instance. In addition, in Dopler models, the type of each dependency is usually presented in the models.

Our approach in this thesis is to exploit the metamodel of Figure 4.2 to specify verification criteria. In particular, this section will exploit the metamodel of Figure 4.2 in order to derivate eight conformance checking criteria. The strategy to achieve this was similar to business rules derived from a conceptual model: each constraint in this figure becomes a conformance checking criterion and each criterion is implemented as a conformance rule (cf. Definiton 1.6).

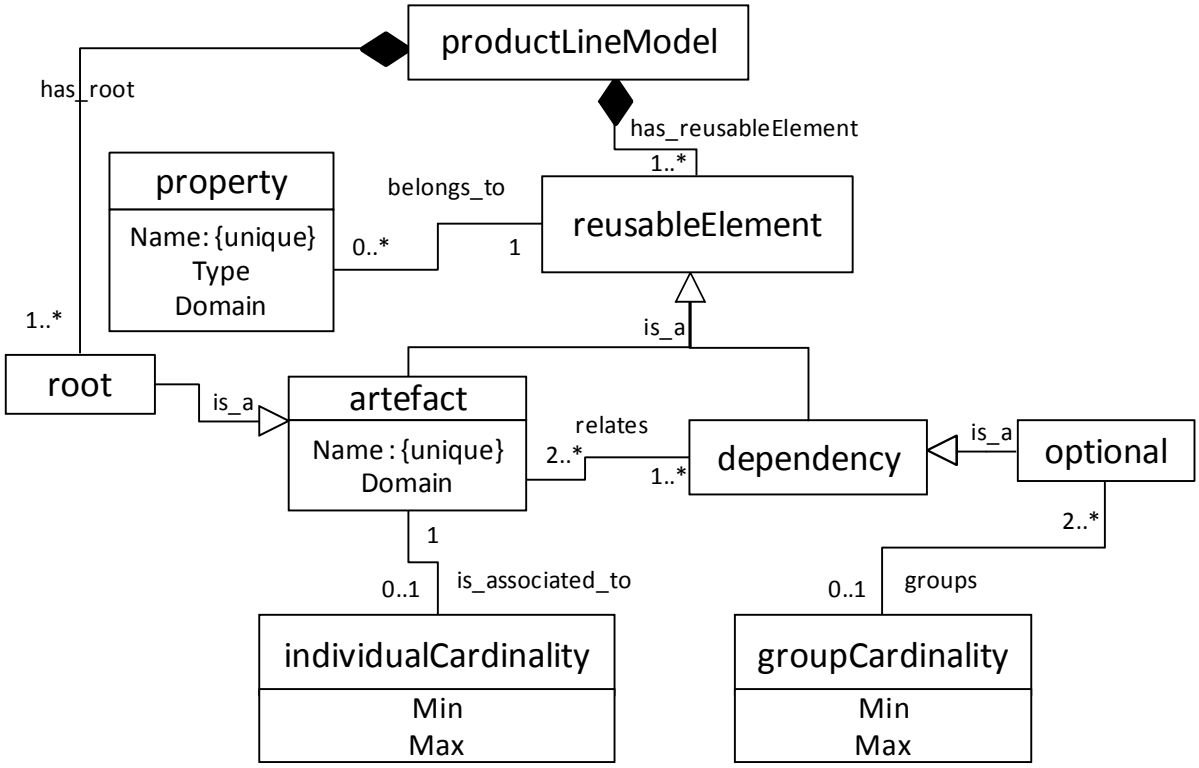



**Figure 4.2** UML class diagram representation of common elements of several product line metamodels

Prior to formalizing the verification criteria in first order logic (FOL), a certain number of predicates (Osman *et al.* 2008), derived from the metamodel of Figure 4.2, must be defined. In order to do that, each class of the metamodel is defined as a predicate that evaluates if a given variable corresponds to the type of element represented by the class. In addition, each dependency of the metamodel is defined as a predicate, as follows:

- *has_root*: evaluates the fact that an artefact of type *root* is the root of a product line model. For example, *has_root*(M, R) is a FOL predicate that evaluates if the product line model M has a root artefact R.
- *has_reusableElement*: evaluates if there is a reusable element R in the product line model M. For example, *has_reusableElement*(M, R) returns *true* if there is a reusable element R in M.
- *belongs_to*: evalutes if there is a property P that belongs to a *reusable element* R. For example, *belongs_to*(R, P) reuturns *true* if P is a property of R.
- *is_a*: evaluates if a given class B, of the metamodel represented in Figure 4.2, is a subclass of another class A. For example, *is_a*(A, B) returns *true* if B is a subclass of A, and *false* otherwise.

- *relates*: evaluates if there a list of artefacts [A1, A2, …, Ai] that are related to a dependency D. For example, *relates*(D, [A1, A2]) returns *true* if there is a dependency D that relates the artefacts A1 and A2.
- *is_associated_to*: evaluates if there is an individual cardinality IC associated to an artefact A. For example, *is_associated_to*(A, IC).
- *groups*: evaluates if a collection of optional dependencies [O1, O2, …, Oi] belong to a group cardinality GC. For example, *groups*(GC, [O1, O2]) returs *true* if the optional dependencies O1 and O2 belong to the group cardinality GC.

And the following function:

- *isPLMofOptional*: this function returns, for a given dependeny of type optional, the product line model to which the dependency belongs. For example, *isPLMofOptional*(O).

The eight criteria are: Root uniqueness/multiplicity, Reusable elements uniqueness/multiplicity, Dependency completeness, Non-overlap of properties for each artefact, Association of individual cardinalities to artefacts, Optional dependencies in group cardinalities, Non-overlap of the artefact in a same model and Properties belonging to a single reusable element. Each of these criteria is defined, formalized in FOL according to Bradley & Manna (2007 pp. 51) and illustrated with our running example (cf. Figures 3.5, 3.6 and 3.7) as follows.

**A. Root uniqueness/multiplicity**

Definition: a product line model is composed of one or several root elements, which are special kinds of artefacts; even if some languages use other words to describe the same concept as is the case in OVM or Dopler models.

Formalization: $\forall M, \exists R . productLineModel(M) \wedge root(R) \wedge has\_root(M, R)$

Example: while the FODA metamodel (cf. Kang *et al*. 1990) constrains the existence of one and only one root artefact, the FOPLE (Feature Oriented Product Line Software Engineering, cf. Kang *et al*. 2002) metamodel considers the existence of several root features. In our running example, `UNIX` is the root of both feature models (cf. Figures 3.5 and 3.6). In the same way, `Means of installation` is the root decision of the Dopler model of Figure 3.8. Therefore the three models comply with this conformance criterion.

**B. Reusable elements uniqueness/multiplicity**

Definition: a PLM is composed of one or several reusable elements. This criterion is a necessary but not sufficient condition to enable the derivation of several products from a product line model.

Formalization: $\forall M, \exists R . productLineModel(M) \wedge reusableElement(R) \wedge has\_reusableElement(M, R)$

Example: the feature models of Figures 3.5 and 3.6 have 25 reusable elements (15 features and 20 dependencies) and 32 reusable elements (14 features and 18 dependencies), respectively. The Dopler running example shown in Figure 3.8 has 11 artefacts (i.e., 4 decisions and 7 assets) and 12 dependencies. Therefore the three models comply with this conformance criterion.

**C. Dependency completeness**

Definition: in a PLM, each dependency relates two or several artefacts. Indeed, all product line models have dependencies, at least one in order to represent the variability and the commonality of the product line. It is worth noting that for the "classical" dependencies (i.e., optional or mandatory or requires or excludes) the related artefacts should be different; however, there are particular dependencies in some product line formalisms in which an artefact can be related with itself (e.g., the validity condition that relates the decision "Width?" with itself by means of the constraint "Width $\geq$ 800 && Width $\leq$ 1366"). Due to the fact that the most of dependencies in product line models are optional, mandatory, requires and excludes dependencies (cf. Tables 10.1 to 10.5 on Appendix), our formalization and implementation considers the case where each dependency relates two or several different artefacts.

Formalization: for every PLM, and for each dependency D of M, there are two different artefacts related by D:

$\forall M, \forall R1, \forall D, \exists R2, \exists R3, \exists A1, \exists A2 . productLineModel(M) \wedge reusableElement(R1) \wedge reusableElement(R2) \wedge reusableElement(R3) \wedge dependency(D) \wedge has\_reusableElement(M, R1) \wedge has\_reusableElement(M, R2) \wedge has\_reusableElement(M, R3) \wedge is\_a(R1, D) \wedge artefact(A1) \wedge artefact(A2) \wedge is\_a(R2, A1) \wedge is\_a(R2, A2) \wedge relates(D, [A1, A2]) \wedge A1 \neq A2.$

It is also necessary to express that every PLM has at least one dependency:

$\forall M, \exists D, \exists R.\ producLineModel(M) \wedge dependency(D) \wedge reusableElement(R) \wedge has\_reusableElement(M,R) \wedge is\_a(R,D)$

Example: Figure 3.6 has an optional dependency between the father feature `UNIX` and the child feature `UtilityProgram`. The dependency `contains(Utility program, "OnlineInfo")` $\Rightarrow$ `GraphicalResolution` of the Dopler model depicted in Figure 3.8 represents a requirement dependency. This dependency implies that if the user chooses the utility program called `OnlineInfo`, he/she should also choose a `GraphicalResolution`.

**D. Non-overlap of the artefact in a same model**

Definition: each artefact of a PLM should be identified in a unique manner. Most of the time, the name of the artefact permits the identification of the artefact. Therefore, this name should be unique (within each model) to avoid redundancies and evolutions problems. However, this conformance criterion is not violated when two different models contain artefacts with the same name.

Formalization: $\forall M, \forall R1, \forall R2, \forall A1, \forall A2\ .producLineModel(M) \wedge has\_reusableElement(M,R1) \wedge has\_reusableElement(M,R2) \wedge reusableElement(R1) \wedge reusableElement(R2) \wedge artefact(A1) \wedge artefact(A2) \wedge is\_a(R1,A1) \wedge is\_a(R2,A2) \wedge (A1 \neq A2) \rightarrow (A1.\text{Name} \neq A2.\text{Name})$

Example: in our running example, the artefacts of each model have a unique name in the model. For example, the feature called `UserInterface` appears only one time in the model of Figure 3.5 and only one time in the model of Figure 3.6.

**E. Properties belonging to a single reusable element**

Definition: a property belongs to one and only one reusable element.

Formalization: $\forall P, \forall R1, \forall R2\ .property(P) \wedge reusableElement(R1) \wedge reusableElement(R2) \wedge belongs\_to(R1,P) \wedge belongs\_to(R2,P) \rightarrow R1 = R2$

Example: in Figure 3.5, the artefact called `Graphical` has two properties: `WidthResolution` and `HeightResolution`. Due to the fact that these properties belong to only one artefact, the model of Figure 3.5 complies with this conformance criterion. This criterion allows, for instance, distinguishing the property `price` of an artefact `A` from the `price` of another artefact `B`.

**F. Non-overlap of the properties that belong to the same reusable element**

Definition: in a same product line model, a property has a unique name. Indeed, the name of each property should be different from one another in order to avoid ambiguity problems during product configuration, verification management (as presented in the general frameworks proposed by Finkelstein *et al*. (1994, 1996), Nuseibeh *et al*. (2000) and Spanoudakis & Zisman (2001)) and maintenance stages.

Formalization: $\forall M, \forall R, \forall P1, \forall P2 \,.\, producLineModel(M) \wedge reusableElement(R) \wedge property(P1) \wedge property(P2) \wedge has\_reusableElement(M, R) \wedge belongs\_to(R, P1) \wedge belongs\_to(R, P2) \wedge P1 \neq P2 \rightarrow P1.Name \neq P2.Name$

Example: in Figure 3.5, the attributes `WidthResolution` and `HeightResolution` have a unique name and a domain [800, 1024, 1366] and [600, 768] respectively. In Figure 3.8, the dependency of type "Visibility condition" has the value "isTaken(Means of installation)" and is identified with an artificial name, e.g. `Depd01`.

**G. Association of individual cardinalities to artefacts**

Definition: an individual cardinality is associated to one and only one artefact. Even if two individual cardinalities have the same values, each one of them would be associated to different artefacts. This conformance criterion is useful from the point of view of maintenance since the elimination of an individual cardinality for a particular artefact entails its elimination for the corresponding artefact, and this action does not affect the other artefacts that have an identical cardinality.

Riebisch *et al*. (2002) propose to extend FMs with cardinalities similar to those found in UML class diagrams. This kind of cardinality, called individual cardinality, is a concept used in several feature metamodels (e.g., Riebisch *et al*. 2002, Sun *et al*. 2005, Czarnecki *et al*. 2005, Michel *et al*. 2011); however, it is not used in most of product line metamodels in literature, e.g., FODA (Kang *et al*. 1990), OVM (Pohl *et al*. 2005) and Dopler (Dhungana *et al*. 2010).

Formalization: $\forall C, \forall A1, \forall A2 \,.\, individualCardinality(C) \wedge artefact(A1) \wedge artefact(A2) \wedge is\_associated\_to(A2, C) \wedge is\_associated\_to(A1, C) \rightarrow A1 = A2$

Example: artefact `Process` in Figure 3.5 has an individual cardinality [0..*] indicating that the artefact `Process` can be instantiated several times in a same product. Therefore the model of Figure 3.5 complies with this conformance criterion.

**H. Optional dependencies in group cardinalities**

Definition: a group cardinality gathers two or more optional dependencies. By definition 3.8, a group cardinality is about the selection of a certain number of artefacts among a collection of them. In this selection each artefact must have the same possibility to be chosen as the others, which is why dependencies must be optional. This thesis considers this conformance criterion, initially presented in (Czarnecki *et al*. 2005), as a good practice. It is worth noting that this thesis relates group cardinalities and dependencies (not artefacts). This choice is due to the fact that graphically a group cardinality groups two or more dependencies and not two or more artefacts. This choice was also made to improve the performance of the criterion implementation.

The following FOL formula represents the fact that for all groupCardinality there are two optional dependencies, of the same product line model, associated to the groupCardinality.

Formalization: $\forall G, \exists X1, \exists X2 \,.\, groupCardinality(G) \wedge optional(X1) \wedge optional(X2) \wedge X1 \neq X2 \wedge isPLMofOptional(X1) = isPLMofOptional(X2) \wedge groups(G, [X1, X2])$

Example: Figure 3.6 presents three group cardinalities: `<1..1>` that group dependencies between the father feature `UNIX` and the child features `Cdrom`, `Usb` and `Net`, `<1..2>` that group the dependencies between the father feature `UserInterface` and the child features `Graphical` and `Shell`, and `<1..3>` grouping the dependencies between the father feature `Shell` and the child features `SH`, `TCSH` and `BASH`.

# 4.2 Domain-specific Criteria

In the context of product lines, domain-specific verification criteria are properties that every model has to respect in order to be a “real” PLM. In addition the model should correctly represent the domain intended to be represented whit the PLM. This thesis, due to the fact that all these verification criteria are related with the domain of the product lines, groups all these verification criteria under the name of domain-specific verification.

This thesis proposes four groups of domain-specific verification criteria: expressiveness, error-free, consistency and redundancy-free. Each group is composed of one or several domain-specific verification criterion. Each one of these criterion is defined, formalized and exemplified with our running example (cf. Figures 3.5, 3.6 and 3.7) as follows.

The formalization corresponds to FOL formulas that use the following predicates (Osman *et al.* 2008). These predicates are implemented in the Analysis module of the tool VariaMos (Mazo & Salinesi 2011).

- *isOptional*: this predicate evaluates if an artefact A of a product line model M is modeled as an optional artefact (i.e., if the artefact is the child of an optional dependency). For example, *isOptional*(M, A) returs *true* if the artefact A is modeled as optional in the model M, or false otherwise.
- *oneProduct*: returns *true* if a given product line model M allows the configuration of at least one product, and *false* otherwise. For example, *oneProduct*(M).
- *oneProductWithConstraint*: returns *true* if a given product line model M allows the configuration of at least one product that respects a given constraint C, and *false* otherwise. To be precise, constraints are mathematical expressions over variables, representing Properties and Artefacts, and constants over a given domain. For example, *oneProductWithConstraint*(M, C), where C is '$Property_1$+$Property_2$'.
- *find*: returns *true* if a certain number of different products can be derived from a *product line model* M, and *false* otherwise. For example, *find*(M, 2) is *true* if the product line model M allows the derivation of at least 2 products.

We also need to define the following functions, other than the function *isPLMofOptional* defined in Section 4.1:

- *semanticOf:* retunrs the collection of products that are possible to be derivated from a product line model M or *false* if no product can be derivated from the model. For example, *semanticOf*(M).
- *eraseOneDependency:* this function takes a product line model M and a dependency D, and returns the product line model whitout the dependency D. For example: eraseOneDependency(M, D).
- *isPLMofDependency(D)*: this function returns the product line model associated to the dependency D.

### 4.2.1 Expressiveness

Every PLM should permit the configuration of more than one product, i.e., the model should not be void, and be expressive enough to permit the configuration of more than one product (Benavides *et al.* 2005). Indeed, the purpose of PLMs is to represent at least two products – otherwise, there is no reuse. The former is called "non-void" and the latter "non-false". Each

criterion can be used to verify the expressiveness of PLMs at its turn. It is however worth noting that the latter includes the former: if a PLM is non-false, then it is non-void too.

**A. Non-void PLMs**

Definition: a void PLM is defined as a model that does not permit the configuration of any product.

Formalization: to formalize this criterion, we define the following Boolean function:

$nonVoidPLM(M) \triangleq producLineModel(M) \wedge find(M,1)$

Example: FMs of Figures 3.5 and 3.6 are not void. Dopler model of Figure 3.8 is not void. Each of these models allows configuring at least one product.

**B. Non-false PLMs**

Definition: a false PLM is defined as a model that permits configuring one product only. In this case, the model cannot be considered as a PLM, but as a product model. This criterion can be automated by means of an operation that takes a PLM as input and returns "False PLM" if at most one valid product can be configured with it. Although this operation would also help to detect when PLMs are void (our precedent operation), the converse is not true. The two criteria have then a separate definition.

Formalization: $nonFalsePLM(M) \triangleq producLineModel(M) \wedge find(M,2)$

Example: FMs of Figures 3.5 and 3.6 are not false. Dopler model of Figure 3.8 is not false. All these models permit the configuration of at least two products each one.

### 4.2.2 Error-free

The Dictionary of Computing defines an error as "A discrepancy between a computed, observed, or measured value or condition, and the true, specified, or theoretically correct value or condition" (Howe 2010). In PLMs, an error represents a discrepancy between what the engineer wants to represent and the result obtained from the model. For instance, this is the case when the engineer includes a new reusable element (in a given domain) in a PLM, but this element never appears in a product. Our ontology proposes three criteria to identify errors in PLMs: non-attainable domain values of PLM's reusable artefacts, i.e. an artefacts `A` with a domain [0, 1, 2, 3] but can never attain the values 1 and 2; dead artefacts, i.e. artefacts of the PL that are never used in a product: this means that their value is all time 0; and the third criterion permits the identification of the reusable elements modeled as optional but that appear in all the products of the PL, i.e. an artefacts `A` with a domain [0, 1, 2, 3] but can never attain de value 0. It is worth noting that the former criterion includes the second (artefacts

that can never attain non-cero values), and the third one, for optional artefacts (i.e., when the domain of the artefact includes the 0 value). However, the opposite is not true.

**C. Non-dead Artefacts**

Definition: An artefact is dead if it cannot appear in any product of the product line. From a mathematical point of vew, an artefact is dead if it is always setted to 0 in each one of the products that can be derived from the product line model. Our formalization expresses the fact that it should be possible to configure at least one product with each artefact of a product line model.

Formalization: $\forall M, \forall R, \forall A . productLineModel(M) \wedge reusableElement(R) \wedge artefact(A) \wedge has\_reusableElement(M, R) \wedge is\_a(R, A) \wedge onePoductWithConstraint(M, 'A > 0')$

Example: There are no dead artefacts in our running example. In other words, artefacts in feature and Dopler models have the possibility to be selected almost one configuration.

**D. Non-false Optional Artefacts**

Definition: An optional artefact is an artefact playing the role of child in an optional dependency. An artefact is false optional if it is included in all the products of the product line despite being declared optional (Von der Maßen & Lichter 2004, Benavides *et al.* 2005, Trinidad *et al.* 2008). Our formalization expresses the fact that if an artefact is optional, it should be possible to configure at least one product whitout this artefact (i.e., setted to 0).

Formalization: $\forall M, \forall R, \forall A . productLineModel(M) \wedge reusableElement(R) \wedge artefact(A) \wedge has\_reusableElement(M, R) \wedge is\_a(R, A) \wedge isOptional(M, A) \rightarrow onePoductWithConstraint(M, 'A = 0')$

Example: Feature `Process`, in Figure 3.5, with individual cardinality [0..5] is included by several futures bellowing to the core of the product line. Therefore, feature `Process` is a false optional feature due to the fact that it appears in all the configurations of the product line despite the zero value of its individual cardinality.

**E. Attainable Domains**

Definition: A non-attainable domain value is the value of an artefact, or a property, that never appears in any product of the product line. For example, if an artefact `A` has the domain `[0,1]`, value `1` is non-attainable if `A` can never be integrated in a product line;

i.e., it never takes the value of `1`. Non-attainable values are clearly undesired since they give the user a wrong idea about domain of reusable elements.

Formalization: First, we evaluate each value $\texttt{Domain}_i$ of the artefacts' domain:

$\forall M, \forall R, \forall A.\ \text{Domain}_i\ .\ producLineModel(M) \wedge reusableElement(R) \wedge artefact(A) \wedge has\_reusableElement(M, R) \wedge is\_a(R, A) \wedge oneProductWithConstraint(M, \text{'A= A.Domain}_i\text{'})$

Second, we evaluate each value $\texttt{Domain}_i$ of the propertie's domain:

$\forall M, \forall R, \forall P.\text{Domain}_i\ .\ producLineModel(M) \wedge reusableElement(R) \wedge Property(P) \wedge has\_reusableElement(M, R) \wedge belongs\_to(R, P) \wedge oneProductWithConstraint(M, \text{'P= P.Domain}_i\text{'})$

Example: In Figures 3.5 and 3.6 the feature `UNIX` can never take the value of 0 due to the fact that this feature plays the role of root and therefore its value is constant to `1` even if it is a Boolean feature. Indeed, all core features of each model take the constant value of `1`. In the Dopler model of Figure 3.8, the decision `Means of installation` that is the root decision of the model only takes the value of `1`; i.e., this decision must be considered in all configurations.

### 4.2.3 Redundancy-free

According to the Oxford Dictionary, something redundant is something "able to be omitted without loss of meaning or function" (Oxford University 2008). Therefore, redundancy in a PLM is about the presence of reusable elements and variability constraints among them that can be omitted from the PLM without loss of semantics on the PLM. Redundant dependencies in FMs are undesired because, although they do not alter the space of solutions, they may consume extra computational effort in derivation and analysis operations, as demonstrated in (Yan *et al.* 2009), and they are likely to generate inconsistencies when the PL evolves. For the sake of evolution, it is certainly better to detect and correct these redundancies. However, and due to the fact that this thesis represents PLMs as constraint programs, it is also worth noting that in constraint models the presence of redundant dependencies is not necessarily undesired. While the space of solutions remains the same, more propagation might occur. Quite frequently, particular redundant dependencies are added explicitly to improve the solving performance and then removed to not affect the evolution of the model, as presented in (Borrett & Tsang 2001). In order to detect redundant dependencies in a PLM this thesis proposes an operation that takes a PLM and a constraint (and its

negation) as input and returns *true* if removing the constraint does not change the space of solutions.

**F. Non-redundant Dependencies**

Definition: a redundant dependency is a dependency that does not reduce the semantics of a PLM. In other words, the collection of products that can be generated with or without the constraint are identical. It is worth noting that redundancy is not a bijection as a dependency can in fact be subsumed by the conjunction of several other dependencies. The following formalization corresponds to the *redundantDependency* function that returns *true* if the dependency `D` is redundant and *false* otherwise.

Formalization: $redundancyDependency(D) \triangleq dependency(D) \wedge (semanticOf(isPLMofDependency(D)) = semanticOf(eraseOneDependency(isPLMofDependency(D), D))$

Example: In Figure 3.5, the dependency `Shell` *requires* `Executing Instructions` and the constraint `Shell` *requires* `Interpreting Instructions` are both redundant due to the fact that features `Executing Instructions` and `Interpreting Instructions` are included in all products and therefore they do not need to be included by the feature `Shell` in order to be included in a particular product.

## 4.3 Multi-model Verification Criteria

Representing a PL with several models permits tackling various aspects of the product line. This happens in particular, in the presence of multiple stakeholders with various viewpoints (executives, developers, distributors, marketing, architects, testers, etc.; cf. Nuseibeh *et al.* 1994). For example, a UNIX product line can be composed of several models, each one developed by a different team or developing a particular view of the PL. Thus, while the team responsible for the kernel develops a model, the team responsible of the user interface develops another model. Motivated by the fact that (a) this practice is current in industry (Dhungana *et al*. 2010); (b) even if each individual model is consistent, once the models are integrated, they can be inconsistent; and (c) current state of the art lacks proposals for multi-model PL verification; this thesis proposes a method to verify multi-model PLs. Overall, the approach is to integrate the models after having transformed them into a CP. The proposed method is composed of three steps: (i) the base models are transformed into constraint programs that grasp their semantics; (ii) the resulting CPs are integrated using a series of

different integration strategies and rules according to each language; and (iii) the collection of verification criteria, initially proposed in this thesis for standalone models, are applied on the integrated model in the same way as for standalone PLMs. From the point of view of integration, it is worth noting that the case where the models come from different metamodels was not considered in this thesis and instead, proposed as future work. The multi-model verification approach is further explained in Chapter 7.

## 4.4 Summary

In order to verify models against the verification criteria classified in the typology presented in this chapter, it is necessary to represent PLMs in an expressive-enough language to represent both the semantics and the syntax of PLMs. Experience shows that the semantics of PLMs can be represented as a collection of variables over different domains and constrains among these variables. While the variables specify what can vary from a configuration to another one, constraints expressed under the form of restrictions specify what combinations of values are permitted in the products. This approach extends to the structure of a PLM, which can be represented as a collection of logic facts with the elements of each PLM and a collection of relationships among them to represent abstract syntax of the particular model.

**The typology of verification criteria** emphasizes the difference between domain-specific and conformance defects. In the case of domain-specific verification, the defects are associated with non-expressiveness, errors, inconsistencies and redundancies. In the case of conformance checking, the purpose is to verify if the abstract syntax of a model is correct with regards to the corresponding metamodel. Once the abstract syntax of the PLM is translated into a constraint logic program, users can verify it against a collection of conformance criteria. These criteria are generic due to the fact that they are derived from a generic PL metamodel; then, they could be adapted to the particular metamodel at hand. For example, if the metamodel specifies that PLMs can have only one root artefact, the user can propose a conformance checking rule for this criterion. The implementation and evaluation of the PLM verification approach presented in this chapter is presented in the remaining chapters of this thesis.

# Chapter 5

# Conformance Checking of Product Line Models

In general, conformance checking, cf. Definition 1.5, is a kind of consistency checking focused on verification of a model (a product line model in our case) against a collection of conformance criteria generated from the corresponding metamodel. The difference between conformance checking and consistency checking is that consistency checking consists of "analyzing models to identify unwanted configurations defined by the inconsistency rules" (Cabot & Teniente 2006), whereas conformance checking focuses on the verification of the PLM itself and not on its potential configurations. The choice to deal with conformance checking instead of consistency checking is supported by the fact that this thesis exclusively focuses on verification of PLMs in the domain engineering stage (cf. Definition 1.1) and not on finding inconsistent configurations.

Another issue in the context of product lines is that product verification should be achieved at the level of product line models first. Indeed, product models are not instantiated from their meta-models, but by configuration of PLMs. The assumption is that any product model configured from a correct PLM is itself correct. On the semantic level, a product line model is defined as the collection of all the product models that can be derived from it. Therefore checking the conformance of the product line model is equivalent to checking the conformance of all the possible product models (Djebbi & Salinesi 2007). However, we would like to avoid verifying all the product models because of a scalability issue: their number can be simply too high (Mendonça *et al*. 2009). The naïve approach that consists of achieving product model verification by checking their conformance with the product line meta-model is also not scalable to real world constraints: one does not want to deal at the application engineering stage with errors that should have been detected during domain engineering.

This chapter is structured as follows. Section 5.1 presents a generic approach to check conformance of product line models, based in a generic product line metamodel. The generic approach is based in a collection of verification criteria classified in the ontology of verification criteria presented in the previous chapter. For each conformance checking

criterion, Section 5.1 provides an explanation with regards to the generic product line metamodel and an algorithm to implement each criterion. Section 5.2 presents the conformance checking approach applied to a particular formalism: feature models. This specific approach specialises the generic conformance checking criteria presented in Section 5.1 to the feature model metamodel. Other criteria are particular to the FM metamodel and therefore they are not present in the ontology of generic verification criteria. For each FM conformance checking criteria, an explanation with regards to the FM metamodel, an algorithm in pseudo-code, and its implementation in CP, is provided. Section 5.3 summarises the chapter.

## 5.1 Generic Conformance Criteria for Product Lines Models

The approach proposed in this thesis to check the conformance of PLMs exploits (i) a collection of generic conformance criteria, as presented in Chapter 4, and (ii) a collection of specific conformance criteria.

The *generic conformance criteria* are the ones generated from the generic product line metamodel presented in Chapter 4 (cf. Figure 4.2).

The specific conformance criteria are directly related to specific aspects of each metamodel, i.e. aspects that are not grasped by the generic metamodel. For instance, the FM metamodel, presented in Figure 3.4, contains the notion of mandatory dependency. This concept is not handled in the generic metamodel of Figure 4.2. It is therefore a specific concept that requires specific verification criteria.

This section focuses on the generic conformance checking criteria highlighted in Figure 5.1. For each one of these generic conformance checking criteria, an algorithm is proposed. It is worth noting that even if the conformance checking criteria are generic, they should be adapted to each particular metamodel. Therefore the implementation of each criterion should be specialized for each particular metamodel too. For instance, the generic criterion “Root uniqueness/multiplicity”, which specifies that a PLM is composed of one or several root artefacts, uses the concepts and constraints of the generic PL metamodel (cf. Figure 4.2). The FM criterion “root uniqueness”, which specifies that a feature model is composed of one and only one root feature, uses the concepts and constraints of the FM metamodel used in this thesis (cf. Figure 3.4). As this example shows, there are no contradictions between both criteria. Instead, the specific criterion is a particular instance of the generic criterion. In this

case, the specific conformance criterion uses the concepts and constraints of the FM metamodel instead of the ones of the generic PL metamodel. Defining this criterion can be simply done by specializing the concepts of the generic criterion:

Product Line Model → Feature Model,

artefact → feature,

root artefact → root feature.

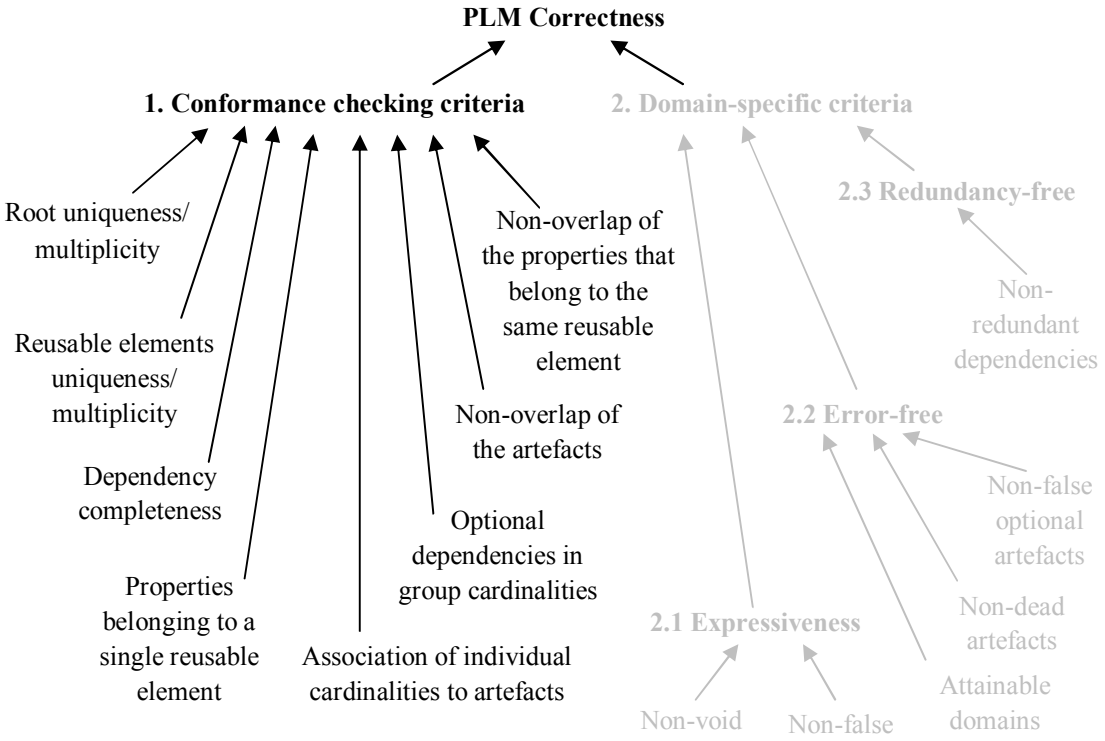



**Figure 5.1.** Generic conformance checking criteria

### CC.1. Root uniqueness/multiplicity

This conformance checking criterion is highlighted in Figure 5.2, over the generic PLM metamodel presented above in Figure 4.2. This criterion specifies that a PLM is composed of one or several root artefacts.

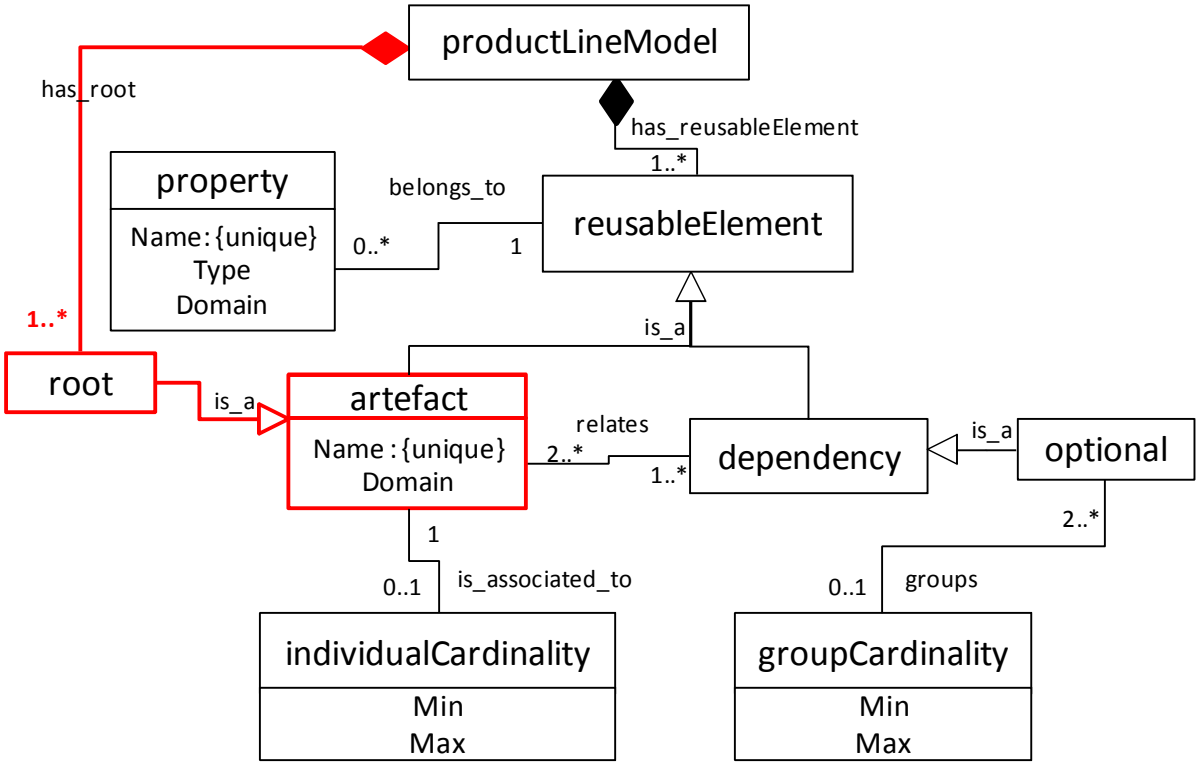



**Figure 5.2.** CC.1 highlighted in the generic PLM metamodel.

Algorithm: The algorithm that implements this conformance criterion looks for root artefacts in the PLM. Then, each root artefact found is kept in a list. Once all the artefacts of the PLM are evaluated, the algorithm computes the number of elements in the list. If this number is equal to 0, a conformance defect is raised. The algorithm of this conformance criterion is specified as follows.

```
For each productLineModel M
{
   RootArtefactList ='';
   For each root artefact RA
   {
      RootArtefactList += RA;
   }
   N = length(RootArtefactList);
   If (N = 0)
   {
      return 'defect found in model:' M;
   }
}
```

A more interesting problem consists of determining the root artefacts among the collection of artefacts of the product line model when the type of the root artefacts is not known in advance. However, the root artefacts in the product line modelling formalism referenced in this thesis uses a particular tag to indicate that an artefact is a root artefact, for instance: in the Dopler formalism, even if the root concept does not exist in the Dopler

metamodel (cf. Figure 3.7), the visibility condition of certain decisions is set to *true* to indicate that the decision is a root decision. In feature models the root concept is explicitly presented in the FM metamodel (cf. Figure 3.4) and therefore the concept should be explicit in the FMs. For this reason, the algorithm searches for the root artefacts assuming that there are some criteria to define if the artefact is root or not.

### CC.2. Reusable elements uniqueness/multiplicity

This conformance checking criterion is highlighted in Figure 5.3, over the generic PLM metamodel shown in Figure 4.2. This criterion specifies that a PLM is composed of one or several reusable elements.

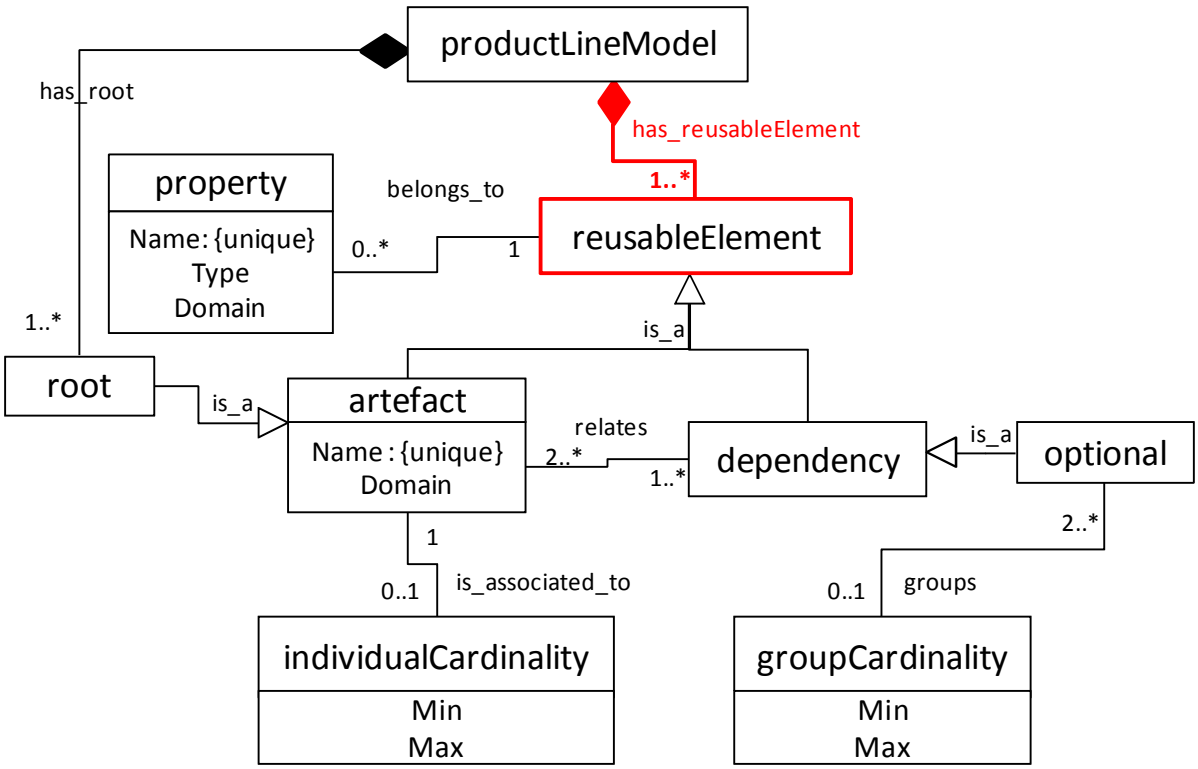



**Figure 5.3.** CC.2 highlighted in the generic PLM metamodel.

Algorithm: this algorithm looks for at least one reusable element. Once a reusable element is found, the algorithm breaks. The break is because of the first reusable element is found and therefore there is no conformance violation in the model at hand regarding this conformance criterion. If no reusable element is found in the model, the algorithm shows a conformance violation to the user.

```
For each productLineModel M
{
   fly = false;
   For (each reusableElement R) {
      fly = true;
      break;
```

```
        }
        If (fly = false) {
            returns 'defect found in model:' M;
        }
    }
```

The break avoids useless computations when a first reusable element is found in a given model. If no reusable elements are found in a given model, the variable *fly* keeps its value *false* and therefore at the end of each model the algorithm evaluates the value of this variable and if the value is *false* violation of this conformance criterion is signalled to the user.

### CC.3. Non-overlap of the artefacts that belong to the same model

This conformance checking criterion is highlighted in Figure 5.4, over the generic PLM metamodel of Figure 4.2. This conformance criterion specifies that each artefact has a unique name in the model to which the artefact belongs. However, another PLM can have another artefact with the same name.

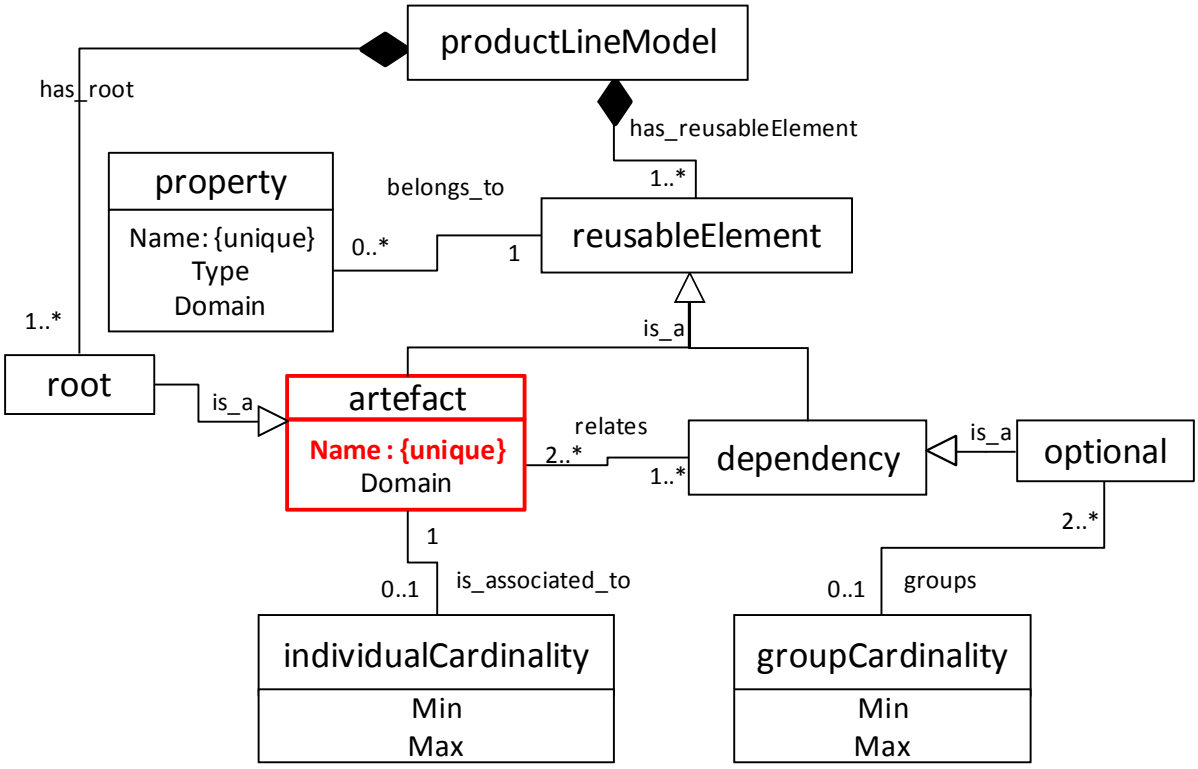



**Figure 5.4.** CC.3 highlighted in the generic PLM metamodel.

Algorithm: The algorithm for this conformance criterion looks for pairs of different artefacts that share the same name. Each time the algorithm identifies a pair of artefacts with the same name, it returns to the user the identifiers and the name of both artefacts, and the model in which the defect was found. The algorithm is as follows:

```
For each productLineModel M
{
   For (each artefact A1)
   {
      If (there is an artefact A2 and A1≠A2 and A1.Name=A2.Name)
      {
            Return (A1, A2, Name, M);
      }
   }
}
```

Several algorithms are proposed in literature in order to detect overlaps (i.e., variables that belong to a same model and sharing the same name). Several of these proposals are: (i) unification algorithms (Knight 1989), where the unification algorithm performs a systematic matching between the terms that they are given to unify. (ii) Shared anthologies, where authors of the models should tag the elements with items in a shared ontology. The tag of a model element is taken to denote its interpretation in the domain described by the ontology and therefore it is used to identify overlaps between elements of different models. A total overlap in this approach is assumed when two model elements are "tagged" with the same item in the ontology (Leite & Freeman 1991, Robinson 1994, Robinson & Fickas 1994, Boehm & In 1996). (iii) Similar analysis, which is an “approach that exploits the fact that modelling languages incorporate constructs which imply or strongly suggest the existence of overlap relations. For instance, the "Is-a" relation in various object-oriented modelling languages is a statement of either an inclusive overlap or a total overlap” (Spanoudakis & Zisman 2001). Due to the fact that the aforepresented algorithm looks for overlapping in each PLM, the more inexpensive way to detect them is by unification of artefacts’ names.

### CC.4. Dependency completeness

This conformance checking criterion is highlighted in Figure 5.5, over the generic PLM metamodel of Figure 4.2. This conformance criterion makes reference to the fact that each dependency relates two or several artefacts.

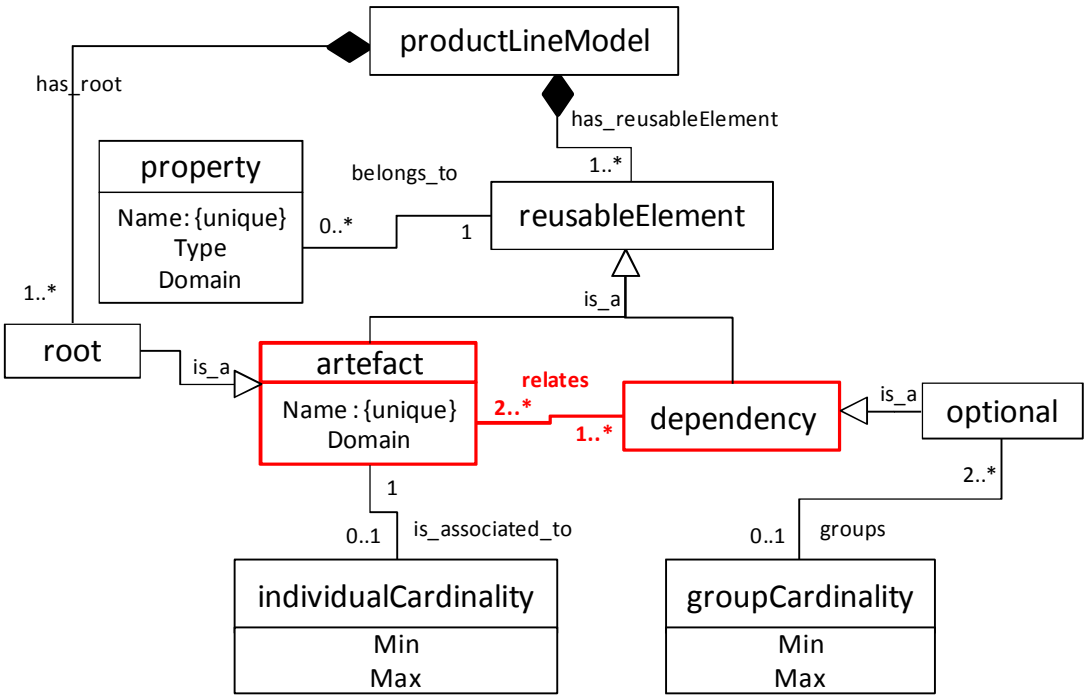



**Figure 5.5.** CC.4 highlighted in the generic PLM metamodel.

Algorithm: The algorithm corresponding to this conformance criterion looks for a dependency (i.e., optional or mandatory or requires or excludes) between two identical artefacts. If such a situation is met, the artefact, its name and the PLM in which the defect was found are returned. The algorithm to check this non-conform situation is presented as follows:

```
For each productLineModel M
{
   For (each artefact A)
   {
      If (there is a dependency D between A and A)
      {
            Return (A, A.Name, M);
      }
   }
}
```

This algorithm searches for wrong dependencies, i.e. dependencies that do not respect this conformance criterion and therefore relate the same artefact. This algorithm evaluates if there are dependencies among the same artefact. If some of these dependencies are found, the elements intervening in the non-conformance are returned.

### CC.5. Properties belonging to a single reusable element

This conformance checking criterion is highlighted in Figure 5.6, over the generic PLM metamodel of Figure 4.2 This criterion refers to the fact that a property belongs to one and only one reusable element.

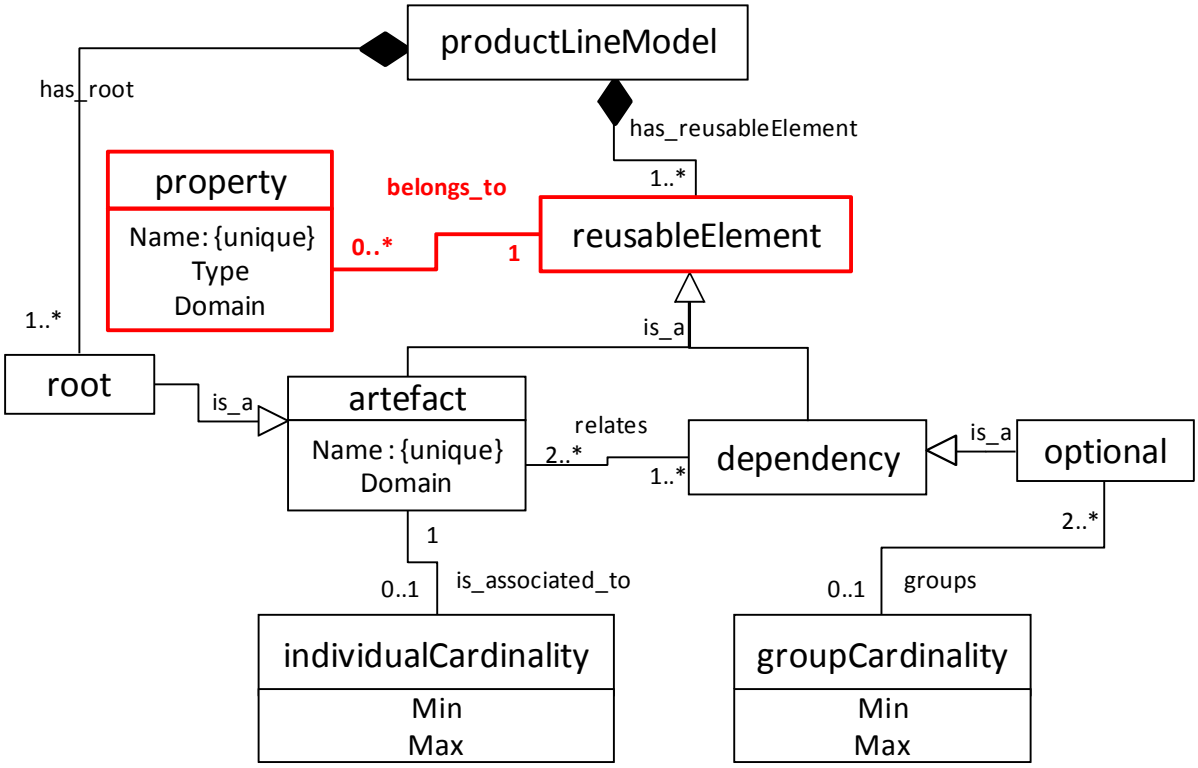



**Figure 5.6.** CC.5 highlighted in the generic PLM metamodel.

Algorithm: The algorithm that implements this conformance criterion looks for property elements in the PLM. For each property, the algorithm puts into a list the reusable elements to which the property is related. Once all the property elements of the PLM are evaluated, the algorithm computes the number of elements in the list. If this number is different to 1, a non-conformant situation regarding this conformance criterion is identified. For each non-conformance detected in the model at hand, the list of reusable elements implied in the defect is returned.

```
For each productLineModel M
{
   ReusableElementsList ='';
   For each property P
   {
      reusableElement RE;
      If (belongs_to(RE, P))
      {
            ReusableElementsList += RE;
      }
   }
   N = length(ReusableElementsList);
   If (N ≠ 1)
   {
      return ReusableElementsList;
   }
}
```

For each product line model, this algorithm looks for properties instead of reusable elements. This decision avoids the evaluation of reusable elements without artefacts, which improves significantly the performance of the algorithm in product line models where only few reusable elements have properties. If the model contains many properties, the performance of the algorithm is in the worst of the cases, similar to the equivalent algorithm that navigates through the reusable elements and of each one of them searches if there are properties that belongs at the same time to other reusable element.

### CC.6. Non-overlap of the properties that belong to the same reusable element

This conformance checking criterion is highlighted in Figure 5.7, over the generic PLM metamodel of Figure 4.2. This criterion specifies that properties of the same reusable elements cannot have the same name.

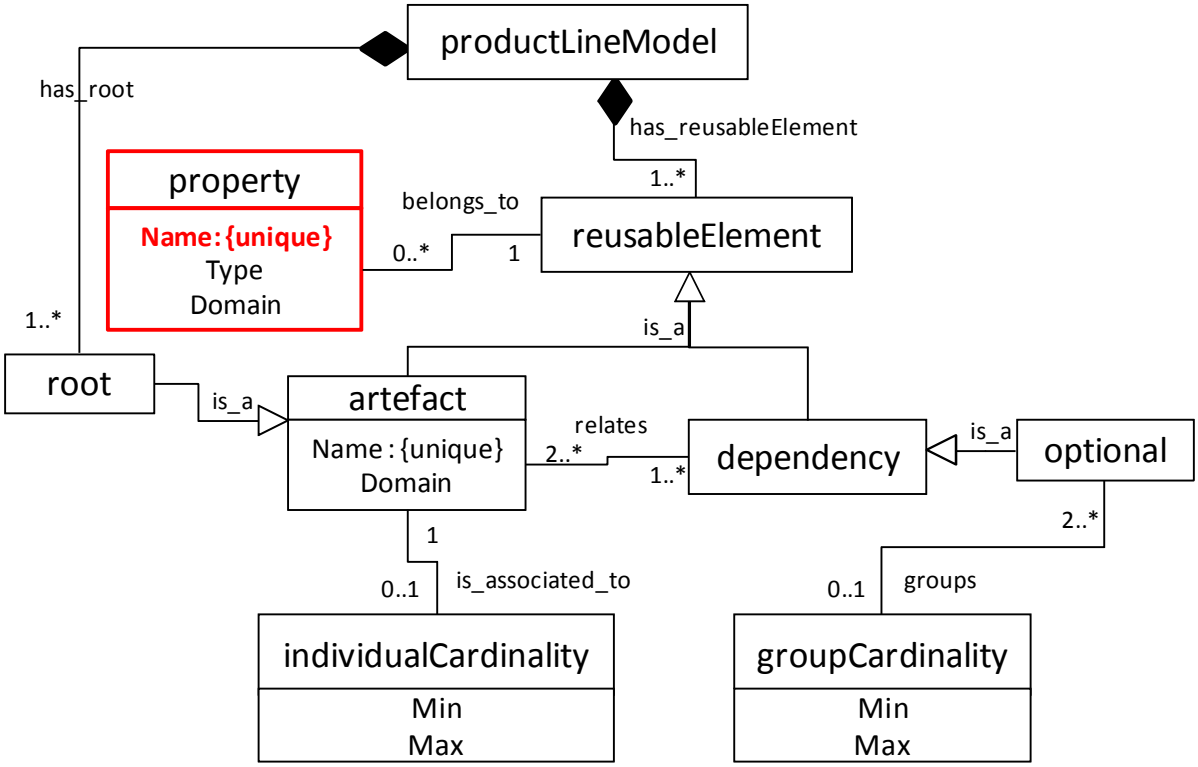



**Figure 5.7.** CC.6 highlighted in the generic PLM metamodel.

Algorithm: The algorithm looks for artefacts with properties. Once an artefact with properties is found, the algorithm looks for two different properties that share the same name. Each time that a couple of different properties with the same name is found, the artefact to which they belong to and the couple of properties are returned to the user. So on, the algorithm recursively evaluates the other artefacts of the model, as follows.

```
For each productLineModel M
{
```

```
For (each reusableElement E)
{
   If (E has properties LProp)
   {
         For (each {Prop1, Prop2} of LProp)
         {
               If ((Prop1 ≠ Prop2) and (Prop1.Name = Prop2.Name))
               {
                     Return (Prop1, Prop2, E, M);
               }
         }
   }
}
```

This algorithm can also be implemented by means of the overlap detection techniques discussed in the conformance checking rule 3 (CC.3) and an extra constraint to guarantee that the two elements under comparison are effectively different one each other. The computational complexity of the implementation will depend on the technique used to find the overlaps.

### CC.7. Association of individual cardinalities to artefacts

This conformance checking criterion is highlighted in Figure 5.8, over the generic PLM metamodel of Figure 4.2. This criterion specifies that an individual cardinality should be associated to one and only one artefact.

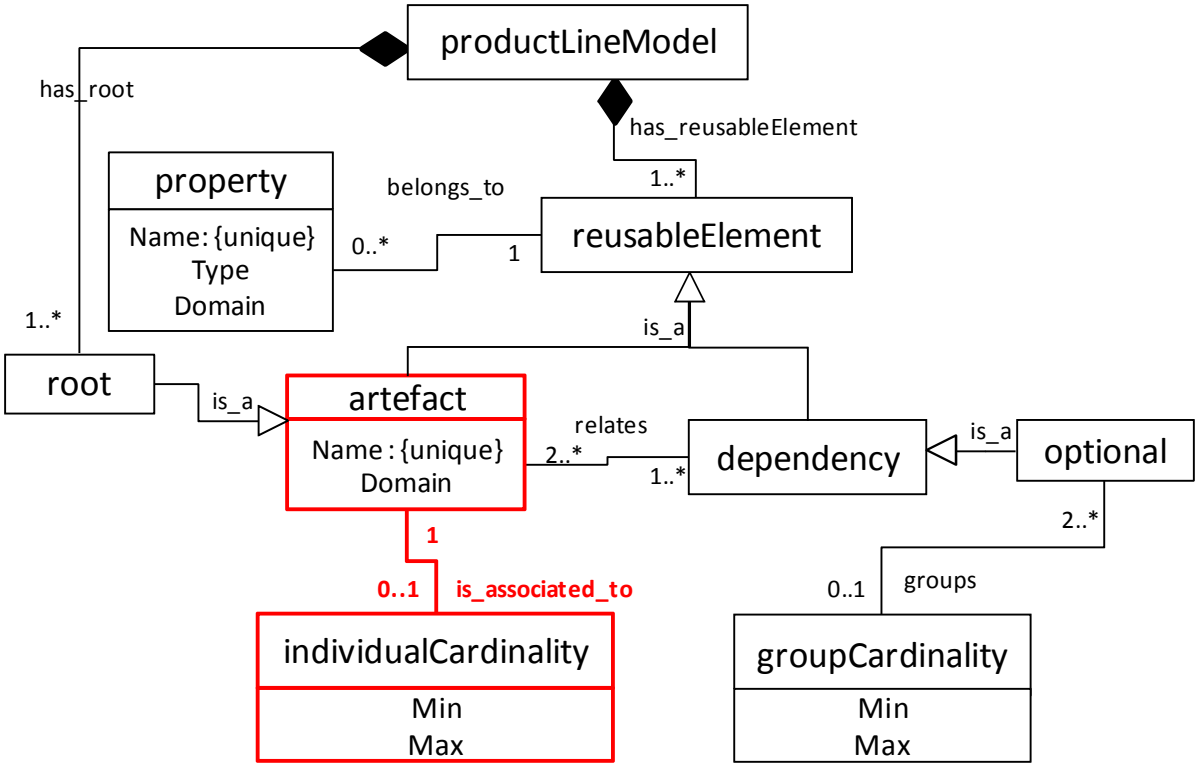



**Figure 5.8.** CC.7 highlighted in the generic PLM metamodel.

Algorithm: The algorithm looks for individual cardinalities in each PLM. For each one of the individual cardinalities, the algorithm puts into a list the artefacts to which the individual cardinality is related. Once all the individual cardinalities of the PLM are evaluated, the algorithm computes the number of elements in the list. If this number is different to 1, one conformance violation regarding this conformance criterion over the model at hand is identified and the list of artefacts is returned to the user. The corresponding algorithm of this conformance criterion is presented as follows.

```
For each productLineModel M
{
   ArtefactList ='';
   For each indivualCardinality I
   {
      If (I is associated to an artefact A)
      {
            ArtefactList += A;
      }
   }
   N = length(ArtefactList);
   If (N ≠ 1)
   {
      return (ArtefactList, M);
   }
}
```

This algorithm searches at first for the individual cardinalities and then, it evaluates if the individual cardinality at hand is associated to more than one artefact. It is worth noting that the algorithm avoids useless evaluations (i.e., evaluation of artefacts without individual cardinalities).

### CC.8. Optional dependencies in group cardinalities

This conformance checking criterion is highlighted in Figure 5.9, over the generic PLM metamodel of Figure 4.2. This criterion specifies that a group cardinality should group two or more optional dependencies.

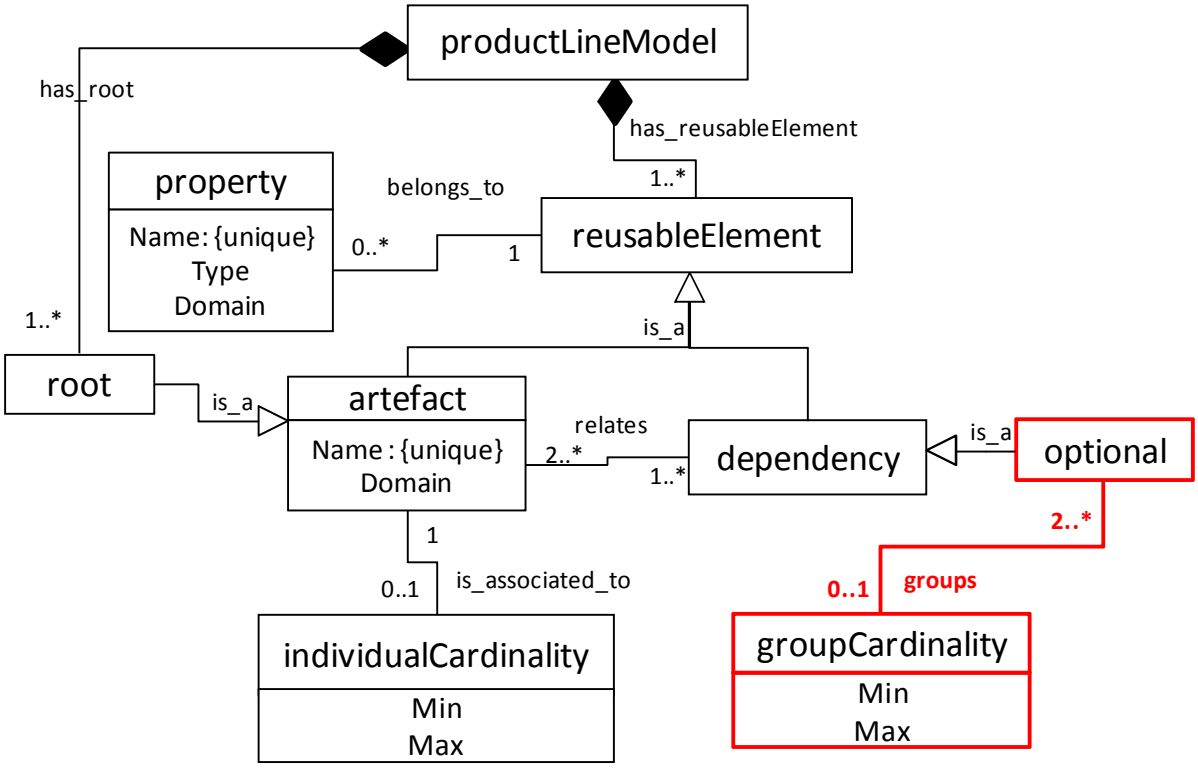



**Figure 5.9.** CC.8 highlighted in the generic PLM metamodel.

Algorithm: The algorithm looks for group cardinalities with only one dependency. This algorithm navigates through the group cardinalities of the PLM and for each one of them, evaluates the number of dependencies belonging to the group cardinality. If this number is equal to one, the algorithm returns the group cardinality and the model to which it belongs to, as follows.

```
For each productLineModel M
{
   For each groupCardinality GC
   {
      If (GC only contains one dependency)
      {
            return (GC, M);
      }
    }
}
```

This algorithm searches for the group cardinalities instead of optional dependencies, based on the fact that in product line models there are fewer group cardinalities than optional dependencies. This decision improves the execution time of the implementation avoiding useless evaluations (evaluation of optional dependencies not associated in a group cardinality).

## 5.2 The Case of Feature Models

This section presents the application of the aforementioned generic conformance checking criteria, in the case of FMs and the implementation of these criteria according to the FM metamodel depicted in Figure 3.4.

The purpose of this section is to show how a collection of conformance criteria (among which several were adapted from the generic criteria, and others are specific as the deltas of the FM metamodel with reference to the generic PL metamodel) can be extracted from the FM metamodel and checked automatically. In this manner, one can extend the conformance checking criteria according to particular requirements and depending on the metamodel at hand. The implementation of each verification criterion is a verification rule (cf. Definition 1.6). Rules implementing FM conformance criteria can be seen as a queries that will be executed over a FM represented as a CLP (cf. Chapter 3). If the rule is evaluated to true in a model, its output is a set of elements that make true the evaluation of the rule. Note that in each algorithm there are only instantiated the elements that need to be analyzed to evaluate the corresponding rule, and each time that a case where the conformance rule is evaluated true, the elements participating in the case are signalled to the user. Therefore, the approach proposed in this thesis identifies not just the presence of conformance violations but also theirs sources.

The following concepts of the FM metamodel are specializations of the concepts presented in the generic metamodel (cf. Figure 4.2):

Product Line Model → Feature Model,
artefact reusableElement → feature,
dependency reusableElement → dependency,
property → attribute,
root artefact → root feature,
individualCardinality → featureCardinality,
optional dependency → optional dependency, and
groupCardinality → groupCardinality.

All the other concepts of the FM metamodel are specific to the FM formalism.

**FM CC. Criterion 1. A feature model should have one and only one root**

As defined by Kang *et al.* (1990), Griss *et al.* (1998), Matthias *et al.* (2002) and Czarnecki *et al.* (2005), we consider that a FM should have only one root feature.

The application of this criterion has for consequence that when someone makes several feature models for the different aspects of a same PL, the collection of feature models is integrated with a single root representing the different aspects of the PL. In Figure 5.10, this criterion is highlighted over the FM metamodel presented in Figure 3.4. Figure 5.10 depicts the fact that a FM is composed of one and only one root feature and that the root is a kind of feature. Thus, it is possible to find the elements that do not respect this conformance criterion.

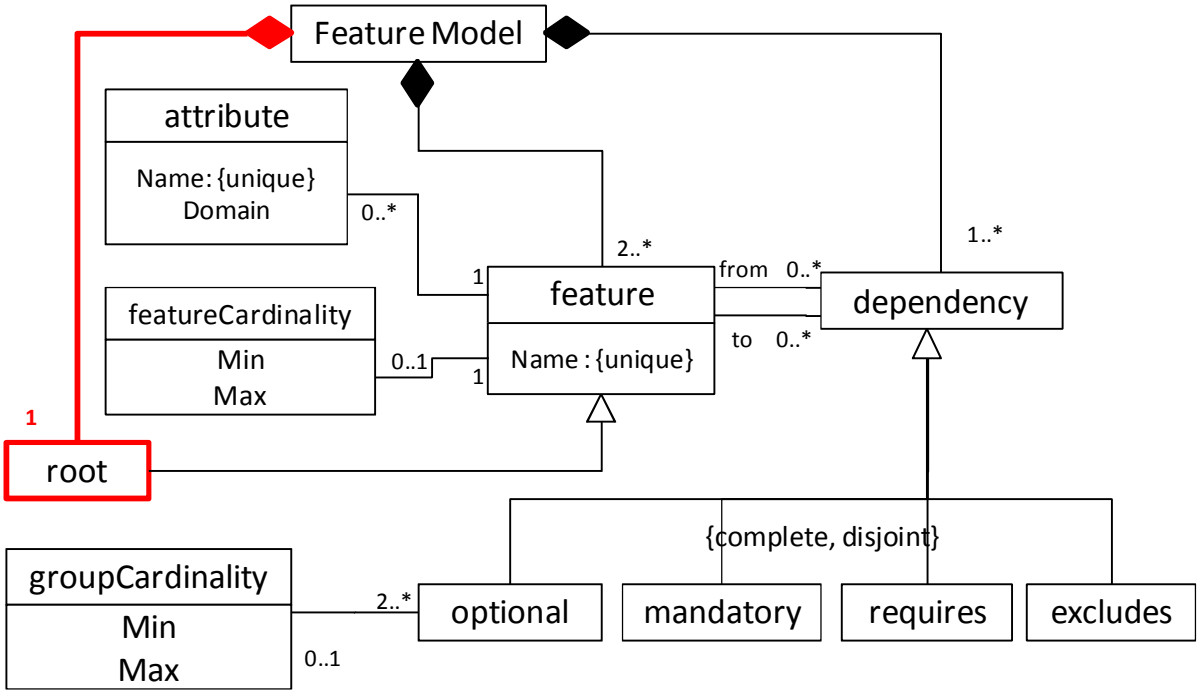



**Figure 5.10.** FM CC criterion 1 on the FM metamodel

Algorithm: The algorithm that implements this conformance criterion for FMs looks for root features in the FMs; then, each root feature found is kept in a list. Once all the root features of the FM at hand are evaluated, the algorithm computes the number of elements in the list. If this number of different to 1, one defect regarding to this conformance criterion over the model at hand is identified and the list of root features is returned to the user. The algorithm is thus as follows.

```
RootFeatureList ='';
For each root feature RF
{
      RootFeatureList += RF;
}
N = length(RootFeatureList);
If (N ≠ 1)
{
      return RootFeatureList
}
```

Implementation: This algorithm is implemented in GNU Prolog as a constraint logic program query as follows:

```
(1) conformance_rule_1(LRootId) :-
(2)   findall(FeatureId, root(FeatureId), LRootId),
(3)   length(LRootId, N),
(4)   N \== 1.
```

Line 1 uses one output variable to return the list of identifiers of the FM only if the number of elements of the list is different to one. The GNU Prolog built-in predicate `findall(FeatureId, root(FeatureId), LRootId)` returns a list `LRootId` with all values for the identifiers of features `FeatureId` corresponding to the identifier of a root feature `root(FeatureId)`. Then, the GNU Prolog built-in predicate `length(LRootId, N)` calculates the length of the list returned by the predicate of line 2 and gives the result in the variable `N` , which is constrained to be different to `1` (line 4) before return the result `LRootId` in line 1.

**FM CC. Criterion 2. Features intervening in a group cardinality relationship should not be mandatory features**

According to Definition 3.8, a group cardinality dependency is about the selection of a certain number of features among a given set. In this selection, each feature should have the same possibility to be chosen as the others. In this case, all the child features participating in a group cardinality should be optional features. We consider this conformance criterion, initially proposed by Czarnecki *et al.* (2005), as a good practice to avoid errors and redundancies. Conversely, the application of this criterion does not have any negative consequence over the group cardinality specification since it is enough to reduce in one the boundaries of the group cardinality when a mandatory feature is erased from the specification. Figure 5.11 highlights this criterion over the FM metamodel presented in Figure 3.4. Figure 5.11 depicts the fact that two or several optional dependencies can or cannot participate in a group cardinality. Likewise, an optional dependency relates two features, where, by definition (cf. Kang *et al*. 1990), the last one is considered as optional feature.

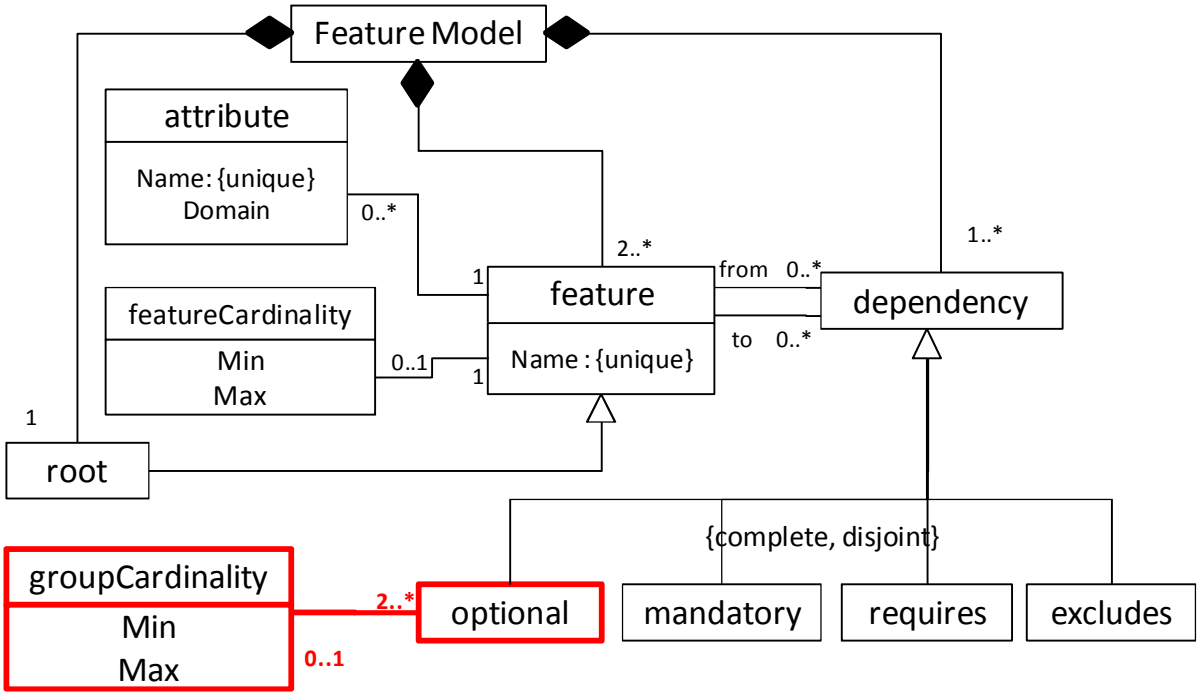



**Figure 5.11.** FM CC Criterion 2 on the FM metamodel

Algorithm: The algorithm that corresponds to this conformance criterion looks for mandatory dependencies participating in group cardinalities. This algorithm instantiates the group cardinalities of the FM and for each one of them, evaluates if the dependency is optional or not. Then, the algorithm returns each mandatory dependency found in the precedent stage, as follows.

```
For each groupCardinality
{
If (groupCardinality contains a mandatory dependency MD)
   {
      Returns MD
   }
}
```

Implementation: This algorithm is implemented as a constraint logic program query as follows:

```
(1) conformance_rule_2(DepId, FeatureId) :-
(2)   groupCardinality(LDepId, _, _),
(3)   member(DepId, LDepId),
(4)   dependency(DepId, _, FeatureId, mandatory).
```

Line 1 uses two output variables to return the identifiers of a dependency and its associated mandatory child feature involved in a group cardinality. The detection of inconsistencies consists of looking for mandatory dependencies (line 4) among the dependencies that belong to a group cardinality (line 2). The built-in predicate of line 2

(`member(DepId, LDepId)`) takes, each time that the predicate is called, the last element (`DepId`) from a list of elements (`LDepId`).

**FM CC. Criterion 3. A feature should not have two attributes with the same name**

The attributes of the product line model should not only be uniquely identified but also they should have different names to avoid redundancies and management issues in a PLM evolution process. In the running example depicted in Figure 3.5, feature `Graphical` has two attributes but they are not violating this conformance criterion because its two attributes have different names (`WidthResolution`, `HeightResolution`). The other way round, two different features may have attributes with the same name without this conformance criterion be violated. Figure 5.12 highlights this conformance criterion over the FM metamodel presented in Figure 3.4. Figure 5.12 depicts the fact that a feature may have zero or several attributes and that each attribute should have a unique name.

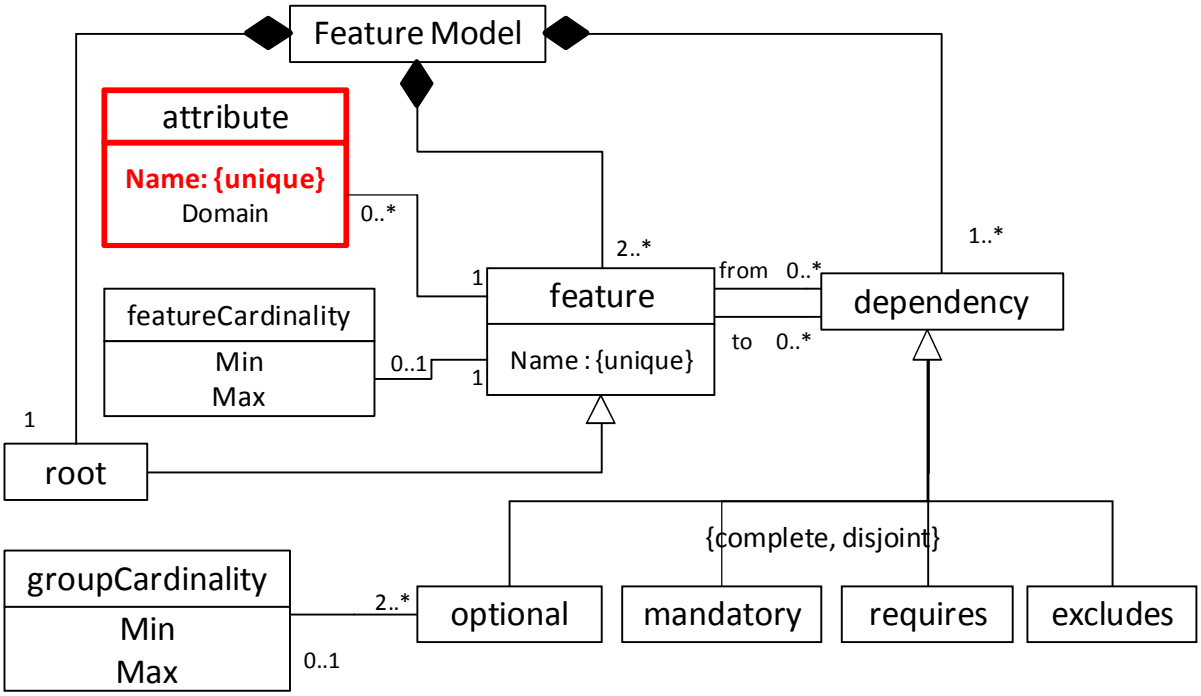



**Figure 5.12.** FM CC Criterion 3 on the FM metamodel

Algorithm: The algorithm that corresponds to this conformance criterion looks for features with attributes. Once a feature with attributes is found, the algorithm looks for two different attributes sharing the same name. Each time a couple of different attributes with the same name is found, the feature to which they belong to and the couple of attributes is returned to the user. Thus, the algorithm iteratively evaluates the other features of the model, as follows.

```
For (each feature F)
{
   If (F has attributes LAtt)
   {
      For (each {Att1, Att2} of LAtt)
      {
            If ((Att1 ≠ Att2) and (Att1.Name = Att2.Name))
            {
                  Return (Att1, Att2, F)
            }
      }
   }
}
```

Implementation: This algorithm is implemented by means of the next constraint logic program query, which searches for two different attributes, of the same feature, with the same name.

```
(1) conformance_rule_3(FeatureName,AttId1,AttId2,AttName) :-
(2)   feature(_, FeatureName, LAttId),
(3)   choose(LAttId, AttId1, LAttId1),
(4)   member(AttId2, LAttId1),
(5)   AttId1 \== AttId2,
(6)   attribute(AttId1, AttName, _),
(7)   attribute(AttId2, AttName, _).
```

Line 1 uses four output variables to return the name of the feature that has the repeated attributes, their two identifiers and the name of the repeated attributes, if any feature where these characteristics exists. These variables will take the values of one feature where two of its attributes have the same name. Usually in CLP other solutions can be obtained thanks to the underlying non-determinism mechanism. The source of non-determinism stands in line 2 that chooses one feature, line 3 that chooses an attribute associated with the feature at hand and line 4 that chooses a second attribute of the feature. Then, line 5 constraints the fact that both features must be different and lines 6 and 7 constraint the fact that the two attributes must have the same name. It is worth noting the declarative formulation of this conformance rule and the fact that we only use relevant elements for the conformance criterion. In this rule we are interested in comparing attributes of a same feature. Therefore, we only consider features with a list of attributes (`LAttId`) and do not use dependencies or cardinalities because they are not relevant for this criterion. The research strategy we use in each conformance rule is exhaustive because we do not avoid evaluating any case even if in our research we only consider elements relevant to the scope of each conformance criterion.

**FM CC. Criterion 4. Two features should not have the same name, in the same model**

The fact that several features share the same name can generate ambiguity problems in product configuration and maintenance stages. In none of the models of our running example (cf. Figure 3.5 and 3.6) have the same names been used for different features in the same view. There are some features with the same name (e.g., `Shell`, `UserInterface` and `Graphical`) but these are in different models. This criterion does not apply in this case as it only considers one model at time as specified with the "Feature Model" entity. Figure 5.13 highlights this conformance criterion in the FM metamodel.

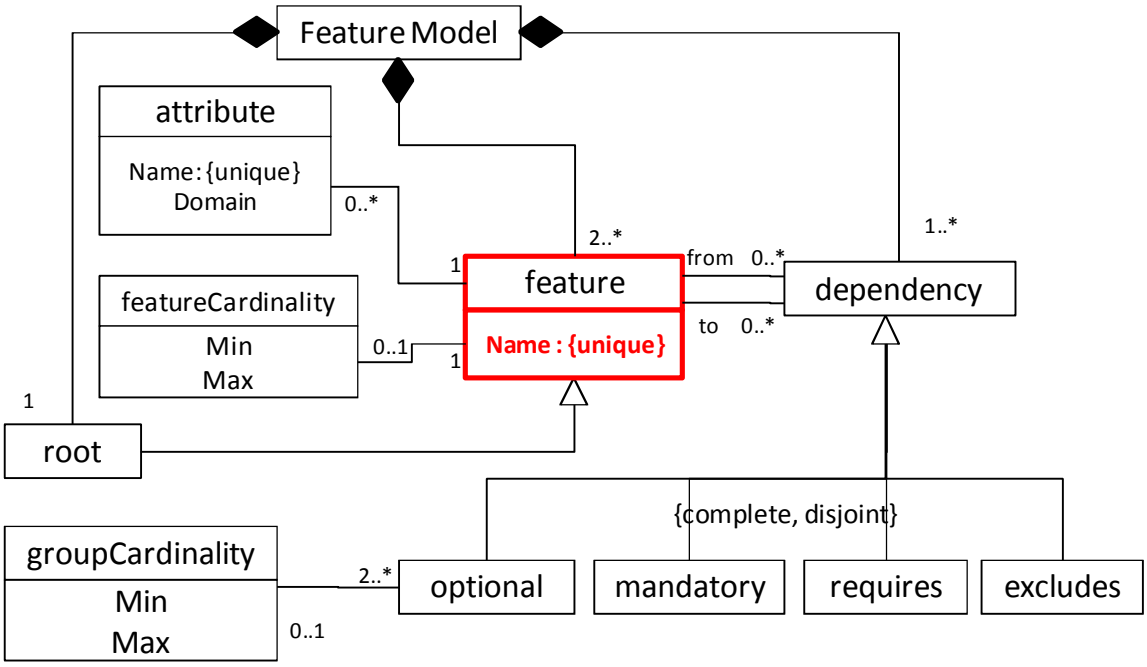



**Figure 5.13.** FM CC Criterion 4 on the FM metamodel

Algorithm: The algorithm that corresponds to this conformance criterion looks for features that although different from each other, share the same name. Once these features are identified, the algorithm returns the identifiers of both features and the corresponding name to the user. The algorithm is as follows:

```
For (each feature F1)
{
      If (there is another feature F2 and F1≠F2 and F1.Name=F2.Name)
      {
            Return (F1, F2, Name)
      }
}
```

Implementation: This algorithm is implemented as follows.

```
(1) conformance_rule_4(FeatureId1, FeatureId2, FeatureName) :-
(2)   findall(FName-FId, feature(FId, FName,_), LNameId),
```

```
(3)   keysort(LNameId ,LNameId1),
(4)   append(_,[FeatureName-FeatureId1, FeatureName-FeatureId2|_],
            LNameId1).
```

Line 1 uses three output variables to return the identifiers and the name of the pairs of different features that have the same name. The GNU Prolog built-in predicate `findall(FName-FId, feature(FId, FName,_), LNameId)` returns a list `LNameId` with all values for the pair of values `FName-FId` corresponding to the name and identifier of a same feature; i.e., that satisfied the predicate `feature(FId, FName,_)`. Then, the GNU Prolog built-in predicate `keysort` sorts the list `LNameId` (and puts the result in `LNameId1`) according to the names of the features in the list. This way, if there are two features with the same name, they are together in the list. Line 4 uses the GNU Prolog built-in predicate `append` to search for two consecutive elements with the same name under the list `LNameId1` obtained from the line 3.

**FM CC. Criterion 5. A child feature cannot be related in an optional and a mandatory dependency at the same time**

When a feature is involved as a child in an optional/mandatory relationship, this feature is also referred to as optional/mandatory. By definition 3.5, a feature that is optional cannot be mandatory at the same time and vice versa. In the metamodel, optional and mandatory are complete and disjoint dependencies. This conformance criterion covers two cases. In the first case, it evaluates if a feature is constrained two times by the same father by means of an optional dependency and a mandatory dependency. In the second case, it evaluates if a feature is mandatory towards one parent and optional towards another one, directly or indirectly (through other features). Figure 5.14 highlights this conformance criterion in the FM metamodel.

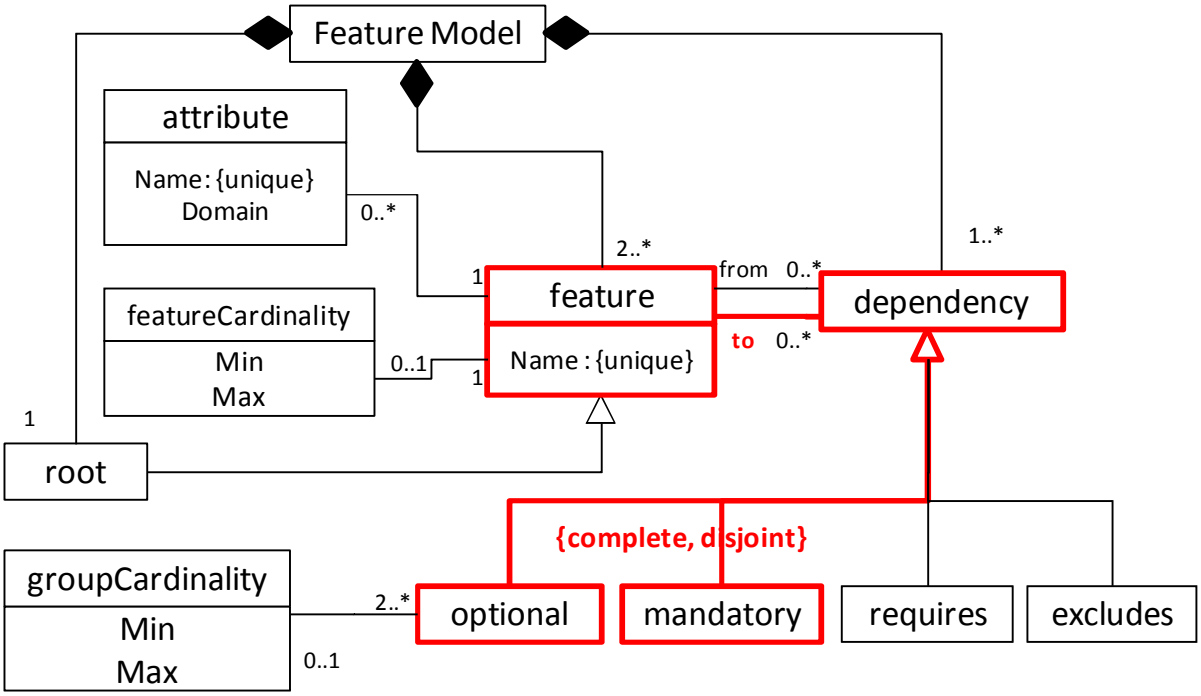



**Figure 5.14.** FM CC criterion 5 on the FM metamodel

Algorithm: The algorithm corresponding to this conformance criterion takes the collection of mandatory dependencies between features `F1` (father) and `F2` (child) and then looks for an optional dependency in which `F2` is again the child feature. If such cases are identified in the FM, the dependencies and the child feature are returned to the user. The corresponding algorithm is as follows:

```
   For (each mandatory dependency D1 between features F1 and F2, in this
order)
   {
         If (there is an optional dependency D2 between features F3 and
F2, in this order)
         {
               Return (D1, D2, F2)
         }
   }
```

Implementation: This algorithm is implemented as a constraint logic program query. This program looks for features constrained at the same time by an optional and a mandatory dependency.

```
(1) conformance_rule_5(FeatureId, DepId1, DepId2) :-
(2)    bagof(TypeId-DepId, FId0^dependency(DepId, FId0, FeatureId,
            TypeId), L),
(3)    % member(mandatory-DepId1, L), member(optional-DepId2,L).
(4)    once((member(mandatory-DepId1, L), member(optional-DepId2, L))).
```

Line 1 uses three output variables to return the identifiers of the feature and its associated dependencies that violate the conformance criterion. Then, the GNU Prolog built-in predicate

`bagof` groups by the variable `FId0` (initial `FeatureId`) all the dependencies in which a given feature is involved. For each `FeatureId` (by backtracking) this predicate returns a list `L` consisting on elements of the form `TypeId-DepId` associated with each particular `FeatureId`. Once this list is ready, the GNU Prolog built-in predicate `once` searches both a mandatory and an optional dependency in each member of `L`. Thus, if this predicate finds some results, then this means that a particular feature is involved in a mandatory and an optional relationship at the same time. By means of the backtracking mechanism, another group of `TypeId-DepId`, for other `FeatureId` is tested until the latest feature. Line 3 is commented and can be used if we want all possibilities for a same `FeatureId`.

**FM CC. Criterion 6. Two features cannot be required and mutually excluded at the same time**

If two features are related in requires and excludes relationships, the model is not-conform to the FM metamodel (cf. Figure 3.4). This conformance criterion is applicable in the cases in which features are related directly (i.e., `F1` requires `F2` and `F1` excludes `F2`) and transitively (i.e., `F1` requires `F2`, `F2` requires `F3` and, `F1` excludes `F3`) in mutual exclusion and requirement. Figure 5.15 highlights this conformance criterion in the FM metamodel.

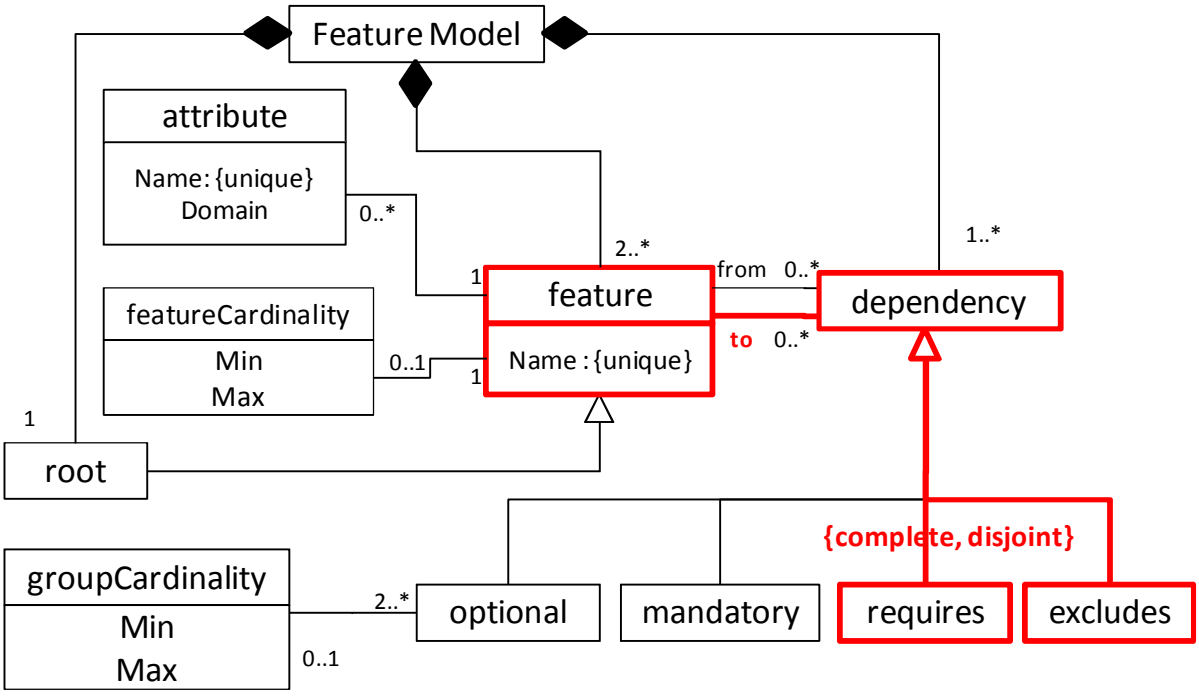



**Figure 5.15.** FM CC criterion 6 on the FM metamodel

Algorithm: The algorithm that corresponds to this conformance criterion takes the collection of requires dependencies between features `F1` (requiring) and `F2` (required) and then looks for

an exclusion dependency between these two features. If such cases are identified in the FM, the features are returned to the user. The corresponding algorithm is as follows:

```
For (each requires dependency D1 between features F1 and F2)
{
      If (there is an excludes dependency D2 between F1 and F2)
      {
            Return (F1, F2)
      }
}
```

It is worth noting that this algorithm does not consider the case in which features are related transitively (i.e., $F_1$ requires $F_2$, $F_2$ requires $F_3$ and, $F_1$ excludes $F_3$) in mutual exclusion and requires due to the fact that this case does not corresponds to a conformance checking criterion but to a semantic criterion. The conformance criteria are directly deduced from the metamodel at hand, and in the case of FM metamodel, requires and mandatory dependencies relate two features. The case of transitivity is treated as a domain-specific criterion (cf. Chapter 6).

Implementation: This algorithm is implemented as a CLP query in the following manner:

```
(1) conformance_rule_6(FeatureName1, FeatureName2) :-
(2)   dependency(_, A, B, requires),
(3)   order2(A, B, A1, B1),
(4)   dependency(_, A1, B1, excludes),
(5)   feature(A, FeatureName1, _),
(6)   feature(B, FeatureName2, _).
(7) order2(A, B, A, B) :-
(8)   A @=< B, !.
(9) order2(A, B, B, A).
```

Line 1 uses two output variables to return the names of features that are required and mutually excluded at the same time in a same model. Line 2 instantiates a requires-type dependency between features `A` and `B`. In requires dependencies there is a predefined order for the identifiers of both features involved in the dependency (i.e., if `A` requires `B`, the identifier of `A` should be placed before the identifier of `B`). On the contrary, in the data structure defined to represent exclusion dependencies there is no predefined order to arrange the identifiers. In line 3, the identifiers of features that will be used to instantiate exclusion dependencies are ordered. This order is necessary to optimize the execution time of the conformance rule. Once the identifiers or both features involved in an exclusion dependency are ordered (lines 7 to 9), line 4 instantiates an exclusion dependency with the order of values `A` and `B` (named `A1` and `B1`), which can also be used in line 2 to instantiate a requires dependency.

**FM CC. Criterion 7. Each dependency relates two or several different features**

According to the FM metamodel shown in Figure 3.4 every dependency should be set between two different features. Figure 5.16 highlights this conformance criterion in the FM metamodel.

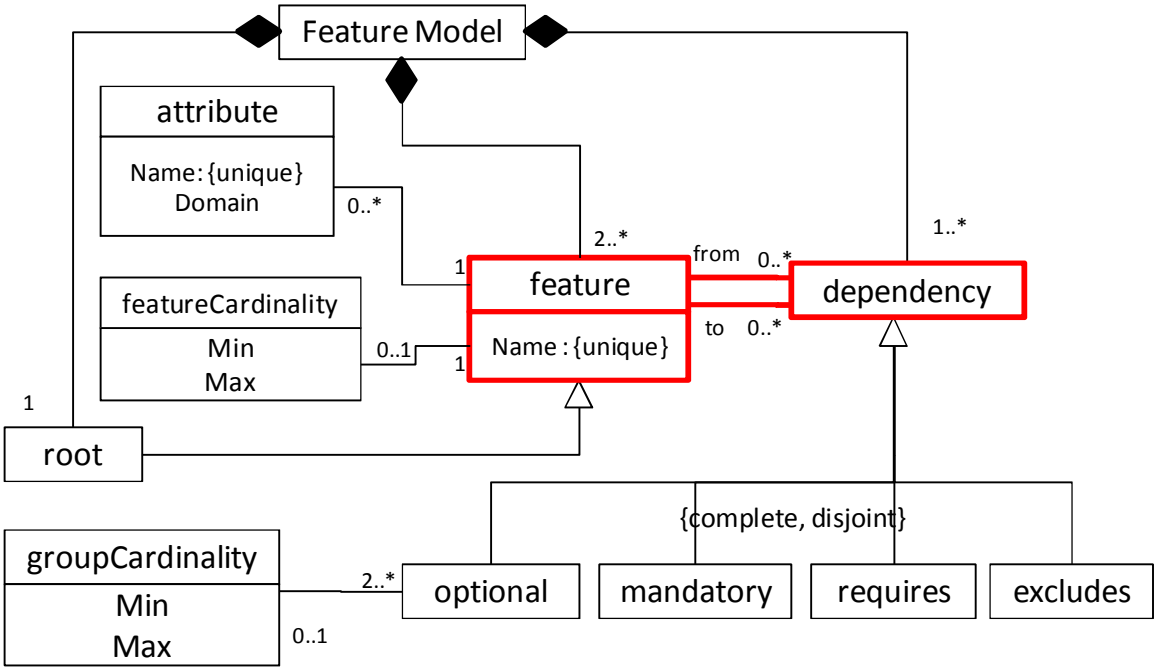



**Figure 5.16.** FM CC criterion 7 on the FM metamodel

Algorithm: The algorithm that implements this FM conformance criterion looks for a dependency (i.e. optional or mandatory or requires or excludes) between two identical features. If such a situation is found, the feature and its name are returned to the user. The algorithm is as follows:

```
For (each feature F)
{
      If (there is an optional or mandatory or requires or excludes
dependency D between F and F)
      {
            Return (F, F.Name)
      }
}
```

Implementation: This algorithm is implemented as a constraint logic program query in the following manner:

```
(1) conformance_rule_7(FeaId, FeaName) :-
(2)    (dependency(_, FeaId, FeaId, optional) ; dependency(_, FeaId,
FeaId,   mandatory)   ;   dependency(_,   FeaId,   FeaId,   excludes)   ;
dependency(_, FeaId, FeaId, includes)),
(3)    feature(FeaId, FeaName, _).
```

Line 1 is the clause that corresponds to this FM conformance criterion and returns to the user the identification and the name of each feature that includes or excludes itself. In line 2

an optional or mandatory or requires or excludes dependency between the feature (`FeaId`) and itself (`FeaId`) is instantiated. At line 3 `FeaId` is used in order to instantiate a feature and save its name in the variable `FeaName`.

## 5.3 Summary

In the case of conformance checking, the purpose is to verify if the abstract syntax of a model is correct with regards to the corresponding metamodel. Once a PLMs' abstract syntax is transformed into constraint logic programming, users can verify them against conformance criteria derived from the corresponding meta-models (Mazo *et al.* 2011b).

Even if the approach presented in this chapter is generic, the algorithms corresponding to each criterion should be adapted to the specific concepts and constraints of the metamodel at hand. Besides, there will be several concepts and constraints in each specific metamodel that are not included in the generic metamodel and therefore new conformance criteria for these concepts and constraints should be defined by the engineer.

To summarize, this chapter proposes a generic conformance checker that uses parameterizable criteria to detect non-conformance on product line models. This generic conformance checking approach can be adapted to specific product line formalisms by (i) means of specialization of the generic criteria, and (ii) implementation of the specific deltas (deltas are concepts of the specific metamodels not considered in the generic metamodel). In addition to this contribution, the generic algorithms can be implemented in different ways, we proposed in this chapter its implementation in constraint programming using the GNU Prolog language (Diaz & Codognet 2001).

# Chapter 6

# Domain-specific Verification of Product Line Models

Looking for undesirable properties in PLMs is not a new subject and several works exist in the literature (Von der Maßen & Lichter 2004, Batory 2005, Benavides *et al*. 2005, Lauenroth & Pohl 2007, Trinidad *et al*. 2008, Mendonça *et al*. 2009). These works, among others, agree with the fact that verification of PLMs consists of "finding undesirable properties, such as redundant or contradictory information" (Trinidad *et al*. 2008). The approach presented in this thesis classifies these undesirable properties in several sub-categories. As the typology presented in Chapter 4 shows, these properties can be grouped in the class of domain-specific verification criteria, under three categories: expressiveness, error-free and redundancy-free.

The *expressiveness* category intends to verify if PLMs are really PLMs. This is probably the most important category due to the fact that a PLM that permits the configuration of no products at all or permits the configuration of only one product is a useless model.

The *error-free* category is about the errors of the PLM, i.e. wrong representations of the product line domain. This thesis identifies three types of errors widely studied in literature: presence of domain values that cannot be attained (Trinidad *et al*. 2008), presence of useless artefact (Von der Maßen & Lichter 2004, Trinidad *et al*. 2008, Van den Broek & Galvão 2009, Benavides *et al*. 2005, 2006, Elfaki *et al*. 2009) and presence of artefacts intended to be optional but appearing in all the products of the PL (Von der Maßen & Lichter 2004, Benavides *et al.* 2005, Trinidad *et al.* 2008). Even if the error criteria are not as crucial as the expressiveness criteria, PLMs should be verified against them in order to identify and correct all the errors before configuring any product. The longer an error goes unnoticed, the more subsequent decisions and product configurations are based on it, and hence the more difficult and expensive it will be to correct it (Boehm 1981).

The *redundant-free* category of verification criteria is about redundancies in PLMs. Redundancies are not errors. However they are undesirable since they increase the computational effort in configuration, analysis and configuration stages, and they undermine the correct evolution of PLMs.

The domain-specific verification criteria considered in this thesis are highlighted in Figure 6.1.

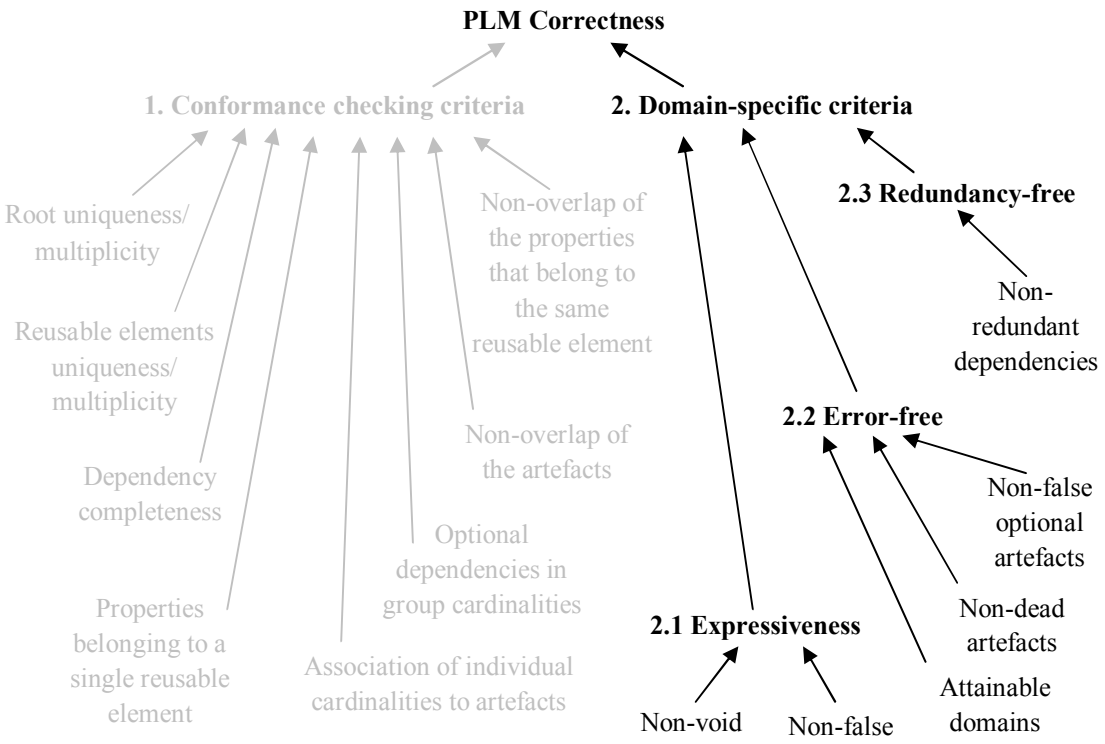



**Figure 6.1.** Domain-specific verification criteria

The rest of this chapter develops the domain-specific verification depicted in Figure 6.1. For each criterion we present a generic algorithm, its application to our running example, its comparison with other algorithms proposed in literature (if some exist) and the implementation of the generic algorithm in java pseudo-code. It is worth noting that these algorithms can also be presented as constraint programs. However, one of the aims of this thesis is to propose optimized algorithms to implement each verification criterion. To this end, we manipulate in java the CP queries and the answers obtained from the solver in order to reuse these answers and avoid useless queries. We use java to implement certain algorithms that are composed of several CP queries arranged in a given order, e.g. the algorithm to check is composed of several CP queries that must be executed according to the results obtained from the solver for each query. It is worth noting that even if the queries are made in constraint programming, the traitement of the results of these queries are made in java in order to reuse these results and avoid future useless queries.

## 6.1 Non-void PLMs

A void PLM is defined as a model that does not permit the configuration of any product.

**Algorithm:**

The proposed approach determines if there is at least one product that can be generated by means of a query to a CP solver. The model is passed to the solver, which is then requested for a solution. If $[V_1, V_2, \ldots, V_i]$ is the vector of variables in the CP that implements the artefacts and properties of the PLM, then a solution is $P1=[V^j_1, V^j_2, \ldots, V^j_i]$ where $V^j_k$ is a possible value of $V_k$ and all the values respects the constraints of the PLM. If such a solution exists, the PLM is not void. The solver will then return one valid product; *false* otherwise.

```
Void_PLM(PLM M, Solver S) {
   Load the model M in the solver S;
   Answer = S.getOneSolution();
   If (Answer ≠ "false") {
         Write (Answer);
   }
   Else {
         Write ("Void PLM");
   }
}
```

**Application to the Running Example:**

The application of this algorithm over the running example gives (within others) the following product that can be configured from the FM of Figure 3.5.

```
P1 = [UNIX=1, Kernel=1, Scheduling=1, ExecutingInstructions=1,
InterpretingInstructions=1, AccomplishingTheTransferOfData=1,
AllocatingTheMachinesMemory=1, Shell=1, FileSystem=1, UserInterface=1,
Graphical=1, Process1=1, Process2=1, Process3=1, Process4=1, Process5=1,
WidthResolution=800, HeightResolution=600, Support_usb=0, Cdrom_atech=1,
Pcmacia_support=2].
```

An example of product that can be configured from the Dopler model of Figure 3.8 is:

```
P2 = [MeansOfInstallation=1, Cdrom=1, Usb=0, Net=0, UtilityProgram=1,
FileMaintenance=1, Editing=1, OnlineInfo=0, ProgrammingSupport=0,
Shell=1, GraphicalResolution=0, Width=0, Height=0, ATwm=0, A4dwm=0,
AMwm=0, AMotif=0, AKDE=0, AQt=0, AGNOME=0].
```

Therefore, both models are not void.

**Discussion:**

Two alternative techniques have been proposed so far to implement this operation: (i) calculating the number of products (Van den Broek & Galvão 2009), and (ii) asking for a product configuration that meets the constraints of a FM (Benavides *et al.* 2005, Trinidad *et al.* 2008). The problem with the former approach is that it is unnecessarily computational costly (if at all possible). Indeed, there is no need to compute all solutions to prove that the model has at least one solution. The proposal of this thesis follows along the lines of the latter alternative as presented above.

**Implementation:**

The Java pseudo-code and the queries to the GNU Prolog solver, corresponding to the above algorithm are presented as follows:

```
(1)   ConnectionProlog connection = new
      ConnectionProlog("localhost",port);
(2)   public String noVoid(String model, String listVariablesOfTheModel)
      {
(3)      String sol = connection.sendMessage("exec("+model+","+
      listVariablesOfTheModel+").");
(4)           if(sol.equals("false")){
(5)                 sol = "Void PLM");
(6)           }
(7)      return sol;
(8)   }
```

Line 2 corresponds to the parameterizable function that automates the non-void verification criterion. Line 3 executes in GNU Prolog the model at hand, and retrieves the solution from the solver in the variable `listVariablesOfTheModel`, which is then returned from the `sendMessage` function and saved into the variable `sol`. Line 7 returns the solutions obtained from the solver, or the string "Void PLM" is there is no products in the PLM at hand (`model`).

## 6.2 Non-false PLMs

A false PLM is defined as a model that permits the configuration of one product only.

**Algorithm:**

The approach proposed in this thesis asks the solver to generate two products in order to decide if the PLM is false. The algorithm of our approach is as follows:

```
False_PLM(PLM M, Solver S) {
   Load the model M in the solver S;
   Answer1 = S.getOneSolution();
   If (Answer1 ≠ "false") {
```

```
            Answer2 = S.getOneSolution();
            If ((Answer1 ≠ Answer2) && (Answer2 ≠ "false")) {
                  Write (Answer1, Answer2);
            }
            Else {
                  Write ("False PLM");
            }
      }
      Else {
            Write ("False PLM");
      }
}
```

**Application to the Running Example:**

The application of this algorithm over the running example of Figure 3.5 gives two products: `P1` already presented in the previous section, and `P3 = = [UNIX=1, Kernel=1, Scheduling=1, ExecutingInstructions=1, InterpretingInstructions=1, AccomplishingTheTransferOfData=1, AllocatingTheMachinesMemory=1, Shell=1, FileSystem=1, UserInterface=0, Graphical=0, Process1=1, Process2=1, Process3=1, Process4=1, Process5=1, WidthResolution=0, HeightResolution=0, Support_usb=0, Cdrom_atech=1, Pcmacia_support=2].`

In the same way, this algorithm applied over the Dopler model of the running example (cf. Figure 3.8) gives two products: `P2` already presented in the previous section and `P4 = [MeansOfInstallation=1, Cdrom=0, Usb=1, Net=0, UtilityProgram=1, FileMaintenance=1, Editing=1, OnlineInfo=1, ProgrammingSupport=0, Shell=1, GraphicalResolution=1, Width=800, Height=600, ATwm=1, A4dwm=0, AMwm=0, AMotif=0, AKDE=1, AQt=1, AGNOME=0].`

**Discussion:**

Although this operation could also help detecting when PLMs are void (our preceding operation), the converse is not true. The two operations have then a separate implementation.

**Implementation:**

The Java pseudo-code and queries to the GNU Prolog solver corresponding to the above algorithm are presented as follows:

```
(1)   ConnectionProlog connection = new
      ConnectionProlog("localhost",port);
(2)   public String FalsePLM(String model, String
      listVariablesOfTheModel) {
(3)      String sol1 = connection.sendMessage("exec("+model+","+
      listVariablesOfTheModel+").");
(4)      String result = "False PLM";
(5)      if(!sol1.equals("false")){
(6)            sol2 = nextSolution();
(7)            if(!sol2.equals("false")){
```

```
(8)                    result = sol1 + sol2;
(9)              }
(10)       }
(11)       return result;
(12)    }
(13)    public String nextSolution(){
(14)       return connection.sendMessage("next.");
(15)    }
```

Lines 2 to 12 represent the function to verify if the product line model `model` is false or not. If the model allows generating two products (lines 3 and 6), both products are returned to the user by means of the variable `result` (lines 8 and 11). If only one product or no products can be generated from the PLM, the functions return "False PLM" (lines 4 and 11). Lines 13 to 15 represent the function to find the next product from the PLM.

## 6.3 Non-dead Artefacts

An artefact is dead if it cannot appear in any product of the product line.

**Algorithm:**

The approach presented in this thesis evaluates each non-zero value of each artefact's domain. If an artefact cannot attain any of its non-zero values, then the reusable element is dead. The corresponding algorithm is presented as follows:

```
DeadArtefacts(PLM M, Solver S) {
   Load the model M in the solver S;
   DeadArtefactsList = all variables of M;
      For (each variable V ∈ DeadArtefactsList) {
         Product = S.getOneSolution("V > 0");
         If (Product = "false") {
               Write ("The artefact " + V + " is  dead");
         }
         Else {
               Erase V and all the other non-zero variables obtained in
      Product from DeadArtefactsList;
         }
   }
}
```

**Application to the Running Example:**

The application of the aforementioned algorithm in our running example is as follows. The initial list of dead artefacts is composed of all the artefacts of the PLM:

```
deadArtefactsList=[UNIX, Kernel, Scheduling, ExecutingInstructions,
InterpretingInstructions, AccomplishingTheTransferOfData,
AllocatingTheMachinesMemory, Shell, FileSystem, UserInterface,
Graphical, Process1, Process2, Process3, Process4, Process5,
WidthResolution, HeightResolution, Support_usb, Cdrom_atech,
Pcmacia_support].
```

Then, the algorithm queries for a configuration based on artefacts for which the algorithm still ignores if they are dead or not, and sieves the selected (and thus alive) elements from this list. For example, to know if `Kernel` is dead or not, it is sufficient to query the solver for a product with `Kernel=1`, which provides a product:

```
P5 = [UNIX=1, Kernel=1, Scheduling=1, ExecutingInstructions=1,
InterpretingInstructions=1, AccomplishingTheTransferOfData=1,
AllocatingTheMachinesMemory=1, Shell=1, FileSystem=1, UserInterface=1,
Graphical=1, Process1=1, Process2=1, Process3=1, Process4=1, Process5=1,
WidthResolution=800, HeightResolution=600, Support_usb=0, Cdrom_atech=1,
Pcmacia_support=2].
```

This not only means that the artefact `Kernel` is not dead, but also that the other artefacts with values different from `0` are not dead. According to the algorithm, these artefacts can be sieved from the list of dead artefacts. Therefore, in the second iteration, the list of dead artefacts is `deadArtefactsList=[Support_usb]`. The next step consists of querying for products with `Support_usb=1`. As answer, the solver provides another product, which means that the `Support_usb` artefact is not dead either. According to our algorithm, the variable `Support_usb` must be erased from the list of dead artefacts. At this point the list of dead artefacts is empty, which means that there are no dead artefacts in the PLM.

The purpose of the list `deadArtefactsList` is to reuse the information gathred from the solver and then reduce the number of future queries. For instance, in this example only two queries were necessary to evaluate all artefacts. In contrast, `21` queries would have been required in the current state of the art algorithm (Broek & Galvão 2009). However, it is not possible to calculate in advance how many queries would be needed, or even, to guarantee that the minimal number of queries will be executed, as this depends on the configuration generated by the solver.

**Discussion:**

Artefacts can be dead because: (i) they are excluded by an element that appears in all products (also known as full-mandatory or core artefacts, c.f. Von der Maßen & Lichter 2004, Benavides *et al.* 2005, Trinidad *et al.* 2008, Van den Broek & Galvão 2009); and (ii) they are wrongly constrained (e.g., an attribute of the feature is > 5 and < 3 at the same time, or a group cardinality is wrong defined).

There are several approaches to detect dead artefacts. Elfaki *et al*. (2009) detect dead features by searching only for predefined cases, i.e. defined dead features in the domain-

engineering process. This approach depends of the feature dialect used in Elfaki *et al*. (2009), therefore this approach is not directly exploitable for other formalisms.

Trinidad *et al*. (2006, 2008) detect dead features by finding all products and then searching for unused features. This approach is not scalable to large models.

Broek & Galvão (2009) detect dead features by transforming the FM into a generalized feature tree, and then searching the feature occurrences that cannot be true. This approach depends on the premise that the product line model should be representable as a generalized feature tree, which has been identified as a problem when Roos-Frantz & Segura (2008) tried to transform OVM models into feature trees without success. To the best of our knowledge there is no detail in literature about the way in which this approach and other ones existing in the literature (e.g. Mendonça *et al*. 2009) were implemented.

The algorithm proposed in this thesis evaluates each non-zero value of each reusable element's domain, and reuses each solution obtained from the solver in order to avoid useless computations. Therefore, our approach is original, and scalable as the paragraph above shows it.

It is worth noting that this algorithm can only be executed once the model at hand is found to be non-void. Otherwise, if the model is void, all the artefacts of the PLM will be dead and therefore every execution of the algorithm to detect dead artefacts will show that the artefact at hand is dead. This observation shows how important it is to respect the order proposed by our typology of verification criteria to identify in which sequence they should be verified.

**Implementation:**

The Java pseudo-code and the queries to the GNU Prolog solver, corresponding to the above algorithm are presented as follows:

```
(1)   findDeadVariables(String[] dataModel, Vector variables){
(2)      deadVariables = vector with all the variables of the PLM;
(3)      while(j<deadVariables.size()){
(4)            String[] value = domains.elementAt(j));
(5)            String wrongValues = new String();
(6)            String sol = new String();
(7)            for(int i=0; i<value.length; i++){
(8)                  if(!value[i].equals("0")){
(9)                  String configuration =
      utilities.makeConfiguration(feature,
      (String)deadFeatures.elementAt(j), value[i]);
(10)                 String prolog = "("+dataModel[0]+"="+configuration;
(11)                 prolog = prolog.concat(", "+dataModel[1]+"),
      "+dataModel[0]);
(12)                 sol = connection.sendMessage("exec("+prolog+").");
(13)                 if(sol.equals("fail.")){
```

```
(14)                        wrongValues = wrongValues + value[i]+", ";
(15)                  }
(16)                  else{
(17)                        break;
(18)                  }
(19)            }
(20)      }
(21)  }
```

Line 1 corresponds to the function to find dead variables in a PLM (`dataModel`). Line 3 evaluates if there remain variables of the PLM, initially classified as dead variables (line 2), in the vector `deadVariables`. For each one of the variables of the PLM, line 4 retrieves all the values that the variable at hand can take, and line 6 saves in `wrongValues` the values that the variable at hand cannot take. Line 8 evaluates each value of the variables' domain, except the value 0. If the domain's value is different to 0, the program creates a configuration with the value at hand, and requests the solver for one solution with this configuration (lines 9 to 13). If the configuration does not generate any solution; the program keeps the value that variables cannot take into the variable `wrongValues` (lines 14 to 17). For each variable of the PLM, the program evaluates if there is at least one solution with each one of the values of the variable's domain in order to determine if the variable is dead or not (lines 7 to 15). Since a solution is found for a given variable, the program breaks the *for* cycle corresponding to the domain's values of the variable at hand in order to avoid useless computations (lines 16 to 18).

## 6.4 Non-false Optional Artefacts

An optional artefact is an artefact playing the role of child in an optional dependency. An artefact is false optional if it is included in all the products of the product line despite being declared optional (Von der Maßen & Lichter 2004, Benavides *et al.* 2005, Trinidad *et al.* 2008).

**Algorithm:**

To verify if an optional artefact is a false optional, this algorithm queries for a product that does not contain the artefact at hand (setting the feature value to `0`). If there is no such product, then the artefact evaluated is indeed a false optional.

```
FalseOptionalReusableElements(PLM M, Solver S) {
   Load the model M in the solver S;
```

```
   FalseOptionalElementsList = all optional elements of M;
      For (each variable V ∈ FalseOptionalElementsList) {
         Product = S.getOneSolution("V = 0");
         If (Product = "false") {
               Write (V + " is  false optional");
         }
         Else {
               Erase V and all the other variables with a Zero
      affectation into Product, from DeadElementsList;
         }
   }
}
```

**Application to the running example:**

If one wants to know whether the optional artefact `Process` is a false optional or not, it is sufficient to request for a product without this artefact (`Process=0`). The solver, in this case, returns "false", which means that this optional artefact always take the value of `1`; i.e., the artefact is false optional, as presented in Figure 3.5. This figure shows that `Process` is included by `Sheduling`, `AccomplishingTheTransferOfData` and `AllocatingMachine'sMemory`, which are part of the core artefacts of the UNIX product line.

**Discussion:** The literature proposes two main approaches to detect false optional artefacts in a PLM. Trinidad *et al*. (2006) detect false optional features in FM based on finding all products, and then searching for common features among those which are supposed to be optional. This technique is not scalable and sometimes even unfeasible due to the fact that it requires to generate all possible products first.

Trinidad *et al*. (2008) present another technique to detect false optional features—they call them *full mandatory features*—that tests the optional dependency instead of the feature itself. Their technique, automated as a constraint satisfaction problem, sets to 0 the optional feature at hand and sets to 1 its father.

The approach presented in this thesis evaluates that there exists one configuration without each presumed optional feature. This approach does not try to check that the father artefact must be set to 1. Indeed if the presumed optional artefact is set to 0 and there is a solution (the model is consistent to this constraint), the father can or cannot take the value 1. Besides, this approach mixes up a structural issue in a semantic verification. In fact, Trinidad *et al*. are evaluating at the same time that the presumed optional feature must be optional and that its father in not a dead feature. This observation demonstrates again the

usefulness of the typology of our verification criteria, in this particular case to separate concerns verification in the algorithm.

**Implementation:**

The Java pseudo-code and the queries to the GNU Prolog solver, corresponding to the above algorithm are presented as follows:

```
(1)   findFalseOptionalArtefacts(String[] dataModel, Vector variables){
(2)      for(int i=0; i< variables.size(); i++){
(3)            String[] valuesCardinality =
      utilities.getCardinality(domains.elementAt(i));
(4)            for(int j=0; j<valuesCardinality.length; j++){
(5)                  if(valuesCardinality[j].equals("0")){
(6)                        String configuration =
      utilities.makeConfiguration(featureAll, (String)
      OptionalElements.elementAt(i), valuesCardinality[j]);
(7)                        String sol =
      connection.sendMessage("exec("+prolog+").");
(8)                        if(sol.equals("fail.")){
(9)                              textFeature = "   The Feature "+
      variables.elementAt(i)+" is a False Optional Variable.";
(10)                       }
(11)                 }
(12)           }
(13)     }
(14)  }
```

Line 1 represents the function to find false optional artefacts on a PLM called `dataModel` among the list of artefacts (called `variables`) of the PLM. In this implementation, artefacts are represented as variables. For each one of the variables on the list `variables` (line 2), the program takes all the values that the variable at hand can take (line 3). For each of the values that is consistent with the variable's domain (line 4), the program verifies if the variable can take the 0 value. If the variable at hand can take the 0 value, one product is requested from the solver with this variable setted to 0 (lines 5 to 7). If there is no solution (i.e., the solver returns *fail)* for this configuration, the variable at hand is false optional (lines 8 and 9).

## 6.5 Attainable Domains

A non-attainable domain value is the value of an artefact, or an artefact's property, that can never appear in any product of the product line.

**Algorithm:**

The algorithm proposed in this thesis to automate this verification criterion evaluates the domain of each artefact and artefact's property of the PLM. For each domain value, the

algorithm requests the solver at hand for a solution. A variable is defined by each artefact and each property. If the solver gives a solution for all the values of the variable's domain, the variable is erased from the list of reusable elements with non-attainable domains. Otherwise, the variable, representing a reusable element, is affected with the non-attainable value(s) and kept in the list of artefacts and properties with non-attainable domains. In each product obtained from the solver, all the artefacts and properties of the PLM are affected with a particular value of the corresponding domain.

Thus, this algorithm reuses the information obtained from the solver and records that information in order to avoid achieving useless requests, i.e., testing the attainability of domain values that have already been obtained in precedent tests. The corresponding algorithm is as follows:

```
NonAttainableDomains(PLM M, Solver S) {
   Load the model M in the solver S;
   For (each variable V ∈ M) {
         For(each Di ∈ domain of V AND not in {PrecedentProducts}){
               Product = S.getOneSolution("V = Di");
               If (Product = "false") {
                     Write ("The domain " + Di + " of " + V + " is non-
               attainable");
               }
               Else {
                     PrecedentProducts += Product;
               }
         }
   }
}
```

**Application to the Running Example:**

For instance in the running example, if when asking for a product with `WidthResolution=800` we get a product:

```
P6 = [UNIX=1, Kernel=1, Scheduling=1, ExecutingInstructions=1,
InterpretingInstructions=1, AccomplishingTheTransferOfData=1,
AllocatingTheMachinesMemory=1, Shell=1, FileSystem=1, UserInterface=1,
Graphical=1, Process1=1, Process2=1, Process3=1, Process4=1, Process5=1,
WidthResolution=800, HeightResolution=600, Support_usb=0, Cdrom_atech=1,
Pcmacia_support=2].
```

This means both that:

- `WidthResolution` can attain the value of `800`, and that
- the rest of variables can attain the values assigned by the solver.

Thus, it is not necessary to ask if the variable `UNIX` can attain those values, e.g. to test if the variable `Pcmacia_support` can take the value of `2`.

**Discussion:** The approach presented in this thesis can assesses the attainability of any artefact or property, for all (or parts of) their domain values. This operation was also implemented by Trinidad *et al*. (2008). However, the approach was specifically restricted to the Boolean domains on FMs, which constitutes a limitation of the approach in terms of reuse it in other notations such as extended FMs. In their approach, Trinidad *et al* try to find a product with each value of the features' domain, i.e, 0 and 1 (true and false).

**Implementation:**

This algorithm is implemented, as follows:

```
(1)   findWrongDomain(String[] dataModel, Vector variables){
(2)      String textVariable = "";
(3)      for(int i=0; i<variables.size(); i++){
(4)            String[] valuesDomain =
      utilities.getDomain(domains.elementAt(i));
(5)            boolean flag = false;
(6)            String wrongValues = new String();
(7)            //evaluate it there is a solution for each domain's value
(8)            for(int j=0; j<valuesDomain.length; j++){
(9)                  String configuration =
      utilities.makeConfiguration(variables,
      (String)variables.elementAt(i), valuesDomain[j]);
(10)                 String prolog = "("+dataModel[0]+"="+configuration;
(11)                 prolog = prolog.concat(", "+dataModel[1]+"),
      "+dataModel[0]);
(12)                 String sol =
      connection.sendMessage("exec("+prolog+").");
(13)                 if(sol.equals("fail.")){
(14)                       wrongValues=wrongValues+valuesDomain[j]+", ";
(15)                       flag=true;
(16)                 }
(17)           }
(18)           if(flag){
(19)                 textVariable = textVariable +
      wrongValues.substring(0, wrongValues.length()-2)+".";
(20)           }
(21)           else{
(22)                 textVariable = "The Variable
      "+features.elementAt(i)+" don't have wrong domain values";
(23)           }
(24)     }
(25)  }
```

Line 1 specifies the function to find the non-attainable domain's values of the variables (`variables`) of a product line model (`dataModel`). The *for* cycle of line 3 retrieves the domain of each variable and evaluates if the variable at hand can attain all the values of its domain. This process creates a configuration with each domain's value (lines 8 and 9) and creating for each value, one configuration to be sent to the solver (lines 10 and 11). If the answer from the solver (line 12) is a *fail*, the variable cannot take the value used in the configuration (lines 13 to 16). Lines 18 to 20 are used to save the values that the variable

at hand cannot take, and lines 21 to 23 are used to inform the user that the variable at hand has no wrong domain values.

## 6.6 Non-redundant Dependencies

A redundant constraint is a constraint that does not reduce the semantics of a PLM.

**Algorithm:**

The approach proposed in this thesis to check non redundant dependencies is based on the fact that if a system is consistent, then the system plus a redundant constraint is consistent too. Therefore, negating the allegedly redundant relation implies contradicting the consistency of the system and thus rendering it inconsistent (Mazo *et al.* 2011a). This approach is more efficient, and thus more scalable, when applied on large models. Our algorithm is in two steps: first, it tries to obtain a solution with the set of constraints; then, if a solution exists, the constraint to check is negated. In the case where no solution is found, the inspected constraint turns out to be redundant. This algorithm to find redundant constraints can be formalized as follows:

```
Non-redundantDependencies(PLM M, Solver S) {
   Load the model M in the solver S;
   If (at least one product can be configured from M under a collection
   of constraints C = {C1,...,Ci})
   {
      Write (C ⊨ M);
      Let take Cr ∈ C a constraint to be evaluated;
      If (C without Cr ⊨ M AND C ∪ ¬Cr ⊭ M)
      {
         Write (Cr is redundant);
      }
      Else
      {
         Write (Cr is not redundant);
      }
   }
}
```

**Application on the running example:**

To check if the constraint `UNIX ≥ UserInterface` is redundant or not, it is sufficient to query the solver for a product. Then, if a product is found, the algorithm proceeds by replacing the constraint by its negation (`UNIX < UserInterface`) and asks again for a product. If the solver does not give a solution (as is the case for the running example presented in Section 3.3), one can infer that the constraint (`UNIX ≥ UserInterface`) is not redundant.

**Discussion:**

The literature offers two alternatives ways to check if a relationship is redundant or not. The *naïve algorithm* consists of calculating all the products of the PLM with the constraint to check; then, remove the constraint; and calculate all the solutions of the new model. If both results are equal (i.e. exactly the same products can be configured with and without the constraint), then the constraint is redundant. This approach is computationally very expensive as it requires (a) to compute all configurations twice and (b) to perform an intersection operation between two potentially very large sets (e.g. $10^{21}$ configurations for the Renault PLM according to Dauron & Astesana (2010)). Not only is this algorithm not scalable, but also it is typically unfeasible.

The *element-oriented approach*, proposed by Yan *et al.* (2009) defines a redundant constraint of a PLM as a constraint in which a redundant reusable element takes part. This approach calculates the redundant reusable elements on feature models —features disconnected from the FM. Then the redundant constraints are those in which the redundant features take part. Though it yields a solution, this algorithm is not sufficiently general: indeed, only trivial cases of redundancy are considered. Many cases of redundant dependencies cannot be discovered using this approach.

**Implementation:**

This algorithm is implemented in Java and using the following code:

```
(1)   findRedudantDependencies(String[] dataModel, Vector dependencies){

(2)      Vector constraints = (Vector) dependencies.elementAt(0);
(3)      Vector negationConstraints = (Vector)
      dependencies.elementAt(1);
(4)      resultProlog =
      connection.sendMessage("exec("+dataModel[1]+","+dataModel[0]+").")
      ;
(5)      if(!resultProlog.equals("fail.")){
(6)            for(int i=0; i < dependencies.size(); i++){
(7)                  String model = utilities.converToString(path,
      (String)constraints.elementAt(i));
(8)                  int begin = model.indexOf("fd_labeling");
(9)                  String message = model.substring(0, begin);
(10)                 message += "\nfd_labeling("+dataModel[0]+")";
(11)                 String messageFinal =  "(("+ message+"),
      "+dataModel[1]+"),"+dataModel[0];
(12)                 resultProlog=
      connection.sendMessage("exec("+messageFinal+").");
(13)                 if(!resultProlog.equals("fail.")){
(14)                       begin = model.indexOf("fd_labeling");
(15)                       message = model.substring(0, begin);
(16)                       message += negationConstraints.elementAt(i)
      +",\n fd_labeling("+dataModel[0]+")";
```

```
(17)                        messageFinal =  "(("+ message+"),
      "+dataModel[1]+"),"+dataModel[0];
(18)                        resultProlog=
      connection.sendMessage("exec("+messageFinal+").");
(19)                        if(!resultProlog.equals("fail.")){
(20)                        VerificationManagerView.txtResultats.append("
         The Relationships "+constraints.elementAt(i)+" is not
      Redundant\n");
(21)                        }
(22)                        else{
(23)                        VerificationManagerView.txtResultats.append("
         The dependencies "+ dependencies.elementAt(i)+" is
      Redundant\n");
(24)                        }
(25)                  }
(26)                  else{
(27)                        VerificationManagerView.txtResultats.append("
         The dependencies "+ dependencies.elementAt(i)+" is not
      Redundant\n");
(28)                  }
(29)              }
(30)      }
(31)      else{
(32)            VerificationManagerView.txtResultats.append("The model is
      inconsistent" + "\n");
(33)      }
(34)  }
```

Line 1 corresponds to the function to find the redundant dependencies from a vector with a collection of dependencies and its corresponding negations (`dependencies`). Line 2 captures the vector with the dependencies to verify and line 3 captures the vector with the dependency negations. The first step of the algorithm consists of verifying if the model is consistent (line 4). If there is at least one solution (line 5), the model is consistent and can be checked for non redundant dependencies.The *for* cycle of line 6 allows to consider each dependency at its turn.

The second step of the algorithm consists of verifying the consistency of the model without the dependency at hand. In order to do that, line 7 creates a model without the constraint at hand, and line 12 executes the solver to check whether the new model has at least one solution. If there is a solution, the new model (the model without the dependency to verify) is consistent (line 13).

The third step of the algorithm consists of verifying the consistency of the model with the negation of the dependency to verify instead of the dependency itself. The new version of the model with the negated constraint is created at lines 15 and 16. Line 18 executes the new model in the solver in order to get one solution. If there is a solution the dependency at hand is not redundant (line 19). If there is no solution, this means that the dependency at hand is redundant (lines 22 and 23).

If the model has at least one solution with the dependency to verify and has no solution without the dependency to verify, this means that the dependency at hand is not redundant (lines 26 and 27).

Lines 31 and 32 deal with the case where the model without modifications has no solutions. Then the model is inconsistent, and therefore identification of redundant dependencies with this technique is not possible.

## 6.7 Summary

Product line modeling is of crucial importance for the quality of product line engineering (Salinesi *et al*. 2010a, Mazo *et al*. 2011c). Thus, it is vital to provide mechanisms to verify that product line models respect certain properties or criteria against with these properties should be verified. This chapter develops one of the two verification categories introduced in Chapter 4. The verification approach developed in this chapter is called *domain-specific verification.* The verification approach is based on constraint programming. This approach represents PLMs as constraints programs and implements the verification criteria presented in this chapter as queries on those models. These verification criteria are grouped in three categories (i.e., expressiveness of the PLM, non-errors and non-redundancies) and are arranged in a typology of PLM verification criteria. The domain-specific verification criteria are: Non-void PLMs, Non-false PLMs, Non-dead Artefacts, Non-false Optional Artefacts, Attainable Domains, Non-redundant Dependencies. This chapter also proposes algorithms to implement each domain-specific verification criterion. A java pseudo code of each algorithm is also presented to show the interactions with the GNU Prolog solver and the reuse of each unswer obtained from the solver to avoid useless queries to the solver. The applicability of each algorithm is shown using the running example. Scalability of our approach is systematically discussed with regards to prior ones.

# Chapter 7

# Verification of Multi-model Product Lines

Multi-model representation of product lines permits tackling various models and aspects of a system, in particular in the presence of stakeholders with multiple viewpoints (executives, developers, distributors, marketing, architects, testers, etc.; cf. Nuseibeh *et al.* 1994). For example, a UNIX product line can be composed of several models, each one developed by a different team or developing a particular view of the PL. Motivated by the fact that (a) this practice is current in industry (Dhungana *et al.* 2010); (b) even if each individual model is consistent, once the models are integrated, they can easily be inconsistent; and (c) the shortcomings of the current state of the art in multi-model product line verification, this thesis proposes a method to verify multi-model product lines. This method uses the transformation and integration approach presented in Chapter 3. Once models are integrated, the collection of generic conformance checking and domain-specific verification criteria proposed in this thesis for standalone models can be applied on the integrated model. This verification approach can be applied on the integrated models in the same manner as for standalone models. It is worth noting that to apply the conformance checking approach proposed in this thesis, the resulting model should previously have a well defined metamodel.

This method applies in two different cases: when the integration mechanisms are not defined in the metamodel and when the metamodel specifies the mechanism in which the model should be integrated. Both cases are illustrated in the following sections: (a) by integration of two feature models, and (b) by integration of the two views of a Dopler model.

## 7.1 Verification of Integrated Feature Models

Integrating two models that are individually without defects can generate a model with several defects. For instance, if in a feature model *FM1*, feature `C` is an optional child of feature `A`, and in another feature model *FM2*, feature `C` is an optional child of feature `B`, the resulting FM will have two fathers of C (`A` and `B`). Even worse, if in *FM1* feature `B` is an optional child of feature `A`, and in *FM2* feature `A` excludes feature `B`, the resulting FM will be a void model.

To develop our integration approach we will consider two input FMs called *Base Model 1* and *Base Model 2*, which, after integration, will produce a resulting model called *Result*.

### Strategy N° 1: restrictive and keeping common features and attributes

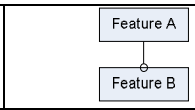
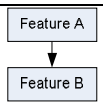
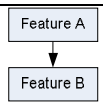
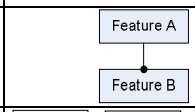
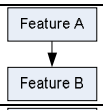
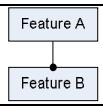
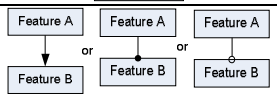
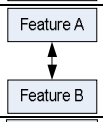
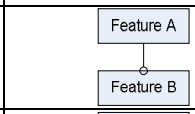
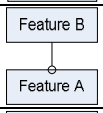
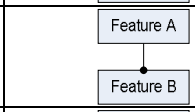
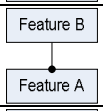
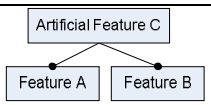
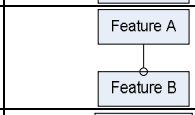
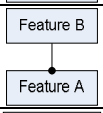
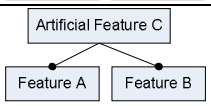
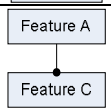
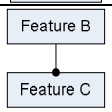
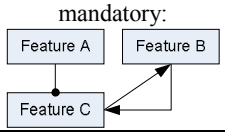
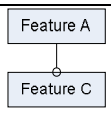
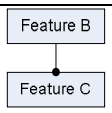
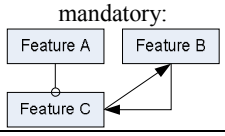
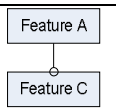
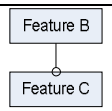
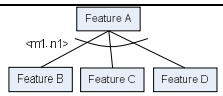
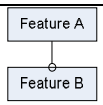
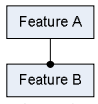
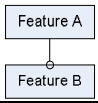
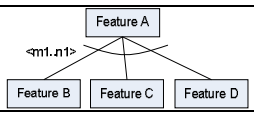
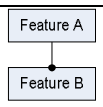
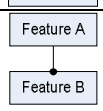
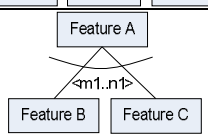
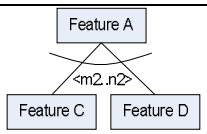
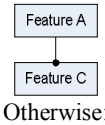
This strategy is restrictive in the sense that it permits representing in the resulting FM the common products represented in both input models that can be configured with the common reusable elements and attributes. In the restrictive strategy we chose the constraints corresponding to the most restrictive dependencies. For example:

| **Base model 1** | **Base model 2** | **Resulting model** |
|---|---|---|
| $A = B$ | $(A \geq B) \Rightarrow C$ | $A = B$ |

We identify two categories to match common features of two FMs. The first category consists of a perfect match of two features (each one bellowing to one input model) in name, quantity and domain of each corresponding pair of attributes. In this case, the resulting feature is exactly a copy of one of the input features. In the second category, this strategy gives priority to the common features and attributes due to the fact that this strategy keeps only common elements. The integration rules that formalize the application of this integration strategy in feature models are presented in Table 7.1. Both categories (perfect and partial match) are considered in the integration rules of Table 7.1.

**Table 7.1.** Integration strategy N° 1. Rules for the restrictive strategy, keeping only common features and attributes

| N | Base1 – Feature representation | Base2 – Feature representation | Result – Feature representation | Strong Result – CP representation |
|---|---|---|---|---|
| 1 | Feature A<br>AttA1 : DomainA11<br>AttA2 : DomainA21 | Feature A<br>AttA2 : DomainA22<br>AttA3 : DomainA32 | Feature A<br>AttA2: DomainA21 ∩ DomainA22 | FeatureA ∈ {0,1} ∧<br>Att2 ∈ {DomainA21}<br>∩{DomainA22} ∧<br>FeatureA ⇔ Att2 > 0 |
| 2 | Feature A<br>or<br>Feature B | Feature A<br>FeatureB ∉ Features(Base2) | Feature A | FeatureA ∈ {0,1} |
| 3 | Feature A<br>or<br>Feature B | Feature A<br>Feature B | Feature A<br>or<br>Feature B<br>respectively | FeatureA = FeatureB<br>Or<br>FeatureA ≥ FeatureB<br>Or<br>FeatureA ⇒ FeatureB<br>Or<br>FeatureA * FeatureB = 0<br>respectively |
| 4 | Feature A<br>Feature B | Feature A<br>Feature B | Feature A<br>Feature B | FeatureA = FeatureB |
| 5 | Feature A<br>Feature B | Feature A<br>Feature B | Feature A<br>Feature B | (FeatureA = FeatureB)<br>∧(FeatureA ≥FeatureB)<br>It is:<br>FeatureA = FeatureB |

| | | | | |
|---|---|---|---|---|
| 6 |  |  |  | (FeatureA ≥ FeatureB)<br>⋀(FeatureA ⇒ FeatureB)<br>It is:<br>FeatureA ⇒FeatureB |
| 7 |  |  |  | (FeatureA = FeatureB)<br>⋀(FeatureA ⇒FeatureB)<br>It is:<br>FeatureA = FeatureB |
| 8 |  |  | Contradiction | Contradiction to be solved by a domain expert |
| 9 |  |  | Mismatch with FMs' syntax | (FeatureA ≥ FeatureB) ⋀<br>(FeatureB ≥ FeatureA)<br>It is:<br>FeatureA = FeatureB |
| 10 |  |  |  | FeatureA = FeatureB |
| 11 |  |  |  | FeatureA = FeatureB |
| 12 |  |  | If features A and B are full-mandatory:<br>  | If features A and B are full-mandatory<br>FeatureA = FeatureC ⋀<br>FeatureB = FeatureC |
| 13 |  |  | If features A and B are full-mandatory:<br>  | If features A and B are full-mandatory<br>(FeatureA ≥ FeatureC)⋀<br>FeatureB = FeatureC |
| 14 |  |  | If there is a path between features A and B, it is necessary to determine the hierarchical order among features A, B and C. Otherwise there is a mismatch with FMs' syntax | If features A and B are full mandatory:<br>(FeatureA ≥ FeatureC) ⋀<br>(FeatureB ≥ FeatureC)<br>If only Feature A is full-mandatory:<br>(FeatureA ≥ FeatureC)<br>If only Feature B:<br>(FeatureB ≥FeatureC)<br>If none is full-mandatory<br>Feature C ∈{0,1} |
| 15 |  |  | If Feature B is mandatory in Base 1:<br>  <br>Otherwise:<br>  | If Feature B is full mandatory in Base 1:<br>FeatyreA = FeatureB<br>Otherwise:<br>FeatureA ≥FeatyreB |
| 16 |  |  |  | FeatyreA = FeatureB |
| 17 |  |  | If Feature C is mandatory in Base 1 or Base 2:<br>  <br>Otherwise: | If Feature C is full mandatory in Base 1:<br>FeatyreA = FeatureC<br>Otherwise:<br>FeatyreA ≥FeatureC |

| | | | | |
|---|---|---|---|---|
| | | | Feature A<br>Feature C | |

This strategy can be used in multi-team development, that is, when the product line is represented with several models, each one complementing the other ones. Besides, this strategy is also useful in the case where two companies on the same market decide to offer together a common portfolio, i.e., a portfolio with products that can be produced at the same time for both companies. However, it is not possible to directly apply this strategy in two FMs (e.g. Figures 3.5 and 3.6). Indeed this strategy keeps only the common features and attributes of both base models. This implies eliminating some core features (e.g., `Kernel`, `FileSystem`, `Scheduling`, `ExecutingInstructions`, `InterpretingInstructions` and `AccomplishingTheTransferOfData`). With this loss, the resulting FM has no sense due to the fact that a Unix system needs one kernel, for instance. In order to avoid that, we propose a version of this strategy that consists of a restrictive strategy keeping only (i) common features and attributes; and (ii) core features and the relationships among them. In order to identify the core features (features that appear in all the products), one can use the corresponding operation—fully automated in the VariaMos tool (Mazo & Salinesi 2010)—to get them. With this modification, the resulting FM, presented in Figure7.1 and represented as a CP in Table 7.2, contains the core features of both base models and also the common features and attributes integrated according to the rules of Table 7.1.

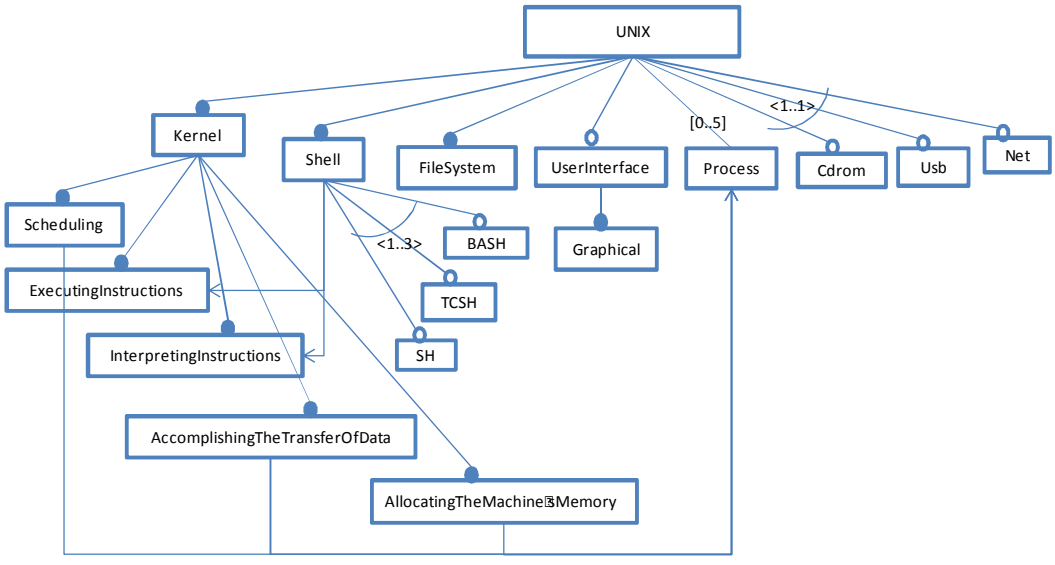



**Figure 7.1.** Feature models of Figure 3.5 and 3.6 integrated by means of the "conservative strategy keeping features and attributes of the original models"

Note that in case of features grouped in a cardinality where, for instance, at least one feature must be selected (i.e., cardinalities <1..1>, <1..2> and <1..3> on Figure 3.6), the intervention of the user is necessary in order to specify the feature(s) that she/he wants to keep in the resulting model. In our case, we kept the two original group cardinalities (Shell ≤ SH + TCSH + BASH ≤ Shell * 3 and UNIX = Cdrom + Usb + Net).

**Table 7.2.** UNIX running example of Figures 3.5 and 3.6 integrated with the strategy N° 1

| | |
|---|---|
| Application of Rule 1 | [UNIX, Kernel, Scheduling, ExecutingInstructions, InterpretingInstructions, AccomplishingTheTransferOfData, AllocatingTheMachinesMemory, Shell, FileSystem, UserInterface, Graphical, Process, Cdrom, Usb, Net, SH, TCSH, BASH] ∈ {0,1} ⋀ |
| 2 | UNIX = Kernel ⋀ |
| 3 | UNIX = Shell ⋀ |
| 2 with user intervention | Shell ≤ SH + TCSH + BASH ≤ Shell * 3 ⋀ |
| No matching with the other model | Kernel1 = AllocatingTheMachinesMemory1 ⋀<br>AllocatingTheMachinesMemory1 ⇒ Process ⋀<br>Kernel1 = Scheduling1 ⋀<br>Scheduling1 ⇒ Process ⋀<br>Kernel1 = AccomplishingTheTransferOfData1 ⋀<br>AccomplishingTheTransferOfData1 ⇒ Process ⋀<br>Shell1 ⇒ InterpretingInstructions1 ⋀<br>Kernel1 = InterpretingInstructions1 ⋀<br>Shell1 ⇒ ExecutingInstructions1 ⋀<br>Kernel1 = ExecutingInstructions1 ⋀ |
| 2 with user intervention | Process ≤ Process1 + Process2 + Process3 + Process4 + Process5 ≤ Process * 5 ⋀<br>Process ≤ UNIX ⋀ |
| 2 | Shell ⇒(Kernel=InterpretingInstructions) ⋀ |
| 2 | Shell ⇒(Kernel = ExecutingInstructions) ⋀ |
| 2 keeping full-mandatory | UNIX = FileSystem ⋀ |
| 3 | UNIX ≥ UserInterface ⋀ |
| 16 | UserInterface = Graphical ⋀ |
| 2 with user intervention | UNIX = Cdrom + Usb + Net |

In this resulting model it is possible to configure products that do not exist in any of the base models (cf. Figures 3.5 and 3.6). For example, the product {UNIX, Kernel, ExecutingInstructions, InterpretingInstructions, Shell, FileSystem, Process, Cdrom, SH} can be configured from the resulting model; however, it does not exist in any of the base models. This is due to the fact that this strategy keeps the full mandatory features of both models and some other features in group cardinalities according to the criteria of the user. Therefore, the resulting model will represents the common features of both input models but also the core features of each input model even if there is not a

corresponding feature in the other input model. The integrated model permits the configuration of 43 different UNIX systems[2] (with five `Processes` at maximum).

### Strategy N° 2: restrictive and keeping all features and attributes

This strategy is also restrictive; however, in contrast to the first strategy, products can be configured with all reusable elements and attributes available on both input models (Acher *et al*. 2010). In this strategy the idea is to keep the most restrictive relationships but keeping the features and attributes presented in both base models. This integration strategy gives to the domain expert the possibility to represent, in an integrated model, the products presented at the same time on both input models and to enrich the expressivity power of the resulting PLM with the reusable elements of both input models. For example:

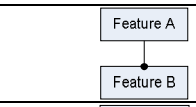
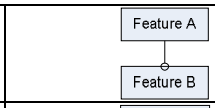
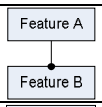
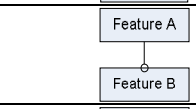
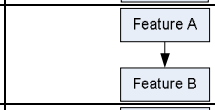
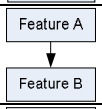
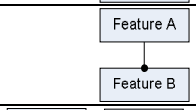
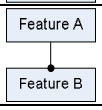
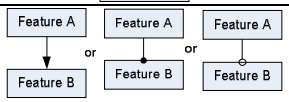
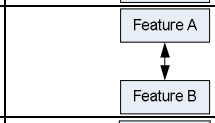
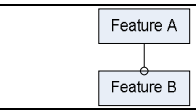
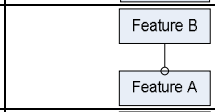
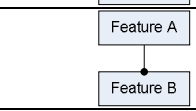
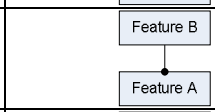
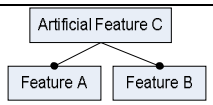
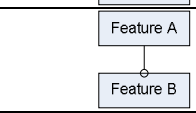
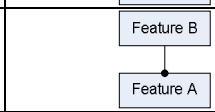
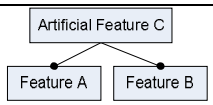
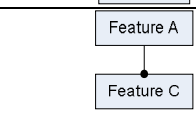
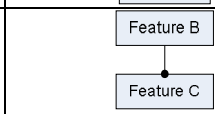
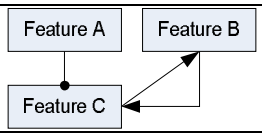
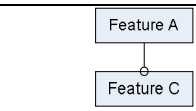
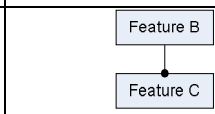
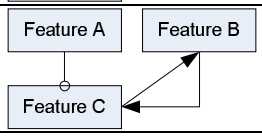
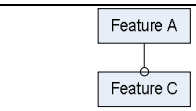
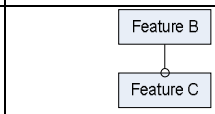
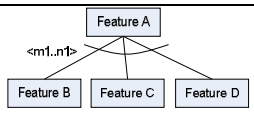
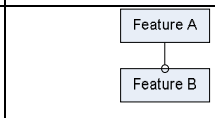
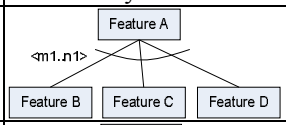
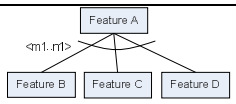
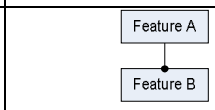
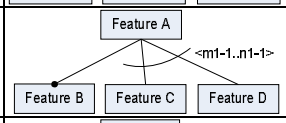
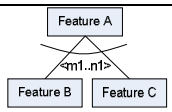
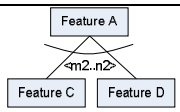
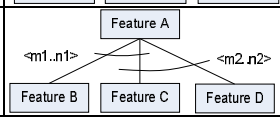
| **Base model 1** | **Base model 2** | **Resulting model** |
|---|---|---|
| $A = B$ | $(A \geq B) \Rightarrow C$ | $(A \geq B) \Rightarrow C$ |

Now we will integrate our running example by using the merging rules proposed in Table 7.3. In this case, we keep features and attributes of the original models and intersect the domains of common attributes and the constraints of common features.

**Table 7.3.** Integration strategy N° 2. Rules for the restrictive strategy, keeping all features and attributes.

| N | Base1 – Feature representation | Base2 – Feature representation | Result – Feature representation | Weak Result – CP representation |
|---|---|---|---|---|
| 1 | Feature A<br>AttA1 : DomainA11<br>AttA2 : DomainA21 | Feature A<br>AttA2 : DomainA22<br>AttA3 : DomainA32 | Feature A<br>AttA1 : DomainA1<br>AttA2: DomainA21 ∩ DomainA22<br>AttA3 : DomainA3 | `FeatureA ∈ {0,1},`<br>`Att1 ∈ {DomainA1},`<br>`Att2 ∈ {DomainA21} ∩{DomainA22},`<br>`Att3 ∈ {DomainA3},`<br>`FeatureA ⇔ Att1> 0,`<br>`FeatureA ⇔ Att2> 0,`<br>`FeatureA ⇔ Att3> 0` |
| 2 | Feature A / Feature B (or) | Feature A<br>`FeatureB ∉ Features(Base2)` | Feature A / Feature B (or)<br>respectively | `FeatureA = FeatureB`<br>`Or`<br>`FeatureA ≥ FeatureB`<br>`Or`<br>`FeatureA ⇒ FeatureB`<br>`Or`<br>`FeatureA * FeatureB = 0`<br>respectively |
| 3 | Feature A / Feature B (or) | Feature A<br>Feature B | Feature A / Feature B (or)<br>respectively | `FeatureA = FeatureB`<br>`Or`<br>`FeatureA ≥ FeatureB`<br>`Or`<br>`FeatureA ⇒ FeatureB`<br>`Or`<br>`FeatureA * FeatureB = 0`<br>respectively |
| 4 | Feature A<br>Feature B | Feature A<br>Feature B | Feature A<br>Feature B | `FeatureA = FeatureB` |

[2] Calculated in GNU Prolog by means of the query: g_assign(cpt,0), productline(_), g_inc(cpt), fail;g_read(cpt,N). Where productline is the fact that represents the product line model loaded into the solver, cpt is a counter variable and N the number of products of the product line at the end of the query. g_assign, g_inc and g_read are predicates of GNU Prolog.

| | | | | |
|---|---|---|---|---|
| 5 |  |  |  | FeatureA = FeatureB |
| 6 |  |  |  | FeatureA ⇒ FeatureB |
| 7 |  |  |  | FeatureA = FeatureB |
| 8 |  |  | Contradiction | Contradiction to be solved by a domain expert |
| 9 |  |  | Mismatch with FMs' syntax | (FeatureA ≥ FeatureB) ∧ (FeatureB ≥ FeatureA)<br>It is:<br>FeatureA = FeatureB |
| 10 |  |  |  | FeatureA = FeatureB |
| 11 |  |  |  | FeatureA = FeatureB |
| 12 |  |  |  | FeatureA = FeatureC ∧ FeatureB = FeatureC |
| 13 |  |  |  | (FeatureA ≥ FeatureC) ∧ FeatureB = FeatureC |
| 14 |  |  | If there is a path between features A and B, it is necessary to determine the hierarchical order among features A, B and C. Otherwise there would be a mismatch with FMs' syntax | (FeatureA ≥ FeatureC) ∧ (FeatureB ≥ FeatureC) |
| 15 |  |  |  | FeatyreB ≤ FeatureA ∧ (m1*FeatureA ≤ FeatyreB + FeatyreC + FeatyreD ≤ n1*FeatureA) |
| 16 |  |  |  | FeatyreA = FeatureB ∧ (m1-1)*FeatureA ≤ FeatyreC+ FeatyreD ≤ (n1-1)*FeatureA |
| 17 |  |  |  | (m1*FeatureA ≤ FeatyreB + FeatyreC ≤ n1*FeatureA) ∧ (m2*FeatureA ≤ FeatyreC + FeatyreD ≤ n2*FeatureA) |

The application of these integration rules in our running example gives as result the LINUX product line model presented as a constraint program in Table 7.4.

**Table 7.4.** UNIX running example of Figures 3.5 and 3.6 integrated with the strategy N° 2

| | |
|---|---|
| Application of Rule 1 | [UNIX, Kernel, Scheduling, ExecutingInstructions, InterpretingInstructions, AccomplishingTheTransferOfData, AllocatingTheMachinesMemory, Shell, FileSystem, UserInterface, |

| | |
|---|---|
| | Graphical, Process1, Process2, Process3, Process4, Process5, Cdrom, Usb, Net, UtilityProgram, FileMaintenance, Editing, OnlineInfo, ProgrammingSupport, SH, TCSH, BASH] ∈ {0, 1} ∧ [WidthResolution] ∈{800, 1024, 1366]) ∧ [HeightResolution] ∈ {600, 768} ∧ [Support_usb, Cdrom_atech, Pcmacia_support] ∈ {0,1,2}∧ [A, B, C] ∈ {0, 1} ∧ |
| 2 | UNIX = Kernel ∧ |
| No matching with the other model | Kernel1 = AllocatingTheMachinesMemory1 ∧ AllocatingTheMachinesMemory1 ⇒ Process ∧ Kernel1 = Scheduling1 ∧ Scheduling1 ⇒ Process ∧ Kernel1 = AccomplishingTheTransferOfData1 ∧ AccomplishingTheTransferOfData1 ⇒ Process ∧ Shell1 ⇒ InterpretingInstructions1 ∧ Kernel1 = InterpretingInstructions1 ∧ Shell1 ⇒ ExecutingInstructions1 ∧ Kernel1 = ExecutingInstructions1 ∧ |
| 2 | Process ≤ Process1 + Process2 + Process3 + Process4 + Process5 ≤ Process * 5 ∧ UNIX ≥ Process ∧ |
| 2 | FileSystem = UNIX ∧ |
| No matching with the other model | Support_usb1 ⇔ A ∧ Cdrom_atech1 ⇔ B ∧ Pcmacia_support1 ⇔ C ∧ 0 ≤ A + B + C ≤ 3 * Kernel1 ∧ |
| Rule 2 with user intervention and Rule 14 | UNIX ≥ UtilityProgram ≥ UserInterface ∧ |
| 16 | UserInterface = Graphical ∧ Graphical = 1 ⇔ (WidthResolution= W1 ∧ HeightResolution= H1) ∧ Graphical = 0 ⇔ (WidthResolution = 0 ∧ HeightResolution = 0) ∧ fd_relation([[800, 600], [1024, 768], [1366, 768]], [W1,H1]) ∧ UserInterface ≥Shell ∧ |
| 2 | Shell ≤ SH + TCSH + BASH ≤ Shell * 3 ∧ |
| 2 | UNIX = Cdrom + Usb + Net ∧ |
| 2 | UtilityProgram = Editing ∧ |
| 2 | UtilityProgram = FileMaintenance ∧ |
| 2 | UtilityProgram ≥ UserInterface ∧ |
| 2 | UtilityProgram ≥ OnlineInfo ∧ |
| 2 | UtilityProgram ≥ ProgrammingSupport |

Note that some auxiliary variables are created, like R1 to record the result of the computation Cdrom + Usb + Net. R1 is used in the constraint UNIX ≤ R1 ∧ R1 ≤ UNIX. In addition, we use the Boolean variables A, B and C in order to keep the state corresponding to the selection of each kernel module. Thus, if the module Support_Usb is selected (set to 1 when it is changed in a static way, and set to 2 when it is charged in a dynamic way) the value of A will be set to 1, if Cdrom_Atech is selected (set to 1 or 2 accordint to its mode of charging) the variable B will be set to 1, and if Pcmcia_Support is selected (set to 1 or 2

accordint to its mode of charging), c will be set to 1. The resulting model still allows configuring 244219 different products, which illustrates that even in a restrictive strategy; the expressivity power of the resulting model significantly increases when all the features and attributes of the input models are kept.

**Strategy N° 3: Conservative and keeping only common features and attributes**

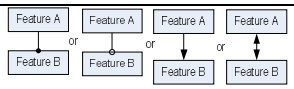
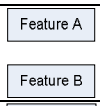
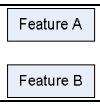
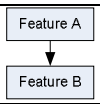
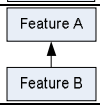
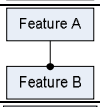
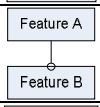
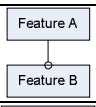
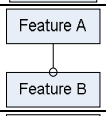
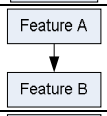
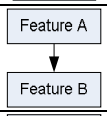
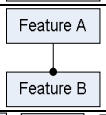
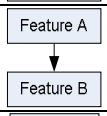
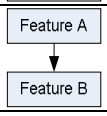
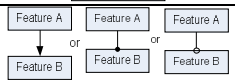
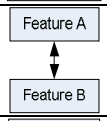
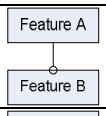
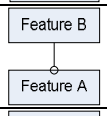
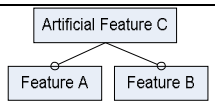
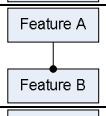
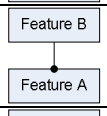
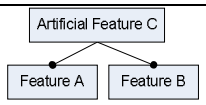
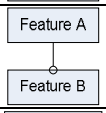
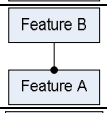
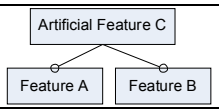
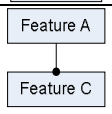
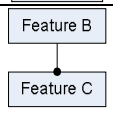
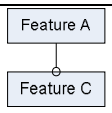
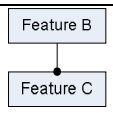
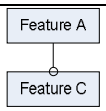
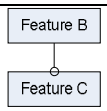
This strategy is conservative in the sense that it permits configuring the products represented in both input models by using only the common reusable elements and attributes. This strategy represents in the resulting model a collection of products that are encompassed by the first base model or by the second model, with the features and attributes that are common to both base models. This integration strategy is for instance useful when a company wants to propose a new series of products that can be produced on two different headquarters and that are offered indistinctly by means of a Web site. A generic example of the application of this strategy can be:

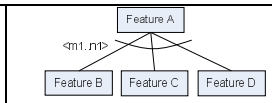
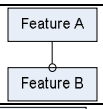
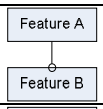
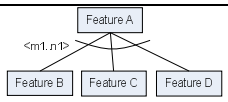
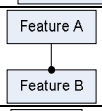
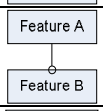
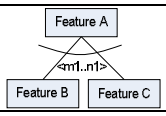
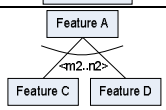
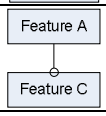
| **Base model 1** | **Base model 2** | **Resulting model** |
|---|---|---|
| $A = B$ | $(A \geq B) \Rightarrow C$ | $A \geq B$ |

It is worth noting that our integration approach with constraint programs avoids using artificial features to represent situations where the structure of the feature representation forces their use (cf. rules 10, 11 and 12 in Table 7.5). However, these artificial features do not contribute in anything from the semantic point of view because the collection of products that can be generated with and without these artificial features is the same. On the contrary, they increase the complexity of the product line and the time to be configured and analyzed. Thus, with the Conservative strategy, keeping common features and attributes, it would be necessary to keep the core features on the resulting model. In addition, the involvement of a modeling expert would be necessary to determine unsolved situations, as on with the group cardinality of Figure 3.6.

**Table 7.5.** Integration strategy N° 3. Rules for the conservative strategy, keeping only common features and attributes.

| | Base1 – Feature representation | Base2 – Feature representation | Result – Feature representation | Result – Cp representation |
|---|---|---|---|---|
| 1 | Feature A<br>AttA1 : DomainA11<br>AttA2 : DomainA21 | Feature A<br>AttA2 : DomainA22<br>AttA3 : DomainA32 | Feature A<br>AttA2: DomainA21 ∪ DomainA22 | FeatureA ∈{0,1},<br>Att2 ∈{DomainA21}<br>∪{DomainA22},<br>FeatureA ⇔ Att2 > 0 |
| 2 | Feature A / Feature B (or) Feature A / Feature B (or) Feature A / Feature B (or) Feature A / Feature B | Feature A<br>FeatureB ∉ Features(Base2) | Feature A | FeatureA ∈{0,1}<br>Or<br>Keep the full mandatory features and their |

| | | | | corresponding constraints |
|---|---|---|---|---|
| 3 |  |  |  | [FeatureA,FeaureB] ∈{0,1} |
| 4 |  |  | Mismatch with de FMs' syntax | (FeatureA ⇒FeatureB) ∨(FeatureB ⇒FeatureA) |
| 5 |  |  |  | (FeatureA = FeatureB) ∨(FeatureA ≥ FeatureB)<br>It is:<br>FeatureA ≥ FeatureB |
| 6 |  |  |  | (FeatureA ≥FeatureB) ∨(FeatureA ⇒FeatureB)<br>It is:<br>FeatureA ⇒ FeatureB |
| 7 |  |  |  | (FeatureA = FeatureB) ∨(FeatureA ⇒FeatureB)<br>It is:<br>FeatureA ⇒ FeatureB |
| 8 |  |  | Contradiction | Contradiction to be solved by a domain expert |
| 9 |  |  |  | (FeatureA ≥ FeatureB) ∨ (FeatureB ≥ FeatureA) |
| 10 |  |  |  | FeatureA = FeatureB |
| 11 |  |  |  | (FeatureA≥FeatureB) ∨(FeatureB=FeatureA)<br>It is:<br>FeatureA ≥ FeatureB |
| 12 |  |  | If there is a path between features A and B, it is necessary to determine the hierarchical order among features A, B and C. Otherwise there is a mismatch with FMs' syntax | If features A and B are full mandatory:<br>(FeatureA = FeatureC)∨ (FeatureB = FeatureC)<br>If only Feature A is full-mandatory:<br>(FeatureA = FeatureC)<br>If only Feature B:<br>(FeatureB = FeatureC)<br>If none is full-mandatory<br>Feature C ∈{0,1} |
| 13 |  |  | If there is a path between features A and B, it is necessary to determine the hierarchical order among features A, B and C. Otherwise there is a mismatch with FMs' syntax | If features A and B are full mandatory:<br>(FeatureA ≥ FeatureC)∨ (FeatureB = FeatureC)<br>If only Feature A is full-mandatory:<br>(FeatureA ≥ FeatureC)<br>If only Feature B:<br>(FeatureB = FeatureC)<br>If none is full-mandatory<br>Feature C ∈{0,1} |
| 14 |  |  | If there is a path between features A and B, it is necessary to determine the hierarchical order among features A, B and C. Otherwise there is a mismatch with FMs' syntax | If features A and B are full mandatory:<br>(FeatureA ≥ FeatureC)∨ (FeatureB ≥ FeatureC)<br>If only Feature A is full-mandatory:<br>(FeatureA ≥ FeatureC)<br>If only Feature B:<br>(FeatureB ≥FeatureC)<br>If none is full-mandatory<br>Feature C ∈{0,1} |

| | | | | |
|---|---|---|---|---|
| 15 |  |  |  | FeatureA ≥ FeatureB |
| 16 |  |  |  | FeatureA ≥ FeatureB |
| 17 |  |  |  | FeatureA ≥ FeatureC |

This strategy, applied to our running example, gives a PLM with the common features and attributes where the conservative relationship prevails over the restrictive one. Besides, we keep the group cardinalities (except the optional group cardinality <0..3> among Support_usb, Cdrom_atech and Pcmacia_support) of the input models without modifications and the core features with their respective relationships. The resulting model is presented in Table 7.6.

**Table 7.6.** UNIX running example of Figures 3.5 and 3.6 integrated with the strategy N° 3

| | |
|---|---|
| Application of Rule 1 | [UNIX, Kernel, Scheduling, ExecutingInstructions, InterpretingInstructions, AccomplishingTheTransferOfData, AllocatingTheMachinesMemory, Shell, FileSystem, UserInterface, Graphical, Process, Cdrom, Usb, Net, SH, TCSH, BASH] ∈ {0,1} ∧ |
| 2 | UNIX = Kernel ∧ |
| No matching with the other model | Kernel1 = AllocatingTheMachinesMemory1 ∧<br>AllocatingTheMachinesMemory1 ⇒ Process ∧<br>Kernel1 = Scheduling1 ∧<br>Scheduling1 ⇒ Process ∧<br>Kernel1 = AccomplishingTheTransferOfData1 ∧<br>AccomplishingTheTransferOfData1 ⇒ Process ∧<br>Shell1 ⇒ InterpretingInstructions1 ∧<br>Kernel1 = InterpretingInstructions1 ∧<br>Shell1 ⇒ ExecutingInstructions1 ∧<br>Kernel1 = ExecutingInstructions1 ∧ |
| 2 | Process ≤ Process1 + Process2 + Process3 + Process4 + Process5 ≤ Process * 5 ∧<br>UNIX ≥ Process ∧ |
| 2 | UNIX = FileSystem ∧ |
| 12 | (UNIX = Shell ∨ UserInterface ≥ Shell) ∧ |
| 14 | (UNIX ≥ UserInterface ∨ UtilityProgram ≥ UserInterface) ∧ |
| 16 | UserInterface ≥ Graphical ∧ |
| 2 | UNIX ≥ UtilityProgram ∧ |
| Rule 2 with user intervention | Shell ≤ SH + TCSH + BASH ≤ Shell * 3 ∧ |
| Rule 2 with user intervention | UNIX = Cdrom + Usb + Net ∧ |

All features, except those that are common to the input models or core features in one of the models, are not included in the resulting model. In contrast with the model resulting from

application of Strategy 1, which permits the configuration of 43 products, the resulting model after application of Strategy 3 permits the configuration of 73 products. The difference is due to the fact that this strategy intends to keep the maximum number of possible products with the common reusable elements of both input models.

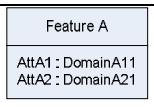
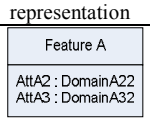
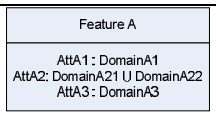
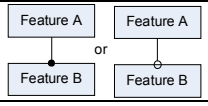
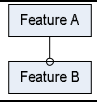
### Strategy N° 4: Conservative and keeping all features and attributes

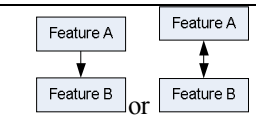
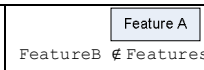
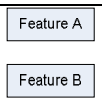
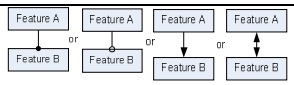
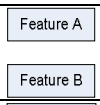
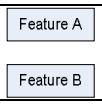
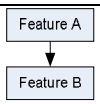
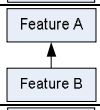
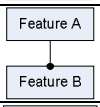
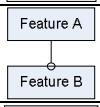
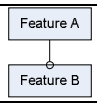
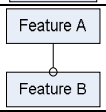
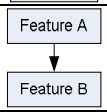
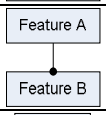
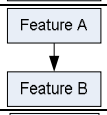
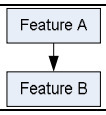
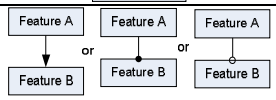
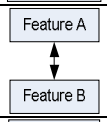
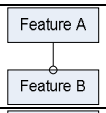
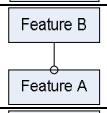
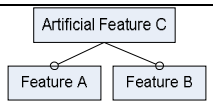
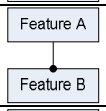
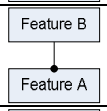
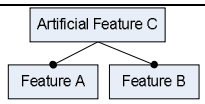
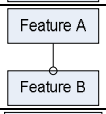
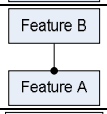
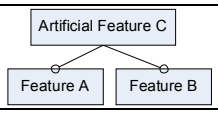
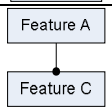
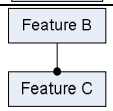
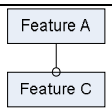
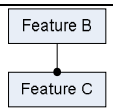
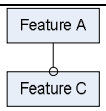
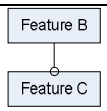
As strategy N° 3, this strategy is also conservative, but this time it permits the configuration of products with all reusable elements and attributes available in both input models (Segura *et al*. 2008, Acher *et al*. 2010). When two elements of the input models match according to the rules of Table 7.7, the conservative strategy keeps the constraint corresponding to the most general relationship. In other words, this strategy keeps the construct that permits configuring the products represented in both input models, for example:

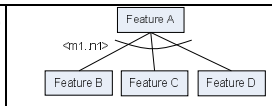
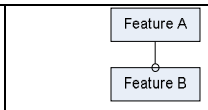
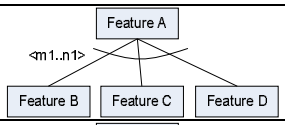
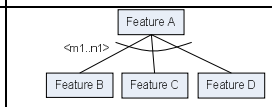
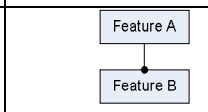
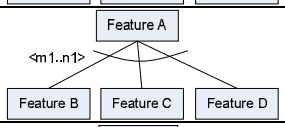
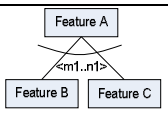
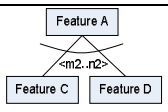
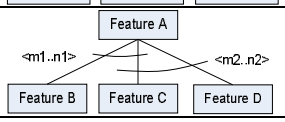
| **Base model 1** | **Base model 2** | **Resulting model** |
|---|---|---|
| $A = B$ | $(A \geq B) \Rightarrow C$ | $(A \geq B) \Rightarrow C$ |

Now, we use the rules of Table 7.7 to integrate our running example by means of the conservative strategy keeping features and attributes of the original models. This strategy seems to be one of the most appropriate ways to integrate FMs in a multi-team working environment where each team models a particular aspect of the PL. This claim is supported by the fact that the variability of each model complements the variability of the other models and in that way the resulting model combines in a generative way the features and the variability of the input models. Another context in which this strategy would be useful is when two companies are merged and the new company uses the reusable components of the legacy FMs to offer a larger portfolio of products that combine the products individually offered by the original companies.

**Table 7.7.** Integration strategy N° 4. Rules for the conservative strategy, keeping all features and attributes.

| | Base1 – Feature representation | Base2 – Feature representation | Result – Feature representation | Result – Cp representation |
|---|---|---|---|---|
| 1 |  |  |  | FeatureA ∈ {0,1},<br>Att1 ∈ {DomainA1},<br>Att2 ∈ {DomainA21}<br>U{DomainA22},<br>Att3 ∈ {DomainA3},<br>FeatureA ⇔ Att1 > 0,<br>FeatureA ⇔ Att2 > 0,<br>FeatureA ⇔ Att3 > 0 |
| 2 |  |  <br>FeatureB ∉ Features(Base2) |  | FeatureA ≥ FeatureB |

| | | | | |
|---|---|---|---|---|
| 3 |  |  <br>FeatureB ∉ Features(Base2) |  | [FeatureA,FeatureB]∈{0,1} |
| 4 |  |  |  | [FeatureA,FeatureB]∈{0,1} |
| 5 |  |  | Mismatch with de FMs' syntax | (FeatureA ⇒ FeatureB) ∨(FeatureB ⇒FeatureA) |
| 6 |  |  |  | (FeatureA = FeatureB) ∨(FeatureA ≥FeatureB)<br>It is:<br>FeatureA ≥ FeatureB |
| 7 |  |  | Mismatch with de FMs' syntax | (FeatureA ≥ FeatureB) ∨(FeatureA ⇒FeatureB) |
| 8 |  |  |  | (FeatureA = FeatureB) ∨(FeatureA ⇒FeatureB)<br>It is:<br>FeatureA ⇒ FeatureB |
| 9 |  |  | Contradiction | Contradiction to be solved by a domain expert |
| 10 |  |  |  | (FeatureA ≥ FeatureB) ∨ (FeatureB ≥ FeatureA) |
| 11 |  |  |  | FeatureA = FeatureB |
| 12 |  |  |  | (FeatureA ≥ FeatureB) ∨ (FeatureB=FeatureA) |
| 13 |  |  | If there is a path between features A and B, it is necessary to determine the hierarchical order among features A, B and C. Otherwise there would be a mismatch with FMs' syntax | (FeatureA=FeatureC) ∨ (FeatureB=FeatureC) |
| 14 |  |  | If there is a path between features A and B, it is necessary to determine the hierarchical order among features A, B and C. Otherwise there would be a mismatch with FMs' syntax | (FeatureA ≥FeatureC) ∨ (FeatureB=FeatureC) |
| 15 |  |  | If there is a path between features A and B, it is necessary to determine the hierarchical order among features A, B and C. Otherwise there would be a mismatch with FMs' syntax | (FeatureA ≥ FeatureC)∨ (FeatureB ≥ FeatureC) |

| | | | | |
|---|---|---|---|---|
| 16 |  |  |  | (m1*FeatureA ≤ FeatureB + FeatureC + FeatureD ≤ n1*FeatureA) |
| 17 |  |  |  | (m1*FeatureA ≤ FeatureB + FeatureC + FeatureD ≤ n1*FeatureA) |
| 18 |  |  |  | (m1*FeatureA ≤ FeatureB + FeatureC ≤ n1*FeatureA) ∧ (m2*FeatureA ≤ FeatureC + FeatureD ≤ n2*FeatureA) |

The application of the matching and merging rules shown in Table 7.7 to the models presented in Figures 3.5 and 3.6, results in the product line model presented in Table 7.8.

**Table 7.8.** UNIX running example of Figures 3.5 and 3.6 integrated with the strategy N° 4

| | |
|---|---|
| Application of Rule 1 | [UNIX, Kernel, Scheduling, ExecutingInstructions, InterpretingInstructions, AccomplishingTheTransferOfData, AllocatingTheMachinesMemory, Shell, FileSystem, UserInterface, Graphical, Process1, Process2, Process3, Process4, Process5, Cdrom, Usb, Net, UtilityProgram, FileMaintenance, Editing, OnlineInfo, ProgrammingSupport, SH, TCSH, BASH] ∈ {0, 1} ∧<br>[WidthResolution] ∈{800, 1024, 1366} ∧<br>[HeightResolution] ∈ {600, 768} ∧<br>[Support_usb, Cdrom_atech, Pcmacia_support] ∈ {0,1,2}∧<br>[A, B, C] ∈ {0, 1} ∧ |
| 2 | UNIX ≥ Kernel ∧ |
| No matching with the other model | Kernel1 = AllocatingTheMachinesMemory1 ∧<br>AllocatingTheMachinesMemory1 ⇒ Process ∧<br>Kernel1 = Scheduling1 ∧<br>Scheduling1 ⇒ Process ∧<br>Kernel1 = AccomplishingTheTransferOfData1 ∧<br>AccomplishingTheTransferOfData1 ⇒ Process ∧<br>Shell1 ⇒ InterpretingInstructions1 ∧<br>Kernel1 = InterpretingInstructions1 ∧<br>Shell1 ⇒ ExecutingInstructions1 ∧<br>Kernel1 = ExecutingInstructions1 ∧ |
| 2 | Process ≤ Process1 + Process2 + Process3 + Process4 + Process5 ≤ Process * 5 ∧<br>UNIX ≥ Process ∧ |
| 2 | UNIX ≥ FileSystem ∧ |
| No matching with the other model | Support_usb1 ⇔ A ∧<br>Cdrom_atech1 ⇔ B ∧<br>Pcmacia_support1 ⇔ C ∧<br>0 ≤ A + B + C ≤ 3*Kernel1 ∧ |
| 14 | (UNIX = Shell ∨ UserInterface ≥ Shell) ∧ |
| 15 | (UNIX ≥UserInterface ∨ UtilityProgram ≥UserInterface)∧ |
| 17 | UserInterface ≥ Graphical ∧<br>Graphical = 1 ⇔ (WidthResolution= W1 ∧ HeightResolution= H1) ∧<br>Graphical = 0 ⇔ (WidthResolution = 0 ∧ HeightResolution = 0) ∧<br>fd_relation([[800, 600], [1024, 768], [1366, 768]], [W1,H1]) ∧ |
| 2 with user intervention | Shell ≤ SH + TCSH + BASH ≤ Shell * 3 ∧ |

| | |
|---|---|
| 2 with user intervention | UNIX = Cdrom + Usb + Net ∧ |
| 2 with user intervention | UtilityProgram = FileMaintenance ∧ |
| 2 with user intervention | UtilityProgram = Editing ∧ |
| 2 with user intervention | UtilityProgram ≥ OnlineInfo ∧ |
| 2 with user intervention | UtilityProgram ≥ ProgrammingSupport |

Note that the supplementary restrictions among features present in the resulting model (e.g., FileMaintenance ⇒ UserInterface, Editing ⇒ UserInterface) are not included in the resulting model in order to keep the essence of this strategy (i.e., to be the less restrictive as possible). With this strategy, it is also worth noting that in Table 7.7 relationships conditioning the presence of features that are neither common nor core features should be included in the resulting model unless the integrator engineer decides otherwise. In the same way, the integrator engineer must decide how to integrate the features and the constructs that belong to one of the base models but not to the other one; as the case of the group cardinalities that have been kept without modifications on the resulting model.

The resulting model allows configuring 967221 different products, which shows the expressivity of the resulting model when models are integrated with a conservative strategy.

**Strategy N° 5**: **Disjunctive and keeping the features and attributes of the original models**

This strategy is disjunctive in the sense that the resulting model permits configuring the products presented on one of the input models by using the reusable elements and attributes of one of the particular models but not those of the other model, for example:

| **Base model 1** | **Base model 2** | **Resulting model** |
|---|---|---|
| A = B | (A ≥ B) ⇒C | (A = B) ⊕ ((A ≥ B) ⇒C) |

This strategy can be justified by cases where two different companies, in progressive merge, integrate their FMS but at the beginning they want to keep in the integrated PLM the possibility to generate with only one model, the products of each company.

This integration strategy creates an artificial root feature that will be related to the root features of each base model by means of an exclusion relationship. It is worth noting there is not necessary any integration rule to implement this integration strategy. It is due to the fact that the collection of products that this strategy is intended to allow in the resulting model is composed of the union of both models, and then the cardinality of the resulting model is mathematically represented by card(BaseModel1 ∪ BaseModel2) =

card(BaseModel1) + card(BaseModel2) - card(BaseModel1 ∩ BaseModel2).

This shows that there are no necessary integrations rules in order to apply this strategy and that a simple disjunction between both models is enaught (the disjoction is represented as a <1..1> group cardinality between the root features of both models, as presented in Figure 7.2). An example of the application of this strategy is presented in Figure 7.2 with our running example (cf. Figures 3.5 and 3.6).

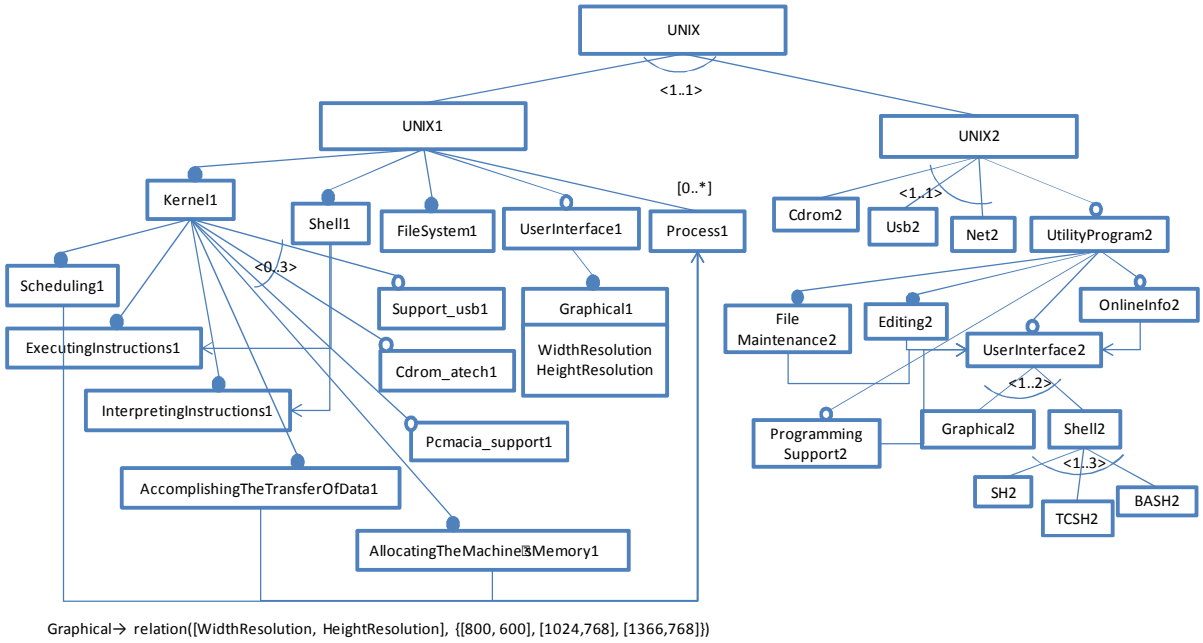



**Figure 7.2.** Application of the disjunctive integration strategy on our running example.

Thus, the FMs of our running example can be integrated, following the disjunctive strategy, as a CP as follows.

```
[UNIX, UNIX1, Kernel1, Scheduling1, ExecutingInstructions1,
InterpretingInstructions1, AccomplishingTheTransferOfData1,
AllocatingTheMachinesMemory1, Shell1, FileSystem1, UserInterface1,
Graphical1, Process1, Process2, Process3, Process4, Process5, UNIX2,
Cdrom2, Usb2, Net2, UtilityProgram2, FileMaintenance2, Editing2,
OnlineInfo2, ProgrammingSupport2, UserInterface2, Shell2, SH2, TCSH2,
BASH2] ∈ {0, 1} ∧
[WidthResolution] ∈ {0, 800, 1024, 1366} ∧
[HeightResolution] ∈ {0, 600, 768} ∧
[Support_usb1, Cdrom_atech1, Pcmacia_support1] ∈ {0,1,2} ∧
[A, B, C] ∈ {0, 1} ∧

UNIX = 1 ∧

%this is the disjunction:
UNIX = UNIX1 + UNIX2 ∧

UNIX1 = Kernel1 ∧
Kernel1 = AllocatingTheMachinesMemory1 ∧
AllocatingTheMachinesMemory1 ⇒ Process ∧
Kernel1 = Scheduling1 ∧
Scheduling1 ⇒ Process ∧
```

```
Kernel1 = AccomplishingTheTransferOfData1 ∧
AccomplishingTheTransferOfData1 ⇒ Process ∧
Shell1 ⇒ InterpretingInstructions1 ∧
Kernel1 = InterpretingInstructions1 ∧
Shell1 ⇒ ExecutingInstructions1 ∧
Kernel1 = ExecutingInstructions1 ∧

Support_usb1 ⇔ A ∧
Cdrom_atech1 ⇔ B ∧
Pcmacia_support1 ⇔ C ∧
0 ≤ A + B + C ≤ 3 * Kernel1 ∧
UNIX1 = Shell1 ∧
UNIX1 = FileSystem1 ∧

UNIX1 ≥ UserInterface1 ∧
UserInterface1 = Graphical1 ∧
Graphical1=1 ⇔ (WidthResolution=W1 ∧ HeightResolution=H1)∧
Graphical1=0 ⇔ (WidthResolution=0 ∧ HeightResolution=0) ∧
fd_relation([[800, 600], [1024, 768], [1366, 768]], [W1,H1]) ∧

UNIX1 ≥ Process ∧
R1 = Process1 + Process2 + Process3 + Process4 + Process5 ∧
Process ≤ R1 ≤ Process * 5 ∧

UNIX2 = Cdrom2 + Usb2 + Net2 ∧
UNIX2 ≥ UtilityProgram2 ∧

UtilityProgram2 = FileMaintenance2 ∧
FileMaintenance2 ⇒ UserInterface2 ∧
UtilityProgram2 = Editing2 ∧
Editing2 ⇒ UserInterface2 ∧
UtilityProgram2 ≥ UserInterface2 ∧
UtilityProgram2 ≥ OnlineInfo2 ∧
OnlineInfo2 ⇒ UserInterface2 ∧
UtilityProgram2 ≥ ProgrammingSupport2 ∧
ProgrammingSupport2 ⇒ UserInterface2 ∧

UserInterface2 ≤ Graphical2 + Shell2 ≤ UserInterface2 * 2 ∧
R2 = SH2 + TCSH2 + BASH2 ∧
Shell2 ≤ R2 ≤ Shell2 * 3
```

This resulting model permits the configuration of 3324 different products. It is worth noting that the *null* product (all the variables set to 0) present in both input FMs is counted only one time in the resulting model. This fact explains the result 3324 = 3225 + 100 – 1; where 3225 is the number of products of the base model presented in Figure 3.5, and 100 is the number of products of the base model presented in Figure 3.6.

In order to develop our multi-model verification approach, we use the integration of the two FMs of our running example by means of the “conservative strategy keeping features and attributes of the original models” (also called Strategy N° 1).

### 7.1.1 Conformance Checking

The abstract syntax of the resulting model presented in Figure 7.1 can be represented as a constraint logic program, by means on the approach presented in Chapter 3. The resulting constraint program is as follows:

```
(1)   root(fea1).
(2)   feature(fea1, 'UNIX',[]).
(3)   feature(fea2, 'Kernel', []).
(4)   feature(fea3, 'Scheduling', []).
(5)   feature(fea4, 'ExecutingInstructions', []).
(6)   feature(fea5, 'InterpretingInstructions', []).
(7)   feature(fea6, 'AccomplishingTheTransferOfData', []).
(8)   feature(fea7, 'AllocatingTheMachinesMemory', []).
(9)   feature(fea8, 'Shell', []).
(10)  feature(fea9,  'SH', []).
(11)  feature(fea10, 'TCSH', []).
(12)  feature(fea11, 'BASH', []).
(13)  feature(fea12, 'FileSystem', []).
(14)  feature(fea13, 'UserInterface', []).
(15)  feature(fea14, 'Graphical', []).
(16)  feature(fea15, 'Process', []).
(17)  feature(fea16, 'Cdrom', []).
(18)  feature(fea17, 'Usb', []).
(19)  feature(fea18, 'Net', []).
(20)  dependency(dep1, fea1, fea2).
(21)  dependency(dep2, fea2, fea3).
(22)  dependency(dep3, fea2, fea4).
(23)  dependency(dep4, fea2, fea5).
(24)  dependency(dep5, fea2, fea6).
(25)  dependency(dep6, fea2, fea7).
(26)  dependency(dep7, fea1, fea8).
(27)  dependency(dep8, fea8, fea9).
(28)  dependency(dep9, fea8, fea10).
(29)  dependency(dep10, fea8, fea11).
(30)  dependency(dep11, fea1, fea12).
(31)  dependency(dep12, fea1, fea13).
(32)  dependency(dep13, fea13, fea14).
(33)  dependency(dep14, fea1, fea15).
(34)  dependency(dep15, fea1, fea16).
(35)  dependency(dep16, fea1, fea17).
(36)  dependency(dep17, fea1, fea18).
(37)  dependency(dep18, fea8, fea4).
(38)  dependency(dep19, fea8, fea5).
(39)  dependency(dep20, fea3, fea14).
(40)  dependency(dep21, fea6, fea14).
(41)  dependency(dep22, fea7, fea14).
(42)  mandatory(dep1).
(43)  mandatory(dep2).
(44)  mandatory(dep3).
(45)  mandatory(dep4).
(46)  mandatory(dep5).
(47)  mandatory(dep6).
(48)  mandatory(dep7).
(49)  optional(dep8).
(50)  optional(dep9).
(51)  optional(dep10).
(52)  mandatory(dep11).
(53)  optional(dep12).
(54)  mandatory(dep13).
(55)  optional(dep14).
(56)  optional(dep15)
(57)  optional(dep16).
(58)  optional(dep17).
(59)  requires(dep18).
```

```
(60)  requires(dep19).
(61)  requires(dep20).
(62)  requires(dep21).
(63)  requires(dep22).
(64)  groupCardinality([dep8, dep9, dep10], 1, 3).
(65)  groupCardinality([dep15, dep16, dep17], 1, 1).
(66)  individualCardinality(fea14, 0, 5)
```

Where lines 1 and 2 define the root feature, lines 3 to 19 define the rest of features, lines 20 to 41 defines the dependencies among features, lines 44 to 63 define the type of each dependency, lines 64 and 65 define the two group cardinality of the model and line 66 defines the individual cardinality of the feature `Process` (identified by the atom `fea14`).

The generic conformance checking approach presented in Chapter 5 can be adapted to the feature dialect used in this thesis (cf. Figure 3.4) and used to check the conformance of the FM depicted in Figure 7.1 against the FM metamodel depicted in Figure 3.4. The eight generic conformance criteria (CC) to check conformance of PLM, adapted to the feature dialect used in this thesis give the following results:

**CC.1.** A FM is composed of one or several root features: line 1 presents one instance of a root feature.

**CC.2.** A FM is composed of one or several features and dependencies: lines 2 to 19 represent 18 instances of features and lines 20 to 41 represent 21 instances of dependencies.

**CC.3.** Each feature has a unique name: logical facts of lines 2 to 19 have, each one, three atoms (an atom is a general-purpose name with no inherent meaning); the second atom of each fact corresponds to its name and is unique to each fact.

**CC.4.** Each dependency relates two or several different features: each fact instantiating a dependency (cf. lines 20 to 41) has three atoms; the first atom represents the identifier of each dependency and the second and third atoms represent the features related by the corresponding dependency.

**CC.5.** An attribute belongs to one and only one feature: there is not attributes on the model of Figure 7.1 and therefore there is not facts representing attributes in the constraint logic program representation of the model.

**CC.6.** An attribute has a unique name: the model of Figure 7.1 has no attributes to be evaluated.

**CC.7.** An individual cardinality is associated to one and only one feature: the individual cardinality instantiated in line 66 has three atoms, the first atom represents the name of the feature with a cardinality [0..5].

**CC.8.** A group cardinality groups two or more optional dependencies: lines 64 and 65 instantiate the two group cardinalities of the FM represented in Figure 7.1. The first group cardinality has three atoms, the first atom is a list of features, and the second and third atoms represent the lower and upper boundaries of the cardinality. The list of features contains three atoms (i.e., `dep8, dep9, dep10`), each one of them is the identifier of one optional (cf. lines 49, 50 and 51 respectively) dependency (cf. lines 27, 28 and 29 respectively).

### 7.1.2 Domain-specific Verification

The semantics of the resulting model depicted in Figure 7.1 can be represented as a constraint program by means of the approach presented in Chapter 3. The resulting constraint program is presented below, where a coma between two constraints means an *and*.

```
(16)  domain([UNIX, Kernel, Scheduling, ExecutingInstructions,
      InterpretingInstructions, AccomplishingTheTransferOfData,
      AllocatingTheMachinesMemory, Shell, FileSystem, UserInterface,
      Graphical, Process, Cdrom, Usb, Net, SH, TCSH, BASH], 0, 1),
(17)  UNIX = Kernel,
(18)  UNIX = Shell,
(19)  Shell ≤ SH + TCSH + BASH,
(20)  SH + TCSH + BASH ≤ Shell * 3,
(21)  Kernel = AllocatingTheMachinesMemory,
(22)  AllocatingTheMachinesMemory ⇒ Process,
(23)  Kernel = Scheduling,
(24)  Scheduling ⇒ Process,
(25)  Kernel = AccomplishingTheTransferOfData,
(26)  AccomplishingTheTransferOfData ⇒ Process,
(27)  Shell ⇒ InterpretingInstructions,
(28)  Kernel = InterpretingInstructions,
(29)  Shell ⇒ ExecutingInstructions,
(30)  Kernel = ExecutingInstructions,
(31)  R1 = Process1 + Process2 + Process3 + Process4 + Process5,
(32)  Process ≤ R1,
(33)  R1 ≤ Process * 5,
(34)  Process ≤ UNIX,
(35)  UNIX = FileSystem,
(36)  UserInterface ≤ UNIX,
(37)  UserInterface = Graphical,
(38)  UNIX = Cdrom + Usb + Net
```

Line 1 defines the domain of the variables of the product line, lines 2, 3, 6, 8, 10, 13, 15, 20 and 22 define the mandatory dependencies on the product line, lines 7, 9, 11, 12 and 14 define the requirement dependencies, lines 19 and 21 define the optional dependencies and lines 4, 5 and 23 define the group cardinalities, and lines 16, 17 and 18 represent the individual cardinality of the product line.

Now the domain-specific verification approach presented in Chapter 6 for standalone models can also be applied on this integrated model. The application of the domain-specific verification criteria depicted in the typology of Figure 4.1 gives the following results:

1. Non-void.

   The integrated model presented above is not void and its semantic richness permits the configuration of 12441600 different products. It is worth noting that the variable `UserInterface` that was an optional feature in the models of Figures 3.3 and 3.4 becomes a mandatory feature in the integrated model. This is because of the restrictive strategy in which, the requires dependencies with the feature `UserInterface` in the model of Figure 3.6 and the direct dependency with the root feature in Figure 3.5, makes this feature mandatory.

2. Non-false.

   The integrated model is non-false. In fact, the base model of Figure 3.5 permits the configuration of 3225 products, and the base model of Figure 3.6 permits the configuration of 100 products; which means that even better, the expressiveness of the resulting model was significantly increased.

3. Attainable domains.

   No features, except the core features that can take the value of 0. All the features of the integrated model can attain their domain.

4. Non-dead artefacts.

   There are no dead features in the integrated model.

5. Non-false optional artefacts.

   Feature `Process` is modeled as optional (because of its [0..5] individual cardinality); however, `Process` appears in all the products of the PL due to the fact that this feature is included by other core features like `Scheduling`.

6. Non-redundant constraints.

   On the one hand `UtilityProgram ≤ UNIX`, and on the other hand `UserInterface ≤ UtilityProgram`. Therefore, the constraint: `UserInterface ≤ UNIX` is redundant. This redundancy can be explained by the fact that the variable `UserInterface` is already an optional variable of `UNIX` through `UtilityProgram`. The technique presented in Section 5.2 verifies the consistency of the model with the redundant constraints, and then changing the constraint by its negation (`UserInterface >`

UNIX) and proving the inconsistency of the new model was used to identify this redundancy.

## 7.2 Verification of Dopler Variability Models

Decisions and assets are linked with *inclusion conditions* defining traceability from the solution space to the problem space (e.g., the asset Tab Window Manager must be included in the solution space if the option OnlineInfo of the decision Utility program is selected in a particular configuration). In our integration approach, these inclusion conditions are constraints that will be added to the collection of constraints representing the decision and asset model. Once these constraints are added, both viewpoints of the PL are integrated, and the model is ready to be verified against the criteria depicted in Section 5.2 with minor variants in some criteria. The application of these verification criteria over the Dopler model depicted in Figure 3.8 and the explanation regarding the minor variants are presented as follows:

1. Void model.

   This model is not a void because it permits the configuration of at least one product; for instance C1 = {USB, Editing, ProgrammingSupport, Shell}

2. False model.

   This model is not a false model because it permits the configuration of more than two products; for instance: C2 = {Cdrom, Editing, OnlineInfo, Shell, Twm, KDE, Qt, GraphicalResolution = "800x600", Width = 800} and C3 = {USB, Editing}.

3. Non-attainable validity conditions' and domains' values.

   This operation either (i) takes a collection of decisions as input and returns the decisions that cannot attain one or more values of its validity condition; or (ii) takes a collection of assets as input and returns the assets that cannot attain one of the values of its domain. A non-attainable value of a validity condition or a domain is a value that can never be taken by a decision or an asset in a valid product. Non-attainable values are undesired because they give the user a wrong idea of the values that decisions and assets modeled in the product line model can take. In our example (see Figure 3.8), the validity condition Width ≥ 800 && Width ≤ 1366 determines a very large range of values that can take the variable Width, however this variable can really take three values: 800, 1024 and 1366 which means that values like 801, 802,..., 1023, 1025, ..., 1365 are not attainable values.

4. Dead reusable elements.

In the Dopler language, the reusable elements are Decisions and Assets. This operation takes a collection of decisions and assets as input and returns the set of dead decisions and assets (if some exist), or *false* otherwise. A decision is dead if it never becomes available for answering it (Mazo *et al*. 2011a). An asset is dead if it cannot appear in any of the products of the product line (Mazo *et al*. 2011a). The presence of dead decisions and assets in PLMs indicates modeling errors and intended but unreachable options. A decision can become dead (i) if its visibility condition can never evaluate to true (e.g., if contradicting decisions are referenced in a condition); (ii) a decision value violates its own visibility condition (e.g., when setting the decision to true will in turn make the decision invisible); or (iii) its visibility condition is constrained in a wrong way (e.g., a decision value is `> 5 && < 3` at the same time). An asset can become dead (i) if its inclusion depends on dead decisions, or (ii) if its inclusion condition is false and it is not included by other assets (due to *requires* dependencies to it). Dead variables in CP are variables than can never take a valid value (defined by the domain of the variable) in the solution space. Thus, our approach evaluates each non-zero value of each variable's domain. If a variable cannot attain any of its non-zero values, the variable is considered dead. For instance, in the Dopler model of Figure 3.8, there are not dead decisions or assets.

5. Redundancy-free.

   In the asset model (cf. the right of Figure 3.8) the asset `4dwn` requires `MwM`, which at the same time requires the asset `Motif`, therefore the dependency `4dwm` requires `Motif` is redundant according to the redundancy-free algorithm presented in Section 5.2.

It is worth noting that the domain-specific operation "false optional reusable elements" is not applicable in Dopler models due to the fact that this language does not have explicitly the concept of optional. Decisions and assets are optional in Dopler models according to the evaluation of the visibility conditions (in the case of decisions) and inter-assets dependencies in the case of assets.

## 7.3 Gaps and Challenges

The CP-based approach to verify multi-model product lines is a first step in this direction that complements the related works found in literature. Some of these works are:

Alves *et al.* (2006) propose an approach to FM refactoring, which, in contrast to FM specialization formalized by Czarnecki *et al.* (2005), is a transformation that either maintains

or increases the set of all FM configurations, whereas FM specialization is a transformation that decreases the set of configurations. Both approaches propose a collection of operations allowing, for example, merge optional and alternative relations of two FMs. Alves *et al*. present the refactoring as a sequence of modification operations applied to both original FMs separately; for example: change, add or remove a relationship, collapse two relationships and pull up/push down a feature. Alves *et al.* can derive other refactorings between more than two FMs by taking a base FM and applying on it a sequence of operations (corresponding to relationships among the products configured from the other input models). The authors encode FMs in the Prototype Verification System language (PVS) in order to prove the FM refactorings proposed in their work. Unfortunately, this approach only considers one strategy to merge features models. In addition, due to the fact that the merge operation of FMs is based in the relationships among the features of the products derived from the input FM and not on the FM themselves, this approach is not realizable in very large FMs because in some cases, it is impossible derive all their products.

Schobbens *et al.* (2006) survey feature diagram variants and generalize the various syntaxes through a generic artefact called Free Feature Diagrams (FFD). In their work, the authors identify and define three kinds of merging strategies on FMs: intersection, union and reduced product. To the best of our knowledge, they do not provide automated support for the merging of FMs neither details about the implementation of these three strategies to deal with problems of coherence, redundancies and situations difficult to integrate. One example of these difficulties is when the resulting model needs a new concept to represent the correct semantics of the input models with regard to the selected merging strategy. In this chapter we consider these cases and complement their approach with two other integration strategies. In this context, we presume that our proposal complements their work and offer some indications about the implementation of FM integration strategies.

Liu *et al.* (2006) study PL refactoring at the code level and propose what they call Feature Oriented Refactoring (FOR). They provide a semi-automatic refactoring process to enable the decomposition of a program into features. The authors propose two operations on FOR. The first is the so-called introduction sum: “a binary operation that aggregates base modules by disjoint set union. A base module is a set of unique variables and methods that belong to one or more classes”. The second operation is called function composition or weaving, a function “used to weave the changes of a derivative module into a base module, yielding a woven base module”. A derivative module is the collection of refinements that modify the methods of a

module. This approach complements our work, since it could be applied in our approach at the code level in software product lines.

Fleurey *et al.* (2007) propose a generic framework for merging models. The framework is independent from a modeling language and has been implemented in the tool Kompose (Fleurey 2007a). The generic framework is specialized by decorating the metamodel of the language with signatures (e.g., the type). These signatures permit capturing semantic elements of the modeling language in order to produce a meaningful composition operator. The main advantage of the proposed approach is that permits the definition of merging operators for new modeling languages. The main limitation is that the framework relies on the structure of the models to compose. The signatures are the only elements which can be used to take into account semantics of models to compose. This is an issue in FMs due to the hierarchical nature of this kind of model and the typical mismatch problems related with the structure of the resulting model (as in Tables 7.1, 7.3, 7.5 and 7.7).

Apel *et al.* (2007) present an algebra for feature-oriented software development. The authors present a procedure for composing (merging) feature trees using tree superimposition. This recursive procedure, in the words of the authors, composes "two nodes to form a new node (1) when their parents have been composed already (this is not required for composing root nodes) and (2) when they have the same name and type. If two nodes have been composed, their children are composed as well, if possible. If not, they are added as separate child nodes to the composed parent node". As in our work, they assume that nodes with the same name refer to the same software artefacts and that the granularity levels of both FMs are the same. Compared to our work, they explore only one strategy to compose FMs and do not consider cross-tree constraints or feature attributes as we explore in our approach, or the syntactical mismatches present on some composition situations as presented in Tables 7.1, 7.3, 7.5 and 7.7. In contrast, they explore the problem of superimposition of features, which is a complement of the work presented in this chapter.

Jayaraman *et al.* (2007) propose an approach to integrate FMs by using graph transformation rules. One year later, Segura *et al.* (2008) presented a similar approach. This approach presents a catalogue of merge rules describing how to build a FM including all the products represented by two given FMs (a conservative strategy) previously represented as graphs. Both approaches use a free Java graphical tool for editing and transforming graphs called Attributed Graph Grammar System (AGG). To do that, the authors of both works make two assumptions: (i) "input FMs represent related products using a common catalogue of features"; and (ii) "the parental relationship between features is equal in all the FMs". In our

approach, we also assume that features with the same name refer to the same artefact and that FMs to be integrated must have the same level of granularity. However, we present several strategies to integrate FMs and not only the conservative one. As an improvement of Jayaraman *et al*'s work, Segura *et al*. use FMs with attributes and two kinds of cross-tree constraints (requires and excludes). In our approach we consider features with attributes, with cardinalities and with external constraints (additional to requires/excludes) often presented in industrial FMs (Salinesi *et al.* 2011). Our approach is inspired on Segura *et al*.'s work; in addition, this Chapter provides a more complete scenario for integration of extended feature models with the details of mismatches and contradictions omitted in (Segura *et al*. 2008).

Acher *et al.* (2010) propose two strategies to integrate the features of two FMs without cross-tree relationships, attributes and cardinalities. The fist strategy makes an intersection among the features of both input models. The second strategy unifies the features of both input models and therefore the resulting model will contain the features of both input models. Achar *et al*. hold that (i) the first strategy preserves the products that are represented in both input models at the same time; and (ii) the second strategy preserve the products of both input models. However, due to the fact that in this approach authors only consider the features of both input models and not the relationships among them, the semantics of the resulting model cannot be defined by the intersection/union of both input models. Unfortunately, their approach does not provide details about how these two strategies have been implemented neither about how to treat the contradictory and mismatch situations.

Rosenmüller *et al.* (2011) provide three alternative mechanisms to compose variability models in order to improve composition. The three mechanisms are inheritance, superimposition and aggregation. As in object-oriented programming, the authors use inheritance to create a new variability model that extends an existing model with new features and constraints. Indeed, Rosenmüller *et al* aim at deriving (i) the union of all features and (ii) the union of all constraints. The first operation increases variability by permitting all feature combinations of the merged models. The second operation limits variability by joining the constraints of the models using conjunctions. In order to do that, they translate the models into their Boolean propositional formula, merge these representations, and create a new feature model from the merged formula. This approach is similar to ours, but unfortunately no details about how to merge these propositional formulas are provided in their work. Besides, we consider not only the union strategy but other four composition strategies and instead of Boolean propositional formulas we use constraints, which can be used to merge not only FODA models but others PLMs. A more restrictive mechanism is the superimposition, in

which the propositional formulas corresponding to each base model are preserved in the resulting model. However, the authors do not provide details about the implementation of this mechanism, or about how to deal with the contradictions and redundancies in the resulting model. In contrast, inheritance and aggregation are more appropriate when input models have different names for the same artefact due to the fact that these two mechanisms permit the creation of new concepts when the names do not match. Unfortunately, these two mechanisms do not scale, because the composition in these cases is done manually.

However, even with all these works and the increasing effort of the product line community to allow the configuration of products from a multi-model product line, some questions still remain unsolved, as for example:

1. How to deal with multi-model PLs where each model is represented in a different language (i.e., where there is no a common metamodel)?
2. Is the constraint language enough expressive to represent the semantics of a multi-model PL expressed with several variability languages? Is there a sort of "assembler language" to compile every PLM independent of their metamodel?
3. Is the verification approach presented in this chapter valid for these kinds of multi-view models where each view is represented in a different formalism?
4. The verification criteria proposed for standalone product line models is enough for verifying multi-model product lines?
5. Is configuring a product from two models (of the same PL) equivalent to configuring a product from an integrated model? How to avoid or to deal with the contradictions and mismatches present on the resulting PLM even if the base models were correct themselves?
6. How to guarantee that the resulting model represent the right semantics that the actor wants to represent in the resulting model?
7. How to deal with the terminology and structural incoherencies and mismatches? How to integrate PLMs where each structure of each PLM is very different from one another?
8. How to deal with base models in which the level of granularity of requirements specified in them is different?
9. How to deal with addition of supplementary constraints specifying model inter-dependencies? How to deal with specification of dependencies that cannot be defined in any of the languages of the base models (e.g., constraints on Integer variables, or on 3 or more variables) or definition of constraint preferences (e.g., Maximize (2 * C1 + C2))?
10. What is the formal semantics of each strategy to integrate FMs?

11. Is our collection of strategies to integrate FMs complete? Is the catalogue of integration rules correct and complete according to the given semantics?

Even if this thesis does not have the answer to all these questions, the constraint-based approach presented here is a step ahead for future works willing to solve these questions.

## 7.4 Summary

This chapter presents the extension of our verification approach to the context of multi-model product lines. Our proposal way to handle this is the UNIX running example, represented by means of a Dopler model and two FMs. The running example was transformed into constraint logic programs in order to integrate them and make them automatically verifiable. Once the models were transformed, they were individually verified against the typology of verification criteria presented in Chapter 4. The results obtained from these two cases show the applicability of this verification approach in at least these two product line modelling languages. However, several research questions still remain unsolved, representing new research topics for future works.

# Chapter 8

# Evaluation

This chapter reports the results of several empirical experiments carried out to evaluate the ideas proposed in this thesis. In particular, this chapter's goal is to evaluate:

- the implementability of our verification approach,
- its generality against different product lines, specified with different languages and in different manners (stand alone and multi-model),
- the scalability of the proposed approach.

The performance results of the tool that was implemented is compared with one of the most popular PLM verification tools of the literature. The verification operations related with conformance checking could not be compared with other solutions due to the fact that, to the best of our knowledge, there are not implementations in literature to be compared to.

This chapter provides details of the resources used in the experiments and report and discuss the observed results.

## 8.1 Hardware and Software

Evaluation was made in the following environment: Laptop with Windows Vista of 32 bits, processor AMD Turion 64 bits X2 Dual-Core Mobile RM-74 2,20 GHz, RAM memory of 4,00 GB, GNU Prolog 1.3.0 and Eclipse RPC Galileo-SR2-Win32.

## 8.2 Benchmarks

### 8.2.1 Real Models

We selected a large collection of feature models used in the field of software product lines to construct the corpus of 34 FMs that served as basis for our experiments. Those models have been used in a variety of ways by their proposers and served mostly as a mean to illustrate approaches and techniques applied to software product lines. Table 8.1 presents the name, the number of features and a small description of 16 of these models, sorted by the number of features in the model.

**Table 8.1.** 16 FMs taken from the literature of software PLs and used in our experiments

| Feature Models Taken From the Literature | Number of Features | Description |
|---|---|---|
| TelecommunicationSystem | 12 | This model represents the functionalities of a telecommunication switch, which can be extended by installing additional software modules onto the hardware component such as management software or application packages for messaging and IP-services (Felfernig *et al.* 2001) |
| James_fm | 14 | JAMES is a framework to develop web collaborative systems with a particular kind of database, user interface and one or several modules (Benavides *et al.* 2005) |
| CellPhone_fm | 15 | This model is a SPL for applications that manipulate photo, music, and video on mobile devices, such as mobile phones (Figueiredo *et al.* 2008) |
| GraphBatory_fm | 20 | This model is a family of graph applications where each graph PL application implements one or more graph algorithms (Batory 2005) |
| MobilePhone_fm | 20 | This model represents a mobile phone family with several utility functions, settings, media facilities and types of connectivity (Segura 2008) |
| Fame_dbms_fm | 21 | This model corresponds to a SPL of embedded database management systems (DBMS) with different types of index, access methods, statistics (e.g., buffer hit, ratio, table size, etc.) and transactions (Kastner *et al.* 2009) |
| Insurance_Product_fm | 25 | PL Model for Insurance Systems. This model proposes several types of insurance objects, insurance options, payment modes and insurance conditions (Tekinerdogan & Aksit 2003) |
| KeyWordInContext_fm | 25 | This model represents the most important features of the KeyWord in Context problem formulated by Parnas (1972) to contrast different criteria for modular software decomposition (Sun *et al*. 2005) |
| DigitalVideoSystem_fm | 26 | This model represents a Digital Video System that need to be controlled remotely and optionally, by a telephone or by a net (Streitferdt *et al.* 2003) |
| GraphMei_fm | 30 | This models represent the family of operation of a graph editor tool such as adding, content moving, outline moving, removing, etc (Mei *et al*. 2003) |
| WebPortal_fm | 43 | It is a feature model for a web portal product line with persistency, security and performance features and some services (Mendonca *et al.* 2008) |
| DocGen_fm | 44 | DocGen is the FM of a commercial documentation generator. This documentation includes textual summaries, overviews, control flow graphs, architectural information, etc., at different levels of abstraction (Van Deursen & Klint 2002) |
| Thread_fm | 44 | The thread FM represents indeed the decision model a programmer has to go through when deciding which kind of thread support his or her application really needs (Beuche 2003) |

| | | |
|---|---|---|
| HIS_fm | 67 | This is a FM of a Home Integration System (HIS) product line constraining the services, controls and quality attributes of a HIS (Kang *et al.* 2002) |
| ModelTransformationTaxonomy | 88 | This is FM representing the variation elements of a model transformation process such as transformation rules, rule application strategies, tracing, directionality and source-target relationships (Czarnecki. & Helsen 2003) |
| EShopping_fm | 287 | This model corresponds to the product line of Business-to-Consumer systems. Elements like the catalogue of products, the characteristics of products, payment options and customer services are related in this model (Lau 2006) |

In our experiments we also used two Dopler models provided by the authors of the Dopler formalism (Dhungana *et al.* 2010). These models and the number of variables presented in each one of them are presented in Table 8.2, sorted by the number of variables in the model.

**Table 8.2.** Dopler variability models.

| Dopler Models | Number of Variables | Description |
|---|---|---|
| Kamera8655 | 39 | This model defines the variability of a fictitious product line of digital cameras. This model has been created by analyzing datasheets of all available digital cameras of a well-known digital camera manufacturer. The model comprises 7 decisions and 32 assets (Mazo *et al.* 2011a). |
| DOPLER tool suite model | 121 | This model represents the variability of the configurable DOPLER tool suite and comprises 14 decisions, 40 decision options and 67 assets (Mazo *et al.* 2011a). The model has been created by the developers of the DOPLER tool suite. |

Three product line models represented as constraints programs were developed by us, based in our academic and industrial experiences. They were also used in our experiments and are briefly presented in Table 8.3, sorted by the number of variables in the model.

**Table 8.3.** Benchmark of PLMs represented from scratch as constraint programs.

| Constraint Program Product Line Models | Number of Variables | Description |
|---|---|---|
| Vehicle movement control systems | 21 | This model represents a vehicle movement control systems product line with different kind of sensors (Salinesi *et al.* 2010b). |
| UNIX | 37 | Multi-model UNIX product line model used in this these; cf. Figures 3.5 and 3.6. |
| Stago | 49 | This model represents a family of blood analysis automatons fabricated by the company Stago (Djebbi & Salinesi 2007), (Salinesi *et al.* 2010b). |

Despite the number of models that we collected from literature, our academic partners and our own models, it was extremely challenging to find larger models to be representatives of the large-scale industrial product line models. For instance, Batory *et al.* (2006) points out that product lines in the automotive industry can contain up to 10000 features. We are aware that such models exist, however they are usually part of commercial projects that offer limited access to their resources. Therefore, we used several automatically generated models. They are used in well known benchmarks for the software PL community. These models are presented in the next section.

### 8.2.2 Automatically-Generated Models

The automatically-generated FMs we used in this thesis were taken from the SPLOT tool (Mendonça *et al.* 2009b), which provides a large collection of automatically-generated FMs to support empirical studies on the performance and scalability of automated techniques for reasoning on FMs. According to Mendonça *et al.* (2009a), in each automatically-generated model, each type of mandatory, optional, inclusive-OR (`<0..n>` group cardinality) and exclusive-OR (`<0..1>` group cardinality) feature was added with equal probability. The branching factor (number of children per parent node) of the feature tree varied from 1 to 6. The cross-tree constraints were generated as a single *Random 3-CNF formula* [3] over a subset of features in the tree.

Each model in the benchmark corresponds to a random 3-CNF formula, depending on the CTCR. Mendonça *et al.* (2009a) define Cross-Tree Constraint Ratio (CTCR) as the ratio of the number of features in the cross-tree constraints to the number of features in the feature tree. For instance, for a model with 1000 features and 30% CTCR, 300 distinct variables are selected randomly from the model and combined randomly into ternary CNF clauses. According to the Mendonça *et al* (2009a) description of the automatically-generated FMs, variables are negated in each clause with a 0.5 probability and identical clauses were not permitted. The number of clauses is controlled by clause density. For instance, given clause density of 2.3 for a model of 1000 features, the tool generates 690 (= 2.3×300) random ternary clauses. The clause density refers to the density of clauses in the cross-tree constraints not in the formula induced by the entire feature model.

Five of the 15 automatically-generated FMs used in this thesis are presented in Table 8.4.

[3] In Boolean logic, a formula is in Conjunctive Normal Form (CNF) if it is a conjunction of clauses, where a clause is a disjunction of literals. In a clause, a literal and its complement cannot appear in the same clause. In a *Random 3-CNF formula*, at most 3 variables per clause are randomly combined to create a formula; for example, the formulas ¬A□B and (A□B)□(A□C) are in 3-CNF; the formula A□¬A is not in CNF and the formula (A□B)□(A□C)□D is not in 3-CNF.

**Table 8.4.** Five automatically-generated FMs taken from SPLOT (Mendonça *et al.* 2009b)

| Some of the automatically-generated FMs with SPLOT | Number of Features | Description |
|---|---|---|
| SPLOT-3CNF-FM-500-50-1.00-UNSAT-1 | 500 | Number of 3-CNF Variables: 50 and 3-CNF Clause Density: 1.0 |
| SPLOT-3CNF-FM-1000-100-1.00-UNSAT-10 | 1000 | Number of 3-CNF Variables: 100 and 3-CNF Clause Density: 1.0 |
| SPLOT-3CNF-FM-2000-200-1.00-UNSAT-1 | 2000 | Number of 3-CNF Variables: 200 and 3-CNF Clause Density: 1.0 |
| SPLOT-3CNF-FM-5000-500-0.30-SAT-10 | 5000 | Number of 3-CNF Variables: 500 and 3-CNF Clause Density: 0.3 |
| SPLOT-3CNF-FM-10000-1000-0.10-SAT-1 | 10000 | Number of 3-CNF Variables: 1000 and 3-CNF Clause Density: 1.0 |

## 8.3 Evaluating the Domain-specific Verification Approach

We performed a series of experiments to evaluate the domain-specific verification approach proposed in this thesis. The goal was to measure:

- the effectiveness or precision of the defect's detection,
- the computational scalability, and
- the usability of the approach to verify different kinds of product line models.

These measurements are presented in the next sections, grouped by the kind of product line models against which they were measured.

### 8.3.1 The Case of Feature Models

We assessed the feasibility, precision and scalability of our approach with 46 models, out of which 44 were taken from the SPLOT repository (Mendonca *et al.* 2009b) and the other two models are the Vehicle movement control system and the Stago model (cf. Table 8.3). The size of the models is distributed as follows:

32 models of sizes from 9 to 49 features,

4 from 50 to 99, 5 from 100 to 999 and

6 from 1000 to 2000 features.

The six feature models with sizes from 5000 to 10000 features were not considered in this experiment due to the fact that the GNU Prolog solver (the used version) does not accept more that 5000 variables. Note that SPLOT models do not have attributes, in contrast to our two industrial models. Therefore artificial attributes were introduced in a random way, in order to have models with 30%, 60% or 100% of their features with attributes. In order to do

that, we have created a simple tool[4] that translates models from SPLOT format to constraint programs. Then we integrated the artificial attributes. In order to test that the transformation respects the semantics of each feature model, we have compared the results of our models without attributes with the results obtained with the tools SPLOT (Mendonca *et al.* 2009b) and FaMa (Trinidad *et al.* 2008b). In both comparisons we have obtained the same results in all the shared functions: detection of void models, dead features, and false optional features. These results show that our transformation algorithm respects the semantics of initial models.

**Precision of the detection**

Not only must the transformation of FMs into CPs be correct, but also the detection of defects. As aforementioned, the results obtained with our tool VariaMos against the results obtained with two other tools: SPLOT and FaMa, were compared. These comparisons were made over models without attributes (the original models taken from SPLOT do not have attributes) and with restricted group cardinalities, due to the fact that the group cardinalities used in SPLOT and FaMa must be able to be transformed into OR, AND and XOR operations. For example, a `<0..3>` group cardinality over three features can be represented as an OR among the three features, but a `<2..5>` group cardinality cannot be represented with an OR, AND or XOR operators. These comparisons show the same results, for the common verification functions on the three tools, but due to the fact that our own models contain attributes and group cardinalities `<m..n>`, for any `m` and `n` bellowing to non negative Integer numbers, a manual inspection has been necessary. A manual inspection on two samples of `28` and `56` features has shown that our approach identifies `100%` of the anomalies with `0%` false positives.

**Computational Scalability**

The execution time of the verification criteria in our implementation shows that the performance obtained with our approach is acceptable in realistic situations. In the worst case, verification criteria were executed in less than `19` seconds for models up to `2000` features. Figure 8.1 shows the execution time of each one of the six verification criteria in the `50` models. Each plot in the figure corresponds to a verification criterion: Figure 8.1(1) corresponds to criterion 1, Figure 8.1(2) corresponds to criterion 2 and so on. Times in the Y axis are expressed in milliseconds (ms). The X axis corresponds to the number of features. It

[4] parserSPLOTmodelsToCP.rar available at: https://sites.google.com/site/raulmazo/

is worth noting that most of the results overlap the other ones; we avoid the use of a logarithmic scale in the X axis, to keep the real behaviour of the results.

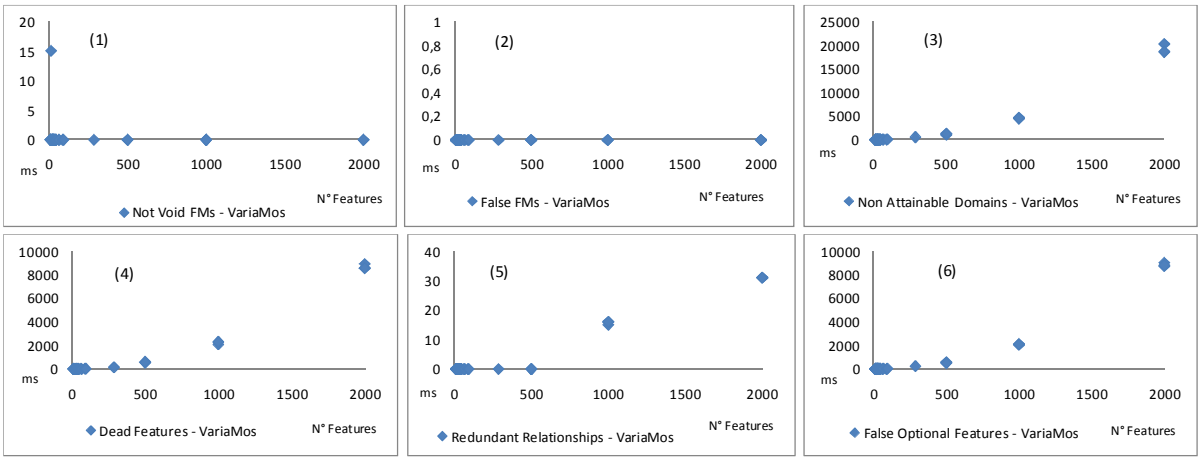



**Figure 8.1.** Execution time of the six verification criteria, per number of features

Let us now present the results in more detail.

For the models with sizes between 9 and 100 features our approach verified all criteria in less than 1 second on average.

For the models with sizes between 101 and 500 features, our apporach verified dead features and false optional features in 0.4 seconds, took 1 second to calculate the non attainable domains and 0 milliseconds in the rest of verification criteria. It is worth noting that GNU Prolog does not provide time measures of microseconds ($10^{-6}$ seconds); thus, 0 milliseconds ($10^{-3}$ seconds) must be interpreted as less than 1 millisecond. In general, over the 46 FMs, the execution time to detect dead features, false optional features and non attainable domains was less than 8.679, 8.819 and 19.089 seconds respectively. For the rest of verification criteria, the execution time is lower than 0.016 seconds even for the largest models. We can only make projections to evaluate the behaviour of our approach with larger models. Following the projection of our results, our approach is probably able to be used in larger FMs with a quadratic increase. To finish, the verification operations like redundant relationships, false feature models and void feature models are executed in less than 0.03 seconds. According to the results of our experiment, we can conclude that our domain-specific verification approach presented in Chapter 5 is scalable to large FMs.

### 8.3.2 The Case of Dopler Variability Models

The verification approach presented in this thesis was also tested with two Dopler models (cf. Table 8.2) named "digital camera" and "DOPLER". In both models, 33 defects in the DOPLER model and 22 defects in the camera model were seeded. The defects cover different types of problems to show the feasibility of the verification approach. For instance, the decision Wizard_height cannot take the values 1200, 1050, 1024 and 768 and the asset VAI_Configuration_DOPLER cannot take the value 1 (is never included for any product), even if these values take part in the corresponding variables' domain. Furthermore, the execution time of applying the approach for both models, for the different verification criteria, has been measured. The results of this experiment are presented below.

The DOPLER model was not void (it could generate 23016416 products). However, 18 defects related with non-attainable domain values and 15 dead decisions and assets (these together are the 33 defects we have seeded before) have been discovered. The verification of the digital camera model showed that the model is not void (it can generate 442368 products). In this model, 11 defects related with non-attainable domain values as well as 11 dead decisions and assets (these together are the 22 defects we have seeded before) have been discovered. It is noteworthy that the same number of defects was identified in a manual verification of both models. The automated verification found all of the seeded defects in the DOPLER model and all of the seeded defects in the camera model.

**Table 8.5.** Results of Dopler models verification: Execution time (in milliseconds) and number of defects found with each verification operation.

| | | Void model | False model | attainable . | Decisions | redundant relationships |
|---|---|---|---|---|---|---|
| DOPLER 81 Variables | Defects | No | No | 18 | 15 | No |
| | Time | 0 | 0 | 125 | 47 | 0 |
| Camera 39 Variables | Defects | No | No | 11 | 11 | No |
| | Time | 0 | 0 | 16 | 15 | 0 |

Table 8.5 shows the number of defects found and the execution time (in milliseconds) corresponding to the verification operations on the models. No defects were found regarding the "Void model", "False model" and "Redundant relationships" operations and the execution time was less than 1 millisecond for each one of these operations in each model. The model

transformations from Dopler models to constraint programs took about 1 second for each model.

## 8.4 Evaluating the Conformance Checking Approach

We performed a series of experiments to evaluate the effectiveness or precision, the scalability and the usability of the conformance checking approach proposed in this thesis. In order to do so, we executed the approach presented in Chapter 5 over 50 feature models taken from the SPLOT repository (Mendonça *et al.* 2009b).

The size of the models were distributed as follows: 30 models of sizes from 9 to 49 features, 4 from 50 to 99, 4 from 100 to 999, 9 from 1000 to 5000 and 3 of 10000 features.

Note that SPLOT models neither support attributes nor multi root features. Therefore artificial attributes (a variable followed by a domain, for example A:String) were introduced in a random way, in order to have models with 30%, 60% or 100% of their features with attributes. Following the same logic, we introduced one artificial root on the 50% of the SPLOT models. In order to do that, we created a simple tool that transforms models from SPLOT format to facts and automates the assignation of artificial attributes, permiting repeated attributes inside each affected feature (between 1 and 5 features per affected feature), and roots.

**Precision of the detection**

One example of the effectiveness of our approach is the 56 conformance anomalies of the models taken from SPLOT, violating conformance criteria (CC) 4, 6 or 7. The list of conformance criteria is:

FM CC. Criterion 1: A feature model should have one and only one root.

FM CC. Criterion 2: Features intervening in a group cardinality relationship should not be mandatory features.

FM CC. Criterion 3: A feature should not have two attributes with the same name.

FM CC. Criterion 4: Two features should not have the same name, in the same model.

FM CC. Criterion 5. A child feature cannot be related in an optional and a mandatory dependency at the same time.

FM CC. Criterion 6. Two features cannot be required and mutually excluded at the same time.

FM CC. Criterion 7. Each dependency relates two or several different features.

For example, in the *Model transformation taxonomy* feature model (cf. Table 8.1), features like `Form`, `Semantically_typed`, `Interactive`, `Source`, `Syntactically_typed`, `Target` and `Untyped` appear twice. In addition, we found `1553` conformance defects with regards to criteria 1 and 3. These came from the attributes and root features that were intentionally introduced in the SPLOT models. A manual inspection on a sample of `56` conformance defects showed that the tool identified `100%` of the anomalies with `0%` false positive. This confirms our belief that our tool has a 100% precision and a 100% recall.

**Computational Scalability**

The execution times of our tool during the experiment show that our approach is able to support a smooth interaction during a conformance checking process. Indeed, each conformance rule was executed within milliseconds. Figure 8.2 shows the execution time of each one of the seven conformance rules in the `50` models. In Figure 8.2 each plot corresponds to a conformance rule: Figure 8.2 (1) corresponds to confornace criterion 1, Figure 8.2 (2) corresponds to conformance criterion 2 and so on. Times in the Y axis are expressed in milliseconds (ms) and X axis corresponds to the number of features in a $Log_{10}$ scale to facilitate the distribution of the results and avoid the overlapping of results.

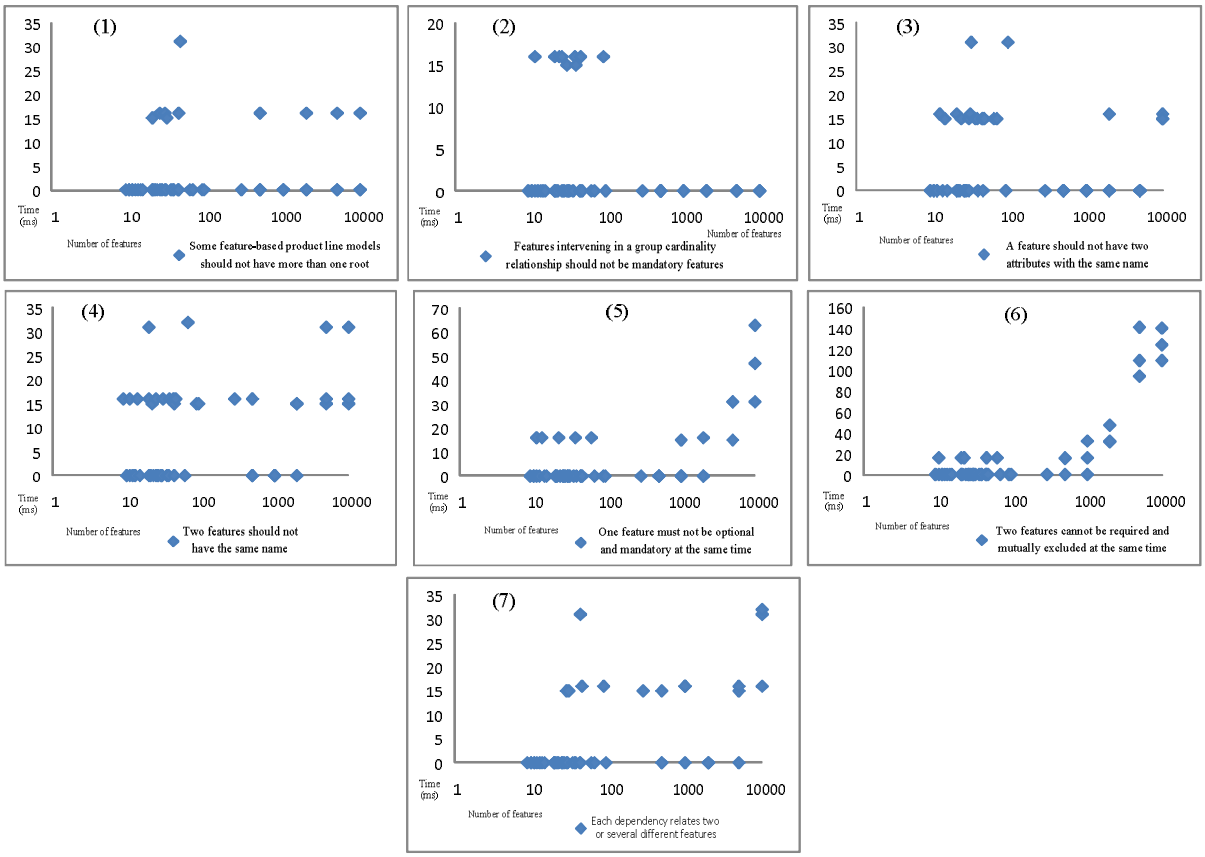



**Figure 8.2.** Execution time, of the 7 FM conformance criteria, per number of features (in a $Log_{10}$ scale)

Initial analyses indicated us that 74.2% of the queries take 0 ms. This actually means that the execution time is less than 1 ms. In small models (9 to 100 features) the worst rule execution time was 32 ms. In large models (100 to 10000 features), execution time of each rule was less than 140 ms. The maximal time taken by the tool to execute all nine conformance rules on complete models was 265 ms, which is still a ¼ of a second.

Table 8.6 shows the correlation coefficient ($R^2$) between the number of features in the models and the time that each rule takes to be executed. Of course, the $R^2$ does not prove independency between these variables. However, it gives a good indication of their dependency/independency. In the case of criteria 1, 2, 3, 4, and 6, the correlation coefficient is next to 0. This means that, despite the NP complexity of verification of product line models (Mendonça *et al*. 2009), (Yan *et al*. 2009), our tool seems to be scalable to large models when cheking these criteria. It seems that every criterion can be checked in a linear (criteria 1, 2, 3, 4, and 7) and polynomial (criteria 5 and 6) time, according to the correlation coefficient of Figure 8.2. As presented in Chapter 5, this good scalability is due to the fact that our conformance checking approach was optimised so as to avoid evaluating whole models, but only through series of queries combined in an appropried way.

**Table 8.6.** Correlation coefficients between "number of features" and "criteria execution time" per each rule and over the 50 models.

| Rule | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| *R*□ | 0.01 | 0.04 | 0.01 | 0.15 | 0.74 | 0.87 | 0.35 |

## 8.5 Tool Support

Several tools were developed in our research to support automated model transformation, integration and verification. The tools were built on Java and Prolog.

To transform feature models into constraint logic programs we used the SPLOT transformation API (Mendonca *et al.* 2009b). We have also built a tool based on ATL (Atlas Transformation Language) to implement transformation rules. Both transformation strategies are developed in Chapter 3 and details about the tools automating these strategies are presented in (Mazo *et al.* 2011e).

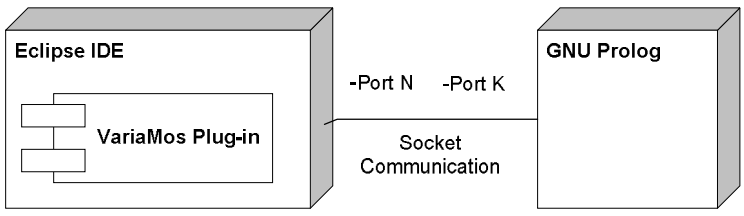
To transform Dopler models into constraint programs we used a navigation API provided by the developers of the Dopler language (Dhungana *et al.* 2010).

To check the conformance of feature models we used a tool developed in Prolog (cf. Appendix), which was executed in GNU Prolog (Diaz & Codognet 2001).

To manage product line models and their integration, to configure the connection with GNU Prolog and to implement the domain-specific verification algorithms proposed in this thesis, we developed an eclipse plug-in. The resulting tool, called VariaMos (Variability Models). VariaMos is can be accessed online at https://sites.google.com/site/raulmazo/

### 8.5.1. VariaMos

VariaMos uses the solver GNU Prolog as the executor engine of the verification operations. This connection with GNU Prolog is made using a client-server architecture through a socket connection as detailed in Figure 8.3. The architecture of VariaMos and its user interface to manage, integrate and execute the domain-specific verification proposed in this thesis, are presented in the next sections.




**8.3.** Communication chema of VariaMos with GNU Prolog.

As Figure 8.3 shows it, the VariaMos plug-in plays the role of client. Its goal is to:

- Control the flow of data and send the request to the GNU Prolog tool.
- Process the responses received from GNU-Prolog.
- Support the interaction with the end users by means of the user interface, in particular to manipulate PLM, and call verification functions.

The GNU Prolog tool plays the role of server. It is intended to:

- Wait for any request. It plays a passive role in the communication.
- Once a request is received, process it and then send the result to the client.
- Be transparent to the end user. The end user does not interact directly with the GNU Prolog server.

A screenshot of the user interface to configure the connection of VariaMos with the GNU Prolog solver is presented in Figure 8.4.

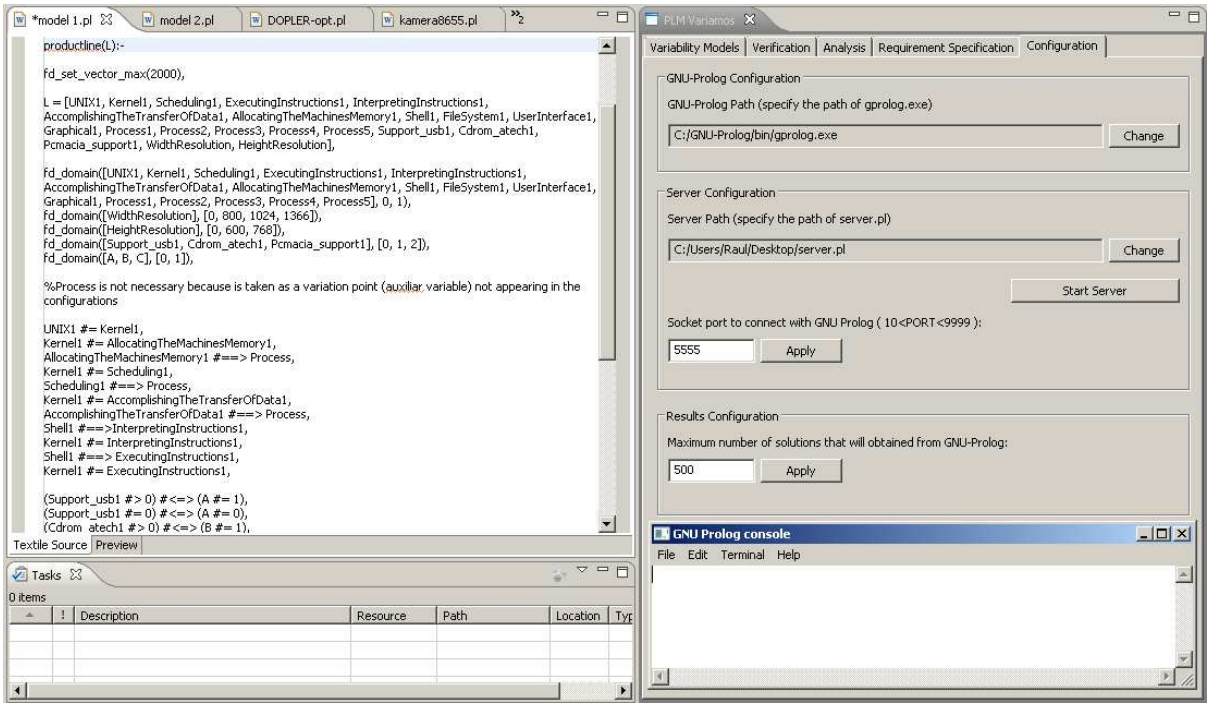



**Figure 8.4.** Screenshot of the configuration tag for the connection of VariaMos with GNU Prolog.

### Technical Architecture

VariaMos is thus composed of two packages, the MANIFEST.MF with the business rules (or Model) and VariaMos.jar with the classes that implement the view. The classes of these two packages use other Eclipse packages like jdt.jar, jface.jar, resources.jar and ui.jar. An overview of the relationships among these packages is presented in Figure 8.5.

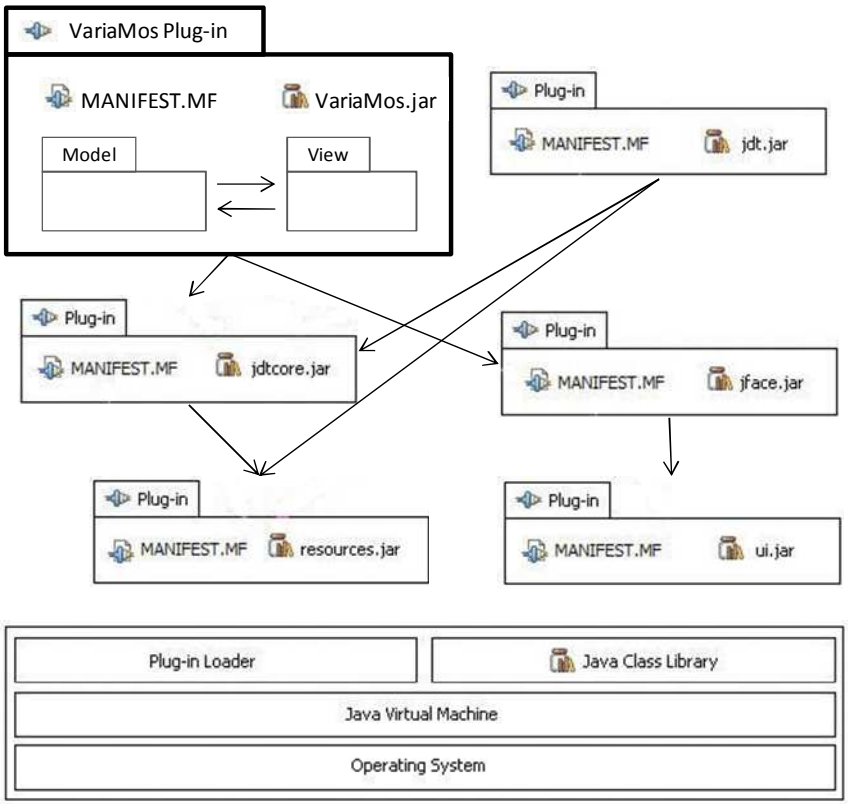



**Figure 8.5.** Architecture of the VariaMos Eclipse plug-in.

The functions of each VariaMos's package are presented as follows:

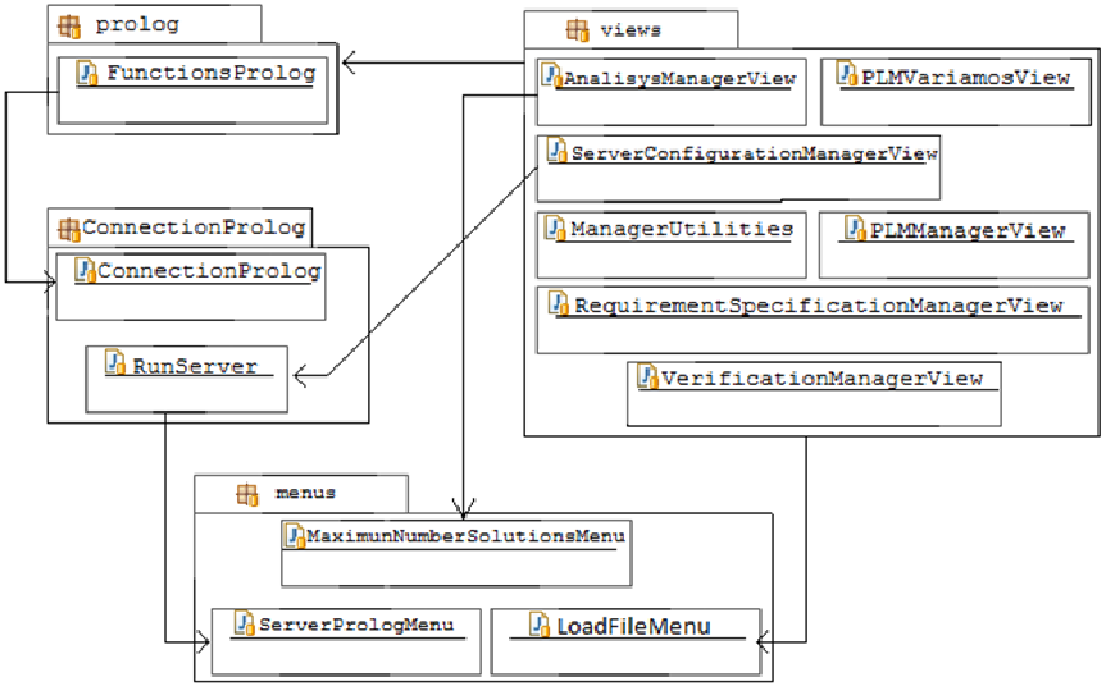
**jface.jar** gives the collection of tools to create and manage the components that belong to the user interface (UI). Jface works together with the Standard Widget Toolkit (SWT) library in order to manage the functionalities of components like text areas, fonts, windows and all the actions and objects provided by the library ui.jar. Jface and SWT are necessary in an eclipse plug-in environment because they permit defining the location of each UI component into the Eclipse workbench.

**Ui.jar** permits adding UI components and the corresponding set of actions into the plug-in. In addition, Ui.jar adds particular actions to the action bar and adding the plug-in view to the set of Eclipse's views.

**Jdt.jar**, it is the acronym of Java Development Tools, which is a collection of plug-ins that add the capabilities of a full-featured Java IDE to the Eclipse platform. Due to the fact that VariaMos was developed in Java, we used some particular functions of the plug-ins provided by the JDT library, especially **jdtcore.jar**, in order to create, debug, edit, compile, execute and interface the collection of Java programs used in VariaMos.

**Resource.jar:** it provides the necessary elements in order to manage the relationship between Eclipse and our plug-in VariaMos, as for example the position that the plug-in's UI will have in the Eclipse workspace.

Our plug-in, packaged in the file VariaMos.jar, contains 13 classes to manage the user interface, the execution of the domain-specific verification operations proposed in this thesis and the connection with GNU Prolog. The distribution of these classes in the corresponding packages is presented in Figure 8.6.




**8.6.** Packages diagram of VariaMos

The Java classes included in the *views* package are responsible for interaction with the *menus* package. This interaction shows to the user the changes made to the plug-in through the actions added in the Eclipse menu bar. The *view* package also uses the *prolog* package due to the fact that all the instructions that will be sent to the solver are generated in the *prolog* package. These instructions are generated in the *prolog* package according to the GNU Prolog syntax and then, they are sent to the solver by means of the methods provided by the built-in classes in the *connectionProlog* package. The *connectionProlog* package also executed the server file (server.pl) in the solver in order to establish a socket connection and communicate with the server by means of the facts provided in the server.pl file (cf. Appendix). The functions or facts, provided in the server.pl file, that the *connectionProlog*

package can execute in the solver are: *server* to assign a communication port, *server_next* to read a next instruction from the socket, *server_stop* to close the socket connection, *server_loop* to execute a collection of instruction passed as parameter and get the answers in other parameter, *server_exec* to execute a goal passed as parameter and get the answer in other parameter, *server_msg* to format a message, *server_read* to read a string from the socket and *server_write* to write a string in the socket.

**User Interface**

Figure 8.7 presents a screenshot of the product line models management tag provided in VariaMos. The left part shows the constraint programming representation of our UNIX running example (cf. Chapter 3). The right side is the user interface to all functions to manage the edition, transformation into CP and integration of product line models.

As the tabs show, other functions are available to support: verification, analysis, requirement specifications, configuration.

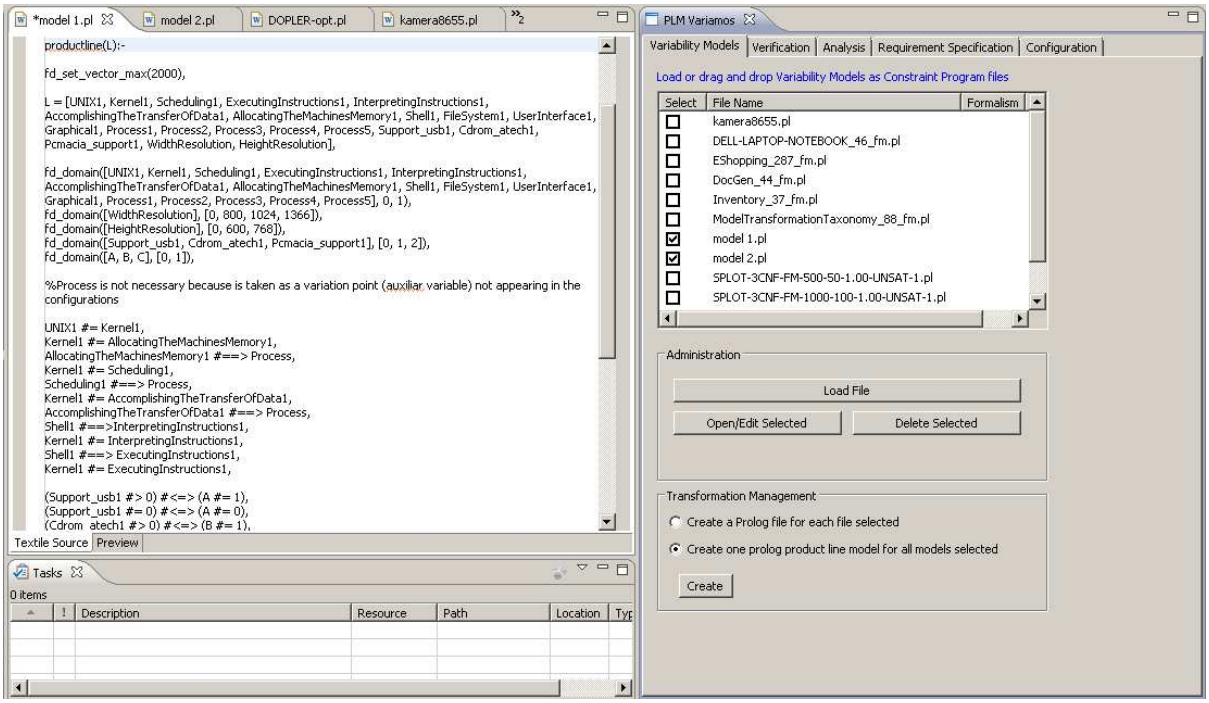



**Figure 8.7.** Screenshot of the PLMs management interface provided by VariaMos.

Figure 8.8 presents a screenshot of the main window of the VariaMos tool, once the verification operations “Richness or no false PLMs” and “dead variables” are executed over the first model of our running example (cf. Chapter 3).

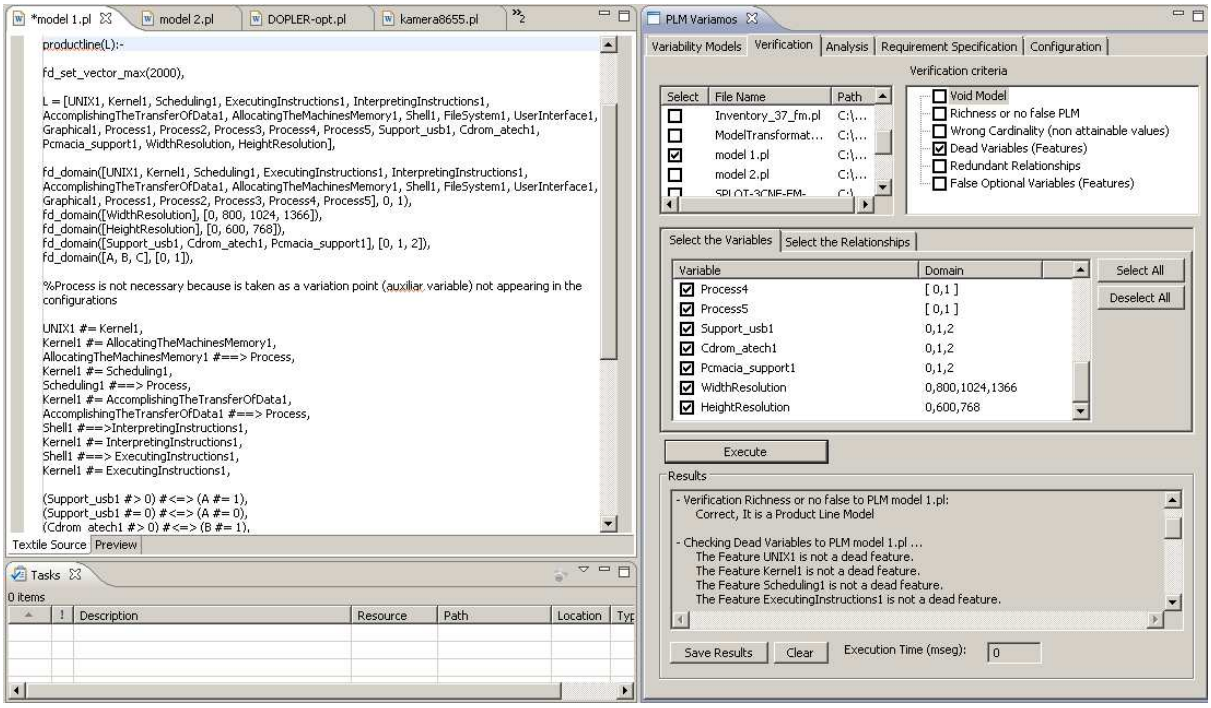



**Figure 8.8.** Screenshot of the domain-specific verification functions provided by VariaMos.

As the screenshot shows, the VariaMos tool allows to choose which verification to perform and on which model to perform it. The results of verification are shown at the bottom of the window together with the execution time. It is also possible to verify views on models, which makes it possible to explore progressively the validity of sub-PLMs (i.e. from a semantic point of view subspaces of the collections of products that can be generated from a PLM).

## 8.5.2. Conformance Checker of Feature Models

The conformance checker of feature models developed to validate this thesis is a tool that verifies if a feature model satisfies the constraints captured in the FMs metamodel. This tool was constructed in the GNU Prolog language (Diaz & Codognet 2001) and executed in the solver with the same name. The source code of this tool is presented in the Appendix section of this thesis. A screenshot of the user interface, once executed in GNU Prolog over the FM called `CocheEcologico_94_fm` (a feature model taken from SPLOT and corresponding to the PL of an ecologic car with `94` features), is presented in Figure 8.9. This screenshot shows how our FM conformance checker discovered six defects regarding the conformance criterion "Two features should not have the same name" that correspond to the rule number two in our tool. For instance, in this feature model, feature identified with numbers `17` and `61` have the same name "Blanco" and features identified with the numbers `19`, `59` and `69` have the same name "Negro".

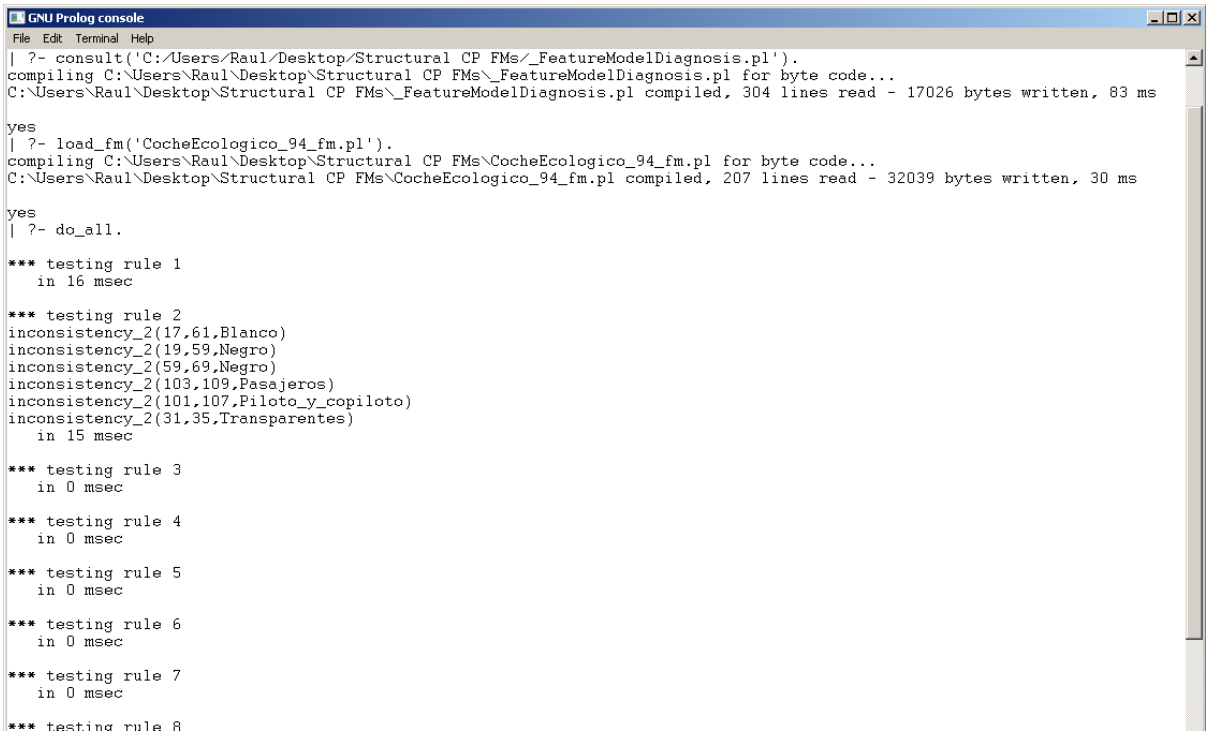



**Figure 8.9.** Conformance Checker of Feature Models

## 8.6 Comparison with FaMa

The VariaMos tool was compared with the state-of-the-art implementation of domain-specific verification. For this purpose, we selected FaMa, a Framework for AutoMated Analyses and verification of feature models integrating some of the most commonly used logic representations and solvers proposed in the literature such as BDD, SAT and CSP solvers (Trinidad *et al.* 2008b).

We selected FaMa to be compared to, because it is the only tool, to the best of our knowledge, that uses a constraint solver to execute the verification operations implemented in the tool and it is the type of solver that we also used to implement our approach.

We did not compare the execution time of *redundant relationships* and *false product line models (or richness)* operations because they are not implemented in FaMa. To make the comparison as fair as possible, we set up FaMa to use the Choco solver as the reasoner used because it is also based on Constraint Programming and FaMa implements, for this solver, all the operations we are interested in. It is worth noting that the verification operations *dead features, false-optional features* and *wrong cardinalities* are compared as a package and not individually due to the fact that FaMa does not compute the time separatly. The VariaMos versus FaMa comparison results is presented in Figure 8.10. The figure ares given with a logarithmic scale in the X axis for the sake of presentation.

VariaMos shows quite extensive performance gains over FaMa. In our tool, the execution time (CPU time) of operations dead features, false optional features and non attainable domains, goes from 14 milliseconds (ms) in models up to 50 features to 64 ms in models up to 100 features. As aforementioned, these operations are aggregated because they are implemented that way in FaMa, in other words, it is not possible to measure their performance separately. These same three functions are executed in 8829 ms in models up to 1000 features and 36587 ms in models up to 2000 features. In FaMa, the execution time of these three operations goes from 151 ms in models up to 50 features to 596 ms in models up to 100 features. The same three operations are executed in 28006 ms in models up to 500 features, in 192685 ms (3,2 minutes) in models up to 1000 features and in 1979458 ms (33 minutes) in models up to 2000 features. In the other operation (detection of void models) our approach is constant and always gives times lower than 1 second. In contrast, for that verification operation, the execution time in FaMa is linear. As we can see in Figure 8.10, the difference between the execution time in FaMa and VariaMos grows with the size of models, which means that our improvement increases more as larger are the feature models. In summary, the gain, in terms of time, when we execute the first three verification operation in FaMa and in Prolog is not in terms of *number of times faster*, but in computational complexity. According to the results depicted in Figure 8.10(1), it seems that the computational complexity to execute the first three verification operations is exponential in FaMa and polynomial in VariaMos. In the same way, the time to execute the verification operation *void feature model* seems to be polynomial in FaMa and linear in VariaMos as presented in Figure 8.10(2). It must be noted that in terms of the precision and recall, both approaches are equal, that is, both find exactly the same defects.

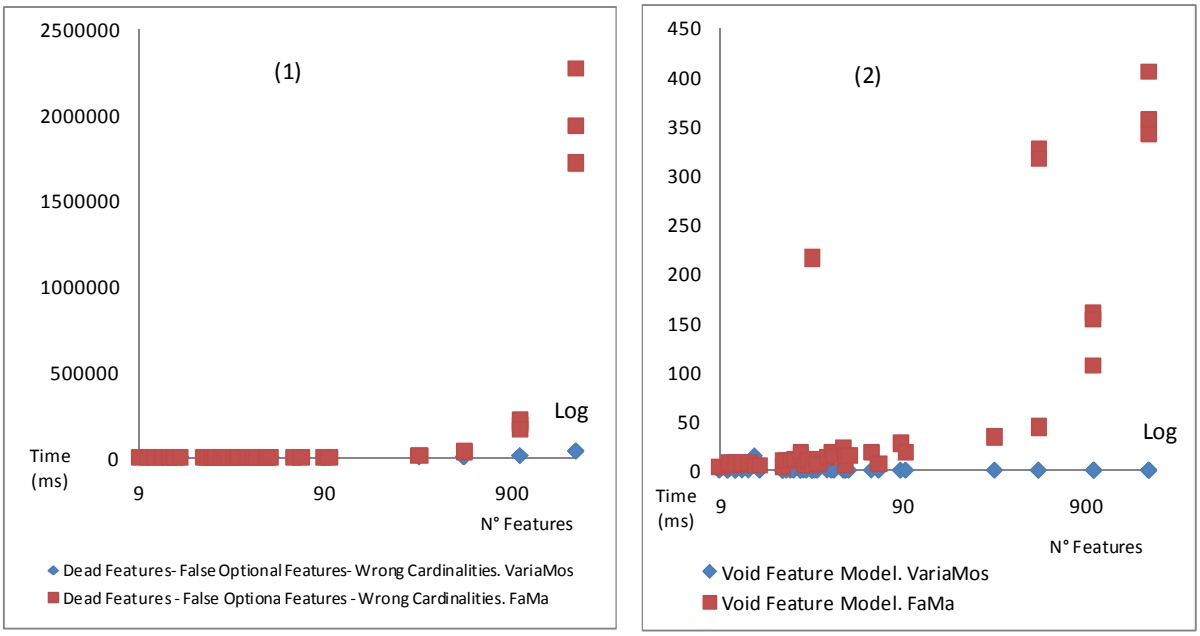



**Figure 8.10.** FaMa versus VariaMos (X axis is in a $Log_{10}$ scale).

## 8.7 Summary

This chapter has presented an evaluation of our verification approach using a corpus of 54 models specified in several languages and with sizes from 9 to 10000 artefacts. This chapter shows how the application of the verification approach, presented in earlier chapters, in the 54 PLMs gives sufficient evidence to support the hypothesis of this thesis. In particular:

- The verification approach can be used to verify product line models specified in different kind of languages and PL specified with only one or with several models.
- Our experiments show that the verification approach is correct, useful, and our tool implementation is fast and scalable.
- To the best of our knowledge, our conformance checking approach offers the first implementation of a FM conformance checker. The domain-specific verification approach considers more verification criteria than each one of the works found in the literature, offers an ordered way to verify FMs, and improves the computational scalability of PLM verification: (i) passing from verification of PLMs with 1000 artefacts at maximum in 41.67 minutes (dead features, false optional and wrong cardinalities in FaMa) to verification of PLMs up to 2000 artefacts (and even more) in 18 seconds in VariaMos for the same three verification criteria; and (ii) passing from verification of the not void (consistency) PLMs with 1200 artefacts in 64 seconds (before elimination of redundant artefacts) and 1900 artefacts in 64 seconds (after

elimination of redundant artefacts from the model) in Yan *et al*. (2009), to identification of not void PLM with 2000 artefacts in 8 seconds in VariaMos.

The tools developed in this thesis are used together accordint to the architecture depicted in Figure 8.11.

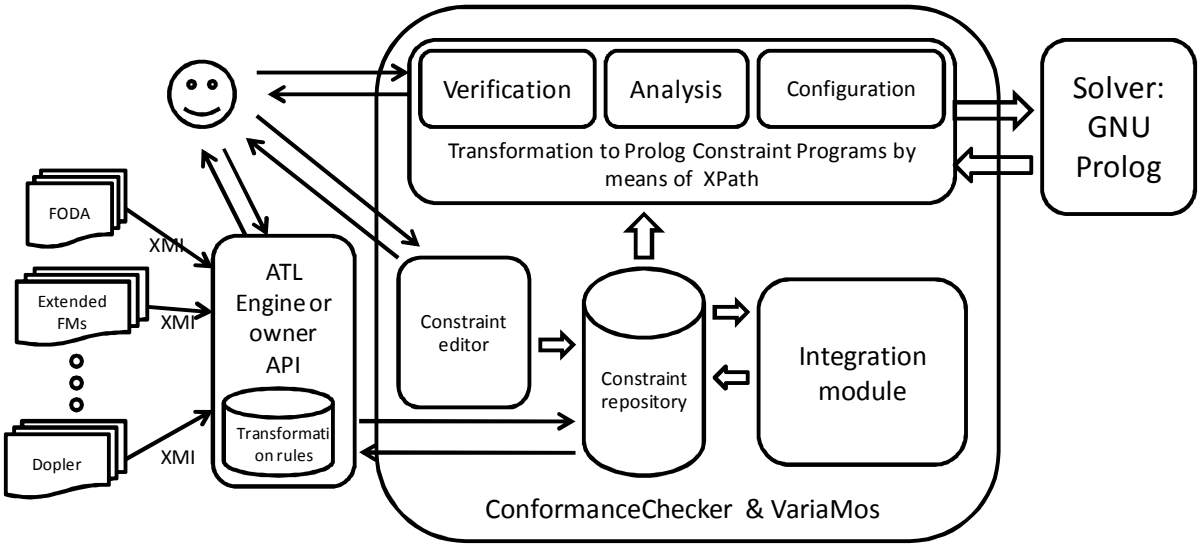



**Figure 8.11.** General architecture of the transformation, integration and verification tools.

Figure 8.11 relates in a unified architecture the different tools developed to validate the PLM verification approach proposed in this thesis. To verify PLMs, we first transform them into constraint (logic) programs by means of ATL rules or in-house applications (Application Programming Interfaces APIs) to navigate through the elements of PLMs. Once PLMs transformed, they can be edited, integrated into a single model or formatted into the GNU Prolog language. To format PLMs into GNU Prolog we used XPath to navigate through the tags of each model, initially represented as XMI (XML Metadata Interchange) files. The PLMs represented in the GNU Prolog language can be verified, analyzed and configured using the GNU Prolog solver. Our tools are connected to GNU Prolog by means of a socket that allows executing PLMs, executing verification rules and retrieving the results from the solver. Then, answers retrieved from the solver are treated by our tools and presented to the user.

# Chapter 9

# Conclusions and Future Research

The main objective of this thesis is to answer the research question: *How can product line models be automatically verified in a generic and reusable way?* To answer this question, this thesis proposes:

a. A state of the art in verification of product line models
b. A typology of verification criteria that classify and dispose the verification criteria found in literature according to the nature of criteria, and inside of each category, according to the order in which they should be executed. These two categories of verification criteria are called "conformance checking" and "domain-specific verification". The order indicates the sequence in which verification criteria should be executed according to their impact on the overall quality of the PLM and the logic sequence of the verification process.
c. Verification criteria formalized in first order logic and generic algorithms to implement them.
   - In the case of conformance checking, the generic algorithms should be adapted to particular metamodels.
   - In the case of domain-specific verification the algorithms reuse the precedent answers obtained from the solver in order to reduce the execution time of each criterion.
d. An approach to verify multi-model product lines.
e. A tool to automate and validate the verification approach.
f. A benchmark of 54 product line models with size up-to 10000 artefacts.
g. An evaluation and comparison of our implementation with another tool.

## 9.1. Conclusions

The proposals used to the answer the research question of this thesis are summarized in the next sub-sections.

### 9.1.1 State of the art in verification of product line models

As for as product line model verification, literature review shows:

a. Most of approaches existing in literature focus in verification of feature models, as presented in Table 2.1.
b. Table 2.1 shows that there is no approach that covers all the verification criteria, or all the PL formalisms.
c. Most of these criteria are overlapped, i.e., different names are used to refer to the same criterion.
d. There is no approach to verify multi-model product lines.
e. There is no comprehensive approach, i.e., an approach that handles all the criteria in a consistent way.
f. All the techniques we found in the literature transform PLMs in another formalism: sometimes a conjunctive normal form formula, other times in an if-then-else structure (i.e., BDD), also constraint satisfaction problem (CSP), OCL and in-house representations.

The following challenges arise from our literature review:

g. Can the verification approaches originally created for FMs also be used on other notations? And if it is possible, then, how to do that?
h. How to verify a PLM independently of the language in which the model at hand is represented and for any verification criteria? There is no generic approach to verify product lines models.
i. How to verify PLMs in a scalable way?

### 9.1.2 Typology of verification criteria

The typology of verification criteria emphasizes the difference between domain-specific and conformance defects. The outcomes of the typology are multiple:

j. It classifies the criteria from a semantic perspective, allowing the identification of similarities and differences among the criteria.
k. The typology helps the identification of some defects for which no verification criterion is available in the literature.
l. The classification behind the typology produces a standard and reusable approach to verify PLMs.

m. The typology can be used to select the criteria that one wants to use to verify a PLM according to the impact that these criteria have or the expected quality level of a particular PLM.
n. Due to the fact that not all the verification criteria have the same impact, they have neither the same priority nor the same execution order; the typology can be used to guide the verification process and to propose specific defects management strategies and policies.

### 9.1.3 Conformance Checking, a Generic and Adaptable Verification Approach

From the point of view of conformance checking, this thesis proposes a collection of verification criteria to check conformance of product line models with their corresponding metamodels. This approach is generic and adaptable. Genericity was obtained by developing these criteria from a generic metamodel that entails common structures of several PL metamodels. Adaptability is obtained by specialising the generic metamodel with a specific metamodel. The conformance checking approach needs a transformation of the PLMs' abstract syntax into constraint logic programs. Thus, the abstract syntax of PLMs, once represented as a collection of logic facts, can be evaluated against a collection of conformance criteria taken from the corresponding metamodel. It is worth noting that some of these criteria are generic to several PLM metamodels and other criteria are particular to the metamodel at hand. These conformance criteria are implemented as CLP queries over PLMs and intend to find the elements of the PLMs not satisfying the conformance criteria.

The conformance checking approach is fully automated in a Prolog-based tool. This tool is used to execute a series of experiments to test the feasibility, the performance and the computational scalability of the conformance checking approach over feature models. As presented in Chapter 8, the experiment supports the fact that the approach presented in this thesis to check conformance is correct, useful, and our tool implementation is fast and scalable to PLMs up-to 10000 variables.

### 9.1.4 Domain-specific Verification of PLMs, a Generic Verification Approach

This thesis presents a generic and reusable approach to verify domain-specific properties of PLMs against a collection of verification criteria. To use these verification criteria the semantics of the PLM at hand should be represented as a constraint program. While the

variables of the constraint programs specify what can vary from one configuration to another one, the constraints express, under the form of restrictions, what combinations of values are permitted in the products. Once PLMs represented as CPs, this thesis proposes the use of CP solvers in order to execute the verification criteria. The approach was applied to extended feature models, Dopler models and constraint-based PLMs, which gives an idea of the genericity of this approach.

The PLM verification approach proposed in this thesis is validated against a series of experiments (cf. Chapter 8) to evaluate the hypotheses raised in Chapter 1. The experiment supports the fact that the approach proposed in this thesis to verify domain-specific properties of PLMs is correct, useful, and our tool implementation is fast and scalable to PLMs up-to 2000 variables.

### 9.1.5 Verification of Multi-model Product Lines

From the point of view of multi-model product line specification, this thesis proposes an approach that captures in a unified representation the various models of the PL. As a result, domain and application engineering activities such as PLM verification and analysis, or product configuration will be facilitated. The multi-model verification approach presented in this thesis consists of re-using the verification approach proposed for standalone PLM in order to verify multi-model product line specifications. This approach was validated in the case of feature models (where several FMs are used to specify a PL and they are integrated by means of five integration strategies) and Dopler models (where a PL is specified by means of a decision and an asset models integrated by inclusion rules). However, this thesis does not consider the case where a PL is specified by several models, each one specified in a different modelling notation. This case is proposed in Chapter 7 as a future work.

### 9.1.6 Automation and Validation of the Verification Approach

Several tools were developed in our research to support automated model transformation, integration and verification. To transform feature models into constraint logic programs we (i) used the SPLOT transformation API (Mendonca *et al.* 2009b), and (ii) we build a tool based on ATL (Atlas Transformation Language) transformation rules. To transform Dopler models into constraint programs we used a navigation API provided by the Software Engineering and Automation Institute of the Johannes Kepler University, Austria. To check the conformance of feature models we build a tool in GNU Prolog (Diaz & Codognet 2001). To manage product line models, their integration, to configure the connection with GNU

Prolog and use the domain-specific verification criteria proposed in this thesis, we developed a tool called VariaMos (Variability Models). The algorithms in which these tools are based are presented in the Apendix.

The verification approach proposed in this thesis was evaluated with a collection of 54 product line models with sizes up-to 10000 artefacts. The verification approach can be used to verify product line models specified in different kinds of languages and PLs specified with only one or with several models.

Our experiments show that the verification approach is correct, useful, and our tool implementation is fast and scalable. The conformance checking approach offers the first implementation of a FM conformance checker. The domain-specific verification approach considers more verification criteria than the works found in the literature, offers an ordered way to verify FMs, and improves the computational scalability of PLM verification: (i) passing from verification of PLMs with 1000 artefacts at maximum in 41.67 minutes (dead features, false optional and wrong cardinalities in FaMa) to verification of PLMs up to 2000 artefacts (and even more) in 18 seconds in VariaMos for the same three verification criteria; and (ii) passing from verification of the not void (consistency) PLMs with 1200 artefacts in 64 seconds (before elimination of redundant artefacts) and 1900 artefacts in 64 seconds (after elimination of redundant artefacts from the model) in Yan *et al*. (2009), to identification of not void PLM with 2000 artefacts in 8 seconds in VariaMos.

## 9.2. Future Research Agenda

A desirable aspect of any research is that in addition to providing solutions to initial issues or questions, it should identify new research topics that would allow researchers to further work to eventually produce more useful knowledge and progress. This section presents some research directions and required additional work on verification of product line models and also some particular research directions in conformance checking, domain specific verification of PLMs and further validation of the approach presented in this thesis.

### 9.2.1 Further Challenges in Verification of Product Line Models

Two general frameworks (proposed by Finkelstein *et al*. 1996 and by Nuseibeh *et al*. 2000) describe the process of "inconsistency management". These processes can be adapted to the context of PLM verification and in that way complement the verification approach presented in this thesis. These approaches share the premise that the process of managing inconsistencies includes activities for detecting, diagnosing and handling them. It is probably

that the generic PLM verification approach presented in this thesis (which deals with detection of defects and, in the most of cases with identification of the defects' source) would evolve following the "inconsistency management process" proposed by Finkelstein *et al*. (1996) and Nuseibeh *et al*. (2000). According to them, the verification management process, after detection of defects, must be conducted as follows:

### *A. Diagnosis of defects*

This activity is concerned with the identification of the source, the cause and the impact of each defect found in the previous detection stage. Adapting the definitions of "source of an inconsistency" given by Nuseibeh *et al*. (2000) and by Spanoudakis & Zisman (2001) to the domain of product lines, the source of a PLM's defect is the set of elements of the model which have been used in the construction of the argument that shows that the models violate a verification rule. In the same way, the cause of a PLM's defect is defined as the conflict(s) in the perspectives and/or the goals of the stakeholders which are expressed by the elements of the models that give rise to the defect. The impact of a defect is defined as the consequences that a defect has for a system, e.g., in terms of performance and evolvability.

The verification approach proposed in this thesis also deals with identification of defects' source, except for three verification criteria: Non-redundant dependencies, Non-void and Non-false PLMs. For all the other verification criteria, our approach identifies the elements participating in the model's construct that violates the corresponding verification rule. The three cases where the source of defects is not known are part of the future work.

The source and the cause of a defect play an important role in the defect management process since they can be used to determine what options are available for resolving or ameliorating them and the cost and the benefits of the application of each of these options. Establishing the impact of a defect category in qualitative or quantitative terms is also necessary for deciding with what priority the category of defect has to be handled and for evaluating the risks associated with the actions for handling each category. The classification of the verification criteria by means of our typology, is a first effort to determine the impact of each verification criterion.

### *B. Handling of defects*

According to van Lamsweerde *et al*. (1998), Robinson (1997) and Spanoudakis & Zisman (2001) "handling [defects] is a central stage in the process of [defect] management". This stage, adapted to the domain of product lines, concerns with the following activities:

(i) the identification of the possible actions for dealing with verification criteria,
(ii) the evaluation of the cost and the benefits that would arise from the application of each these actions,
(iii) the evaluation of the risks that would arise from not resolving the defect and in general, the verification criterion to with the defect is associated with, and
(iv) the selection of one of the actions to execute.

***C. Specification and application of a defect management policy***

To manage a defects' detection and correction process in a coherent and effective way it is necessary to have a policy about the defect management that should be applied to a particular project (Finkelstein *et al*. 1996). According to Spanoudakis & Zisman (2001), this policy must specify:

(i) the agent(s) that should be used to verify the product lines models (and for each particular PLM in multi-model product lines).
(ii) the verification criteria that should be checked against the models
(iii) the circumstances that will trigger the execution of each verification criterion
(iv) the mechanisms that should be used for diagnosing each defect according to the criterion to which it belongs and the circumstances in which the defect happened.
(v) the mechanisms that should be used for assessing the impact of each verification criterion and the circumstances that should trigger this activity
(vi) the mechanisms that should be used for assessing the cost, benefits and risks associated with different verification criteria handling options, and
(vii) the stakeholders who would have responsibility for handling defects and responsible of the models' quality.

***D. A transversal tracking activity.***

This activity is charged to record what happens in each stage and activity of the defect management process. Keeping track of what has happened in the process makes the understanding of the findings, the decisions and the actions taken by those who might need to use or refer to the PLMs in subsequent stages of the development life-cycle of the PL easier.

### 9.2.2 Future Work in Conformance Checking

Future works in conformance checking of PLMs include the following items:

(i) Implement an incremental checker with rule scopes such as the one proposed by Egyed (2006). It is expected that this improvement will reduce the execution time of some of the conformance rules.

(ii) Devise the classification of conformance criteria according to their severity and complexity.

(iii) Another important research direction is about man-machine interface and usability of the results obtained from verification. This future work proposition is about how to best present feedback to users and how improve the ergonomic of the implementations developed in this thesis when it comes to dealing with dozens of models that contain thousands of artefacts and dependencies. The efficiency of the verification approach depends highly on how to navigate in models and verification results.

(iv) Another future work consists of investigating how to automatically generate specific conformance criteria and their associated implementation (conformance rule, i.e., criteria that cannot be generated from the generic metamodel) and if possible optimize them automatically.

### 9.2.3 Future work in Domain-specific Verification

Future works in domain-specific verification of PLMs include the following items.

(i) Extend the set of available verification criteria (e.g., regarding the evolution and the temporal properties of PLMs). To guarantee defect-free product line models, the collection of considered defects must be as exhaustive as possible.

(ii) Classify the verification criteria according to their severity, complexity and implementability. With such a classification, it should be possible to guide the verification stage, propose the corresponding fixing actions according to each category of defect and improve the execution time of each verification operation.

(iii) Exploration of model-checking techniques, further to the ones developed by Zhang *et al*. (2004) and Lauenroth *et al*. (2010), which are in an initial stage.

### 9.2.4 Future Work in Evaluation

Even if our experiments showed very promising results to usability, recall and scalability of VariaMos, some important assumptions must be kept in mind and several questions still remain open.

The most important assumption is that using our approach, engineers could verify product line models specified in other languages. Genrality is not proven per se: we can only guarantee that the proposed approach cover several formalisms (FODA, DOPLER, extended feature models). In so far as generality is concerned, we cannot guarantee that users won't come up with their own verification criteria (and require to use other CP solvers than GNU Prolog) Some of the future works that would improve the validation of our approach are:

- Address how to best visualize the defects found with our product line verification approach. Much of this problem has to do with human-computer interaction and further studies in this topic would be part of the future work.
- Another future work consists in adressing the downstream economic benefits of the verification approach presented in this thesis. For example, one could raise the question how does fast detection of defects really benefit software engineering at large? How much does it cost to fix a defect early on as compared to later on? These complex issues have yet to be investigated and measured from a socio-economical perspective.
- Even if the performance and scalability of our experiment with Dopler models indicate scalability and good performance of our verification approach, complementary tests with larger real size models are necessary.
- How do the users explore models to verify them? It would be interesting to log verification activities to learn which criteria are verified first, if there is a systematic order, and even if verification is systematically complete. Such logs would also be useful to build empirically a process model of PLM verification.

# Appendix A: Publications

This appendix contains the titles and venues of the publications the PhD research has produced so far:

Salinesi C, Mazo R. Defects in Product Line Models and how to Identify them. "Software Product Lines - The Automated Analysis", edited by Abdelrahman Elfaki, InTech editions, ISBN 979-953-307-700-9. To appear in January 2012.

Mazo R., Salinesi C, Djebbi O., Diaz D., Lora-Michiels A. Constraints: the Heard of Domain and Application Engineering in the Product Lines Engineering Strategy. International Journal of Information System Modeling and Design IJISMD (accepted). Sweden, July 2011.

Mazo R., Salinesi C., Diaz D. Abstract Constraints: A General Framework for Solver-Independent Reasoning on Product Line Models. Accepted on INSIGHT - Journal of International Council on Systems Engineering (INCOSE), November 2011.

Mazo R., Salinesi C., Diaz D., Lora-Michiels A. Transforming Attribute and Clone-Enabled Feature Models Into Constraint Programs Over Finite Domains. 6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Springer Press, Beijing–China, 8-11 June 2011.

Mazo R., Lopez-Herrejon R., Salinesi C., Diaz D., Egyed A. Conformance Checking with Constraint Logic Programming: The Case of Feature Models. In 35th Annual International Computer Software and Applications Conference (COMPSAC), IEEE Press, Munich-Germany, 18-22 July 2011. **Best Paper Award.**

Salinesi C., Mazo R., Djebbi O., Diaz D., Lora-Michiels A. Constraints: the Core of Product Line Engineering. Fifth IEEE International Conference on Research Challenges in Information Science (RCIS), IEEE Press, Guadeloupe-French West Indies, France, May 19-21 2011. **Best Paper Award.**

Mazo R., Grünbacher P., Heider W., Rabiser R., Salinesi C., Diaz D. Using Constraint Programming to Verify DOPLER Variability Models. 5th International Workshop on Variability Modelling of Software-intensive Systems (VaMos'11), Namur-Belgium, January 27th-29th, 2011.

Salinesi C., Mazo R., Diaz D., Djebbi O. Solving Integer Constraint in Reuse Based Requirements Engineering. 18th IEEE International Conference on Requirements Engineering (RE'10). Sydney - Australia. September-October 2010.

Salinesi C., Mazo R., Diaz D. Criteria for the verification of feature models, In 28th INFORSID Conference, Marseille - France, May 2010.

Lora-Michiels A., Salinesi C., Mazo R. A Method based on Association Rules to Construct Product Line Model. 4th International Workshop on Variability Modelling of Software-intensive Systems "Celebrating 20 Years of Feature Models". Linz-Austria, Janvier 2010.

Lora-Michiels A., Salinesi C., Mazo R. The Baxter Return of Experience on the Use of Association Rules to Construct its Product Line Model. Journée SPL, Lignes de produits logiciels et usines logicielles. Nantes-France. Octobre 2009.

Salinesi C., Rolland C., Mazo R. VMWare: Tool Support for Automatic Verification of Structural and Semantic Correctness in Product Line Models, International Workshop on Variability Modelling of Software-intensive Systems (VaMoS), Sevilla-Spain, pp. 173 - 176, January 2009.

Mazo R. Aperçu d'une Méthode Automatisée basée sur des Contraintes Génériques pour la Vérification de Modèles Multi-vues de Lignes de Produits/ Overview of an Automated Method based on Generic Constraints for Verifying Multi-View Product Line Models. Poster in Forum Academie-industrie AFIS, Bordeux 2,3 Décembre 2010

Salinesi C, Diaz D., Mazo R., Djebbi O. Spécification d'Exigences dans le Contexte de Lignes de Produits. Journée Action IDM - INFORSID "Exigence, Traçabilité et Co-conception dans les processus de développement". Paris-France. Octobre 2009.

Salinesi C., Diaz D., Djebbi O., Mazo R. Exploiting the Versatility of Constraint Programming over Finite Domains to Integrate Product Line Models. Poster in 17th IEEE International Conference on Requirements Engineering (RE'09). Atlanta-USA. Septembre 2009.

Salinesi C., Rolland C., Diaz D., Mazo R. Looking for Product Line Feature Models Defects: Towards a Systematic Classification of Verification Criteria. Poster in 17th IEEE International Requirements Engineering Conference (RE'09). Atlanta-USA, September 2009.

Mazo R. Processus pour la Vérification et Validation de Modèles de Lignes de Produits. Porter in 27^th Francophone Conference on Information Systems and Data Bases INFORSID, Toulouse-France. May 2009.

# Appendix B: Implementation Details

Several tools were developed in this thesis in order to validate our approach and its associated hypothesis. The most of these tools were presented along in this thesis. In the next section we provide the algorithms or relevant code source of each one of these tools. The execution files of each tool are available to download at https://sites.google.com/site/raulmazo/

## 1. Representation of other variability languages as constraint programs

The semantics of a product line model can be specified as a constraint program (Salinesi *et al.* 2010b, Mazo *et al.* 2011a, 2011e) by means of: (i) a set of variables $X=\{x_1,...,x_n\}$; (ii) for each variable $x_i$, a finite set $D_i$ of possible values (its domain); and (iii) a set of constraints restricting the values that they can simultaneously assume. A variable in a PLM has a domain of values, and the result of the configuration process is to provide it a value. This representation of the semantics of a PLM is very similar to the representation of a Constraint Satisfaction Problem (CSP), which is defined as a triple $(X, D, C)$, where $X$ is a set of variables, $D$ is a set of domains, and $C$ a set of constraints restricting the values that the variables can simultaneously take (cf. Section 1). Indeed, we have shown in Section 4-A how to represent the syntax and the semantics of feature models as constraint logic programs and constraint programs, respectively. However, not only feature models can be represented by means of constraints. Our experience and related works (Kang *et al.* 2002, Riebisch *et al.* 2002, Mannion 2002, Von der Maßen & Lichter 2002, Czarnecki *et al.* 2005, Benavides *et al.* 2005c, Pohl *et al.* 2005, Ziadi 2004, Korherr & List 2007, White *et al.* 2009, Boucher *et al.* 2010, Dhungana *et al.* 2010) show that PLMs can also represented in other notations with the aim of verify and analyse them. Thus, we proposed in this thesis a collection of representation patterns of the semantics of PLMs into constraint programs. Tables 10.1 to 10.5 compile the constructs, and its corresponding CP representations, of the most popular languages used to specify PLMs.

**Table 10.1.** Compilation of the feature-based languages' constructs and the corresponding representation as CPs.

| **Constructor and domains vs. Languages** | **FODA-like models** (Kang *et al*., 1990, 2002) | **Feature models with cardinalities** (Riebisch *et al*. 2002, Czarnecki *et al*. 2005) **and attributes** (Streitferdt *et al*. 2003, Benavides *et al*. 2005c, White *et al*. 2009) |
|---|---|---|
| **Root.** The root element must be selected in all the configurations. | If {Root} ∈ {true, false} then Root = true<br>If {Root} ∈ {0, 1} then Root = 1 | If {Root} ∈ {true, false} then Root = true<br>If {Root} ∈ ℤ then Root ≥ 1 |
| **Optional.** If the father element is selected, the child element can but needs not be selected. Otherwise, if the child element is selected, the father element must as well be selected. | If {Father, Child} ∈ {true, false} then Child ⇒ Father<br>If {Father, Child} ∈ {0, 1} then Father ≥ Child | If {Father, Child} ∈ {true, false} then Child ⇒ Father<br>If {Father, Child} ∈ {0, 1} then Father ≥ Child<br>If {Father, Child} ∈ ℤ then Child ≥ 1 ⇒ Father ≥ 1 |
| **Mandatory.** If the father element is selected, the child element must be selected as well and vice versa. | If {Father, Child} ∈ {true, false} then Father ⇔ Child<br>If {Father, Child} ∈ {0, 1} then Father = Child | If {Father, Child} ∈ {true, false} then Father ⇔ Child<br>If {Father, Child} ∈ {0, 1} then Father = Child<br>If {Father, Child} ∈ ℤ then Child ≥ 1 ⇔ Father ≥ 1 |
| **Requires (includes)**. If the requiring element is selected, the required element(s) has(have) to be selected as well, but not vice-versa. | If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required | If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required<br>If {Requiring, Required} ∈ ℤ then Requiring ≥ 1 ⇒ Required ≥ 1 |
| **Exclusion.** Indicates that both excluded elements cannot be selected in one product configuration. | If {Excluding, Excluded} ∈ {true, false} then Excluding ⊕ Excluded<br>If {Father, Child} ∈ {0, 1} then Excluding + Excluded ≤ 1 | If {Excluding, Excluded} ∈ {true, false} then Excluding ⊕ Excluded<br>If {Father, Child} ∈ {0, 1} then Excluding + Excluded ≤ 1<br>If {Father, Child} ∈ ℤ then Excluding * Excluded = 0 |
| **Alternative/xor-decomposition**. A set of child elements are defined as alternative if only one element can be selected when its parent element is part of the product. | If {Father, Child1, …, ChildN} ∈ {true, false} then: (Child1⇔(¬Child2 ⋀...⋀ ¬ChildN ⋀ Father) ⋀ Child2 ⇔ (¬Child1 ⋀...⋀ ¬ChildN ⋀ Father) ⋀ ChildN ⇔ (¬Child1 ⋀...⋀ ¬ChildN-1 ⋀ Father)) | |
| **Or-Relation.** A set of child elements are defined as an or-relation if one or more of them can be included in the products in | If {Father, Child1, …, ChildN} ∈ {true, false} then: Father ⇔ Child1 ∨...∨ ChildN | |

| | | |
|---|---|---|
| which its parent element appears. | | |
| **Group cardinality**. Cardinality determines how many variants (with the same father) may be chosen, at least M and at most N of the group. Besides, if one of the children is selected, the father element must be selected as well. | | If {Father, Child1, …, ChildN} ∈ {0, 1} then<br>Father ≥ Child1 ⋀ …⋀ Father ≥ ChildN ⋀<br>M*Father ≤ Child1+...+ChildN ≤ N*Father |
| A **feature cardinality** is represented as a sequence of intervals [min..max] determining the number of instances of a particular feature that can be part of a product. | | If {Father, Clone1, …, CloneN} ∈ { 0, 1} then:<br>Clone1 ⇒ Father ⋀ ...⋀ CloneN ⇒ Father ⋀<br>Father ⇒ (M ≤ Clone1 + … + CloneN ≤ N) |
| **Attribute.** An attribute is a variable associated to a reusable element. | | value ∈ Domain ⋀ Attribute = value ⋀<br>ReusableElement⇔Attribute >0 |

**Table 10.2.** Compilation of the OVM and TVL's constructs and the corresponding representation as CPs.

| **Constructor and domains vs. Languages** | **Orthogonal Variability Models (OVM)** (Pohl *et al*. 2005) | **Textual Variability Language (TVL)** (Boucher *et al*. 2010), (Classen *et al*. 2011) |
|---|---|---|
| **Root.** The root element must be selected in all the configurations. | | root Element |
| **Dependency/and-decomposition: operator allOf.** The selection of the children depends of the selection of the father element and vice versa | | Father ⇔ (Child1 ⋀…⋀ ChildN) |
| **Optional.** If the father element is selected, the child element can but needs not be selected. Otherwise, if the child element is selected, the father element must as well be selected. | If {Element1, Element2} ∈ {true, false} then Element2 ⇒ Element1<br>if {Element1, Element2} ∈ {0, 1} then Element1 ≥ Element2<br>If {Element1, Element2} ∈ ℤ then Element2 ≥ 1 ⇒ Element1 ≥ 1 | Child ⇒ Father |
| **Mandatory.** If the father element is selected, the child element must be | If {Element1, Element2} ∈ {true, false} then Element1 ⇔ Element2 | Father ⇔ Child |

| | | |
|---|---|---|
| selected as well and vice versa. | If {Element1t, Element2} ∈ {0, 1} then Element1 = Element2<br>if {Element1, Element2} ∈ ℤ then Element2 ≥ 1 ⇔ Element1 ≥ 1 | |
| **Requires (includes)**. If the requiring element is selected, the required element(s) has(have) to be selected as well, but not vice-versa. | If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required<br>If {Requiring, Required} ∈ ℤ then Requiring ≥ 1 ⇒ Required ≥ 1 | If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required |
| **Exclusion.** Indicates that both excluded elements cannot be selected in one product configuration. | If {Excluding, Excluded} ∈ {true, false} then Excluding ⊕ Excluded<br>If {Father, Child} ∈ {0, 1} then Excluding+Excluded ≤ 1<br>If {Father, Child} ∈ ℤ then Excluding * Excluded = 0 | |
| **Alternative/xor-decomposition**. A set of child elements are defined as alternative if only one element can be selected when its parent element is part of the product. | | If {Father, Child1, …, ChildN} ∈ {0, 1} then:<br>(Child1⇔(¬Child2 ⋀...⋀ ¬ChildN ⋀ Father) ⋀ Child2 ⇔ (¬Child1 ⋀...⋀ ¬ChildN ⋀ Father) ⋀ ChildN ⇔ (¬Child1 ⋀...⋀ ¬ChildN-1 ⋀ Father)) |
| **Or-Relation.** A set of child elements are defined as an or-relation if one or more of them can be included in the products in which its parent element appears. | | If {Father, Child1, …, ChildN} ∈ {0, 1} then:<br>Father ⇔ Child1 ∨...∨ ChildN |
| **Group cardinality**. Cardinality determines how many variants (with the same father) may be chosen, at least M and at most N of the group. Besides, if one of the children is selected, the father element must be selected as well. | If {VariationPoint, Variant1, …, VariantN} ∈ {0, 1} then VariationPoint ≥ Variant1 ⋀ …⋀ VariationPoint ≥ VariantN ⋀ M*VariationPoint ≤ Variant1+...+ VariantN ≤ N*VariationPoint | If {Father, Child1, …, ChildN} ∈ {0, 1} then<br>Father ≥ Child1 ⋀ …⋀ Father ≥ ChildN ⋀ M*Father ≤ Child1+...+ChildN ≤ N*Father |
| **Individual cardinality** is represented as a sequence of intervals [min..max] determining the number of instances of a particular feature that can be part of | If {Father, Clone1, …, CloneN} ∈ { 0, 1} then:<br>Clone1 ⇒ Father ⋀ ...⋀ CloneN ⇒ Father ⋀ Father ⇒ (M ≤ Clone1 + … + CloneN ≤ N) | If {Father, Clone1, …, CloneN} ∈ { 0, 1} then:<br>Clone1 ⇒ Father ⋀ ...⋀ CloneN ⇒ Father ⋀ Father ⇒ (M ≤ Clone1 + … + CloneN ≤ N) |

| | | |
|---|---|---|
| a product. | | |
| **Attribute.** An attribute is a variable associated to a reusable element. | | Attribute ∈ { integer, real, boolean, enumeration} ⋀ Attribute = value ⋀ ReusableElement ⇔Attribute>0 |

**Table 10.3.** Compilation of the Class-based and Use case-based variability languages' constructs and the corresponding representation as CPs.

| **Constructor and domains vs. Languages** | **Class-based PLMs** (Ziadi 2004; Korherr & List 2007) | **Use case-based PLMs** (Van der Maßen & Lichter 2002) |
|---|---|---|
| **Optional.** If the father element is selected, the child element can but needs not be selected. Otherwise, if the child element is selected, the father element must as well be selected. | If {Element1, Element2} ∈ {true, false} then Element2 ⇒ Element1<br>If {Element1, Element2} ∈ {0, 1} then Element1 ≥ Element2 | If {Element1, Element2} ∈ {true, false} then Element2 ⇒ Element1<br>If {Element1, Element2} ∈ {0, 1} then Element1 ≥ Element2 |
| **Mandatory.** If the father element is selected, the child element must be selected as well and vice versa. | If {Element1, Element2} ∈ {true, false} then Element1 ⇔ Element2<br>If {Element1t, Element2} ∈ {0, 1} then Element1 = Element2 | If {Element1, Element2} ∈ {true, false} then Element1 ⇔ Element2<br>If {Element1t, Element2} ∈ {0, 1} then Element1 = Element2 |
| **Requires (includes)**. If the requiring element is selected, the required element(s) has(have) to be selected as well, but not vice-versa. | If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required<br>If {Requiring, Required} ∈ ℤ then Requiring ≥ 1 ⇒ Required ≥ 1 | If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required<br>If {Requiring, Required} ∈ ℤ then Requiring ≥ 1 ⇒ Required ≥ 1 |
| **Group cardinality**. Cardinality determines how many variants (with the same father) may be chosen, at least `M` and at most `N` of the group. Besides, if one of the children is selected, the father element must be selected as well. | | If {Father, Child1, …, ChildN} ∈ {0, 1} then<br>Father ≥ Child1 ⋀ …⋀ Father ≥ ChildN ⋀<br>M*Father ≤ Child1+...+ChildN ≤ N*Father |
| **Individual cardinality** is represented as a sequence of intervals [`M..N`] determining the number of instances of a particular reusable element that can be part of a product. | If {FatherClass, Clone1, …, CloneN} ∈ { 0, 1} then:<br>Clone1 ⇒ FatherClass ⋀ ...⋀ CloneN ⇒ FatherClass ⋀<br>FatherClass ⇒ (M ≤ Clone1 + … + CloneN ≤ N) | |

**Table 10.4.** Compilation of the Dopler and CEA variability languages' constructs and the corresponding representation as CPs.

| **Constructor and domains vs. Languages** | **Dopler variability language** (Dhungana *et al*. 2010) | **CEA - variability language** |
|---|---|---|
| **Root/Visibility Condition.** The root decision must be solved in all the configurations. | Decision = true ∨ Decision = false | |
| **Mandatory.** If the father element is selected, the child element must be selected as well and vice versa. | | If {Element1, Element2} ∈ {true, false} then Element1 ⇔ Element2<br>If {Element1, Element2} ∈ {0, 1} then Element1 = Element2 |
| **Requires/Decision Effects/ Inclusion Conditions.** If the requiring element is selected, the required element(s) has(have) to be selected as well, but not vice-versa. | Constraint1 ⇒ Constraint2;<br>Asset ⇒ Decision | |
| **Validity condition. RDL equivalent: "sauf"**. The Validity Condition constrains the range of possible values for a particular reusable element | | If {Element1, Element2} ∈ {true, false, 0, 1} then Element1 ⇒ Element2<br>If {Element1, Element2} ∈ ℤ then Element1 ≥ 1 ⇒ Element2 ≥ 1 |
| **Or.** Almost one of the reusable elements related in the OR dependency must be selected in a particular configuration. | | If {Element1, Element2, …, ElementN} ∈ {true, false} then: Element1 ∨...∨ ElementN = true<br>If {Element1, Element2, …, ElementN} ∈ {1, 0} then: Element1 + ...+ElementN ≥ 1 |
| **Group cardinality/ Enumeration Decision Type/**. Cardinality determines how many Decision options of the same Decision may be chosen in a configuration, at least `M` and at most `N` of the group. | Decision ∈ ValidityCondition ⋀<br>Decision ≥ DecisionOption1 ⋀ …⋀ Decision ≥ DecisionOptionN ⋀<br>M*Decision ≤ DecisionOption1+...+ DecisionOptionN ≤ N*Decision | |

**Table 10.5.** Compilation of the RDL and Latice variability languages' constructs and the corresponding representation as CPs.

| **Constructor and domains vs. Languages** | **Renaul Documentary Language (RDL)** | **Latice** (Mannion 2002) |
|---|---|---|
| **Root.** The root element must be selected in all the | root Projet_Vehicule | |

| | | |
|---|---|---|
| configurations. | | |
| **Dependency.** The selection of the children depends of the selection of the father element and vice versa | | Father ⋀ (Child1 ⋀…⋀ ChildN) |
| **Optional.** If the use case is selected, the child element can but needs not be selected. Otherwise, if the child element is selected, the use case must as well be selected. | if {Use_Case, Element} ∈ {true, false} then Element ⇒ Use_Case<br>if {Use_Case, Element} ∈ {0, 1} then Use_Case ≥ Element | |
| **Mandatory.** If the father element is selected, the child element must be selected as well and vice versa. | if {Use_Case, Element} ∈ {true, false} then Use_Case ⇔ Element<br>if {Use_Case, Element} ∈ {0, 1} then Use_Case = Element | Father ⇔ Child |
| **Requires (includes)**. If the requiring element is selected, the required element(s) has(have) to be selected as well, but not vice versa. | if {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required<br>If {Requiring, Required} ∈ ℤ then Requiring ≥ 1 ⇒ Required ≥ 1 | |
| **Exclusion.** Indicates that both excluded elements cannot be selected in one product configuration. | if {Excluding, Excluded} ∈ {true, false} then Excluding ⇒ ¬Excluded<br>if {Excluding, Excluding} ∈ {0, 1} then Excluding - Excluded ≥ 1 | Excluding ⊕ Excluded |
| **Alternative/xor-decomposition**. A set of child elements are defined as alternative if only one element can be selected when its parent element is part of the product. | if {Use_Case, Element1, …, ElementN} ∈ {true, false} then:<br>(Element1⇔(¬Element2 ⋀...⋀ ¬ElementN ⋀ Father) ⋀ Element2 ⇔ (¬Element1 ⋀...⋀ ¬ElementN ⋀ Use_Case) ⋀ ElementN ⇔ (¬Element1 ⋀...⋀ ¬ElementN-1 ⋀ Use_Case))<br>if {Use_Case, Element1, …, ElementN} ∈ℤ then:<br>Use_Case - (Element1 + ...+ ElementN) = 0 | |
| **Or-Relation.** A set of child elements are defined as an or-relation if one or more of them can be included in the products in | if {Father, Child1, …, ChildN} ∈ {true, false} then:<br>Father ⇔ Child1 ∨...∨ ChildN<br>if {Use_Case, Element1, …, ElementN} ∈ℤ then: | Father ⇔ Child1 ∨...∨ ChildN |

| | | |
|---|---|---|
| which its parent element appears. | Use_Case - (Element1 + ...+ ElementN) ≥ 0 | |
| **Validity condition.** RDL equivalent: "sauf". It constrains the range of possible values for a particular use case. | if {Relation1, Use_Case} ∈ {true, false} then Relation1 ⇒ Use_Case<br>if {Relation1, Use_Case} ∈ {0, 1} then Use_Case - Relation1 ≥ 0 | |
| **Conjunction of subgraphs.** If Gi and Gj are the logical expressions for two different subgraphs of a lattice, the PLM is con conjunction of Gi and Gj | | Gi ∧ Gj |

# 2. Parser to Transform the Semantics of Feature Models into Constraint Programs

The next algorithm uses the SPLOT API to transform the semantics FMs represented in XML files into CPs. This algorithm navigates the XML file and creates, for each element of the feature model, the corresponding representation on constraints. For each FM, two important sub-sets must be transformed: the feature tree and the set of constraints. The algorithm starts at the root feature in depth first search in order to transform the feature tree. Next, the solitaire features (optional and mandatory) and its corresponding father are transformed into constraints according to the rules proposed in Table 10.1. Then, the features grouped in a cardinality and the corresponding father are transformed into constraints. Once the current transformation is made, the algorithm uses the current feature to recursively call the function to traverse the feature tree in depth first search, until the end of the tree. Then, the next algorithm transforms the set of constraints, represented as Conjunctive Normal Form (CNF) formulas, into constraint programs. There are two kind of constrains: requires and excludes. The algorithm transforms each CNF constraint into a CP consisting on a sum of variables being greater than 0 (e.g., `Feature1 +...+ FeatureN > 0`). To finish, features without negation are transformed into CP variables and features with a negation (e.g., `¬Feature`) are transformed into the features' complement, it is: `1-Feature`.

**Algorithm 10.1** Parser to Transform the Semantics of Feature Models into Constraint Programs using the SPLOT API

```
TransformationOfFMsSemanticIntoCPs(Feature Model FM){
      node = FM.GetRoot();
      traverseDFS(node);
      traverseConstraints(FM);
}

traverseDFS(FeatureTreeNode node){
      if ( node instanceof RootNode ) {
            transform node into a root constraint;
      }
      else if (node instanceof SolitaireFeature) {
            // Optional Feature
            if(node.isOptional()){
                  transform optional dependency into CP
            }
            // Mandatory Feature
            else{
                  transform mandatory dependency into CP
            }
      }
      // Feature Group
      else if ( node instanceof FeatureGroup ) {
            transform group cardinality dependency into CP
      }
      //we call the method traverseDFS for each children in a recursive way
      for( int i = 0 ; i < node.getChildCount() ; i++ ) {
            traverseDFS(node.getChildAt(i));
      }
}

traverseConstraints(FeatureModel FM) {
      for(PropositionalFormula formula : FM.getConstraints() ) {
      Iterator iter = formula.getVariables().iterator();
            while(iter.hasNext()){
                  artefact = iter.next();
                  if(artefact.isPositive()){
                        constraint += artefact.getName()+" + ";
                  }
                  else{
                        constraint += "(1-"+artefact.getName()+") + ";
                  }
            }
      }
}
```

# 3. Parser to Transform the Semantics of Feature Models into Constraint Programs using ATL Rules

The next ATL (Atlas Transformation Language) rules allows transforming features into CP variables, group cardinality boundaries into CP constants, and neutral, optional, mandatory, requires and excludes dependencies into constraints. For instance, the `Feature2Variable`

rule takes each source feature and transforms it into a variable. In the *modus operandi* of this rule, the feature's name is affected to the variable's name and the `haveDomain` variables' relationship is the collection of the `haveCardinality` features' relationship. If the feature to be transformed has a cardinality, then the subordinated rule (lazy rule) `Cardinality2Domain` is called to represent the corresponding cardinality as a domain of the feature.

**Source code 10.1** Parser to Transform the Semantics of Feature Models into Constraint Programs using ATL Rules

```
module Features2CP;

create OUT: CPs from IN: Features;

--function to get the source of a Neutral dependency
helper context Features!Neutral def: getSourceN(): String =
      self.source.name;

--function to get the target of a Neutral dependency
helper context Features!Neutral def: getTargetN(): String =
      self.target.name;

--function to get the source of an Optional dependency
helper context Features!Optional def: getSourceO(): String =
self.source.name;

--function to get the target of an Optional dependency
helper context Features!Optional def: getTargetO(): String =
self.target.name;

--function to get the source of a Mandatory dependency
helper context Features!Mandatory def: getSourceM(): String =
self.source.name;

--function to get the target of a Mandatory dependency
helper context Features!Mandatory def: getTargetM(): String =
self.target.name;

--function to get the source of a Requires dependency
helper context Features!Require def: getSourceR(): String =
      self.source.name;

--function to get the target of a Requires dependency
helper context Features!Require def: getTargetR(): String =
      self.target.name;

--function to get the source of an Excludes dependency
helper context Features!Exclude def: getSourceE(): String =
      self.source.name;

--function to get the target of an Excludes dependency
helper context Features!Exclude def: getTargetE(): String =
      self.target.name;
```

```
--function to concatenate with the symbol + a sequence of strings
helper def: concatenateStrings(strings: Sequence(String), before: String,
after: String): String = strings->iterate(s; acc: String = '' | acc +
before + s + after);

--function to get the first element of a sequence of strings
helper def: firstOfSequence(strings: Sequence(String)): String = strings-
>first();


rule Feature2Variable {
      from s : Features!Feature

      to    t1 : CPs!Variable (
                  name <- s.name,
                  haveDomain <- s.haveCardinality-> collect(e |
thisModule.Cardinality2Domain(e))
            )
      }

lazy rule Cardinality2Domain {
      from s : Features!Cardinality

      to   cardi : CPs!Domain (
                  min <- s.min,
                  max <- s.max
            )
      }

rule GroupCardinality2Constraint{
      from s: Features!GroupCardinality

      to t1: CPs!Constraint (
                  constraint <-
thisModule.firstOfSequence(s.haveRelationship->collect(e | e.getSourceN()))
+ ' >= 1 <==> ' + thisModule.concatenateStrings(s.haveRelationship-
>collect(e | e.getTargetN()), '', '+') + ' >= ' + s.min
                  ),
            t2: CPs!Constraint (
                  constraint <-
thisModule.firstOfSequence(s.haveRelationship->collect(e | e.getSourceN()))
+ ' >= 1 <==> ' + thisModule.concatenateStrings(s.haveRelationship-
>collect(e | e.getTargetN()), '', '+') + ' <= ' + s.max
                  )
      }

rule Neutral2Constraint{
      from s: Features!Neutral

      to t: CPs!Constraint (
            constraint <- s.getSourceN() + ' >= ' + s.getTargetN()
            )
      }

rule Optional2Constraint{
      from s: Features!Optional

      to t: CPs!Constraint (
            constraint <- s.getSourceO() + ' >= 1 <==> ' + s.getTargetO() +
' >= 0'
            )
```

```
	}

rule Mandatory2Constraint{
	from s: Features!Mandatory

	to t: CPs!Constraint (
		constraint <- s.getSourceM() + ' >= 1 <==> ' + s.getTargetM() +
' >= 1'
		)
	}

rule Require2Constraint{
	from s: Features!Require

	to t: CPs!Constraint (
		constraint <- s.getSourceR() + ' >= 1 ==> ' + s.getTargetR() +
' >= 1'
		)
	}

rule Exclude2Constraint{
	from s: Features!Exclude

	to t: CPs!Constraint (
		constraint <- s.getSourceE() + ' != 0 <==> ' + s.getTargetE() +
' == 0'
		)
	}
```

# 4. Parser to Transform the Semantics of Dopler Models into Constraint Programs

The conversion algorithm has two main phases presented in the following pseudo-code. First, the algorithm navigates through the decision model and then through the asset model. In both cases, we gather the relevant information of decisions and assets and translate them into constraints in CP. Relevant information means information affecting the variability as described above; for example, a description attribute does not affect the variability of the product line model. Our algorithm for converting Dopler variability models is implemented as an Eclipse plug-in that uses the API of the DOPLER tool suite. In the next algorithm, the variable DM represents the Dopler model to be transformed and the variable CP accumulates the results of each transformation. CP is the resulting constraint program representing DM.

**Algorithm 10.2** Parser to Transform the Semantics of Dopler Models into Constraint Programs using the API of the DOPLER tool suite

```
CP = "";
for each decision D in DM{
	if D.type == Boolean {
	  CP += "D ∈ {0, 1}";
```

```
        visc = D.getVisibilityCondition();
        if visc == false { CP += "D = 0";}
        else if visc == true {CP += "D = 1";}
        else { CP += "D ⇒ visc";}
        df = D.getDecisionEffect();
        CP += "D ⇒ df";
      }
      else if D.type == Enumeration {
        CP += "D ∈ {0, 1}";
        m, n = D.getCardinality();
        DOpt1, DOpt2,...,DOpti=D.getDecOptions();
        CP += "DOpt1, DOpt2,...,DOpti ∈ {0, 1}";
        CP += "D ⇔ m≤DOpt1+DOpt2+...+DOpti≤n";
        visc = D.getVisibilityCondition();
        if visc == false { CP += "D = 0";}
        else if visc == true {CP += "D = 1";}
        else { CP += "D ⇒ visc";}
        df = D.getDecisionEffect();
        CP += "D ⇒ df";
      }
      else if D.type == Number {
        val = representValidityConditionAsCP();
        CP += "D ∈ val";
        visc = D.getVisibilityCondition();
        if visc == false { CP += "D = 0";}
        else if visc == true {CP += "D = 1";}
        else { CP += "D ⇒ visc";}
        df = D.getDecisionEffect();
        CP += "D ⇒ df";
      }
      else if D.type == String {
        valc = representValidityConditionAsCP();
        CP += "D ∈ valc";
        visc = D.getVisibilityCondition();
        if visc == false { CP += "D = 0";}
        else if visc == true {CP += "D = 1";}
        else { CP += "D ⇒ visc";}
        df = D.getDecisionEffect();
        CP += "D ⇒ df";
      }
}
for each asset A in DM{
      CP += "A ∈ {0, 1}";
      ic = A.getInclusionCondition();
```

```
      if ic is not null {
        CP += "A ⇒ ic";
      }
      ad = A.getDependency();
      if A.type == requires {
        CP += "A ⇒ ad";
      }
      else if A.type == excludes {
        CP += "A * ad = 0";
      }
}
   Write ("The constraint program representation of the DOPLER model DM is:
" + CP);
```

# 5. Parser to Transform the Syntax of Feature Models into Constraint Programs

The next algorithm uses the Mendonça's API for navigation over SPLOT's XML-based feature models. The algorithm to transform the syntax of FMs into constraint logic programs navigates the XML file and creates, for each element of the feature model, the corresponding representation on facts. For each FM, two important sub-sets must be transformed: the feature tree and the set of constraints. We start at the root feature in depth first search in order to transform the feature tree. Next, the solitaire features (optional and mandatory) and its corresponding father are transformed into facts as presented in Chapter 3. Then, the features grouped in a group cardinality and the corresponding father are transformed into facts. Once the current transformation is made, the algorithm uses the current feature to recursively call the function to traverse the feature tree in depth first search, until the end of the tree. Then, the algorithm transforms the set of constraints, represented as Conjunctive Normal Form (CNF) formulas, into facts. There are two kind of constrains: requires and excludes. Thus, each CNF formula, corresponding to an exclusion dependency, is transformed into two facts (one dependency fact and one fact excludes) and each CNF formula corresponding to a requirement dependency is transformed into two facts (one dependency fact and one requires fact) as we explained in Chapter 3.

**Algorithm 10.3** Parser to Transform the Syntax of Feature Models into Constraint Programs using the SPLOT API

```
TransformationOfFMsSyntaxIntoCPs(Feature Model FM){
      node = FM.GetRoot();
      traverseDFS(node);
      traverseConstraints(FM);
}

traverseDFS(FeatureTreeNode node){
      if ( node instanceof RootNode ) {
            transform node into a root constraint;
      }
      else if (node instanceof SolitaireFeature) {
            // Optional Feature
            if(node.isOptional()){
                  transform optional dependency into CP
            }
            // Mandatory Feature
            else{
                  transform mandatory dependency into CP
            }
      }
      // Feature Group
      else if ( node instanceof FeatureGroup ) {
            transform group cardinality dependency into CP
      }
      //we call the method traverseDFS for each children in a recursive way
      for( int i = 0 ; i < node.getChildCount() ; i++ ) {
            traverseDFS(node.getChildAt(i));
      }
}

traverseConstraints(FeatureModel FM) {
      for(PropositionalFormula formula : FM.getConstraints() ) {
      relationType="excludes";
      Iterator iter = formula.getVariables().iterator();
            while(iter.hasNext()){
                  var[i] = iter.next();
                  //if one artefact > 0 ==> requires
                  if(var[i].isPositive()){
                        relationType="requires";
                  }
                  i++;
            }
      Build a fact according to relationType for variables in var[]
      }
}
```

# 6. Domain-specific Verification of Product Line Models

In order to execute the domain-specific verification operations proposed in this thesis, we developed a tool called VariaMos. VariaMos is an Eclipse plug-in presented in Chapter 7. In this section we present the algorithm in Java of each verification operation implemented in VariaMos.

**Algorithm 10.4** Find a valid solution

```
      giveOneSolution(String model, String listVariables){

      //obtain a valid solution
      String sol =
connection.sendMessage("exec("+model+","+listVariables+").");
      return sol;
      }
```

**Algorithm 10.5** Get the next solution

```
      public String nextSolution(){
            return connection.sendMessage("next.");
      }
```

**Algorithm 10.6** Non void PLM

```
      private ConnectionProlog connection = new
      ConnectionProlog("localhost",port);
      public String noVoid(String model, String
      listVariablesOfTheModel) {
            String sol = connection.sendMessage("exec("+model+","+
            listVariablesOfTheModel+").");
            if(sol.equals("false")){
                  sol = "Void PLM");
            }
            return sol;
      }
```

**Algorithm 10.7** Non false PLM

```
      private ConnectionProlog connection = new
ConnectionProlog("localhost",port);
      public String FalsePLM(String model, String
      listVariablesOfTheModel) {
            String sol1 = connection.sendMessage("exec("+model+","+
            listVariablesOfTheModel+").");
            String result = "False PLM";
            if(!sol1.equals("false")){
                  sol2 = nextSolution();
                  if(!sol2.equals("false")){
                        result = sol1 + sol2;
                  }
            }
            return result;
      }
```

**Algorithm 10.8** Non-attainable domains

```
findWrongDomain(String[] dataModel, Vector variables){
      String textVariable = "";
      //in order to evaluate each variable selected by the user
      for(int i=0; i<variables.size(); i++){
            //get the domain of each variable
            String[] valuesDomain =
utilities.getDomain(domains.elementAt(i));

            //flag to identify fails
            boolean flag = false;
```

```
            String wrongValues = new String();
            //evaluate it there is a solution for each domain's value
            for(int j=0; j<valuesDomain.length; j++){
                  //configuration with each domain's value
                  String configuration =
utilities.makeConfiguration(variables,
(String)variables.elementAt(i), valuesDomain[j]);
                  //create the configuration to be send to GNU Prolog
                  String prolog = "("+dataModel[0]+"="+configuration;
                  prolog = prolog.concat(", "+dataModel[1]+"),
"+dataModel[0]);
                  //to execute the instruction
                  String sol =
connection.sendMessage("exec("+prolog+").");
                  //if there is a fail,the vairiable cannot take this
value
                  if(sol.equals("fail.")){
                        wrongValues=wrongValues+valuesDomain[j]+", ";
                        flag=true;
                  }
            }
            //save the values that the variable cannot take
            if(flag){
                  textVariable = textVariable +
wrongValues.substring(0, wrongValues.length()-2)+".";
            }
            //the variable has not wrong domain's values
            else{
                  textVariable = "The Variable
"+features.elementAt(i)+" don't has wrong domain's values";
            }
      }
}
```

**Algorithm 10.9** Dead reusable elements

```
findDeadVariables(String[] dataModel, Vector variables){
      deadVariables = vector with all the variables of the PLM;
      //for each one of the variables in deadVariables
      while(j<deadVariables.size()){
            //get all the values that the current variable can take
            String[] value = domains.elementAt(j));
            //identify when we have a solution or a fail from Prolog
            boolean flag = false;
            //save the values that the variables cannot take
            String wrongValues = new String();
            String sol = new String();
            //evaluate each value of the domain(except 0)
            for(int i=0; i< value.length; i++){
                  if(!value[i].equals("0")){
                  //create a configuration with the variables' domain
values
                  String                configuration                =
utilities.makeConfiguration(feature,
(String)deadFeatures.elementAt(j), value[i]);
                  //create the instruction to be executed in Gnu-
Prolog
                  String prolog = "("+dataModel[0]+"="+configuration;
                  //send the instruction to GNU Prolog
```

```
                    prolog    =    prolog.concat(",    "+dataModel[1]+"),
   "+dataModel[0]);
                    sol = connection.sendMessage("exec("+prolog+").");
                    //if  the  configuration  does  not  generate  any
   solution; we kept the value that the variables cannot take
                    if(sol.equals("fail.")){
                          wrongValues = wrongValues + value[i]+", ";
                          flag=true;
                    }
                          //it is not necessary to evaluate all the values of
                    each domain, with the first product obtained we know that
                    the variables is not dead
                          else{
                                break;
                          }
                    }
             }
      }
```

**Algorithm 10.10** False optional reusable elements

```
   findFalseOptionalFeatures(String[] dataModel, Vector
   OptionalElements){
         //for each one of the variables to be verified
         for(int i=0; i< OptionalElements.size(); i++){
               //take all the values that this variable can take
               String[] valuesCardinality =
   utilities.getCardinality(domains.elementAt(i));
               for(int j=0; j<valuesCardinality.length; j++){
                     //verify if the variable can take the 0 value
                     if(valuesCardinality[j].equals("0")){
                           //create a configuration with the variable=0
                           String configuration =
   utilities.makeConfiguration(featureAll, (String)
   OptionalElements.elementAt(i), valuesCardinality[j]);
                           //get a sol. from GNU Prolog with this conf
                           String sol =
   connection.sendMessage("exec("+prolog+").");
                           //if there is not solution, the variable is
   false optional
                           if(sol.equals("fail.")){
                                 textFeature = "   The Feature "+
   OptionalElements.elementAt(i)+" is a False Optional Variable.";
                           }
                     }
               }
         }
   }
```

**Algorithm 10.11** Redundancy-free

```
findRedudantRelatioships(String[] dataModel, Vector relatonships){
   //vector with the variables to verify
   Vector constraints = (Vector)relatonships.elementAt(0);
   //vector with the negations of the variables to verify
   Vector negationConstraints = (Vector)relatonships.elementAt(1);

   //1. Verify if the model is consistent
   resultProlog =
connection.sendMessage("exec("+dataModel[1]+","+dataModel[0]+").");
```

```
   //if there is a solution, the model is consistent
   if(!resultProlog.equals("fail.")){
         //Verify each one of the constraints selected by the user
         for(int i=0; i<constraints.size(); i++){
               //2.verify the consistency of the model without the const
               //create a string with the model without the constraint
               String model = utilities.converToString(path,
(String)constraints.elementAt(i));
               int begin = model.indexOf("fd_labeling");
               String message = model.substring(0, begin);
               message += "\nfd_labeling("+dataModel[0]+")";
               String messageFinal =  "(("+ message+"),
"+dataModel[1]+"),"+dataModel[0];
               //execute the model to get a solution
               resultProlog=
connection.sendMessage("exec("+messageFinal+").");

               //if there is a solution the model without the constraint
at hand is consistent
               if(!resultProlog.equals("fail.")){

                     //3.consistenty of the model with the negation
                     begin = model.indexOf("fd_labeling");
                     message = model.substring(0, begin);
                     message += negationConstraints.elementAt(i) +",\n
fd_labeling("+dataModel[0]+")";
                     messageFinal =  "(("+ message+"),
"+dataModel[1]+"),"+dataModel[0];
                     //execution of the sentence to get one solution
                     resultProlog=
connection.sendMessage("exec("+messageFinal+").");
                     //if there is a solution the const is not redundant
                     if(!resultProlog.equals("fail.")){
                           VerificationManagerView.txtResultats.append("
   The Relationships "+constraints.elementAt(i)+" is not Redundant\n");
                     }
                     //if there is not solutions the const is redundant
                     else{
                           VerificationManagerView.txtResultats.append("
   The Relationships "+constraints.elementAt(i)+" is Redundant\n");
                     }
               }
               //if without the constr there is no solts: not redundant
               else{
                     VerificationManagerView.txtResultats.append("   The
Relationships "+constraints.elementAt(i)+" is not Redundant\n");
               }
         }
   }
   //the model is not consistent because there is no valid solutions
   else{
         VerificationManagerView.txtResultats.append("The model is
inconsistent" + "\n");
   }
}
```

# 7. Conformance Checker of Feature Models

A conformance checker is a tool that verifies if a certain model is a correct instance of its metamodel, it is, verify if the model respects the syntax rules of the metamodel. The next program check the conformance of Feature Models with the metamodel presented in Chapter 3. Detailed explanations about each line of code below are also provided in Chapter 3.

**Source code 10.2** Conformance Checker of Feature Models

```
/* Format of Feature Model Files
root(FeatureId) - defines a root feature. NB: root Id(s) should be the
lowest feature Id(s)

feature(FeatureId, FeatureName, LAttId) - defines a feature with a unique
Id, a name and a list of attributes.  NB: LAttId is ascending sorted

attribute(AttId, AttName, LAttDomain, AttValue) - defines an attribute with
a unique Id, a name a list of domain values, and a value

dependency(DepId, FeatureId1, FeatureId2, DepType) - defines a dependency
between the source FeatureId1 and the target FeatureId2 whose type is
DepType = mandatory / optional / requires / excludes.  NB: for excludes
(which is commutative) it is expected FeatureId1 < FeatureId2.

groupCardinality(LDepId, Min, Max) - defines a cardinality for a set of
dependencies LDepId to be between Min and Max.
*/

load_fm(File) :-
      '$remove_predicate'(root, 1),
      '$remove_predicate'(feature, 3),
      '$remove_predicate'(attribute, 3),
      '$remove_predicate'(dependency, 4),
      '$remove_predicate'(groupCardinality, 3),
      consult(File),
      mk_index.

:- dynamic(inclusion_dep/2).

mk_index :-
      retractall(inclusion_dep(_, _)),
      dependency(_, Id1, Id2, Type),
      inclusion_type(Type),  % this is used by rule 10
      assertz(inclusion_dep(Id1, Id2)),
      fail.

mk_index.

inclusion_type(mandatory).
inclusion_type(optional).
inclusion_type(requires).

/* run all rules */

do_all :-
      user_time(T),
      g_assign(t, T),
      write('\n*** testing rule 1'), nl,
      conformance_1(FeatureName, AttId1, AttId2, AttName),
```

```
      write(conformance_1(FeatureName, AttId1, AttId2, AttName)), nl,
      fail.

do_all :-
      disp_time,
      write('\n*** testing rule 2'), nl,
      conformance_2(FeatureId1, FeatureId2, FeatureName),
      write(conformance_2(FeatureId1, FeatureId2, FeatureName)), nl,
      fail.

do_all :-
      disp_time,
      write('\n*** testing rule 3'), nl,
      conformance_3(LRootId),
      write(conformance_3(LRootId)), nl,
      fail.

do_all :-
      disp_time,
      write('\n*** testing rule 4'), nl,
      conformance_4(DepId, FeatureId),
      write(conformance_4(DepId, FeatureId)), nl,
      fail.

do_all :-
      disp_time,
      write('\n*** testing rule 5'), nl,
      conformance_5(FeatureId, DepId1, DepId2),
      write(conformance_5(FeatureId, DepId1, DepId2)), nl,
      fail.

do_all :-
      disp_time,
      write('\n*** testing rule 6'), nl,
      conformance_6(LDepId, Min, Max),
      write(conformance_6(LDepId, Min, Max)), nl,
      fail.

do_all :-
      disp_time,
      write('\n*** testing rule 7'), nl,
      inconsistecy_7(FeatureName1, FeatureName2),
      write(inconsistecy_7(FeatureName1, FeatureName2)), nl,
      fail.

do_all :-
      disp_time,
      write('\n*** testing rule 8'), nl,
      conformance_8(RootName, FeatureName),
      write(conformance_8(RootName, FeatureName)), nl,
      fail.

do_all :-
      disp_time,
      write('\n*** testing rule 9'), nl,
      conformance_9(ChildName, AncName),
      write(conformance_9(ChildName, AncName)), nl,
      fail.

do_all :-
      disp_time,
```

```
      write('\n*** done !'), nl.

disp_time :-
      user_time(T2),
      g_read(t, T1),
      T is T2 - T1,
      g_assign(t, T2),
      format('   in ~w msec~n', [T]).
```

# References


Acher M., Collet P., Lahire P, France R. Comparing Approaches to Implement Feature Model Composition. In 6th European Conference on Modelling Foundations and Applications (ECMFA), vol. 6136, pages 3--19, Springer, France, 2010.

Alves V., Gheyi R., Massoni T., Kulesza U., Borba P., Lucena C. Refactoring product lines. In: GPCE'06, ACM pp. 201–210, 2006.

Apel S., Kastner C., Lengauer C. FeatureHouse: Language-Independent, Automatic Software Composition. In Proc. Int'l. Conf. Software Engineering (ICSE), pages 221-231. IEEE CS, 2009.

Apel S., Lengauer C., Batory D., Moller B., Kastner C. An algebra for feature-oriented software development. Technical Report MIP-0706, Department of Informatics and Mathematics, University of Passau, Germany, 2007.

Apt K.R., Wallace M.G. Constraint Logic Programming Using ECLiPSe. Cambridge University Press, Cambridge, 2006.

Barthe G., Beringer L., Cregut P., Gregoire B., Hofmann M., MÄuller P., Poll E., Puebla G., Stark I., Vetillard E. Mobius: Mobility, ubiquity, security: Objectives and progress report. In Trustworthy Global Computing'06, LNCS, 2007.

Bass L., Clements P., Donohoe P., McGregor J., Northrop L. Fourth Product Line Practice Workshop Report. Technical report CMU/SEI-2000-TR-002 ESC-TR-2000-002

Batory D. Feature Models, Grammars, and Propositional Formulas. In 9th International Software Product Lines Conference Rennes, France, volume 3714 of Lecture Notes in Computer Sciences. Springer–Verlag 2005.

Batory D., Benavides D., Ruiz-Cortes A. Automated analysis of feature models: challenges ahead. Communications of ACM 49, 12, December 2006.

Beckert B., Hahnle R., Schmitt P.H., editors. Verification of Object-Oriented Software: The KeY Approach, volume 4334 of Lecture Notes in Computer Science. Springer-Verlag, 2007.

Benavides D. On the Automated Analysis of Software Product Lines Using Feature Models. A Framework for Developing Automated Tool Support. University of Seville, Spain, PhD Thesis, June 2007.

Benavides D., Trinidad P., Ruiz-Cortés A. Automated Reasoning on Feature Models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg, 2005.

Benavides D., Ruiz-Cortés A., and Trinidad P.: Using constraint programming to reason on feature models. In The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE 2005, pages 677–682, 2005

Benavides D., Trujillo S., Trinidad P. On the modularization of feature models. In First European Workshop on Model Transformation, September 2005.

Benavides, D., Segura, S., Trinidad, P., and Ruiz-Cortés, A. Using Java CSP solvers in the automated analyses of feature models. In Post-Proceedings of The Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE). LNCS 4143, 2006.

Benavides D., Segura S., Ruiz-Cortés A. Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems . Elsevier, 2010.

Bessiere Ch. Constraint propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, Handbook of Constraint Programming. Elsevier, 2006.

Beuche D. Composition and Construction of Embedded Software Families. PhD thesis, Otto-von-Guericke-Universit□at Magdeburg, Germany, December 2003.

Blanc X., Mougenot A., Mounier I., Mens T. Incremental Detection of Model Inconsistencies Based on Model Operations. In CAiSE'09, pages 32-46, 2009.

Boehm B. Software Engineering Economics. Englewood Cliffs, New Jersey: Prentice-Hall, ISBN 0-13-822122-7. 1981.

Boehm B., In H. Identifying Quality Requirements Conflicts. IEEE Software, , pp. 25-35. March 1996

Borrett J . E., Tsang E. P. K.. A Context for Constraint Satisfaction Problem Formulation Selection. Constraints 6, no. 4, pp. 299-327, October 2001.

Bosch J. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach, Addison-Wesley, Boston, 2000.

Boucher Q., Classen A., Faber P., Heymans P. Introducing TVL, a text-based feature modelling language. In Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29. University of Duisburg-Essen, January 2010.

Bradley A., Manna Z. The Calculus of Computation - Decision Procedures with Applications to Verification. ISBN 978-3-540-74112-1 Springer Berlin Heidelberg New York, 2007.

Broek P., Galvão I. Analysis of Feature Models using Generalised Feature Trees, Third Int. Workshop VaMoS, 2009.

Bruns D., Klebanov V., Schaefer I. Verification of software product lines with delta-oriented slicing. On the 2010 Int. Conf. on Formal verification of object-oriented software Springer-Verlag Press 2011.

Cabot J., Teniente E. Incremental evaluation of OCL constraints. In: Dubois, E., Pohl, K. (eds.) CAiSE'06. LNCS, vol. 4001, pp. 81–95. Springer, Heidelberg, 2006.

Clarke E., Grumberg O. Peled D. Model Checking. The MIT Press, 1999.

Classen A., Boucher Q., Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. Sci. Comput. Program. 76(12): pages 1130-1143, 2011.

Clements P., Northrop L. Software Product Lines: Practices and Patterns, Addison Wesley, Reading, MA, USA, 2001.

Colmerauer, A. Prolog II reference manual and theoretical model. Technical report, Groupe Intelligence Artificielle, Universit`e Aix-Mareseille II, October 1982.

Czarnecki K. Helsen S. Classification of model transformation approaches. In Online Proceedings of the 2nd OOPSLA03 Workshop on Generative Techniques in the Context of MDA, Anaheim, October 2003.

Czarnecki K., Helsen S., Eisenecker U. Staged Configuration Using Feature Models. Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004.

Czarnecki K., Helsen S., Eisenecker U. W. Formalizing cardinality-based feature models and their specialization, Software Process: Improvement and Practice, 10 (1) pages 7–29, 2005.

Czarnecki K., Pietroszek K. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints, 5th Int. Conference on Generative Programming and Component Engineering, 2006.

Dauron A., Astesana J-M. Spécification et configuration de la ligne de produits véhicule de Renault. Journée Lignes de Produits. Université Pantéon Sorbonne, Paris, France. October 2010.

Dhungana D., Grünbacher P., Rabiser R. The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study. Automated Software Engineering, 2010 (in press; doi: 10.1007/s10515-010-0076-6).

Dhungana D., Heymans P., Rabiser R. A Formal Semantics for Decision-oriented Variability Modeling with DOPLER, Proc. of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010), Linz, Austria, ICB-Research Report No. 37, University of Duisburg Essen, pp. 29-35. 2010.

Dhungana D., Rabiser R., Grünbacher P. Coordinating Multi-Team Variability Modeling in Product Line Engineering. In 2nd International Workshop on Supporting Knowledge Collaboration in Software Development (KCSD2006), Tokyo, Japan, 2006.

Dhungana, P. Grünbacher, and R. Rabiser, "The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study," Automated Software Engineering, 2010.

Diaz D., Codognet Ph. Design and implementation of the GNU Prolog System. Journal of Functional and Logic Programming (2001). http://www.gprolog.org.

Djebbi O., Salinesi C. RED-PL, a Method for Deriving Product Requirements from a Product Line Requirements Model. In: CAISE'07, Norway, 2007.

Djebbi O., Salinesi C. RED-PL, a Method for Deriving Product Requirements from a Product Line Requirements Model. International Conference on Advances in Information Systems Engineering, CAISE□07. Norway, 2007.

Djebbi O., Salinesi C., Fanmuy G. Industry Survey of Product Lines Management Tools: Requirements, Qualities and Open Issues, International Conference on Requirement Engineering (RE), IEEE Computer Society, New Delhi, India, October 2007.

Egyed A. Instant consistency checking for UML. In: International Conf. Software Engineering (ICSE□06), pp. 381–390. ACM Press, New York, 2006.

Elaasar, M., Brian, L.: An overview of UML consistency management. Technical Report SCE-04-18, August 2004.

Elfaki A., Phon-Amnuaisuk S., Kuan Ho C. Using First Order Logic to Validate Feature Model. Third Int. Workshop VaMoS, 2009.

Felfernig A., Friedrich G., Jannach D., Zanker M. Towards distributed configuration. In KI '01: Proceedings of the Joint German/Austrian Conference on AI, London, UK, 2001.

Figueiredo E., Cacho N., Sant'Anna C., Monteiro M., Kulesza U, Garcia A., Soares S., Cutigi-Ferrari F., Shakil-Khan S., Castor-Filho F., Dantas F. Evolving software product lines with aspects: an empirical study on design stability. ICSE 2008.

Finkelstein, A. & Sommerville I. The Viewpoints FAQ. Software Engineering Journal, Vol. 11, No. 1, pp. 2-4, 1996.

Finkelstein A.C.W., Gabbay D., Hunter A., Kramer J., Nuseibeh B. Inconsistency handling in multiperspective specifications. IEEE Transactions on Software Engineering, pages 569–578, 1994.

Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering 2(1), 1992.

Fleurey F. Kompose: a generic model composition tool. 2007. Available from: http://www.kermeta.org/kompose/.

Fleurey, F., Baudry, B., France, R.B., Ghosh, S.: A generic approach for automatic model composition. In Giese, H., ed.: MoDELS Workshops, Springer, pp. 7–15

Gordon M. J. C., Melham T. F. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, 1993.

Griss M., Favaro J., d'Alessandro M. Integrating feature modeling with the RSEB, in: Proceedings of the Fifth International Conference on Software Reuse, Vancouver, BC, Canada, June 1998.

Harel D., Rumpe B. Meaningful modeling: what's the semantics of 'semantics'?, IEEE Computation, 37, (10), pp. 64–72, 2004.

Harel D., Rumpe B. Modeling languages: syntax, semantics and all that stuff, part I: the basic stuff, Technical Report MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, 2000.

Hemakumar A. Finding Contradictions in Feature Models. Workshop on the Analysis of Software Product Lines (ASPL), September 2008.

Hevner A.R., March S.T., Park J., Ram S. Design science in information system research. MIS Quarterly, 28(1):75-105, March 2004.

Heymans P., Schobbens P.-Y., Trigaux J.-C., Bontemps Y., Matulevicius R., Classen A. Evaluating formal properties of feature diagram languages. IET Software (IEE) 2(3):281-302, 2008.

Howe D. The Free On-line Dictionary of Computing, 2010 http://foldoc.org

Hubaux, A., Classen, A., Heymans, P.: Formal modelling of feature configuration workflow. In: SPLC 2009, San Francisco, CA, USA (2009)

Hubaux A., Heymans P., Schobbens P.Y., Deridder D. Towards Multi-view Feature-Based Configuration. R. Wieringa and A. Persson (Eds.): REFSQ 2010, LNCS 6182, pp. 106–112, Springer-Verlag Berlin Heidelberg (2010).

Igarashi A., Pierce B., Wadler P. Featherweight Java: A minimal core calculus for Java and GJ. In ACM Trans. Program. Lang. Syst., vol. 23, no. 3, 2001.

Jaffar, J., Lassez, J.-L.: Constraint logic programming. In: Proc. 14th symp. On Principles of programming languages. ACM, New York, 1987.

Janota M., Kiniry J. Reasoning about Feature Models in Higher-Order Logic, in 11th Int. Software Product Line Conference (SPLC07), 2007.

Jayaraman, P., Whittle, J., Elkhodary, A., Gomaa, H.: Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In: MoDELS, 2007.

Kang K., Cohen S., Hess J., Novak W., Peterson S. Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

Kang K., Lee J., Donohoe P. Feature-oriented product line engineering. Software, IEEE, 19(4), July-August 2002.

Kang K, Kim S., Lee J., Kim K., Shin E., Huh M. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Ann. Software Eng. (ANSOFT)5:143-168, 1998.

Karataş A., Oğuztüzün H., Doğru A. Mapping Extended Feature Models to Constraint Logic Programming over Finite Domains. SPLC, Korea (2010).

Kastner Ch., Apel S., Rahman S. S., Rosenmuller M., Batory D., Saake G. On the Impact of the Optional Feature Problem: Analysis and Case Studies, SPLC'09, San Francisco, USA, 2009.

Kästner, C., Apel, S. Type-checking Software Product Lines – a Formal Approach. In Proc. of ASE08 , 2008, pp. 258-267.

Kim C.H.P., Batory D., Khurshid S. Reducing Combinatorics in Testing Product Lines. Aspect Oriented Software Development (AOSD), March 2011.

Kitchenham B. Procedures for performing systematic reviews. Technical report, Keele University and NICTA, 2004.

Korherr B., List B. A UML 2 Profile for Variability Models and their Dependency to Business Processes. 1st International Workshop on Enterprise Information Systems Engineering (WEISE 07), September 2007, Regensburg, Germany, IEEE Press, 2007.

Kowalski R.A., Sadri F., Soper P. Integrity checking in deductive databases. In: Proc. International Conference on Very Large Data Bases (VLDB), pp. 61–69. Morgan Kaufmann, San Francisco (1987).

Kuchcinski K. Constraints-driven scheduling and resource assignment. ACM Transactions on Design Automation of Electronic Systems (TODAES), 8(3):355–383, July 2003.

Laburthe F., Jussien N. Choco constraint programming system. Available at http://choco.sourceforge.net/ (2005).

Lauenroth K., Metzger A., Pohl K. Quality Assurance in the Presence of Variability. S. Nurcan *et al.* (eds.), Intentional Perspectives on Information Systems Engineering, Springer-Verlag, Berlin Heidelberg 2010.

Lauenroth K., Pohl K. Towards automated consistency checks of product line requirements specifications, Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, Atlanta, Georgia, USA, November 2007.

Lauenroth K., Pohl K., Toehning S. Model Checking of Domain Artifacts in Product Line Engineering, Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, p.269-280, November 16-20, 2009.

Leavens G. T., Baker A. L., Ruby C. Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes, 31(3):1-38, 2006.

Leite J., Freeman P.A. Requirements Validation through Viewpoint Resolution. IEEE Transactions on Software Engineering, Vol. 12, No. 12, pp. 1253-1269. 1991.

Liu J., Batory D., Lengauer C. Feature Oriented Refactoring of Legacy Applications, ICSE, 2006.

Liu J., Basu S., Lutz R. R. Compositional model checking of software product lines using variation point obligations. Journal Automated Software Engineering, Volume 18 Issue 1, 2011.

Mannion M, Kaindl H. Using Parameters and Discriminants for Product Line Requirements. Published in Wiley InterScience, 21 December 2007.

Mannion M. Using first-order logic for product line model validation. In Proceedings of the Second SPLC, LNCS 2379, San Diego, CA, Springer (2002).

March A.T., Smith G.F. Design and natural science research on information technology. Decision Support Systems, 15(4):251-266, December 1995.

Matthias R., Kai B., Detlef S., Ilka P. Extending feature diagrams with UML multiplicities. Proceedings of the Sixth Conference on Integrated Design and Process Technology, Pasadena, CA, 2002.

Mazo R., Salinesi C. Variability Models (VariaMos) eclipse Plug-in, 2011. Available to download at https://sites.google.com/site/raulmazo/

Mazo R., Grünbacher P., Heider W., Rabiser R., Salinesi C., Diaz D. Using Constraint Programming to Verify DOPLER Variability Models. Proceedings of the VaMos Workshop, ACM Press, Belgium, January 2011. (a)

Mazo R., Lopez-Herrejon R., Salinesi C., Diaz D., Egyed A. A Constraint Programming Approach for Checking Conformance in Feature Models. In 35th International Computer Software and Applications Conference (COMPSAC), IEEE Press, Germany, 2011. (b)

Mazo R., Salinesi C, Djebbi O, Diaz D, Lora-Michiels A. Constraints: the Heart of Domain and Application Engineering in the Product Lines Engineering Strategy. International Journal of Information System Modeling and Design IJISMD. Sweden, November 2011. (c)

Mazo R., Salinesi C., Diaz D. Abstract Constraints: A General Framework for Solver-Independent Reasoning on Product Line Models. Accepted on INSIGHT - Journal of International Council on Systems Engineering (INCOSE), to be released the 15 October 2011. (d)

Mazo R., Salinesi C., Diaz D., Lora-Michiels A. Transforming Attribute and Clone-Enabled Feature Models Into Constraint Programs Over Finite Domains. 6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Springer Press, China, 2011. (e)

Mendonça, M., Branco, M., Cowan, D. S.P.L.O.T.: software product lines online tools. In OOPSLA Companion. ACM, (2009) http://www.splot-research.org.

Mendonça, M., Wasowski, A., Czarnecki, K. SAT–based analysis of feature models is easy. In Proceedings of the Sofware Product Line Conference (2009).

Metzger A., Pohl K., Heymans P., Schobbens P.-Y., Saval G. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In 15th IEEE International Requirements Engineering Conference, RE ’07. 2007.

Michel R., Classen A., Hubaux A., Boucher Q. A Formal Semantics for Feature Cardinalities in Feature Diagrams. 5th International Workshop on Variability Modelling of Software-intensive Systems (VaMos'11), Namur-Belgium, 2011.

Nuseibeh, B., Easterbrook, S., and Russo, A., 2000. "Leveraging Inconsistency in Software Development". IEEE Computer, April.

Nuseibeh B., Kramer J., Finkelstein A. A framework for expressing the relationships between multiple views in requirements specification. IEEE Trans. Software Eng. 20(10) (1994) 760–773.

Olivé A. Integrity Constraints Checking in Deductive Databases, Proceedings of the 17th International Conference on Very Large Data Bases, p.513-523, September 03-06, (1991).

OMG. UML 2.0 Object Constraint Language (OCL) Final Adopted specification. 2003. Available from: http://www.omg.org/cgibin/doc?ptc/2003-10-14.

Oxford University. Concise Oxford English Dictionary. Oxford University Press, 2008.

Paige R. F., Brooke P. J., Ostro J. S. Metamodel-based model conformance and multiview consistency checking. ACM Transactions on Software Engineering and Methodology, (2007).

Parnas D. L. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, December 1972.

Peffers, K., Tuunanen T., Chatterjee M.A, Rothenberger S. A design science research methodology for information systems research. Journal of Management Information Systems 24, n° 3 pp. 45--77. (2007)

Pohl K, Böckle G; van der Linden F. Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, Heidelberg (2005)

Popper, K.R. All Life is Problem Solving. Collection of essays republished by Piper Berlang in “Alles Leben ist Problemlösen” (Munich, 1994); translated as All Life is Problem Solving by Patrick Camiller, 1994.

Popper, K.R. Conjectures and refutations : the growth of scientific knowledge. Routledge and Kegan Paul, 5th edition, 1974.

Riebisch M., Bollert K., Streitferdt D., Philippow I. Extending feature diagrams with UML multiplicities, in: Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT2002), Pasadena, CA, June 2002.

Robinson W. Interactive Decision Support for Requirements Negotiation. In Concurrent Engineering: Research & Applications, 2, pp. 237-252. 1994.

Robinson W., Fickas S. Supporting Multiple Perspective Requirements Engineering. Proceedings of the 1st International Conference on Requirements Engineering (ICRE 94), IEEE Computer Society Press, pp.206-215. 1994.

Rolland C, Nurcan S (2010) Business Process Lines to deal with the Variability. In Proc. Of the Hawaii International Conference on System Sciences (HICSS), Hawaii, USA

Rolland C, Prakash N, Kaabi R (2007): Variability in Business Process Families. Information Resources Management Association (IRMA)

Rosenmüller M., Siegmund N., Thüm T., Saake G. Multi-Dimensional Variability Modeling. Proceedings of the VAMOS Workshop, Belgium, 2011.

Roos-Frantz F., Segura S. Automated Analysis of Orthogonal Variability Models. A First Step. In: 1st. SPLC Workshop on Analysis of Software Product Lines (ASPL 2008). Limerick, Ireland; 2008.

Salinesi C., Diaz D., Djebbi O., Mazo R. Exploiting the Versatility of Constraint Programming over Finite Domains to Integrate Product Line Models. Poster in 17th IEEE International Conference on Requirements Engineering (RE'09). Atlanta-USA. Septembre 2009.

Salinesi C., Etien A., Zoukar I. A Systematic Approach to Express IS Evolution Requirements Using Gap Modelling and Similarity Modelling Techniques, International Conference on Advanced information Systems Engineering (CAISE), Springer Verlag, Riga, Latvia, 2004.

Salinesi C, Mazo R. Defects in Product Line Models and how to Identify them. "Software Product Lines - The Automated Analysis", edited by Abdelrahman Elfaki, InTech editions, ISBN 979-953-307-700-9. to appear in January 2012.

Salinesi C., Mazo R., Diaz D. Criteria for the verification of feature models, In 28th INFORSID Conference, Marseille, France, 2010.

Salinesi C., Mazo R., Diaz D., Djebbi O. Solving Integer Constraint in Reuse Based Requirements Engineering. 18th IEEE Int. Conference on Requirements Engineering (RE'10). Sydney, Australia, 2010.

Salinesi C., Mazo R., Djebbi O., Diaz D., Lora-Michiels A. Constraints: the Core of Product Line Engineering. Fifth IEEE International Conference on Research Challenges in Information Science (RCIS), IEEE Press, France, May 2011.

Salinesi C., Rolland C., Mazo R. VMWare: Tool support for automatic verification of structural and semantic correctness in product line models. In Third International Workshop on Variability Modelling of Software-intensive Systems (VaMos), pages 173–176, 2009.

Sawyer P. Software Requirements. In R. Thayer (Ed), Software Engineering: Vol 1: The Development Process, IEEE Press, September 2005.

Schaefer I., Bettini L., Bono V., Damiani F., Tanzarella N. Delta-oriented programming of software product lines. In Proceedings, 14th International Software Product Line Conference, Lecture Notes in Computer Science, Jeju, South Korea, 2010.

Schobbens P., Heymans P., Trigaux J., Bontemps Y. Feature diagrams: A survey and a formal semantics. In Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06), Minneapolis, Minnesota, USA, September 2006. IEEE Computer Society

Schobbens P., Heymans P., Trigaux J.C, Bontemps Y. Generic semantics of feature diagrams. Journal of Computer Networks. 2007 Feb; 51(2):456–479.

Schulte Ch., Stuckey P. J. Efficient constraint propagation engines. ACM Trans. Program. Lang. Syst., 31(1), (2008).

Segura S. Automated Analysis of Feature Models using Atomic Sets. First Workshop on Analyses of Software Product Lines (ASPL'08), SPLC'08. Limerick, Ireland, 2008.

Segura S., Benavides D., Ruiz-Cortés A., Trinidad P. Automated merging of feature models using graph transformations. In: GTTSE '07. Volume 5235 of LNCS., Springer-Verlag, 2008.

Semmak F., Gnaho Ch., Laleau R. Extended KAOS Method to Model Variability in Requirements. Communications in Computer and Information Science, 69:193-205, 2010

Semmak F., Laleau R., Gnaho Ch. Supporting Variability in Goal-based Requirements. Proceedings of the IEEE International Conference on Research Challenges in Information Science (RCIS), pp. 271-280, 2009.

Simon H.A. The Sciences of the Artificial. The MIT Press, 1981. Second edition.

Sommerville I., Sawyer P. Requirements Engineering - A Good Practice Guide. John Wiley, April 1997.

Spanoudakis G., Zisman A. Inconsistency management in software engineering: Survey and open research issues. Handbook of Software Engineering and Knowledge Engineering, 329–380. 2001.

Stahl T., Völter M., Czarnecki K. Model-Driven Software Development: Technology, Engineering, Management. San Francisco, Wiley, June 2006.

Streitferdt D., Riebisch M., Philippow I. Details of formalized relations in feature models using OCL. In Proceedings of 10th IEEE International Conference on Engineering of Computer–Based Systems (ECBS), Huntsville, USA. IEEE Computer Society, pages 45–54, 2003.

Stropky, M.E., Laforme, D.. An automated mechanism for effectively applying domain engineering in reuse activities. In International Conference on Ada, pp. 332-340, California,USA. (1995)

Sun J., Zhang H., Wang H. Formal semantics and verification for feature modeling. In Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, pages 303-312, Washington, DC, USA, 2005.

Tekinerdogan B., Aksit M. Managing variability in product line scoping using design space models. Proceedings of Software Variability Management Workshop, Groningen, The Netherlands, 2003.

Trinidad P., Benavides D., Ruiz-Cortés A. A first step detecting inconsistencies in feature models. In CAiSE Short Paper Proceedings, Advanced Information Systems Engineering, 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, 2006.

Trinidad P., Benavides D., Durán A., Ruiz-Cortés A., Toro M. Automated error analysis for the agilization of feature modeling, Journal of Systems & Software – Elsevier, 2008.

Trinidad P., Ruiz-Cortés A., Benavides D., Segura S., Jimenez A. FAMA Framework. 12th International Software Product Line Conference (SPLC 2008), Limerick, Ireland, (2008).

Turner D. Miranda: a non-strict functional language with polymorphic types, in: Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science Vol 201, J.-P. Jouannaud (ed.), Springer-Verlag, Berlin, Heidelberg, 1985.

Van den Broek P., Galvão I. Analysis of Feature Models using Generalised Feature Trees, Third Int. Workshop VaMoS, 2009.

Van der Storm T. Variability and Component Composition, in: Proceedings of the 8th International Conference on Software Reuse (ICSR-8), 2004.

Van der Storm T. Generic Feature-Based Composition. In: M. Lumpe and W. Vandeperren, editors, Proceedings of the Workshop on Software Composition (SC'07), volume 4829 of LNCS, pp. 66-80, Springer, 2007.

Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logics to maintain consistency between UML models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003)

Van Deursen A., Klint P. Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology, 10(1):1–18, March 2002.

Van Hentenryck P. Constraint Satisfaction in Logic Programming. The MIT Press, (1989).

Vierhauser M., Grünbacher P., Egyed A., Rabiser R., Heider W. Flexible and Scalable Consistency Checking on Product Line Variability Models. Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE), Antwerp, Belgium, ACM (2010).

Von der Maßen T., Lichter H. Deficiencies in feature models. In Tomi Mannisto and Jan Bosch, editors, Workshop on Software Variability Management for Product Derivation - Towards Tool Support (2004).

Von der Maßen T., Lichter H. Modeling Variability by UML Use Case Diagrams. International Workshop on Requirements Engineering for Product Lines. Avaya Labs Report series editors, (pp. 18-26). Essen, Germany, 2002.

Von der Maßen T., Lichter H. RequiLine: A requirements engineering tool for software product lines, Proceedings of International Workshop on Product Family Engineering PFE-5, Springer LNCS 3014, Siena, Italy, November 2003.

Webster J., Watson R. Analyzing the past to prepare for the future: Writing a literature review. MIS Quarterly, 26(2):xiii–xxiii, 2002.

White J., Doughtery B., Schmidt D. Selecting highly optimal architectural feature sets with filtered cartesian flattening. Journal of Systems and Software, 82(8):1268–1284, 2009.

White J., Schmidt D., Benavides D., Trinidad P., and Ruiz-Cortes A. Automated diagnosis of product-line configuration errors in feature models. In Proceedings of the SPLC (2008).

Yan H., Zhang W., Zhao H., Mei H. An optimization strategy to feature models' verification by eliminating verification-irrelevant features and constraints. In ICSR, pages 65–75, 2009.

Zhang, W., Zhao, H., Mei, H. A Propositional Logic-Based Method for Verification of Feature Models. In: Proceedings of 6th International Conference on Formal Engineering Methods, pp. 115–130 (2004).

Zhao, H., Zhang, W., Mei, H. Multi-view based customization of feature models. Journal of Frontiers of Computer Science and Technology 2(3), 260–273, 2008.

Ziadi, T. Manipulation de Lignes de Produits en UML. Unpublished doctoral dissertation, IRISA-TRISKELL, Université de Rennes 1, France, 2004.