UNIVERSITÉ DE NICE–SOPHIA ANTIPOLIS — UFR Sciences
École Doctorale de Sciences et Technologies de l'Information
et de la Communication (STIC)

T H È S E

pour obtenir le titre de
Docteur en Sciences
de l'UNIVERSITÉ de Nice–Sophia Antipolis

Discipline: Informatique

présentée et soutenue par
Mathieu ACHER

Managing Multiple Feature Models: Foundations, Language and Applications

Thèse dirigée par Philippe LAHIRE et Philippe COLLET
soutenue le 23 – 09 – 2011

Jury :

Rapporteurs	Krzysztof CZARNECKI	Professeur, University of Waterloo (Canada)
	Jean-Marc JÉZÉQUEL	Professeur, Université de Rennes 1
Examinateurs	Robert DE SIMONE	Directeur de Recherche, INRIA Sophia Antipolis
	Patrick HEYMANS	Professeur, University of Namur (Belgium)
	Robert B. FRANCE	Professeur, Colorado State University (USA)
Co-Directeur	Philippe LAHIRE	Professeur, Université Nice - Sophia Antipolis
Co-Directeur	Philippe COLLET	Maître de Conférences, Université Nice - Sophia Antipolis

Abstract

Software Product Line (SPL) engineering is a paradigm shift towards modeling and developing software system families rather than individual systems. It focuses on the means of efficiently producing and maintaining multiple similar software products, exploiting what they have in common and managing what varies among them. This is analogous to what is practiced in the automotive industry, where the focus is on creating a single production line, out of which many customized but similar variations of a car model are produced.

Feature models (FMs) are a fundamental formalism for specifying and reasoning about commonality and variability of SPLs. FMs are becoming increasingly complex, handled by several stakeholders or organizations, used to describe features at various levels of abstraction and related in a variety of ways. In different contexts and application domains, maintaining a single large FM is neither feasible nor desirable. Instead, multiple FMs are now used.

In this thesis, we develop theoretical foundations and practical support for managing multiple FMs. We design and develop a set of composition and decomposition operators (aggregate, merge, slice) for supporting separation of concerns. The operators are formally defined, implemented with a fully automated algorithm and guarantee properties in terms of sets of configurations. We show how the composition and decomposition operators can be combined together or with other reasoning and editing operators to realize complex tasks. We propose a textual language, FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning), which provides a practical solution for managing FMs on a large scale. An SPL practitioner can combine the different operators and manipulate a restricted set of concepts (FMs, features, configurations, etc.) using a concise notation and language facilities. FAMILIAR hides implementation details (e.g., solvers) and comes with a development environment. We report various applications of the operators and usages of FAMILIAR in different domains (medical imaging, video surveillance) and for different purposes (scientific workflow design, variability modeling from requirements to runtime, reverse engineering), showing the applicability of both the operators and the supporting language. Without the new capabilities brought by the operators and FAMILIAR, some analysis and reasoning operations would not be made possible in the different case studies.

To conclude, we discuss different research perspectives in the medium term (regarding the operators, the language and validation elements) and in the long term (e.g., relationships between FMs and other models).

Résumé

L'ingénierie des lignes de produits logiciels (LdPs) est un paradigme pour la modélisation et le développement de familles de systèmes logiciels plutôt que de systèmes logiciels individuels. Son objectif porte sur les moyens de produire et maintenir efficacement des produits logiciels similaires en exploitant ce qu'ils ont en commun et en gérant ce qui varie entre eux. Par analogie, la pratique dans l'industrie automobile est de construire une ligne de production dans laquelle des variations personnalisées mais tout de même similaires de modèles de voitures sont produits. Les feature models (FMs) sont une représentation fondamentale pour spécifier et raisonner sur la commonalité et la variabilité des LdPs en termes de features (caractéristiques). Les FMs deviennent de plus en plus complexes, manipulés par plusieurs développeurs ou organisations, utilisés pour décrire des features à divers niveaux d'abstraction et qui sont mises en relation de différentes façons. Maintenir un seul gros FM n'est ni réaliste ni souhaitable. Au contraire une tendance forte est de considérer de multiples FMs. Dans cette thèse, nous développons les fondations théoriques et un support pratique pour gérer de multiples FMs. Nous concevons et développons un ensemble d'opérateurs de composition et de décomposition (aggregate, merge, slice) pour supporter la séparation des préoccupations. Les opérateurs sont formellement définis et implémentés avec un algorithme qui guarantie des propriétés sémantiques. Nous montrons comment les opérateurs de composition et de décomposition peuvent être combinés ensemble ou avec d'autres opérateurs d'édition ou de raisonnement pour réaliser des tâches complexes. Nous proposons un language textuel, FAMILIAR (pour FeAture Model scrIpt Language for manIpulation and Automatic Reasoning), qui fournit une solution opérationnelle à la gestion de multiples FMs à large échelle. Un utilisateur des FMs peut combiner les différents opérateurs et manipuler un ensemble restreint de concepts (FMs, features, configurations, etc.) en utilisant une notation concise et des facilités linguistiques. FAMILIAR cache les détails d'implémentations (e.g., solveurs) et est supporté par un environnement de développement complet. Nous décrivons plusieurs applications de ces opérateurs et utilisations de FAMILIAR dans différents domaines (imagerie médicale, vidéo protection) et pour différents objectifs (conception de workflows scientifiques, modélisation de la variabilité des exigences à l'exécution, rétro ingénierie), démontrant l'applicabilité à la fois des opérateurs et du langage de support. Sans les nouvelles capacités fournies par les opérateurs et FAMILIAR, certaines opérations d'analyse et de raisonnement n'auraient pas été possibles dans les différents cas d'études. Pour conclure, nous discutons les différentes perspectives de recherche à moyen terme (opérateurs, langage, éléments de validation) et à long terme (e.g., relations entre les FMs et les autres modèles).

Acknowledgments

First of all, I would like to thank my two PhD supervisors, Philippe Collet and Philippe Lahire. I learned almost everything about research and teaching from them. They guided my research in many ways, gave me constant feedbacks and advices in a pedagogical manner and largely contribute to make me progress. I cannot remember how many brain stormings and meetings we have got: I am not sure whether a PhD student already had the same amount of effort and time from supervisors. Philippe Lahire supported me since my first year in my Master's degree (we started with the object-oriented language Eiffel). Philippe gave me the opportunity to start a PhD thesis. He was always here to ensure that everything goes fine, to organize collaborations and to exchange research ideas. I knew Philippe Collet since the bachelor's degree, as a teacher, when I learned the main principles of software engineering and there is no doubt I have chose to continue in this field thanks to his high-quality courses. Philippe showed me what an energetic researcher and a pedagogical advertiser are. I do not really want to distinguish one supervisor from another. My two PhD supervisors are not only brilliant individually, they are also very complementary. We have a lot of fun all together, not necessary in Sophia, and I am sure it will continue! Thank you!

I would like to thank the members of my thesis committee. Thank you for having accepted to review my thesis and for coming in France in your very busy schedule. Believe me or not, I wanted exactly this jury, composed of many researchers that have inspired me a lot. When your research is related to feature modeling, software product line engineering or model-driven engineering, you cannot get out of it: you must read and study carefully the prominent works of Krzysztof Czarnecki. The presence of Krzysztof in the committee was an exciting challenge, especially regarding my research focus. Jean-Marc Jézéquel had also a direct impact on my research regarding his major contributions in the areas of model-driven engineering and aspect-oriented modeling. I was fortunate to attend a talk of Jean-Marc in april 2009 in Sophia Antipolis just before the HDR of Mireille Blay-Fornarino: I had enough inspirations for the next two years, and hopefully, there is more to come. Robert B. France played a crucial role in my PhD thesis. Robert has constantly supported and challenged me, reviewed most of my preliminary ideas and gave me important feedbacks and advices since 2009. In a way, he is my third supervisor. I have this constant feeling we (i.e., not only me) have learned a lot from being with Robert. I hope it will continue. Thank you Robert! I am also very honored to have Patrick Heymans in my committee. Patrick is a leading researcher in software product line engineering and in feature modeling. I have been fascinated by his research methodology and I cannot wait to work with him in Namur! Robert De Simone is the last member of the committee. He was member of another jury in 2008, the one who awards my MESR doctoral grant. So I guess the wheel

has come full circle.

During the last three years, I had the chance to be part of the MODALIS (formerly RAINBOW) team. MODALIS proved to be an excellent place for academic development and collaboration. I would like to thank Johan Montagnat, team leader. Johan gave me the opportunity to pursue a PhD in his team. Very early, he encourages me to investigate further the issues that arose in the medical imaging domain. He pointed out the weaknesses and potential strengths of my approaches and gave me numerous advices I will not forget. More generally, I would like to thank all the members of the MODALIS team, Mireille for the inspiring discussions, Diane for all the feedbacks and your great support during my master's thesis, Alban for his great support (how many SVN accounts created?) as well as his research skills, Sébastien my "co-bureau" for being a great researcher and bearing me during 3 years at the office (and some couple of days in Colorado), Javier one of the most interesting guy I ever met, Tristan and his PhD thesis (the first I read), Tram, Filip, Hélène, Pierre, Nadia, all members of the RAINBOW team (Michel, Jean-Yves, Stéphane, Gaëtan, Nicolas, Vincent, Christophe). Special thanks to Sabine, our team assistant, who efficiently manages all my trips.

The work developed in this thesis would not be the same without the research collaborations I had. Many ideas have emerged thanks to fruitful discussions and case studies. I would like to thank all my co-authors. I consider Jean-Paul Rigault and Sabine Moisan have a major influence in the research direction I toke. We investigated together the role of model-driven engineering in video surveillance systems, we have got a lot of interesting exchanges and the chance to work on real software intensive systems. Thank you very much for having confidence in my work and more generally for your simplicity, your kindness and your research skills. I also want to thank Franck Fleurey which was also part of the collaboration (he visited us in 2009). Another fruitful collaboration is the one with Anthony Cleve, Laurence Duchien and Philippe Merle. I would like to thank Laurence for having initiated and organized the collaboration, Anthony for being at the root of the research proposal and for being one of the first believer of FAMILIAR, Philippe for his expertise in FraSCAti. We will continue, no doubt. I am coming Lille and Namur! Many research colleagues develop and share tools. I thank Steven She and Andrzej Wąsowski for sharing the implementation of the synthesis algorithm, Christian Kästner for his support with FeatureIDE, Marcilio Mendonca for his support with SPLOT, and Sergio Segura for sharing his prototype of catalog rules. I also would like to thank Martin Fagereng Johansen (now PhD candidate in SINTEF, Norway), Charles Vanbeneden and Foudil Bendjabeur for their research or valuable development effort.

I gratefully thank my family and friends for their ongoing support far beyond this thesis. By family, I mean my parents Elise and Alain, my brother Adrien, my grandmother, Chloé my Las Vegas wife, Thomas, Francette, Claire and all related people ;-) Chloé made the exploit to support me for more than 5 years now. I would simply be unable to start a PhD thesis, never mind finish a PhD thesis, without her. Special thanks to my parents that always believe in me and that deploy an extraordinary amount of energy to help me. I have a special thought for Geneviève and my grandfather. I will never forget. Last but not least, I would like to thank the smarter chess international grand master ever Fabien, Gautier and Anaïs, Brice, Chloé (yet another!), Laura, Audrey, Guillaume, Laurent, Jérôme, Grace, Cédric, Mélanie, Stevens and Charlotte, Iannis, my former chess trainers (Jean-Marc, Darko, Josif, Murtas, Vladislav) and the firemen of Var.

Contents

List of Figures

List of Tables

One

Introduction

1.1 Context and Motivation

There is no doubt that the world is becoming increasingly dependent on software. It is now an essential element of many organizations (finance, retail, public sectors) and even our daily lives depend on complex *software-intensive systems*, from banking and communications to transportation and medicine.

It is obvious that these software systems should provide the required capability, be of sufficient quality, be customizable, and be delivered at an acceptable price. Unfortunately, the reality is not like that. Several studies report that software projects are running overtime, are unmanageable, are running over-budget, are producing low-quality software that did not meet the original requirements and that are difficult to maintain [Northrop et al. 2006]. The whole set of phenomena, usually defined as the *software crisis*, has been observed early in history (in the 60s) and is still persistent. The discipline of *software engineering* has been created to cope with the software crisis and has a long tradition. The challenge for the research community and the industry has always been to provide the right languages, abstractions, models, methods, and tools to assist software developers in building well-structured and customizable software. Paradigms and concepts such as structured programming, abstract data types, modularization, object orientation, design patterns or modeling languages were all introduced with the clear objective of simplifying the task of engineering a software system.

Nevertheless, all these attempts though useful and practically applied are still not sufficient to deal with the enormous and ever-increasing complexity of contemporary software systems, as well as the evolving customer expectations. With more and more diversity, software products rapidly evolve and increase in size and complexity. Due to the increasing demand of highly customized products and services, software organizations have now to produce many complex variants accounting not only for differences in software functionalities but also for differences in hardware, operating systems, localization, user preferences, etc. Obviously, one do not want to develop from scratch and independently all of the variants: one wants to achieve reuse and create software systems from existing software. As software engineering, the methodical reuse of software artifacts has a long tradition and was early motivated in 1968 [McIlroy 1968], leading to the development of several languages, methods, and tools. Object-oriented programming, object-oriented frameworks, aspect-oriented programming, components, services, to name a few, have been proposed but current research and practical experience suggest that we need more effective and systematic way to achieve software reuse and customization.

Software Product Line (SPL) engineering is a paradigm shift towards modeling and de-

veloping software system families rather than individual systems. SPL engineering embraces the ideas of mass customization and software reuse. It focuses on the means of efficiently producing and maintaining multiple similar software products, exploiting what they have in common and managing what varies among them. This is analogous to what is practiced in the automotive industry, where the focus is on creating a single production line, out of which many customized but similar variations of a car model are produced. In an SPL approach, different software products of a domain can be assembled from common and reusable artifacts (core assets). Throughout the software development process, SPL engineering institutionalizes systematic reuse of artifacts from various natures that include reusable software components, domain models, requirements statements, documentation and specifications, test cases, source code, etc. The basic assumption behind SPL engineering is that reuse-in-the-large works best in families of related systems (e.g., software products from a same domain) from which common/reusable artifacts have been initially identified and co-developed.

In the vast majority of approaches, reusing artifacts of an SPL requires the systematic identification and exploitation of *commonality* (i.e., the common characteristics of products) and *variability* (i.e., the differences between products). Commonality and variability are everywhere and cross-cut many different types of SPL artifacts. A lack of flexibility in the reusable artifacts or a scope that is too large may have important consequences on the SPL engineering process (e.g., unnecessary software development effort). Therefore a fundamental challenge is to model and manage commonality and variability of an SPL.

In this thesis, we focus on *feature models*, a formalism originally introduced by [Kang et al. 1990] and now widely used in SPL engineering. Feature models allow SPL practitioners to describe commonality and variability of an SPL in terms of features. Feature models characterize, in a very concise way, the valid combination of features (also called configurations) supported by an SPL – for each valid configuration authorized by a feature model should correspond a product of an SPL.

1.2 Contribution

The analysis of the state-of-the-art reveals that feature models are used in a wider scope than originally planned in 1990. The reason is that practitioners need to manage an increasing complexity characterized by a large number of concerns, artifacts and sub-systems that are part of an SPL. Using only one monolithic feature model is not realistic to model variability of an SPL. Feature models are rather *multiple*: They document variability of several (different kinds of) artifacts, at different levels of abstractions, are used and perceived differently by different stakeholders or suppliers and are combined together to model the entire variability of a system. We defend the idea that the phenomenon of increasing complexity of feature models can be handled by applying the principles of *separation of concerns (SoC)*. The contribution of the thesis is as follows:

- a set of composition and decomposition operators dedicated to the formalism of feature models for supporting separation of concerns. The operators are formally defined, implemented with a fully automated algorithm and guarantee properties in terms of sets of configurations. We show how the composition and decomposition operators can be combined together or with other reasoning and editing operators to realize complex tasks ;

- a domain-specific textual language, FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning), which provides a practical solution for managing feature models on a large scale. An SPL practitioner can combine the different operators and manipulate a restricted set of concepts (feature models, features, configurations, etc.) using a concise notation and language facilities. FAMILIAR hides implementation details (e.g., solvers) and comes with a development environment ;
- various applications of the operators and usages of FAMILIAR in different domains (medical imaging, video surveillance) and for different purposes (scientific workflow design, variability modeling from requirements to runtime, reverse engineering), showing the applicability of both the operators and the supporting language.

1.3 OUTLINE

In Part I, we discuss the state-of-the-art and present background on software product line engineering (Chapter 2) and feature models (Chapter 3). We then describe an example that involves the management of set of feature models, called *multiple feature models*, in the domain of medical imaging workflows (Chapter 4). Using this example, we identify different requirements for separation of concerns in feature modeling. We also show that the identified issues are not properly tackled by existing works. The example will be used throughout the document to illustrate and evaluate our contributions.

In Part II, we apply separation of concerns to feature models using a set of complementary *operators*. In Chapter 5, we describe the semantics and the implementation of a set of composition operators (insert, aggregate, merge). In Chapter 6, we focus on the merge operator that produces compact feature models from a set of existing feature models and thus ease their management and analysis. We illustrate the use of the merge operator when building a catalog of image analysis services provided by different suppliers. In Chapter 7, we present a slicing technique to decompose feature models that, combined with other composition/reasoning operators, brings new capabilities to SPL practitioners (update and extraction of feature model views, reconciliation of feature models and reasoning about different kinds of variability, etc.).

In Part III, we present the *language* FAMILIAR. In Chapter 8, we illustrate its syntax, the integration of composition/decomposition operators, as well as other facilities to manipulate and reason about feature models. In Chapter 9, details of the implementation and performance of two important operators (merge and slice) are given.

In Part IV, we show how the operators and FAMILIAR have been used in different *applications*. In Chapter 10, we revisit the example introduced in Chapter 4. We show how we can combine multiple variability artifacts to assemble coherent workflows and facilitate the selection of services from among sets of competing services organized in a catalog. In Chapter 11, we show how in dynamic adaptive systems, such as video surveillance systems, the variability requirements can be expressed and then refined at design time so that the set of valid software configurations to be considered at runtime may be highly reduced. In Chapter 12, we develop automated techniques to extract and combine different variability descriptions of a software architecture. Then, alignment and reasoning techniques are applied to integrate the architect knowledge and reinforce the extracted feature model. We illustrate the reverse engineering process when applied to a representative software system, FraSCAti, and we report on our experience in this context.

In Part V, Chapter 13 summarizes the contribution, while Chapter 14 discusses different perspectives in the medium term (regarding the operators, the language and validation elements) and in the long term (e.g., relationships between feature models and other models).

1.4 Bibliographical Notes

The research presented in this thesis reuses and extends publications of the author. The list of peer-reviewed papers is given below.

Journal

1. [Acher et al. 2011f] Acher, M., Collet, P., Lahire, P., Gaignard, A., France, R., and Montagnat, J. (2011). Composing Multiple Variability Artifacts to Assemble Coherent Workflows. *Software Quality Journal – Special issue on Quality Engineering for SPLs*, LNCS, Springer (Chapter 10) ;

Conference

2. [Acher et al. 2009b] Acher, M., Collet, P., Lahire, P., and France, R. (2009). Composing Feature Models. In *2nd International Conference on Software Language Engineering (SLE'09)*, LNCS, Springer (Chapter 5) ;
3. [Acher et al. 2010a] Acher, M., Collet, P., Lahire, P., and France, R. (2010). Comparing Approaches to Implement Feature Model Composition. In *6th European Conference on Modelling Foundations and Applications (ECMFA'10)*, LNCS, Springer (Chapter 5) ;
4. [Acher et al. 2011e] Acher, M., Collet, P., Lahire, P., and France, R. (2011). Slicing Feature Models. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE'11), short paper*, IEEE/ACM (Chapter 7) ;
5. [Acher et al. 2011b] Acher, M., Collet, P., Lahire, P., and France, R. (2011). A Domain-Specific Language for Managing Feature Models. In *Symposium on Applied Computing (SAC), Programming Languages Track*, ACM (Chapter 8) ;
6. [Acher et al. 2011c] Acher, M., Collet, P., Lahire, P., and France, R. (2011). Decomposing Feature Models: Language, Environment, and Applications. In *Automated Software Engineering (ASE'11), short paper: demonstration track*, IEEE/ACM (Chapter 8);
7. [Acher et al. 2010c] Acher, M., Collet, P., Lahire, P., and France, R. (2010). Managing Variability in Worklow with Feature Model Composition Operators. In *9th International Conference on Software Composition (SC'10)*, LNCS, Springer (Chapter 10) ;
8. [Acher et al. 2011g] Acher, M., Collet, P., Lahire, P., Moisan, S., and Rigault, J.-P. (2011). Modeling Variability from Requirements to Runtime. In *16th International Conference on Engineering of Complex Computer Systems (ICECCS'11)*, IEEE (Chapter 11) ;
9. [Moisan et al. 2011] Moisan, S., Rigault, J.-P., Acher, M., Collet, P., and Lahire, P. (2011). Run Time Adaptation of Video-Surveillance Systems: A Software Modeling Approach. In *8th International Conference on Computer Vision Systems (ICVS'2011)*, LNCS, Springer (Chapter 11) ;
10. [Acher et al. 2011a] Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., and Lahire, P. (2011). Reverse Engineering Architectural Feature Models. In *5th European Conference on Software Architecture (ECSA'11), long paper*, LNCS, Springer (Chapter 12)

Workshop

11. [Acher et al. 2008a] Acher, M., Collet, P., and Lahire, P. (2008). Issues in Managing Variability of Medical Imaging Grid Services. In *MICCAI-Grid Workshop 2008* (Chapter 4) ;
12. [Acher et al. 2008b] Acher, M., Collet, P., Lahire, P., and Montagnat, J. (2008). Imaging Services on the Grid as a Product Line: Requirements and Architecture. In *Service-Oriented Architectures and Software Product Lines - Putting Both Together (SOAPL'08)* – associated workshop issue of SPLC 2008, IEEE, (Chapter 4) ;
13. [Acher et al. 2011d] Acher, M., Collet, P., Lahire, P., and France, R. (2011). Managing Feature Models with FAMILIAR: a Demonstration of the Language and its Tool Support. In *Fifth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'11)*, ACM (Chapter 8) ;
14. [Acher et al. 2009c] Acher, M., Lahire, P., Moisan, S., and Rigault, J.-P. (2009). Tackling High Variability in Video Surveillance Systems through a Model Transformation Approach. In *MiSE '09: Proceedings of the 2009 international workshop on Modeling in software engineering at ICSE 2009*, IEEE (Chapter 11) ;
15. [Acher et al. 2009a] Acher, M., Collet, P., Fleurey, F., Lahire, P., Moisan, S., and Rigault, J.-P. (2009). Modeling Context and Dynamic Adaptations with Feature Models. In *4th International Workshop Models@run.time at Models 2009 (MRT'09)* (Chapter 11) ;
16. [Fagereng Johansen et al. 2010] Fagereng Johansen, M., Fleurey, F., Acher, M., Collet, P., and Lahire, P. (2010). Exploring the Synergies Between Feature Models and Ontologies. In *International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010)*, volume 2 of *SPLC '10*, (Chapter 14)

Under Review

17. [Acher et al. 2012] Acher, M., Collet, P., Lahire, P., and France, R. (2012). Separation of Concerns in Feature Modeling: Support and Applications. In *11th International Conference on Aspect-Oriented Software Development (AOSD'12)*, ACM, submitted, currently under review (Chapter 7);

Part I

State of the Art

Two

Software Product Line Engineering

In this chapter, we give a brief introduction to software product line engineering and its main concepts (customization, reuse, domain, variability). We notably describe the fundamental separation between domain and application engineering. We discuss the possible roles of model-based approaches in this context. Furthermore, we present some approaches, techniques and tools to model, manage and realize variability of a software product lines.

2.1 Software Product Lines

The traditional focus of software engineering is to develop individual software systems, i.e., one software system at a time. A typical development process starts with the analysis of customers' requirements and then several development steps (specification, design, implementation, testing) are performed. The result obtained is a single software product. In contrast, *Software Product Line (SPL)* engineering focuses on the development of multiple similar software systems from a common codebase [Bass et al. 1998, Clements and Northrop 2001, Pohl et al. 2005].

2.1.1 Mass Customization and Reuse

SPL engineering relies on the idea of *mass customization* [Pine 1999] known from many industries. For instance, in the automotive industry, the focus is on creating a single production line, out of which many customized but similar variations of a car model are produced. Mass customization takes advantage of similarity principle and modular design to massively produce customized products. Many industries, such as avionics, telecommunications, or automotive industry, are now building the same *multiple similar software-intensive products* over and over again. Therefore, there is an opportunity to massively reuse common software artifacts. Software mass customization, and so SPL engineering, focus on the *means* of efficiently producing and maintaining multiple similar software products, exploiting what they have in common and managing what varies among them. SPL engineering aims at developing related variants in a systematic, coordinated way and providing tailor-made solutions for different customers. Instead of individually developing each variant from scratch, commonalities are conceived only once.

The following definition given by [Clements and Northrop 2001] captures the general idea behind an SPL:

> *"A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."*

Historically, the idea to develop a set of related software products can be traced back to the idea of program families [Parnas 1976]. SPLs drew attention of the software engineering community only in the mid of 1990s when software begun to be integrated massively in families of electronic products, mobile phones [Maccari and Heie 2005] being the most popular, but several other areas, such as automotive systems, medical systems, aerospace or telecommunication were also targeted by SPLs.

An important characteristic of SPL engineering is its emphasis on *software reuse* [Jacobson et al. 1997]. A general definition of software reuse is given in [Krueger 1992]

> *"Software reuse is the process of creating software systems from existing software rather than building software systems from scratch."*

Reuse, as a software strategy for decreasing development costs and improving quality, is not a new idea. It has been a popular topic since the publication of seminal papers [McIlroy 1968, Parnas 1976]. Past reuse strategies, which focused on reusing relatively small pieces of code or opportunistically cloning code designed for one system for use in another, have not been profitable. Object-oriented programming offers several mechanisms to develop generic units of functionality, but reuse is typically achieved on a small scale. Object-oriented frameworks provide reusable concepts but are often large, complex and/or undocumented so that locating and instantiation these concepts for implementing a software system is hard, even with extensive skills. Components – as independently-deployable units of composition with contractually specified interfaces [Szyperski et al. 2002] – promise reuse-in-the-large, but the optimal trade-off between functionality, size and reusability of components is very hard to find.

Reuse is still a challenging problem in software engineering. Developing software system families rather than individual systems can facilitate reuse, since products of the family share common features (e.g., functionality) and are from the same domain. As noticed by [Glass 2001]:

> *"Reuse-in-the-large works best in families of related systems, and thus is domain dependent."*

SPL engineering is based on this idea. It is a paradigm shift towards planned, proactive and systematic reuse of the common artifacts (also called core assets), where related products are treated as a product family. Their co-development can then be planned from the beginning, instead of starting from scratch or copying and editing from a previous product. SPLs promise several benefits, especially for organizations that develop software-intensive products, but research is more and more needed due to the increasing complexity and diversity of these products. Part of the research on SPL engineering is related to the analysis of business and organizational factors, such as market analysis, responsibilities assignment, risks management, etc. It mostly concerns "product line" management. In this thesis, we mainly focus on technical issues related to software development efforts.

2.1.2 Expected Benefits

Compared to traditional single-product development, SPLs promise several benefits [Bass et al. 1998, Clements and Northrop 2001, Pohl et al. 2005]: Due to co-development and systematic reuse, software products are expected to be produced faster, with lower costs, and higher quality. The ability to customize software products to different contexts opens new perspectives. For instance, although resources in embedded systems are scarce and

hardware is heterogeneous, efficient variants can be tailored to a specific device or scenarios [Beuche et al. 2004]. Real world success stories of software mass customization come from diverse areas such as mobile phones, telecom networks, medical systems, computer printers, diesel engines, cars, ships, and airplanes[1]. There are many companies that report significant benefits from SPLs: Alcatel, Hewlett Packard, Philips, the Boeing Company, and Robert Bosch GmBh7 presented their experiences at the 2000 SPL conference [Northrop 2002]. For example, Bass et al. summarize that, with SPLs, Nokia can produce 30 instead of previously 4 phone models per year; Cummins, Inc. reduced development time for a software for a new diesel engine from one year to one week; Motorola observed a 400 % increase in productivity, etc [Bass et al. 1998].

2.1.3 Extractive, Reactive, Proactive Strategies

Although many organizations were aware of the huge potential benefits of software mass customization, the associated costs, risks, and resources are a prohibitive *adoption barrier* for many [Krueger 2001]. Historically, practices of SPL engineering combine a massive upfront investment. In the worst case, it consists in analyzing, architecting, designing, and implementing all product variations on the foreseeable horizon. This proactive approach might suit organizations that can predict their SPL requirements well into the future and that have the time and resources for a long development cycle. Unfortunately, organizations cannot afford to slow or stop production for a few months, even if the potential payoff is huge.

Clements and Krueger argue that organizations transitioning to SPL engineering need more flexible process, with low-cost adoption methods, that go beyond a pure proactive approach [Clements and Krueger 2002]. Krueger proposes three adoption strategies – proactive, reactive, and extractive approaches – that can be possibly combined by SPL organizations [Krueger 2001; 2006].

With the *proactive* approach, the organization analyzes, designs and implements a complete software mass customization production line to support the full scope of products needed on the foreseeable horizon. From the analysis and design, a complete set of common and varying requirements, product definitions, source code, etc. are implemented.

With the *reactive* approach, the organization incrementally grows their software mass customization production line when the demand arises for new products or new requirements on existing products. The common and varying artifacts are incrementally extended in reaction to new requirements. This incremental approach offers a quicker and less expensive transition into software mass customization.

With the *extractive* approach, the organization capitalizes on existing custom software systems by extracting the common and varying source code into a single production line. This high level of software reuse enables an organization to very quickly adopt software mass customization. For example, developers often take an extractive approach for creating the SPL by refactoring and decomposing one or more legacy applications into reusable and composable units.

These approaches are not necessarily mutually exclusive. For example, a common approach is to bootstrap a software mass customization effort using the extractive approach and then move on to a reactive approach to incrementally evolve the SPL over time.

[1]For a larger overview of famous SPLs, consult SEI's SPL Hall of Fame: http://www.sei.cmu.edu/productlines/plp_hof.html

Northrop proposes a related classification and distinguishes different production strategies for realizing the core assets [Northrop 2002]. The production strategy can be *top down* (starting with a set of core assets and spinning products off of them), *bottom up* (starting with a set of products and generalizing their components to produce the product line as-sets), or a mix of both.

2.2 Framework for Software Product Line Engineering

Organizations that have succeeded with SPLs vary widely in the nature of their software products and their organizational structure. Several methodologies, techniques and tools have been developed in the context of SPL engineering. Nevertheless, there are universal essential activities and practices that emerge, being related to the ability to construct new products from a set of common assets while working under the constraints of various organizational contexts and starting points. As a result, several *frameworks*[2] have been proposed in the ITEA projects, ESAPS, CAFÉ, FAMILIES [van der Linden 2002, Böckle et al. 2004] or in books (e.g., see [Czarnecki and Eisenecker 2000, Clements and Northrop 2001, Pohl et al. 2005]). The SEI also maintains an online document that describes a framework for product line development [SEI 2011]. These frameworks are all based on the differentiation between the domain and application engineering processes originally proposed by [Weiss and Lai 1999] (see Section 2.2.1). Furthermore, numerous paradigms additionally distinguish between problem space and solution space (see Section 2.2.2). In this context, model-based approaches have a role to play: We present their principles and some background (see Section 2.2.3).

2.2.1 Domain Engineering and Application Engineering

SPL engineering is separated in two complementary phases. Domain engineering is concerned with development *for reuse* while application engineering is the development *with reuse* (see Figure 2.1).

The idea behind this approach to SPL engineering is that the investments required to develop the reusable artifacts during domain engineering, are outweighed by the benefits of deriving the individual products during application engineering [Deelstra et al. 2004; 2005]. A fundamental reason for researching and investing in sophisticated technologies for SPLs is to obtain the maximum benefit out of this up-front investment, in other words, to minimize the proportion of application engineering costs (see Figure 2.2).

Domain Engineering. The process to develop a set of related products (i.e., an SPL) instead of a single product is called domain engineering. An SPL must fulfill not only the requirements of a single customer but the requirements of multiple customers in a domain, including both current customers and potential future customers. Therefore the entire domain and its potential requirements are analyzed, for example, to scope the SPL and identify what differs between products, to identify reusable artifacts and plan their development, etc. Domain Engineering is development *for reuse*: common and reusable artifacts (requirements, components, test cases, etc.) are factoring out so that their reuse is facilitated. It is usually composed of four activities: domain analysis, domain design,

[2]The word framework refers here to *conceptual framework* that serves as the guiding principles of research within a particular discipline. There is no intended connection to *software frameworks*.

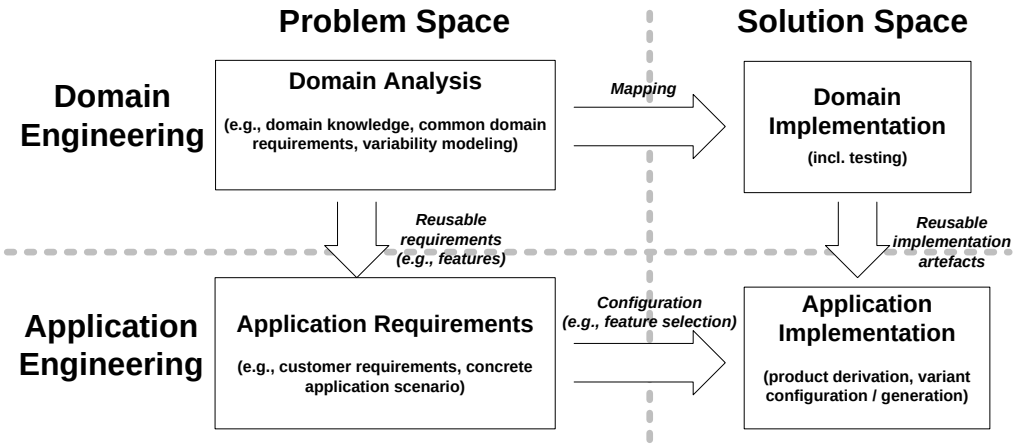


Figure 2.1: Framework for SPL engineering: domain and application engineering, problem and solution space

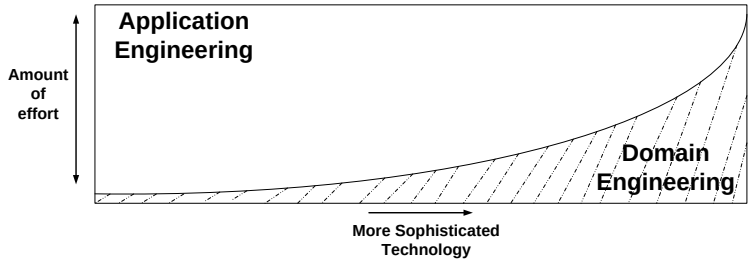


Figure 2.2: Challenge for SPL engineering: decreasing the proportion of application engineering effort [Deelstra et al. 2004]

domain coding and domain testing. In the domain analysis, the commonalities and differences between potential variants are identified and described, for example, in terms of features. In this context, a feature is a first-class domain abstraction, typically an end-user visible increment in functionality [Apel and Kästner 2009] (see Section 3.2.1 for a detailed discussion of the term feature). Then, developers design and implement the SPL such that different variants can be constructed from common and variable parts.

Application Engineering. Application Engineering is development *with reuse* (also called product development). Concrete products are derived using the common and reusable artifacts developed in domain engineering. It is composed of four activities, in line with the activities of domain engineering: application requirements engineering, application design, application coding and application testing. This process is built on the domain engineering one and consists in developing a final product, reusing the reusable artifacts and adapting the final product to specific requirements. Ideally, the customer's requirements can be mapped to elements (e.g., features) identified during domain engineering, so that

the variant can be constructed from existing common and variable parts of the SPL implementation. The process of constructing products from the domain assets is called *product derivation*. Depending on the form of implementation, there can be different automation levels of the application engineering process, from manual development effort to more sophisticated technology including automated variant configuration and generation.

2.2.2 Problem Space and Solution Space

One of the toughest challenges for software engineering, and by extension SPL engineering, is the inherent *complexity* of modern software systems. As noted in [Brooks 1987], there are two types of complexity. On the one hand, *essential complexity* is inherent to the problem being solved and cannot be removed. On the other hand, *accidental complexity* is non-essential to the problem and is more related to implementation issues. In particular, the use of inadequate technology with regard to the targeted problem (e.g., the use of an assembly language to develop a chess engine) can add significant accidental complexity during the software development. The invention of high level programming languages, for instance, had made significant improvement in the area of accidental complexity.

The terms problem and solution (or "problem model" and "solution model", "problem domain" and "solution domain", "problem space" and "solution space", and so on) are commonly employed to denote the contrast between the system under study and its application domain [Génova et al. 2009]. This can also be expressed with the terms "problem analysis" and "solution design". Several paradigms (see below) explicitly distinguish between the problem space and the solution space and propose to separate the two spaces during software (e.g., SPL) development (see Figure 2.1).

The *problem* space comprises domain-specific abstractions that describe the requirements on a software system and its intended behavior. For instance, domain analysis takes place in the problem space, and its results are usually documented in terms of features. The *solution* space comprises implementation-oriented abstractions, such as code artifacts. In the solution space, abstractions have been used in the field of programming languages ranging from assembly languages to object-oriented languages that meaningfully help the programmer to organize both structural and behavioral information constituting software.

Between elements (e.g., requirements, features) in the problem space and elements in the solution space, there is a *mapping* that describes which implementation artifact belongs to which requirements or features. Depending on the implementation approach and the degree of automation, this mapping can have different forms and complexity, from simple implicit mappings based on naming conventions to complex rules encoded in generators or transformations [Czarnecki and Eisenecker 2000].

Figure 2.1 gives a rather idealized, simplified view of the framework. Mappings between the problem and the solution space can be chained ; a mapping could take two or more specifications and map them to one (or more) solution space (this is common when different aspects of a system are represented) ; a mapping can also implement a problem space in terms of two or more solution spaces [Czarnecki and Eisenecker 2000]. Furthermore the transition between domain and application engineering can be more or less complex (e.g., the degree of automation may vary (see Figure 2.2).

2.2.3 Possible Roles of Model-Driven Engineering

Abstraction and Model. In order to be manageable, we need to reduce this number of

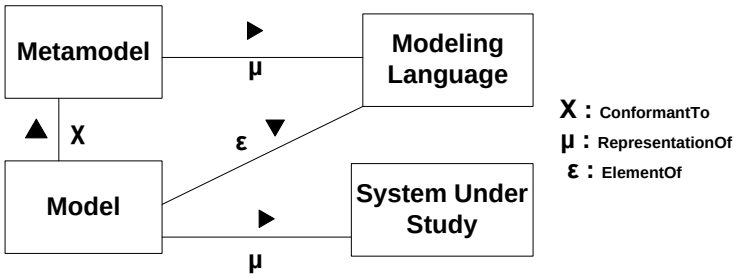


Figure 2.3: Model, Metamodels, Modeling Language, SUS

elements and focus on the important points rather than details. This is exactly the purpose of *abstraction* [Kramer 2007]:

> *"Abstraction is the mapping from one representation of a problem to another which preserves certain desirable properties and which reduces complexity."*

Approaches centered on the use of *models* (e.g., model-driven engineering, aspect-oriented modeling, generative programming) have been proposed in order to effectively represent, define and use abstractions for any part of a software system. Models have been used for ages in various scientific disciplines including biology, economy, house building or geography [Bézivin 2005]. In the context of software engineering, several definitions of the notion of model have been given (e.g., nine definitions are reported in [Muller et al. 2009]), but, in essence, a model is an abstraction of some aspect of a *system under study (SUS)*. A SUS can be an existing system, or a system under development. A model is an abstraction since some details are hidden or removed to *simplify* and focus attention. A model is an abstraction since *general* concepts can be formulated by abstracting common properties of instances or by extracting common features from specific examples.

Models are created to serve particular *purposes*, for example, to present a human understandable description of some aspect of a SUS or to present information in a form that can be *mechanically* analyzed. Various kinds of automated analysis can be performed to reason about some relevant properties of a SUS. Moreover models can processed by computer-based tools in order to derive other useful models and/or some of the artifacts (such as test cases, performance profiles, or documentation) composing a real software system. This derivation process is called *model transformation*. Numerous approaches and techniques have been developed for supporting model transformation (e.g., see [Czarnecki and Helsen 2006] for an overview of features of model transformation approaches).

Model, Metamodel, Language. As a model is an abstraction (here, simplification) of a SUS, the set of questions addressable by a model is necessarily a *subset* of the questions that can be asked to the actual SUS. The set of statements expressed in a model typically uses a set of predefined modeling constructs and constraints restricting statement usage. It is the role of a *metamodel* to define these constructs (i.e., their abstract syntax) and constraints; without a metamodel, models cannot be built, cannot be validated or processed by tools. Simply providing a metamodel is not sufficient to use models. It is also needed to describe what is the meaning of the individual metamodel constructs as well as their composition.

Model *semantics* provides a mapping between the syntactical constructs and the semantic domain [Harel and Rumpe 2004]. Both a metamodel and its associated semantics are required to fully build, validate and understand models.

Modeling languages are used to build valid models (i.e., conformant to a metamodel). Depending on their purposes, modeling languages may propose textual or graphical constructs which are formally defined (formal syntax and semantics) or semi-formally defined (well-defined syntax but semantics defined in natural language). The first category is well adapted for reasoning (proofs, model checking, etc.). The second category is more accessible and yielded general purpose modeling languages such as the UML (Unified Modeling Language) [Rumbaugh et al. 2004].

Model-based approaches. Various approaches consider models as first-class entities and primary building blocks for constructing software. *Model-driven Engineering (MDE)* is primarily concerned with reducing the gap between problem and solution space through the use of technologies that support systematic transformation of problem-level abstractions to software implementations [Schmidt 2006]. The complexity of bridging the gap is tackled through the use of models that describe complex systems at multiple levels of abstraction and from a variety of perspectives, and through automated support for transforming and analyzing models. In the MDE vision of software development, models are the primary artifacts of development and developers rely on computer-based technologies to transform models to running systems. Various MDE flavors have been developed to date, for example the OMG's Model Driven Architecture (MDA) [Kleppe et al. 2003] initiative which contributed to popularize MDE's ideas via standardization. *Aspect Oriented Modeling (AOM)* approaches provide advanced mechanisms for separation of concerns such as mechanisms for encapsulating crosscutting concerns and for composing concerns to form integrated models [Aspect-Oriented Modeling Workshop Series 2011].

Model-based Approaches in SPL engineering. MDE and other related paradigms are not primary targeted to the development of SPLs. However, SPL engineering can benefit from models and transformations (see Figure 2.1). Models can be used both in the problem space and solution space to represent different aspects of an SPL and provide domain-specific abstractions. Model transformations can be performed to generate lower-level models (in the solution space), and eventually code, from higher-level models (in the problem space). In addition, model transformations can ease and automate the transition between domain and application engineering. In the context of SPLs, MDE is gaining more attention as a provider of techniques and tools that can be used to manage the complexity of SPL development. For example, *Generative Software Development (GSD)* has been proposed and aims at modeling and implementing system families in such a way that a given system can be automatically generated from a specification (model) written in one or more textual or graphical domain-specific languages [Czarnecki and Eisenecker 2000].

2.3 Variability Management

Domain engineering and application engineering (and subsequently the separation between the problem space and the solution space) describe a very general process framework (see Figure 2.1) for SPL engineering. Central and unique to SPL engineering is the

management of *variability*, i.e., the process of factoring out commonalities and systematizing variabilities of documentation, requirements, code, test artifacts, models. For each step of the proposed framework, different approaches, formalisms, and tools can be used. For example, there are different scoping approaches [John and Eisenbarth 2009], different mechanisms to perform domain analysis or model variability (see Section 2.3.2) and different implementation mechanisms (see Section 2.3.3).

2.3.1 Variability

Several definitions of variability have been given in the literature.

Variability in Time vs. Variability in Space. Existing work on software variation management can be generally split into two categories. The variability in time and the variability in space are usually considered as fundamentally distinct dimensions in SPL engineering. Pohl et al. define the variability in time as *"the existence of different versions of an artifact that are valid at different times"* and the variability in space as *"the existence of an artifact in different shapes at the same time"* [Pohl et al. 2005]. Variability in time is primarily concerned with managing program variation over time and includes revision control system and the larger field of software configuration management. The goal of SPL engineering is mainly to deal with variability in space [Erwig 2010, Erwig and Walkingshaw 2011].

Commonality and Variability. Weiss and Lai define variability in SPL as *"an assumption about how members of a family may differ from each other"* [Weiss and Lai 1999]. Hence variability specifies the particularities of a system corresponding to the specific expectations of a customer while commonality specifies assumptions that are true for each member of the SPL. [Svahnberg et al. 2005] adopt a software perspective and define variability as the *"the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context"*. At present, these two definitions are sufficient to capture the notion of variability: the former definition is more related to the notions of domain and commonality while the later focuses more on the idea of customization. Nevertheless, there is no one unique perception or definition of variability: [Bachmann and Bass 2001] propose different categories of variabilities, [Svahnberg et al. 2005] have defined five levels of variability while some authors distinguish essential and technical variability [Halmans and Pohl 2003], external and internal variability [Pohl et al. 2005], product line and software variability [Metzger et al. 2007]. We will further discuss the notion of variability in Section 3.2.

2.3.2 Variability Modeling

Managing variability becomes a primary concern to tame complexity of an SPL. It was early recognized that it is necessary to *model* variation points and their variants to express variability [Jacobson et al. 1997]. In the initial phases of a SPL project, an efficient communication between SPL designers and customers about requirements and hence variabilities is a critical success factor. One way to communicate efficiently on variability is to *model* it (see Section 2.2.3). Variability modelling languages are therefore used to produce variability models. Variability modeling languages can be graphical, textual or a mix of both. Variability models document variation points and their variants, facilitate the identification and delimitation (or scope) of variabilities. They are also used for many implementation

approaches and for automated generation of variants (see next section) or for automated reasoning and error detection (see next chapter).

Amalgamated vs Separated Approaches. As noted by [Haugen et al. 2008], there are two categories of techniques to introduce variability *into modeling languages* (represented as metamodels): amalgamated and separated. The amalgamated approach proposes to augment the metamodel with variability concepts. In [Atkinson et al. 2000, Gomaa 2004, Ziadi and Jézéquel 2006], the authors extend the UML metamodel for modeling variability in multiple UML diagrams like Class or Sequence diagram. In [Morin et al. 2009b, Perrouin et al. 2010], the authors propose a more generic solution that can be applied to any kind of metamodel and fully supported by a tool. Clafer [Bąk et al. 2011] is a meta-modeling language with first-class support for feature modeling. More precisely, the language integrates feature modeling (i.e., a formalism to model variability, see below) into class modeling, so that variability can be naturally expressed in class models.

In a separated approach, the metamodel and the variability metamodel are distinct and are related via a mapping. An illustration of this second approach can be found in [Heidenreich et al. 2008] (supported by FeatureMapper [FeatureMapper 2008]) or in [Czarnecki and Antkiewicz 2005]. It directly relates features and model elements and derives product models by removing all the model elements associated with non-selected features. Another example of the separated approach is VML* [Zschaler et al. 2009], which proposes a family of textual languages dedicated to the modeling of relationships between elements. VML* and FeatureMapper have been compared in [Heidenreich et al. 2010] in terms of automation, scalability, expressiveness and evolution of the mapping.

Variability Models. Feature model is currently the most popular technique to model variability. Many extensions and dialects of feature models have been proposed in literature (e.g., FORM [Kang et al. 1998], FeatureRSEB [Griss et al. 1998], [Riebisch 2003]; [Beuche et al. 2004], [Czarnecki et al. 2005b] ; [Schobbens et al. 2007], [Asikainen et al. 2006; 2007]). The Chapter 3 is dedicated to the formalism of feature model.

There are also other approaches to separate variability description (e.g., the orthogonal variability model (OVM) [Pohl et al. 2005] or Covamof [Sinnema et al. 2006, Sinnema and Deelstra 2008]). Metzger et al. show how to convert an OVM into a feature model, so that the two formalisms can be combined to model variability [Metzger et al. 2007]. A similar approach to representing variability in a separate model can also be found in the decision models [Rabiser et al. 2007]. Feature modeling can be also compared with creating a configuration in the manufacturing industry. One such approach is exemplified by a configurator called WeCoTin [Asikainen et al. 2004]. Chen et al. select and survey 32 approaches in the literature and report on the use of variability models: fourteen approaches used feature models, six approaches used the decision modeling and twelve approaches used other kinds of variability models [Chen et al. 2009].

2.3.3 Variability Implementation

Implementing variability is essential to SPL engineering. Various variability implementation techniques have been developed, tool supported, evaluated and/or compared, both at the code or the model level. To realize variability at the code level, SPL methods classically advocate usage of inheritance, components, frameworks, aspects or generative techniques. Svahnberg et al. present a taxonomy of variability realization techniques [Svahnberg et al.

2005]. We distinguish two large groups, annotative and compositional approaches, that can be applied either at the code level or at the model level.

Annotative Approaches. At the code level, code fragments are annotated in a common code base and removed in order to generate variants (e.g., #ifdef and #endif directives of the C preprocessor cpp to conditionally remove feature code before compilation is a typical example). At the model level, Czarnecki and Antkiewicz (e.g., [Czarnecki and Antkiewicz 2005, Czarnecki and Pietroszek 2006]), Heidenreich et al. (e.g., [Heidenreich et al. 2010]), or some model checking approaches (e.g., [Lauenroth et al. 2009, Classen et al. 2010b; 2011]) use annotations on model elements. In these cases, a global model is tailored to a specific product by activating or removing model elements from a combination of features. Similarly, Saval et al. use the term *pruning* [Saval et al. 2009] and Voelter and Groher introduce the term *negative* variability [Voelter and Groher 2007] to describe model-based annotative approach. Recently, the Common Variability Language (CVL) has been proposed to provide a generic and separate approach for modeling variability in any models defined by a metamodel by means of Meta Object Facility [Svendsen et al. 2010].

Compositional Approaches. At the code level, features are implemented separately in distinct modules (files, classes, packages, plug-ins, etc.), which can be composed in different combinations to generate variants. Voelter et al. describe variability implemented with compositional approaches as *positive* variability [Voelter and Groher 2007] since variable elements are added together. Many techniques have been proposed to realize compositional approaches (frameworks, mixin layers, aspects [Mezini and Ostermann 2004], stepwise refinement [Batory et al. 2004], etc.). In model-based SPL engineering, the idea is that multiple models or fragments, each corresponding to a feature, are composed to obtain an integrated model from a feature model configuration. Aspect-oriented modeling techniques have been applied in the context of SPL engineering [Morin, Brice and Barais, Olivier and Jézéquel, Jean-Marc and Ramos, Rodrigo 2007, Morin et al. 2008]. Apel et al. propose to revisit superimposition technique and analyze its feasibility as a model composition technique [Apel et al. 2009]. Perrouin et al. propose a flexible, tool-supported derivation process in which a product model is generated by merging UML class diagram fragments [Perrouin et al. 2008].

Language and Tool Support. In addition, several languages and tools have been developed to support the two approaches. CIDE is a tool for annotating code fragments with #ifdef and #endif directives. FeatureHouse is language-independent composition tool in which software artifacts written in various languages can be composed, for example, source code, test cases, models, documentation, makefiles [Kästner 2010]. FeatureHouse and CIDE are integrated into FeatureIDE development environment [Kästner et al. 2009a]. FeatureMapper is a tool that allows for defining mappings of features to model elements specifying feature realisations [Heidenreich et al. 2010; 2008]. XML-based Variant Configuration Language (XVCL) is a variation mechanism for managing variability in SPLs based on generative techniques [Zhang and Jarzabek 2004]. The software composition community (which encompass Aspect-Oriented Software Development (AOSD), Feature-Oriented Software Development (FOSD), Component-Based Software Development (CBSD), etc.) proposes different but complementary composition mechanisms, such as aspect weaving, mixins, feature composition, component composition to compose software artifact, at code level

or model level. The underlying tools can be adapted to support SPL development.

2.4 Contribution Choices

In this chapter, we have described the roots of SPL engineering (mass customization, reusability, domain, commonality and variability). The SPL engineering framework we present is open for different domain modeling techniques, different ways to express configuration knowledge, and different implementation techniques. We have identified variability management as one crucial challenge that should be tackled to achieve systematic reuse. We have surveyed a number of techniques to model and realize model variability.

Based on these observations, we choose to:

focus on variability modeling: we do not develop specific techniques to implement variability, for example, for automatically deriving products from the different artifacts (e.g., models). We rather consider that existing approaches (feature-oriented, generative, aspect-oriented, model-based, etc.) can be applied in the context of our contribution ;

separate variability description: there are two opposite approaches to model variability (amalgamated or separated approach). We choose a separated approach where the variability description is expressed in a dedicated model and does not alter the artifacts (e.g., models) ;

rely on feature models: different formalisms have been proposed in the literature to model variability and commonality among the products of an SPL. As we will see in the next chapter, feature models have a long history and are the *de facto* standard for separating variability description of an SPL.

Three

Feature Models

Feature modeling is a variability modeling technique, which has generated a lot of interest in SPL engineering since their introduction by [Kang et al. 1990] in the FODA method. Feature models are currently the *de-facto* standard for representing variability. We first describe the essential principles and semantic foundation of feature models (see Section 3.1). We then surveyed in which contexts and for which purposes feature models are used in SPL engineering (see Section 3.2).

3.1 Semantics of Feature Models

Figure 3.1 gives a first visual representation of a feature model. Throughout the thesis, we will rely on the same graphical notation used in this figure, largely inspired by the one proposed in [Czarnecki and Eisenecker 2000]. Features are graphically represented as a rectangles while some graphical elements (e.g., unfilled circle) are used to describe the variability (e.g., a feature may be optional). Intuitively, the feature model depicted in Figure 3.1 compactly describes a family of medical images, where each member of the family is a medical image corresponding to an unique combination of features.

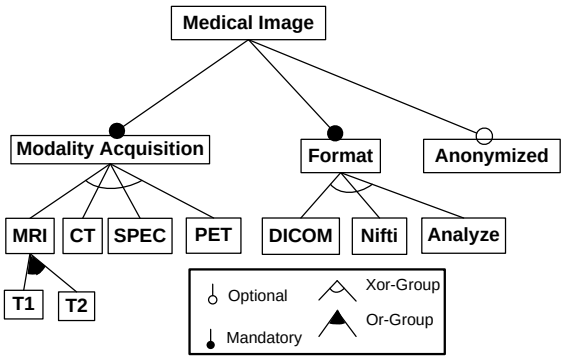


Figure 3.1: A family of medical images described with a feature model

3.1.1 The Essence of Feature Modeling

In essence, a feature model is a hierarchy of features with variability. From a general conceptual perspective, a feature model of a concept describes a set of *valid feature combinations*, each representing an instance of that concept. For example, in Figure 3.1, a feature model

describes the concept of a medical image. An instance of that concept is a medical image whose Modality Acquisition is SPEC, whose Format is DICOM and being not Anonymized. In the rest of the thesis, the font style aFeature is generally used when referring to a feature of a feature model. From an SPL perspective, we can similarly consider that the feature model depicted in Figure 3.1 describes an SPL of medical images, where each product of the SPL is a medical image.

Feature Hierarchy. The primary purpose of a *hierarchy* is to organize a potentially large number of concepts (i.e., features) into multiple levels of increasing detail. The hierarchy is usually represented as a rooted tree, the root feature being the most general concept (here: Medical Image). Edges in feature models model parent-child relations between features and aim at *aggregating* new concepts (i.e., features) or *specializing* a concept. For example, Medical Image has three child features (Modality Acquisition, Format and Anonymized) that suggest that a medical image is composed of a modality acquisition, a format and is potentially anonymized. The feature Modality Acquisition is in turn a general concept that can be specialized (e.g., MRI is a special kind of modality acquisition).

Features and Variability. Another important aspect of feature models is the modeling of *variability*. Variability defines what the allowed combinations of features (also called *configurations*) are, i.e., restricts the set of valid instances of a concept. Variability in a feature model is expressed through a number of mechanisms. When decomposing a feature into subfeatures, the subfeatures may be *optional* or *mandatory*. For example, a medical image has necessary a format and therefore the feature Format is mandatory. However a medical image may not be anonymized (the feature Anonymized is optional). Note that a feature is mandatory or optional *in regards* to its parent feature (e.g., a feature may be modeled as a mandatory feature and not be necessary included in a configuration in the case its parent is not included in the configuration). Features may also form *Or*-, or *Xor*-groups. Features MRI, CT, SPEC or PET form an Xor-group – they are mutually exclusive so that a modality acquisition cannot be MRI and CT at the same time. Features T1 and T2 form an Or-group – an MRI modality acquisition can be either T1, T2 or both T1 and T2. In addition, implies or excludes constraints (called *composition rules* in [Kang et al. 1990]) that cut across the hierarchy can be specified to express more complex dependencies between features. Though implies and excludes constraints can be depicted graphically (and thus can be part of the diagram), we systematically use a textual representation, outside the feature diagram. To be expressively complete regarding propositional logic, any constraint written in propositional logic including $\vee$ (disjunction), $\wedge$ (conjunction), $\neg$ (negation), $\Rightarrow$ (implication), $\Leftrightarrow$ (biimplication) can be used. We consider that a feature model is composed of a feature diagram (see Definition 1) plus a set of constraints expressed in propositional logic (see Definition 2).

A feature model defines a set of valid feature configurations. The validity of a configuration is determined by the semantics of feature models, for example, in Figure 1 DICOM, Nifti and Analyze are mutually exclusive and cannot be selected at the same time. A valid configuration is obtained by selecting features in a manner that respects the following rules:

- if a feature is selected, its parent must also be selected (hence an edge from one feature to another not only denotes a conceptual relationship between two features but also a logical dependency) ;

- if a parent is selected, the following features must also be selected: all the mandatory subfeatures, exactly one subfeature in each of its Xor-groups, and at least one of its subfeatures in each of its Or groups ;
- constraints relating features in different subtrees must hold ;

Feature Diagram. The terms *feature model* and *feature diagram* are employed in the literature, usually to denote the same thing. The word "diagram" suggests a graphical representation of a feature model. We rather consider that a feature model can be graphically represented by means of a feature diagram (e.g., following the rules of the FODA notation [Kang et al. 1990]), but it could be graphically represented also in a tree structure (as it is usually found in the typical left panel in CASE tools), or else in a purely textual form (reports generated by tools, XMI serialization, etc.): that is, feature models are independent of any graphical notation. As summarized in [Czarnecki et al. 2006], the essence of a feature model is its embodiment of a hierarchy and description of variability, rather than its rendering.

We also consider that a feature model does have the properties of a *model* exposed in Section 2.2.3. It is a purposeful abstraction of an SPL, it has a clear semantics from which several kinds of automated analysis and reasoning can be performed (see next section) and textual or graphical languages (see Section 3.3.2) have been developed to specify feature models. For all these reasons, we use the term *feature model*.

Formalism. Four key notions are defined below: feature diagram, feature model, feature hierarchy and configurations. Throughout this thesis, we will rely on this formalism and notation. If needs be, we will recap some important notions, typically at the beginning of each chapter, in a background section. Several formalisms and notations have been proposed in the literature. [Schobbens et al. 2007] survey the large majority of feature model variants. They show that some variants are expressively complete and thereby equivalent in terms of expressiveness. They give them a formal semantics, thanks to a generic construction. The formalism we rely on is expressively complete regarding propositional logics and can be seen as an instance of this generic construction.

Some important details are as follows. First, we consider that the hierarchy of a feature model is represented as a tree (and not as a direct acyclic graph). In particular, a feature cannot have two parents and cannot belong to more than one feature group. Second, we consider only two kinds of feature group (Xor and Or) – we do not consider Mutex-group (as in [She et al. 2011]) or group cardinality (as in [Riebisch et al. 2002, Schobbens et al. 2007]) that defines a minimum number of features and a maximum of features to choose from amongst the total number of features in the feature group. Third, a parent feature can have several feature groups (e.g., two Xor-groups and one Or-group). Fourth, implies and excludes constraints are distinguished from other constraints. Fifth, we do not distinguish primitive features from non-primitive features (as in [Schobbens et al. 2007]) or abstract features from concrete features (as in [Czarnecki and Eisenecker 1999, Thüm et al. 2009]). Finally, features are uniquely identified by their names.

Definition 1 (Feature Diagram)**.** *A feature diagram* $FD = \langle G, r, E_{MAND}, \mathcal{F}_{XOR}, \mathcal{F}_{OR}, Impl, Excl \rangle$ *is defined as follows:*

- $G = (\mathcal{F}, E)$ *is a rooted tree where* $\mathcal{F}$ *is a finite set of features and* $E \subseteq \mathcal{F} \times \mathcal{F}$ *is a finite set of edges (edges represent top-down hierarchical decomposition of features, i.e., parent-child relations between them) ;*

- $r \in \mathcal{F}$ *is the root feature ;*
- $E_{MAND} \subseteq E$ *is a set of edges that defines mandatory features with their parents ;*
- $\mathcal{F}_{XOR} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$ *and* $\mathcal{F}_{OR} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$ *define feature groups and are sets of pairs of child features together with its common parent feature. The child features are either exclusive (Xor-groups) or inclusive (Or-groups) ;*
- *features that are neither mandatory features nor involved in a feature group are optional features ;*
- *a parent feature can have several feature groups but a feature must belong to only one feature group.*
- *a set of implies constraints* $Impl$ *(resp. excludes constraints* $Excl$*), each implies constraint (resp. excludes constraint) being a propositional formula whose form is* $A \Rightarrow B$ *(resp.* $A \Rightarrow \neg B$*) where* $A \in \mathcal{F}$ *and* $B \in \mathcal{F}$*.*

Definition 2 (Feature Model)**.** *A feature model* FM *is a tuple* $\langle FD, \psi_{cst} \rangle$ *where* FD *is a feature diagram and* ψ_{cst} *is a set of cross-tree constraints where each constraint is a propositional formula over the set of features* $\mathcal{F}$ *(but neither an implies constraint nor an excludes constraint).*

Definition 3 (Feature Hierarchy)**.** *The feature hierarchy of a feature model* $FM = \langle FD, \psi_{cst} \rangle$ *corresponds to the hierarchy of its feature diagram* FD*. The hierarchy of a feature diagram* FD *is characterized by its tree* $G = (\mathcal{F}, E)$ *and its root feature* r*.*

Definition 4 (Configuration Semantics)**.** *A configuration of a feature model* FM *is defined as a set of selected features.* $[\![FM]\!]$ *denotes the set of valid configurations of the feature model* FM *and is thus a set of sets of features. A configuration* c *of* FM *is defined as a set of selected features* $c = \{f_1, f_2, \ldots, f_m\} \subseteq \mathcal{F}_{FM}$ *(e.g., see Figure 7.2(b) for the set of valid configurations characterized by the feature model of Figure 7.2(a)).*

The connection between feature model (resp. configuration) and SPL (resp. product) can now be defined.

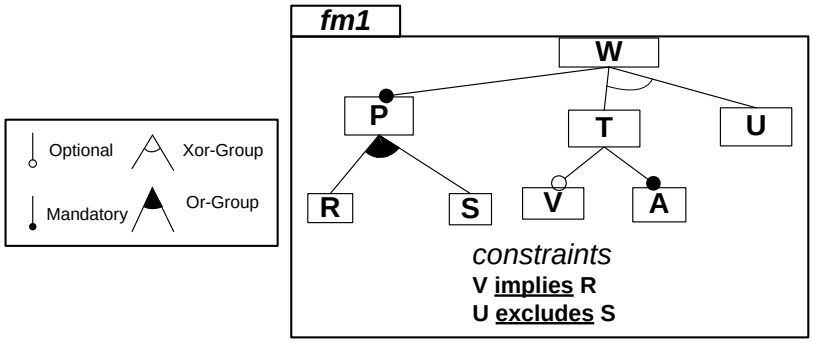
Definition 5 (SPL, Feature Model)**.** *A software product line* SPL *is a set of products described by a feature model* FM*. The set of features of* FM *is denoted* $\mathcal{F}_{FM}$*. Each product of* SPL *is a combination of features and corresponds to a valid configuration of* FM*.*

3.1.2 *Propositional* Feature Models

Figure 3.2(a) depicts a feature model, denoted fm_1. This feature model represents a set of valid configurations, enumerated in Figure 7.2(b), and denoted $[\![fm1]\!]$. The feature model is encoded as a *proposition formula* ϕ_{fm_1} (see Figure 7.2(b)). Using this formula, we can reason about the set of configurations of fm1.

Background on Propositional Logic. In this thesis, we use propositional logic with standard Boolean connectives ($\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ and negation ($\neg$) with their standard meaning. For a set of Boolean variables, a variable assignment is a function from the variables to 1, 0 (or true, false). A variable assignment is often represented as the set of variables that are assigned the value 1 (or true), the other variables are implicitly 0 (or false). A *model* of a formula ϕ is such a variable assignment under which ϕ evaluates to true (a model can be seen as a *solution* to the formula).

Example. The formula $x \vee y$ has the models $\{x\}$, $\{y\}$, and $\{x, y\}$. The assignment $\emptyset$, which assigns false to all variables, is not a model of $x \vee y$.



(a) FODA-like representation

$\phi_{fm_1} = W$ // root
$\wedge\, W \Leftrightarrow P$ // mandatory
// Or-group
$\wedge\, P \Rightarrow R \vee S$
$\wedge\, R \Rightarrow P \wedge\ S \Rightarrow P$
$\wedge\, V \Rightarrow T$ // optional
$\wedge\, A \Leftrightarrow T$ // mandatory
// Xor-group
$\wedge\, T \Rightarrow W$
$\wedge\, U \Rightarrow W$
$\wedge\, \neg T \vee \neg U$
// constraints
$\wedge\, V \Rightarrow R$ // implies
$\wedge\, \neg U \Rightarrow \neg S$ // excludes

(b) corresponding propositional formula

$[\![fm1]\!] = \{$
$\{W, P, R, S, T, A, V\},$
$\{W, P, S, T, A\},$
$\{W, P, R, T, A\},$
$\{W, P, R, U\},$
$\{W, P, R, T, V, A\},$
$\{W, P, R, S, T, A\},$
$\}$

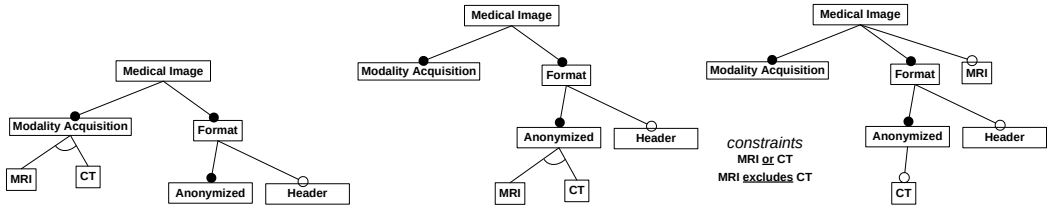
(c) corresponding set of configurations

Figure 3.2: feature model, set of configurations and propositional logic encoding

A formula is *satisfiable* if and only if there is a assignment of the variables for which the formula evaluates to true (the formula has a model); a formula is *unsatisfiable* if and only if it is not satisfiable.

Formula Encoding. In general, the following steps are performed to encode a feature model as a propositional formula [Batory 2005, Czarnecki and Wąsowski 2007, Benavides et al. 2010]: *(1)* each feature of the feature model corresponds to a variable of the propositional formula, *(2)* each relationship of the model is mapped into one or more small formulas depending on the type of relationship (Xor- and Or-groups) *(3)* the resulting formula is the conjunction of all the resulting formulas of step *(2) plus* additional propositional constraints.

Definition 6 (Feature Model)**.** *A feature model can be encoded as a propositional formula, denoted ϕ and defined over a set of Boolean variables, where each variable corresponds to a feature (see also*



(a) fm_{h1} (b) fm_{h2} (c) fm_{h3}

Figure 3.3: Three logically equivalent feature models with different hierarchies

Figure 3.2).

Configuration Semantics Is Not Enough: Hierarchy Matters. While the configuration semantics is most commonly associated with feature modeling, there exist other meanings of feature models. For example, Figure 3.3 depicts three feature models with identical configuration semantics, yet different hierarchies and meanings. In the medical imaging domain, MRI and CT can be considered as a kind of modality acquisition of a medical image. Hence, from an ontological perspective, the feature model fm_{h1} (see Figure 3.3(a)) correctly structures the features MRI and CT that are child-features of the feature Modality Acquisition. The features MRI and CT are differently organized in fm_{h2} (see Figure 3.3(b)) and fm_{h3} (see Figure 3.3(c)). In this specific example, one can consider that there are incorrectly modeled. We have shown that the configuration semantics is not sufficient to characterize the semantics of feature models. The hierarchy reflects the meaning of the features and is another important aspect of feature models [Czarnecki and Wąsowski 2007, She et al. 2011], especially for the modeler[1].

Properties of Feature Models. Feature models have important properties that can be automatically extracted by automated techniques and reported to an SPL practitioner. In particular, a feature model may represent no valid configuration (see Definition 7), typically due to the presence of cross-tree constraints – in this case, the feature model is a void feature model.

Definition 7 (Void feature model)**.** *A feature model is void (or unsatisfiable or invalid or inconsistent) if it represents no configuration.*

A feature model may have features not present in any valid configuration (see Definition 8), called dead features [Benavides et al. 2010]. For example, the feature V is dead in Figure 3.4. Note that all features (including the root) are dead in a void feature model.

Definition 8 (Dead features)**.** *A feature f of FM is dead if it cannot be part of any of the valid configurations of FM. The set of dead features of FM is noted $deads(FM) = \{f \in \mathcal{F} \mid \forall c \in [\![FM]\!], f \notin c\}$*

[1]Though the meaning of edges and feature names in feature models is important for a modeler, this kind of information is usually not explicitly represented and exploited by reasoning tools.

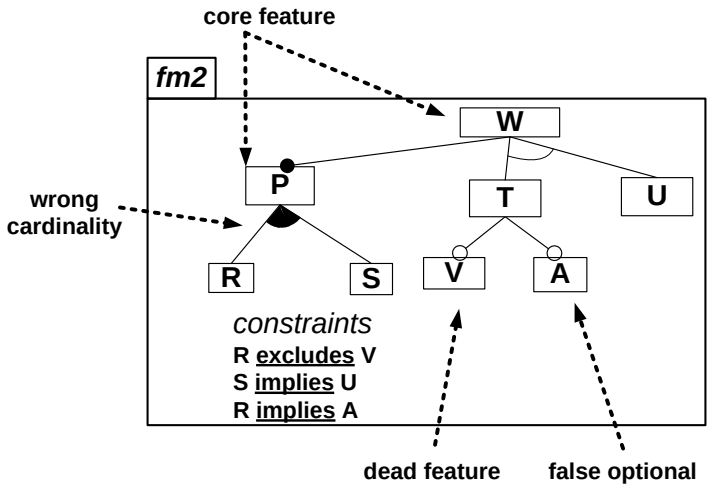


Figure 3.4: Properties of feature models: core features and anomalies

Features may also be part of all valid configurations of a feature model (it is the case of the features W and P in Figure 3.4).

Definition 9 (Core features)**.** *A feature f of FM is a core feature if it is part of all valid configurations of FM. The set of core features of FM is noted* $cores(FM) = \{f \in \mathcal{F} \mid \forall c \in [\![FM]\!], f \in c\}$

Feature models may exhibit anomalies in them (i.e., incorrect definitions of relationships that suggests that the set of products described by a feature model may not match the SPL it describes) [Trinidad et al. 2008a]. Dead features can be considered as anomalies. Considering $fm2$ (see Figure 3.4), we can notice that *i)* though features R and S form an Or-group, they are mutually exclusive (due to the presence of constraints) ; *ii)* though the feature A is an optional feature of its parent feature T, the two features are logically bi-implied (i.e., whenever the feature T is selected, the feature A must also be selected) and thereby the feature A should be modeled as a mandatory feature. Generally, these anomalies are regarded as a negative property of a feature model since it can easily decrease its maintainability or understandability. These anomalies can be automatically detected and some corrections can be suggested and applied [Trinidad et al. 2008a, Benavides et al. 2010].

Reasoning Operations. The automated analysis of feature models is about extracting information from feature models using automated mechanisms [Benavides et al. 2010]. Analysing feature models is an error-prone and tedious task, and it is infeasible to do manually with large-scale feature models. It is an active area of research and is gaining importance in both practitioners and researchers in the software product line community. Since the introduction of feature models, the literature has contributed with a number of operations of analysis, tools, paradigms and algorithms to support the analysis process.

Mannion was the first to identify the use of propositional logic techniques to reason about properties of a feature model [Mannion 2002]. In [Batory 2005], the relationship

that exists between feature model, propositional logic and grammar has been established. Batory also suggests to use logic truth maintenance system (LTMS) to infer some choices during the configuration process [Batory 2005]. Schobbens et al. have formalized some operations and their complexity [Schobbens et al. 2007]. Benavides et al. present a structured literature review of the existing proposals for the automated analysis of feature models [Benavides et al. 2010]. Example analyses proliferate and include consistency check or dead feature detections [Marcilio Mendonca 2009], interactive guidance during configuration [Tun et al. 2009, Mendonca and Cowan 2010], or fixing models and configurations [Trinidad et al. 2008a, White et al. 2008, Janota 2010].

It should be noted that most of the reasoning operations (e.g., satisfiability) are difficult computational problem and are NP-complete [Schobbens et al. 2007].

Algorithms and Automation. Various kinds of automated support have been proposed and can be classified as follows:

- propositional logic: SAT (for satisfiability) solvers or Binary Decision Diagram (BDD) take a propositional formula as input and allow one for reasoning about the formula (validity, models, etc.).
- constraint programming: a constraint satisfaction problem (CSP) consists of a set of variables, a set of finite domains for those variables and a set of constraints restricting the values of the variables. A CSP is solved by finding states (values for variables) in which all constraints are satisfied. In contrast to propositional formulas, CSP solvers can deal not only with binary values (true or false) but also with numerical values such as integers or intervals.
- description logic (DL): DLs are a family of knowledge representation languages enabling the reasoning within knowledge domains by using specific logic reasoners. A problem described in terms of description logic is usually composed by a set of concepts (i.e., classes), a set of roles (e.g., properties or relationships) and set of individuals (i.e., instances). A description logic reasoner is a software package that takes as input a problem described in DL and provides facilities for consistency and correctness checking and other reasoning operations.
- and other contributions not classified in the former groups proposing ad hoc solutions, algorithms or paradigms.

The majority of existing techniques rely on propositional logic tools to reason about propositional feature models (e.g., see [Czarnecki and Wąsowski 2007, Mendonca et al. 2008, Thüm et al. 2009, Marcilio Mendonca 2009, Janota 2010]). Benavides et al. report that CSP solvers or DL solvers may also be used, but mostly for other extensions of feature models (e.g., feature attributes) [Benavides et al. 2010]. In [Janota and Kiniry 2007], The use of a satisfiability modulo theories (SMT) solver is suggested.

Non propositional feature models. Some extensions of feature models have been proposed (e.g., feature attributes [Benavides et al. 2005], cardinality-based feature models [Czarnecki et al. 2005b]). Very early, Kang et al. use an example of attribute in [Kang et al. 1990] while Czarnecki et al. coin the term *"feature attribute"* in [Czarnecki et al. 2002]. However, as reported in [Benavides et al. 2010], the vast majority of research in feature modeling has focused on *"basic"* [Czarnecki et al. 2006, Czarnecki and Wąsowski 2007], propositional feature models. In this thesis, we focus exclusively on this kind of formalism. In the

reminder of the document, the term feature model implicitly refers to propositional feature model.

3.2 On the Use of Feature Models

In SPL engineering, feature models have been employed in a variety of contexts, at different levels of abstractions and for different purposes. It can be somewhat explained by the different notions associated to the notions of features and variability (see Section 3.2.1). In Section 3.2.2, we survey[2] relevant work, for which the use of feature models is the primary interest.

3.2.1 Feature and Variability

Feature models aim at describing the variability of an SPL in terms of features: Variability and features though are broad concepts for which a large number of interpretations exists. It has naturally an impact on how feature models are used.

What is in a Feature? Due to the diversity of software engineering research, there are several definitions of a feature [Classen et al. 2008, Apel and Kästner 2009], for example (ordered from abstract to technical):

- [Kang et al. 1990]: "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"
- [Kang et al. 1998]: "a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained"
- [Czarnecki and Eisenecker 2000]: "a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept"
- [Bosch 2000]: "a logical unit of behavior specified by a set of functional and non-functional requirements"
- [Chen et al. 2005]: "a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements"
- [Batory et al. 2003]: "a product characteristic that is used in distinguishing programs within a family of related programs"
- [Classen et al. 2008]: "a triplet, $f = (R, W, S)$, where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification"
- [Zave and Jackson 1997]: "an optional or incremental unit of functionality"
- [Batory 2005]: "an increment of program functionality"
- [Apel and Kästner 2009]: "a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option"

On Variability: Categories, Levels, Classification. Variability is a cross-cutting concern of an SPL engineering process and appears in every software development phase. During variability identification, for instance, the origin of variabilities may be identified from different perspectives. Bachmann and Bass propose in [Bachmann and Bass 2001] to classify variabilities into *categories*: Variability in function (a particular function may exist in some

[2] We review and compare these works in Chapter 4.

products and not in others); Variability in data (a particular data structure may be used in one product but not in another); Variability in control flow means (a particular interaction may occur in some products and not in others); Variability in technology (OS, hardware, user interface, etc. may vary from one product to another) ; Variability in product quality goals means that goals (qualitative dimensions like performance or security may be unique to one product), etc. [Svahnberg et al. 2005] have defined five *levels* of variability (SPL, product, component, sub-component and code level) where different variability design issues appear . Another *classification*, based on the distinction between essential and technical variability, has been proposed in [Halmans and Pohl 2003]. Essential variability refers to the customer's point of view (and is also called external variability in [Pohl et al. 2005] or product line variability in [Metzger et al. 2007]). Technical variability (also called internal variability in [Pohl et al. 2005] and software variability in [Metzger et al. 2007]) refers to the SPL engineers point of view who is primarily interested in implementation details (i.e., variation points, variants and binding times to implement variability).

Impact on Feature Modeling. The plethora of feature and variability definitions suggests that feature models can be used at different stages of the SPL development, from high-level requirements to code implementation. From an early stage (e.g., requirements elicitation) to components and platform modeling, feature models can be applied to any kind of artefacts (code, documentation, models) and at any level of abstraction. Feature models can play a central role in managing variability and product derivation of SPLs (e.g., see [Czarnecki and Antkiewicz 2005, Ziadi and Jézéquel 2006, Sanchez et al. 2008, Voelter and Groher 2007, Heidenreich et al. 2010]). In this thesis, feature models are considered from a general perspective in that feature models are not restricted to a specific development phase. As a result, a feature model can just as well describe a family of software programs, a family of requirements or a family of models.

3.2.2 Feature Models in a Multiple World

In [Kang et al. 1998], the authors propose several categories of features, organised into a hierarchy. Feature models in the capabilities layer (e.g., functionality of the end user), in the operating environments layer (e.g., the attributes of the environment), in the domain technologies layer and in the implementation technologies layer are combined through composed-of, generalisation/specialisation and implemented-by relationships, modeled as propositional constraints. Metzger et al. distinguish software variability (hidden to customers and internal to an SPL) from product line (PL) variability (visible to customers) [Metzger et al. 2007]. They separate the two kinds of variability description in two distinct variability models: a feature model and an orthogonal variability model (OVM). They describe a bridge between the formalism of feature model and the formalism of OVM. Though they slightly differ, their semantics remain the same: documenting the valid combination of features (or variation points) in an SPL. They use automated reasoning techniques to determine interesting properties between the software variability and the PL variability. For instance, a crucial check mentioned is the property of *realizability*: for each configuration in the software part should correspond a product in the product line offered to customers. In [Tun et al. 2009], several feature models are used to separate feature descriptions related to requirements, problem world context and software specifications. They also consider the modeling of qualitative attributes in these feature models.

Van der Storm considered not only variability at the level of one software product, but also each variable component as an entry-point for a certain software product (obtained through component composition) [van der Storm 2004]. Hartmann et al. dealt with multiple SPLs and identified several compositional issues in the context of software supply chains [Hartmann et al. 2009]. They aim to integrating several highly-variable components provided by different suppliers. Reiser and Weber proposed to use multi-level feature trees consisting of a tree of feature models in which the parent model serves as a reference feature model for its children [Reiser and Weber 2007]. Their purpose is mostly to cope with large diagrams and large-scale organizations.

There have been several attempts to relating features and contextual factors affecting feature selection. Hartmann and Trew proposed a context variability model which constrains a feature model by describing how contextual variations (e.g., different geographic regions) affect selection of feature variations [Hartmann and Trew 2008]. Tun et al. used contextual variations as links between requirement variations and feature variations [Tun et al. 2009]. In summary, both approaches model contextual variations that affect feature selection as a separate contextual feature model and relate the contextual feature model to the application feature model through feature dependencies (e.g., requires and excludes). Lee and Kang argued that *"Although feature dependencies between the contextual feature model and the application feature model affect selection of features in the application feature model, they are not the only one that affects feature selection. Rather than one contextual feature affecting selection of application features, a group of contextual features determines product goals and attributes, which in turn affects selection of application features"* [Lee and Kang 2010].

Segura et al., Alves et al. use feature models to evolve an SPL (e.g., refactoring of an SPL). They provide catalog rules to evolve feature models and describe some properties of the catalog [Segura et al. 2008, Alves et al. 2006]. Schwanninger et al. report an industrial experience in which several feature models are used throughout the development process [Schwanninger et al. 2009]. Zaid et al. propose an approach for modularizing feature models so that they can be reused in another context [Zaid et al. 2010; 2011].

Czarnecki et al. propose a *staged configuration* approach where feature models are stepwise specialized and instantiated according to the stakeholder interests at each development stage [Czarnecki et al. 2005b]. Specialization and configuration are distinguished: specialization is defined as the process in which variabilities in feature models are removed (i.e., a specialized feature model has fewer variabilities than its parent feature model, a fully specialized feature model has no variability while a configuration is an instantiation of a feature model). The concept of multi-level staged configuration is also introduced, referring to a sequential process in which a feature model is configured and specialized by stakeholders in the development stages. Hubaux et al. propose a formal approach to multi-stage configuration [Hubaux et al. 2009]. They notably propose the formalism of feature configuration workflow in order to configure a large feature models in different steps, possibly by different stakeholders.

Dhungana et al. discuss the challenges of structuring the modelling space for software product lines (solution structure, multiple SPLs, asset types, organisational structure, and market needs) [Dhungana et al. 2010]. They argue that maintaining a single feature model for the entire system is not feasible and proceed to suggest strategies for feature modelling from various perspectives. They also present some examples of how these strategies can be applied, supported by existing tools.

3.3 Tool and Language Support

3.3.1 Feature Modeling Tools

Tools supporting the creation of feature models have emerged since 2004. Most of the tools that support the creation of feature models usually also provide support to the configuration process. Feature Modeling Plug-In (FMP) [Czarnecki et al. 2004] has been the starting point of such tool. FMP provides tree views for creating feature models, configurations, and partial configurations in the process of staged configuration. The Software Product Lines Online Tools (SPLOT) is a web-based tool providing a feature model editor as a tree view, a configuration editor with decision propagation, automated analysis on feature models, and example feature models [Mendonca et al. 2009a;b]. FAMA is a Java framework that focuses on the comparison of different solvers for the automated analysis of feature models [FaMa 2008, Trinidad et al. 2008b]. Two commercial tools integrate feature modeling and the configuration process pure::variants [pure::variants 2006, Beuche 2008] and Gears [BigLever – Gears 2006] and provide extensions to multiple IDEs such as Eclipse and Netbeans. FeatureIDE is an Eclipse-based framework to support feature-oriented software development [Kästner et al. 2009a;b]. The main focus of FeatureIDE is to cover the whole development process and to integrate tools for the implementation of SPLs in an integrated development environment. The goal is to ease the development of tool support for new languages and concepts. FeatureIDE provides advanced support for feature modeling, including graphical editors to specify a feature model or configure a feature model and specific graphical views to reason about feature model edits.

3.3.2 Feature Modeling Notation

In the literature, graphical feature model notations based on FODA [Kang et al. 1990] are by far the most widely used. FeatuRSEB [Griss et al. 1998], FORM [Kang et al. 1998] or the graphical notation proposed in [Czarnecki and Eisenecker 2000] are only slightly different from the FODA notation (e.g., boxes are added around feature names or filled circles are used to represent mandatory features).

In addition, a number of *textual* feature model languages were also proposed in the literature. The first textual language was FDL [Deursen and Klint 2002]. Batory proposed the GUIDSL syntax, in which the feature model is represented by a grammar [Batory 2005]. The GUIDSL syntax is used as a file format of the feature-oriented programming tools AHEAD [Batory et al. 2004] and FeatureIDE [Kästner et al. 2009a;b]. TVL is a text-based (as opposed to diagrammatic) language for feature models with a C-like syntax [Boucher et al. 2010, Classen et al. 2010a]. The goal of the language is to be scalable, by being concise and by offering mechanisms for modularity, and to be comprehensive so as to cover most of the feature model dialects proposed in the literature. TVL also supports enumerates and feature attributes.

The language Clafer integrates feature modeling into class modeling and thereby can also be used to specify feature models [Bąk et al. 2011].

3.4 Summary and Contribution Choices

Feature models are a fundamental formalism in SPL engineering. These models hierarchically structure features and compactly define the set of legal combinations of features, where each combination corresponds to a product in an SPL. Our work focuses on an

approach that puts feature models at the center of SPL engineering. In this chapter, we have introduced the essence and the semantics of feature models. Though extended formalisms have been proposed (e.g., feature attributes), we chose to focus on propositional (also called basic) feature models. We have highlighted that a number of (automated) techniques have been developed to facilitate their analysis and manipulation. We have also surveyed several works centered on the use of feature models. They are now used in a wider scope than originally planned in 1990 and employed in a variety of contexts, at different levels of abstractions and for different purposes. The primary focus of the thesis is to investigate further in this research direction.

Four

Multiple Feature Models: Example and Requirements

This chapter shares material with the MICCAI-Grid paper "Issues in Managing Variability of Medical Imaging Grid Services" [Acher et al. 2008a] and the SOAPL'08 paper "Imaging Services on the Grid as a Product Line: Requirements and Architecture" [Acher et al. 2008b].

In this chapter, we describe an example in which several feature models, possibly interrelated, are intensively used to model the variability of a software system. The example is about variability management in scientific workflows and is representative of the kinds of problems that may occur when multiple variability sources have to be managed. It is an excerpt of a larger case study in the medical imaging domain where the overall objective is to develop model-based software product line techniques that facilitate the reuse of legacy services during the design of scientific workflows (see Section 4.1 for the example description). From this example, we identify different issues that have not been properly addressed by existing related work (see Section 4.2). Finally, we give an overview of the approach and the contributions of the thesis (see Section 4.3). The example and underlying issues will be revisited throughout the thesis document to illustrate and evaluate the contributions of the thesis (mainly in Chapter 6 and Chapter 10).

4.1 Variability in Scientific Workflows

Scientific workflows are increasingly used for the integration of *legacy* tools and algorithms to build large and complex applications such as scientific data analysis pipelines or computational science experiments. Despite the growing interest observed in scientific workflow technologies in recent years, workflow design remains a difficult, tedious, and often error-prone process which slows down the adoption of scientific workflows [Gil et al. 2007, McPhillips et al. 2009]. In particular, although catalogs of domain-specific data processing services are common, the low-level interface representations used (e.g., Web Services) usually only provide syntactical information suited to check the technical consistency of individual services. There is absolutely no guarantee regarding the consistency of the whole process *composed*, nor its validity from an applicative point of view. The use of compositional software product lines techniques to gain potential advantages in terms of flexibility and reuse (and thus time, effort and cost) may alleviate these problems.

Our work is illustrated through the medical imaging area, which is typical of the usage of scientific workflows executed over compute-intensive distributed infrastructures such

as grids. In this domain, grids help in building patient-specific models and in reducing computation time for meeting time constraints from clinical practice. The rest of this section introduces our example and identifies its sources of variability to determine associated requirements.

4.1.1 Medical Imaging Workflows

In the medical image analysis area, distributed computing capabilities are used for many purposes, ranging from validation and optimization processes of specific algorithms to overall reduction of computing time. Besides, image analysis pipelines are scientific, data-driven workflows which are undergoing homogenization nowadays, strongly motivated by the need for mutualizing software development and easily comparing results. This homogenization is conducted through the usage of common data formats and means to reuse algorithms.

In order to facilitate it, Service-Oriented Architectures (SOAs) [Foster et al. 2002] are increasingly used and aim at *i)* producing reusable self-contained, distributed imaging services, decoupled and abstracted from technical grid platforms ; *ii)* providing standardized interfaces for invoking wrapped application codes as well as information on exchange protocols and *iii)* composing these atomic services to describe processing pipelines *as* complex workflows. Using SOAs, medical experts essentially compose different kinds of processing on images, each algorithm being provided by a service.

Medical imaging workflow example. We use as an illustration an existing service-oriented workflow designed to conduct experiments on Alzheimer's disease [Lorenzi et al. 2010]. This disease is a neurodegenerative pathology which can be characterized by an atrophy of the brain. The workflow illustrated in Figure 4.1 is based on several image processing activities aimed at tracking the evolution of Alzheimer disease through a longitudinal study. The disease follow up consists in comparing several MRIs from the same patient acquired over time, to detect changes in the volume of the brain and compute a brain tissue atrophy coefficient. In order to be compared in the last steps of the workflow, source MRIs first need to be homogenized both in terms of intensity biases and space alignment. It must be noted that, as in many similar scientific workflows, the complexity lies in the correct pre-processing of data, which is generally frustrating for the scientist end-users.

In Figure 4.1, the blue boxes at the top represent the input images: the *Image sequence* box represents MRIs acquired at a given time (T_0+6 months and T_0+12 months), the *Reference image* represents the MRI acquisition at T_0, considered as the patient's reference. This configuration of the workflow will lead to two invocations, giving two estimations of brain atrophy, at time T_0+6 and T_0+12. The first image processing activity, *Bias correction* is a general restoration procedure which involves removing voxel inhomogeneities in the magnetic field of the MRI equipment, used to improve the result of image post-processing algorithms. Then an *Affine registration* process is performed in which a spatial alignment is estimated so that the MRI considered is translated, reoriented and scaled to be superimposed on the patient's reference MRI. The next step *Longitudinal intensity correction* is another intensity homogenization procedure that normalizes intensities between the different images acquired over time. At the same time, the right branch of the workflow aims at identifying brain tissues (grey and white matter) through the *Brain extraction* and *Tissue segmentation* activities to finally create a mask (*Mask calculation*) delineating grey and white matter of the patient's brain at T_0. Finally, the comparison of MRIs starts by estimating the deformation field, which corresponds to the *Non-linear registration* activity, between the

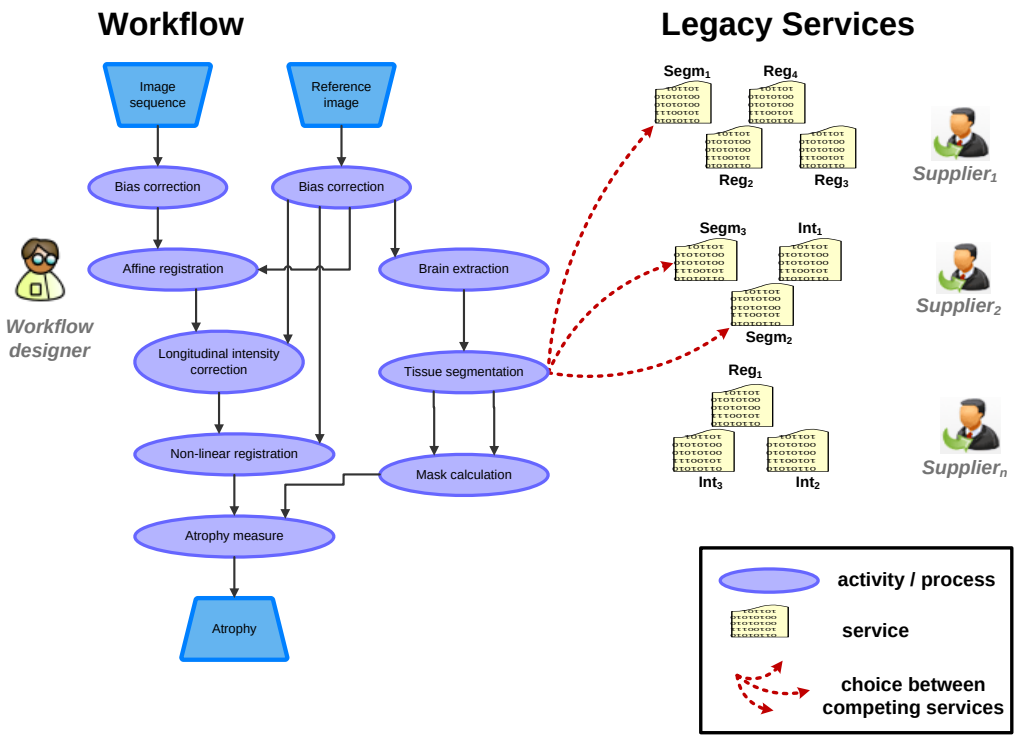


Figure 4.1: From workflow design to selection of services.

reference and the "moving" MRIs. The last step consists in applying the deformation field to the mask of the brain tissues in order to estimate the changes in the volume of brain tissues, and estimate, as the final result of the workflow, a potential atrophy.

4.1.2 Sources of Variability

We analyze here the different sources of variability that are present in the workflow of Figure 4.1.

The pre-processing involved in this workflow are based on three typical categories of image processing activities: *Restoration*, *Registration* and *Segmentation*. By category, we mean a class of activities that are grouped as perceptually similar although they may be discriminable from one another. For instance, the two activities *Bias correction* and *Longitudinal intensity correction* realize the same coarse-grained activity, *Restoration*, but their fine-grained functionality varies because removing magnetic inhomogeneities is different than normalizing intensities from two different MR images. Another example comes from the *Registration* activities in which the same kind of *functional variability* can be observed as there are significant differences between linear and non-rigid transformations. More generally, several *Restoration* services (algorithms) have been developed and are as many candidates to realize a given functionality. In addition, service providers offer different implementations of an algorithm and selection of services are based on the different values of parameters. Hence services can be customized to meet individual needs, for example,

a workflow designer has different kinds of preferences for a service or need a specific behaviour from services.

At workflow runtime, end-users must cope with non functional concerns such as constraints related to the computing infrastructure or restricted access control. For example, to operationalize the *Brain extraction* activity, one may choose the BET tool from the FSL toolbox because it is fast at removing non-brain tissues and can be relocated to the end-user desktop. But if the accuracy of the segmentation is preferred to its computation time, another processing tool might be chosen with different constraints on the infrastructure. Using BET also introduces a deployment constraint as it depends on the full installation of another toolbox (FSL) and needs appropriate environment configuration.

More generally, for each activity of the workflow, numerous existing services are available on the grid and vary from different perspectives: the support of image formats (DICOM, Nifti, Analyze, etc.) and modality acquisition (MR, CT, PET, etc.), the support of network protocols, the algorithm method used to process an image, anatomical structures for which services are supposed to efficiently perform (Brain, Kidney, Breast, etc.), quality of service (QoS) provided in different contexts, etc. It must be noted that not only imaging but nearly all scientific services have a large number of input ports, parameters, data specificities, and dependencies at all levels (functional, non-functional and deployment related). The overall issue for scientific workflow users is thus to deal with services and their dependencies in their workflows while addressing a large amount of concerns. In our medical imaging illustration, from the workflow design time to its run time, both domain-specific and technical knowledge are needed to resolve different forms of variability. This is typically accomplished by manually setting, among others, the choice of tools and the choice in their configuration. This type of manual variability management requires a considerable amount of time and effort, and is generally a tedious and error-prone task.

4.1.3 Challenges in Scientific Workflows Design

There are several issues when designing scientific workflows. Following an SPL approach, our work addresses the following specific challenges. The first challenge is to capture commonalities and variabilities across a family of services in reusable parameterized services, for example, identifying and organizing similar and recurrent imaging tasks such as registrations and corrections. The second challenge is related to providing support for tailoring and composing services to realize consistent workflows.

There are two categories of users for the NeuroLOG platform: (i) image analysis specialist create and deploy image analysis tools that are of interest for neuroscientists; and (ii) neuroscientists design data analysis experiments by composing such tools within specialized workflows. Rather than providing services and hoping that opportunities for reuse will arise during the design of a workflow, a proactive strategy is to plan which characteristics or features of a service are likely to be systematically reused. The ability to efficiently create many variations of a service and capitalize on its commonalities can improve its composability and increase the extent to which service logic is sufficiently generic so that it can be effectively reused. Currently, the difficulty of provisioning and composing parameterized services stems from the lack of mechanisms for managing variabilities *within and across* services.

The goal of the SPL approach promoted in this thesis is to manage not only the variability of the services but also the variability of the resulting composed services. For structuring and managing variability information across a large amount of services, we identified

the following requirements, emerging from the needs of both image analysts and neuroscientists:

- Mechanisms that enable service providers (image analysts) to capture the commonalities and variabilities in parameterized (imaging) services ;
- Assistance to the neuroscientists in selecting the appropriate service from among sets of existing services: They may want to search services matching several criteria to determine whether at least one service can fulfill a specific feature or combination of features ;
- Ensuring that services are consistently composed in the resulting workflow. For example, connected services should inter-operate while exchanging medical images and support compatible formats. A sound formal basis together with tools are needed to support rigorous reasoning on a large number of services for ensuring that these properties are preserved ;
- Evolving services as the variability of services can evolve during time. Similarly, new services from new suppliers and scientists can be proposed. Consequently, there is a need to consistently maintain the set of existing services and favor the integration of new services.

4.2 MANAGING MULTIPLE FEATURE MODELS

We rely on feature models to describe the variability of a service. Several kinds of artifacts (e.g., XML configurations files, source code, tabular data) may provide information about the variability of services. As a starting point of our approach, we *assume* that the variability of services has been explicitly documented in one or more than one feature model, either manually or automatically.

4.2.1 Why Multiple Feature Models?

Multiple concerns. Medical imaging services exhibit multiple sources of variation. The variability may refer to the functionality (e.g., particular function may only exist in some services or can be highly parameterized), the grid deployment technology (e.g., operating system, hardware, libraries required, dependency on middleware), the specificities of data (e.g., medical image format), to non functional property (like security, performance or adaptability), etc. In other words, variability affects different concerns (medical image support, algorithm method, grid deployment, network protocol, etc.) of a medical imaging service. As multiple sources of variation are present within a service, several feature models are used where each feature model focuses on a specific concern of a service.

Multiple services. The management of variability within individual services that constitue the scientific workflow is a first step but is not sufficient. In the same way multiple services have been developed and can be potentially reused in different workflows, there exists multiple feature models that can be exploited by a workflow designer.

Multiple suppliers. In SOA, services are published by a service provider (we call it *supplier*) and registered into a service directory. Several suppliers (e.g., from different companies or research labs) usually exist and offer different ready to use solutions to a workflow

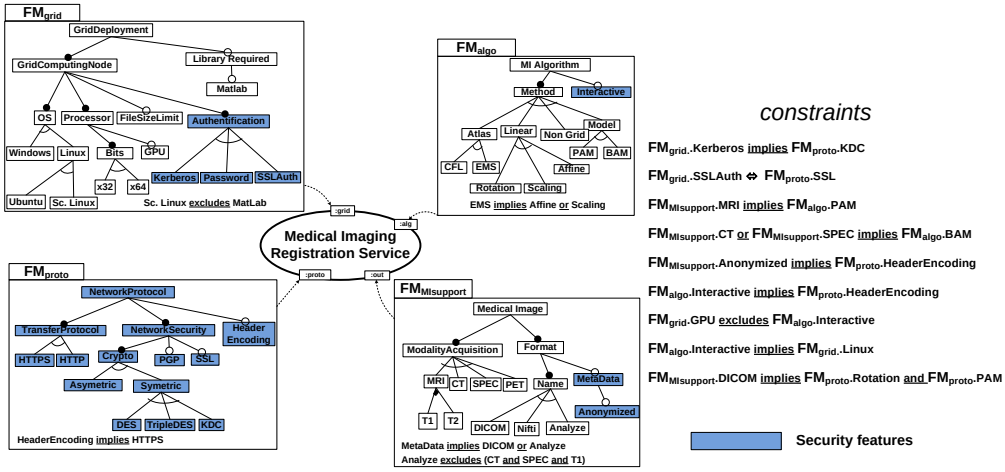


Figure 4.2: Medical Imaging Service: Variability and Concerns

designer. As suppliers document the variability of their own services, multiple feature models exist.

We coin the term *multiple feature models* to characterize a set of feature models, possibly inter-related, that are combined together to model the variability of a system. The use of multiple feature models can be explained by two phenomena. Firstly, the feature models may be originally separated: It is the case when you describe the variability of services that are *by nature* modular entities. It is also the case when independent suppliers describe the variability of their different products. Secondly, it can be the intention of an SPL practitioner to modularize the variability description of the system into different criteria (or concerns). It is the case when describing the variability of service's concerns. Here, an original, typically large feature model is decomposed into smaller and separated feature models.

With feature models being related in a variety of ways and handled by several suppliers, managing them with a large number of features is intuitively a problem of *Separation of Concerns (SoC)*. The sought benefits are indeed similar to the ones of the software engineering discipline (e.g., reduced complexity, improved reusability and simpler evolution). On the one hand, one needs to compose feature models that have originally or intentionally been separated. On the other hand, one needs to separate concerns and decompose (large) feature models. In both cases, the principle of SoC can be an effective way to manage the size and complexity of multiple feature models. As argued by [Tarr et al. 1999] in the abstract of their influential paper "N Degrees of Separation: Multi-Dimensional Separation of Concerns" (ICSE'99)

> *"Done well, separation of concerns (SoC) can provide many software engineering benefits, including reduced complexity, improved reusability, and simpler evolution. The choice of boundaries for separate concerns depends on both requirements on the system and on the kind(s) of decomposition and composition a given formalism supports."*

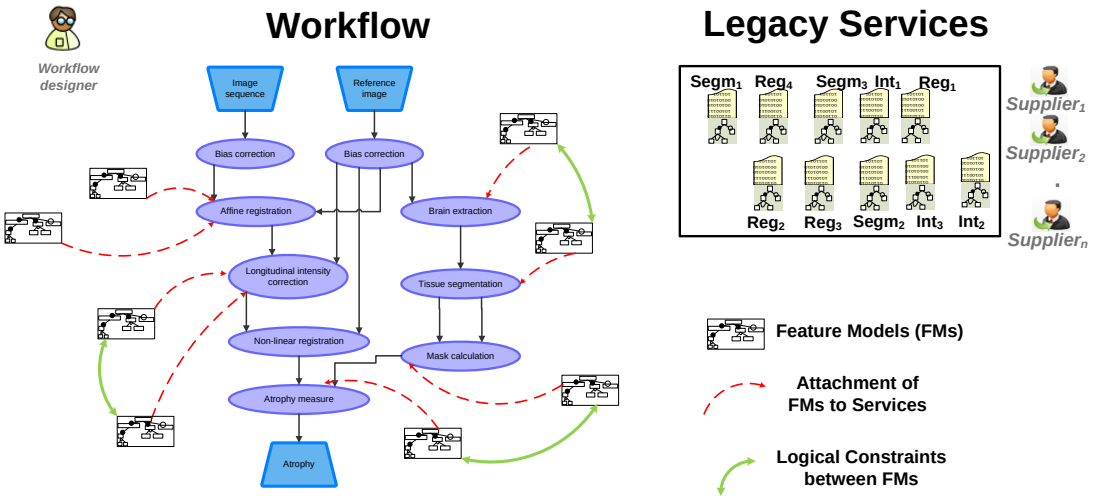


Figure 4.3: Managing Multiple Feature Models: An Example

4.2.2 Requirements

In this specific example, we have shown that multiple concerns, services and suppliers with their variability descriptions have to be managed and therefore why multiple feature models are needed. We think the example is representative of software systems that need to use multiple feature models when dealing with their variabilities. Other examples in the literature corroborate this increasing need (see Section 3.2.2 or Table 4.1 that provides an inventory of possible approaches).

More specifically, we have identified some important issues when dealing with multiple feature models:

consistency checking of multiple feature models: In the example (see green arrows in Figure 4.3), we have to deal with the variability within and across services. Firstly, within services, there are complex relationships between services' concerns (e.g., an algorithm method may require the use of a specific medical image format). Secondly, across services, interactions between services (e.g., a feature of one service may exclude another feature of another service) have to be managed when services are combined to form workflows.

grouping feature models: For each category of activity to be performed in the workflow (e.g., segmentation, registration), there are several candidate services provided by different suppliers (see right part of Figure 4.3). It is time-consuming and labour intensive for a workflow designer to explore the list of services and their features. Grouping similar services (e.g., segmentation services) helps in finding the relevant service and in maintaing the service directory ;

updating feature models: When concerns are inter-related within a service by constraints (see green arrows in Figure 4.3 or constraints in Figure 4.2), some features of some concerns may become dead or mandatory. Hence for each concern of service we need to update the variability information so that each feature model is a correct representation of the set of configurations ;

reasoning locally about some specific parts of the feature models: In the example, when a service processes a medical image and transmits it to another service, the workflow designer should ensure that the involved services are compatible (e.g., see *Tissue Segmentation* and *Mask Calculation* in Figure 4.3). We thus need to reason about some specific parts of the two services (e.g., on the features related to medical image properties).

multiple perspectives support: In the example, the deployment of a medical imaging service is subject to various expertises (grid computing, medical imaging, etc.). Ideally, the different experts should focus on a specific dimension (e.g., security) and the details that are out of the scope of their expertise should be hidden. Dedicated decomposition facilities should be applied to feature models (see Figure 4.2) ;

multi-stage and multi-step process: the design of a scientific workflow is usually not realized in one step or by an unique stakeholder. For example, the specific requirements of an application are obtained by configuring the feature model, i.e., by gradually removing the variability until only those features that are part of the final product remain. At each step of the process, we should be able to reiterate the reasoning tasks previously mentioned (consistency checking of FMs, update of FMs, etc).

4.2.3 Analysis of the State-of-the-art

We survey here some existing and relevant approaches for which the use of multiple feature models has been identified (see Section 3.2.2 for other works). For each approach, we focus on:

- the general motivation of the work and why there is not only one feature model used but rather several feature models ;
- the separation of concerns mechanisms developed, if any, that include composition and decomposition mechanisms ;
- the language and tool support, if any.

Table 4.1 summarizes the description of approaches. This survey shows that the use of several feature models has a growing interest.

Different forms of variability (PL, internal, external, software, contextual variability, layers) are usually considered in the approaches. There is a tendency to expand the scope of feature models: in addition to describing variability in the design, a need for describing the variability in the wider system context has been recognised by feature modeling approaches. Tun and Heymans made a similar observation by surveying how some approaches deal with concerns and their separation in feature model languages [Tun and Heymans 2009].

The need to model several concerns can explain, in part, the phenomenon of increasing size and complexity of feature models. Until now, researchers assume that very large feature models exist – but those feature models are not publicly available. For example, Trujillo et al. claim that the automotive industry has feature models with up to ten thousand features [Trujillo et al. 2006]; Kästner reports some personal communications with industrial partners that mention the existence of large feature models (from 500 features to several thousands of features) [Kästner 2010]; Steger et al. report that Bosch's product line of engine control software has over 1000 features [Steger et al. 2004]; Refstrup that Owen product line has about 2000 features [Refstrup 2009]. Recently, automated extraction of feature models from large implemented software systems have produced feature models with thousands of features [Berger et al. 2010, She et al. 2011].

Previous work has pointed out that dealing with large, monolithic feature models is problematic. In particular, feature model maintenance [Hartmann and Trew 2008, Reiser and Weber 2007] or evolution [Alves et al. 2006, Thüm et al. 2009, Segura et al. 2008, Botterweck et al. 2010] is a difficult process.

As in our running example, an appealing approach is rather to *decompose* large feature models and use multiple feature models during the SPL development. It should allow different stakeholders or software suppliers, at different stages of the software development, to focus on their expertise and integrate their specific concerns [Czarnecki et al. 2005b, Reiser and Weber 2007, Hubaux et al. 2009; 2010b].

In most of the approaches, the separation of feature models, if any, is rather conceptual and there is no clear support. Some operations are suggested but not realized so that only a few work propose automated reasoning support. In case languages and tool support are mentioned, the solutions are rather ad-hoc. For example, feature models are separated from a conceptual point of view but in practice they are manually combined to form a global feature model [Kang et al. 1998, Hartmann and Trew 2008, Hartmann et al. 2009].

Some works mention the need to integrate feature models that come from different suppliers [Hartmann et al. 2009, Bosch 2009] and are concerned with stakeholders and organisational structures [Reiser and Weber 2007]. A few works suggest some composition mechanisms, but do not give a proper semantics or leave it as future work [Czarnecki et al. 2005b, Hartmann and Trew 2008]. In particular, there is inadequate support for *merging* feature models.

Although the feature modeling tool support has a rich history, it is somewhat fragmented. There is no integrated solution that supports both composition and decomposition mechanisms. To the best of our knowledge, existing tools have focused on supporting activities for only one feature model and have not been explicitly developed to support the management of several feature models.

A study of the literature about automated reasoning about feature models demonstrates that providing automated support for composing and decomposing feature models still remains an open challenge [Alves et al. 2006, Segura et al. 2008, Benavides et al. 2010].

Another observation is that existing approaches either focus on composition or decomposition, but do not try to combine the two operations.

4.3 Requirements Summary and Overview of the Contributions

In this chapter, we have shown that feature models are now multiple in SPL engineering. An appropriate support for managing multiple feature models is more and more needed: An example from the medical imaging domain and the analysis of the state-of-the-art support our claims.

4.3.1 Requirements Summary

Central to the requirements described in Section 4.2.2 is *i)* a comprehensive support for separation of concerns and *ii)* automated reasoning techniques.

Separation of Concerns (SoC). By SoC, we mean composition and decomposition mechanisms dedicated to the formalism of feature models.

Composing feature models is important to ensure the consistency of inter-related feature models, to group and evolve a set of similar services or to update inter-related feature models. *Decomposing* feature models is important to reason about local properties of a feature

Work	Motivation	Composition	Decomposition	Support
[Kang et al. 1998]	layers	FMs are conceptually separated and interrelated by constraints	No	not mentioned
[Czarnecki et al. 2005b]	Multi-staged configuration, Suppliers	merge and join (informally described)	fork (informally described)	No
[Pohl et al. 2005]	external variability vs internal variability	no clear mechanism	No	A tool for OVM diagramming
[Metzger et al. 2007]	PL variability vs software variability	an algorithm is described to construct a global FM	No	SAT-based reasoning techniques
[Tun et al. 2009]	Variability in requirements, problem world and specification	similarly as in [Metzger et al. 2007]	No	pure::variants
[Hartmann and Trew 2008]	Contextual and product variability	an FM, combination of the two FMs, is manually build	No	pure::variants
[Hartmann et al. 2009]	Suppliers	algorithm to inter-relate FMs	No	pure::variants
[Reiser and Weber 2006; 2007]	Suppliers	Multi-level feature trees	No	Eclipse-based prototype tool
[Hubaux et al. 2009]	feature-based configuration	inconsistency management	No	YAWL, SPLOT
[Hubaux et al. 2010b;a]	Multiple perspectives, feature-based configuration	No	Views extraction	XPath, SPLOT
[Weston et al. 2009]	Elaboration of FMs from natural language requirements	incremental adding of features	feature clustering	ARBORCRAFT
[Lee and Kang 2010]	Contextual variability	FMs are related by constraints	No	No
[Grünbacher et al. 2009, Dhungana et al. 2010]	solution structure, multiple SPLs, asset types		Variation points detection, model fragments creation	DOPLER tool suite
[Zaid et al. 2010; 2011]	Multiple perspectives, reusability of FMs	Extension of FM formalism	No	No

Table 4.1: Multiple Feature Models (FMs) in the state-of-the-art: motivation, composition and decomposition mechanisms, tool and language support

model or to support multiple perspectives. The two mechanisms are equally important and complementary. For example, for updating the different feature models within a service (see Figure 4.3), we need to compose the feature models (to reason globally about the variability) and decompose the feature model (to retrieve the initial decomposition into different concerns).

Automated reasoning techniques. For all the management activities identified in this chapter, a manual intervention of the workflow designer or different experts is both error-prone, labour-intensive and time-consuming. In practice, there can be hundreds of features whose legal combinations are governed by many and often complex rules. This complexity has already been observed on a scale of one feature model. It becomes even more complex on a scale of multiple feature models. It is thus of crucial importance to be able to *automate* the decision making process as much as possible. Another important aspect is the ability of techniques to *reason* about feature models, for example, to infer choices when configuring some part of a workflow.

4.3.2 Overview of the Contributions

The analysis of the state-of-the-art reveals that there is *i)* no comprehensive solution for composing and decomposing feature models and *ii)* an inadequate support for assisting SPL practitioners when managing multiple feature models.

The contributions of the thesis aim at providing solutions to these problems. It can be summarized in three main points:

- a set of composition and decomposition *operators* to support SoC in feature modeling (see Part II). The operators are formally defined, fully automated, guaranteeing properties in terms of sets of configurations and can be combined together or with other operators, for example, to realize complex reasoning tasks ;
- a domain-specific, textual *language*, FAMILIAR, that provides an operational solution for using the different operators and managing multiple feature models on a large scale (see Part III);
- various *applications* of the operators and FAMILIAR (see Part IV) in different domains (medical imaging, video surveillance) and for different purposes (scientific workflow design, variability modeling from requirements to runtime, reverse engineering). In particular, we revisit the example described in the chapter.

Part II

Applying Separation of Concerns to Feature Modeling

To suport separation of concerns in feature modeling, we describe the design and the semantics of operators to compose and to decompose feature models. We show how the operators can be used to manage and reason about multiple feature models.

In Chapter 5, we present the different composition mechanisms (insert, aggregate, merge).

In Chapter 6, we focus on the merging operators that produce compact feature models from a set of existing feature models and thus ease their management and analysis.

In Chapter 7, we present a slicing technique to decompose feature models that, combined with other composition/reasoning operators, brings new capabilities to SPL practitioners (update and extraction of FM views, reconciliation of FMs and reasoning about different kinds of variability, etc.).

Five

Composing Feature Models

This chapter shares material with the SLE'09 paper "Composing Feature Models" [Acher et al. 2009b] and the ECMFA'10 paper "Comparing Approaches to Implement Feature Model Composition" [Acher et al. 2010a].

Composition has a long tradition in software engineering, be it for merging separate branches of independent development, for combining simple objects or data types into more complex ones, for reconciling the elements of a modeling or language concept or for assembling the fragments of larger component systems. Though a multitude of approaches have been proposed for composing artifacts of various natures (textual documents, code, components, models, aspects, etc.), we should consider the specificities of the formalism of feature models.

We first ask *what does composition mean in the context of feature models*. We identify three important forms of composition (Section 5.1). We design and define the semantics of three composition operators: insert (Section 5.2), aggregate (Section 5.3) and merge (Section 5.4).

In Section 5.5, we study *how composition can be realized in the context of feature models*. In particular, we compare different approaches to implement the merge operator. The study provides some evidence that using generic model composition frameworks are not helping much in the realization, whereas a specific solution, based on propositional logic, is finally necessary and clearly stands out by its qualities.

5.1 Different forms of composition

In their seminal paper, Kang et al. already proposed a compositional mechanism at the level of one feature model, called *composition rules*, in which "features are related to one another primarily through the use of composition rules, which are a type of constraint on the use of a feature" [Kang et al. 1990]. The survey of the literature about feature modeling, our own experience in different application domains and some inspiration about model-based approaches (i.e., aspect-oriented modeling) lead us to further investigate compositional mechanisms for feature models. Figure 5.1 summarizes the different forms of composition we have identified so far.

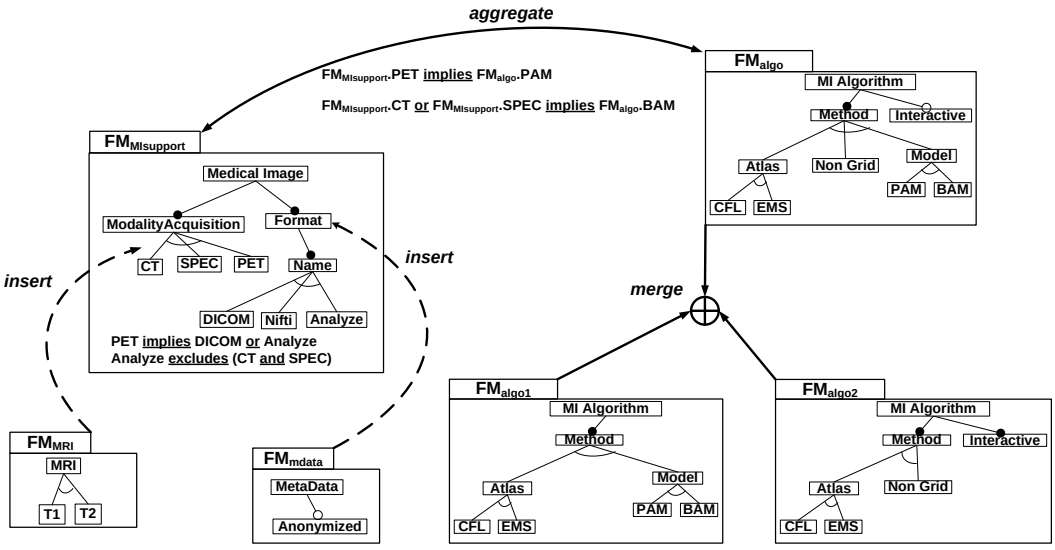


Figure 5.1: Different ways of composing feature models

The first identified mechanism, called *insert*, aims at introducing new features, already organized in a feature model, into a specific location of another existing feature model. It is primarily used to populate an existing feature model with additional information. For example, in Figure 5.1, FM_{mdata} extends the concept of Format modeled in $FM_{MIsupport}$ by incorporating details about metadata support. The insert mechanism may also enable reuse of an existing feature model when creating a larger composed feature model.

The second identified mechanism, called *aggregate*, supports cross-tree constraints between features so that separated feature models can be inter-related. For example, $FM_{MIsupport}$ and FM_{algo}, two concerns of a medical imaging service, are inter-related with two constraints that involve features PET, PAM, CT, SPEC and BAM (see Figure 5.1).

The third mechanism, called *merge*, is dedicated to the composition of feature models that exhibit similar features (i.e., features with the same name). For example, different feature models (see FM_{algo}, FM_{algo1} and FM_{algo2} in Figure 5.1) have been elaborated (e.g., by different suppliers) to describe a family of medical imaging algorithm.

5.1.1 Main Design Choices and Background

For all the identified mechanisms, we prefer to use the term *operator* as usually found in the programming language literature. We see an operator as a special form of function with a limited range of parameter forms. We consider that an operator is similar to an operation in mathematics, i.e., an action or procedure which produces a new value from one or more input values. In our case, the operators produce a new feature model from one or more input feature models. As for any operators, we need to define *i)* syntactical mechanisms that describe how to apply it (e.g., what are its parameters) and *ii)* semantic properties that characterize the input feature models as well as the resulting feature model.

We define the semantics of these operators both in terms of configuration semantics and feature hierarchy.

We consider that the primary meaning of a feature model, known as its *configuration semantics*, is a set of legal configurations, i.e., sets of selected features that respect the dependencies entailed by the diagram and the cross-tree constraints (see Definition 4, page 24). We thus define the semantic properties of each operator in terms of the relationship between the configuration sets of the input models and the resulting feature model. In particular, we rely on the classification proposed in [Thüm et al. 2009] that covers all the changes a designer can produce on a feature model and that provides a sound basis for reasoning about these changes. In [Thüm et al. 2009], the authors distinguish and classify four feature model adaptations[1] (see Definition 10). In this chapter but also in the reminder of the thesis, we will use extensively this classification to characterize the relationship between two feature models.

Definition 10 (Specialization, Refactoring, Generalization, Arbitrary Edit)**.** *Let f and g be two feature models, $[\![f]\!]$ and $[\![g]\!]$ denote their respective sets of configurations.*

- *f is a specialization of g if $[\![f]\!] \subset [\![g]\!]$*
- *f is a generalization of g if $[\![g]\!] \subset [\![f]\!]$*
- *f is a refactoring of g if $[\![g]\!] = [\![f]\!]$*
- *f is an arbitrary edit of g if f is neither a specialization, a generalization nor a refactoring of g.*

The *reduced product*[2] of two sets of sets is also given (see Definition 11) since we will use it to characterize the relationship between two configuration sets.

Definition 11 (Reduced product)**.** *Let A and B being a set of sets.*
The reduced product, denoted $\otimes$, is defined as follows:

$$A \otimes B = \{a \cup b \mid a \in A, b \in B\}$$

Another important property of a feature model is the way features are organized – reflected in the *feature hierarchy*. We recall that two feature models can have identical configuration semantics, yet different hierarchies and thus meaning (see Section 3.1.2, page 24). As a result, we consider that the feature hierarchy (see Definition 3, page 24) should also be part of the semantics of the composition operators.

5.2 INSERT OPERATOR

In Figure 5.1, the concept of medical image is designed from a general perspective and described as a feature model (see $FM_{MIsupport}$). It acts as a *base* or primary model that may not provide all the elements required by an application or system of a medical image, that is, it may be augmented with other features describing different aspects of a medical image. This base model can be composed incrementally with other feature models describing different aspects of features in the base model. These other feature models are called

[1] The authors use the term "edits" since a set of changes to a feature model are applied. An example of edit given in their paper is "moving a feature from one branch to another".

[2] In [Acher et al. 2009b; 2010b], we used the term *"cross product"*. It introduces too much confusion regarding the *"cartesian product"* used in mathematics. As a result, we choose to reuse the term *"reduced product"* introduced in [Schobbens et al. 2007].

aspects[3]. The insert operator aims at introducing newly created elements into any base element or inserting elements from the aspect model into the base model. For example, an SPL practitioner can extend the Format feature associated to a medical image by including the metadata information represented in an aspect feature model FM_{mdata}.

5.2.1 Design Choices

As argued in [Jeanneret et al. 2008], one has to address three concerns when composing two models: Decide which concepts described in the models will be composed – *what* ; Select a location where the insertion or modification will take place in the destination model – *where* ; Determine how to integrate the selected concepts in the composed model to obtain the desired result – *how*.

In the example of Figure 5.1, the dotted arrows suggest that the feature Metadata is inserted below the feature Format while the feature MRI is inserted as a sibling feature of features CT, PET and SPEC. We consider that:

What the entire aspect feature model is inserted and the original feature model is not altered. The precondition of the insert operator requires that the intersection between the set of features of the base feature model and the one of the aspect feature model is empty. This condition preserves the well-formed property of the composed feature model which states that each feature's name is unique ;

Where there is a feature of the base feature model that acts as a *joint point* where the aspect feature model will be inserted ;

How there are various ways to insert an aspect feature model (see Figure 5.1). For example, an SPL practitioner may consider that not all medical images support metadata information (or that the medical imaging service is unable to treat metadata). In this case, the aspect feature model FM_{mdata} can be inserted as *optional* – the feature Metadata becomes an optional, child feature of Format. For the aspect feature model FM_{MRI}, we would like to insert it as part of the *Xor*-group constituted by the features CT, PET and SPEC – the feature MRI becomes a child feature of Modality-Acquisition and belongs to the *Xor*-group of CT, PET and SPEC. As showed by the examples, the variability information should be part of the insert operator.

Syntactic definition. An SPL practitioner needs syntactic mechanisms to precisely define what needs to be inserted, where and how the insertion is achieved. We syntactically define the insert operator as follows:

> insert (aFM: FeatureModel, bFM: FeatureModel, jptFeature: Feature, vop: VariabilityOperator)

It takes four arguments: the aspect feature model to be inserted (aFM), the base feature model (bFM), the targeted feature (a feature in the base model) where the insertion needs to be done ($jptFeature$), and the variability operator vop specified by the user (whose value is either *Mandatory*, *Optional*, *Xor*, *Or*).

[3] We will see afterwards why distinguishing an aspect feature model from a base feature model is important when defining the semantical properties' of the insert operator. We reuse here the terminology of the aspect-oriented programming/modeling (AOP/AOM) community (aspect, join point) since we rely on the principles of AOP/AOM.

5.2.2 Semantics

Two cases are important to distinguish when performing the insertion:

- when the variability operator is either *Mandatory* or *Optional*, the root of the aspect feature model is inserted as a *child* feature of the join point feature. Two examples are given in Figure 5.2. The insertion of $Aspect_1$ with $vop = Mandatory$ (resp. $vop = Optional$) into the feature B of $Base_1$ (see Figure 5.2(a)) produces a new feature model depicted in Figure 5.2(b) (resp. in Figure 5.2(c)) ;
- when the variability operator is either *Xor* or *Or*, the root of the aspect feature model, say aFT, is inserted as a *sibling* feature of the join point feature. For example, when $Aspect_2$ is inserted into the feature N of $Base_2$ with $vop = Xor$ (see Figure 5.2(d), the root feature of $Aspect_2$, R, is inserted as a sibling feature of N (i.e., R is inserted as a child feature of M, the parent of N). In addition,
  - if the join point feature belongs to a feature group (*Xor* or *Or*), then *i)* aFT is part of the feature group and *ii)* the feature group is changed according to the value of vop. For example, the features R, N, O and P form a Xor-group (see Figure 5.2(e)) after the insertion of $Aspect_2$ with $vop = Xor$ into $Base_2$ (see Figure 5.2(d)).
  - in case the join point is a solitary feature (i.e., does not belong to a feature group), a new feature group is created according to the value of vop and constituted by aFT and $jptFeature$.

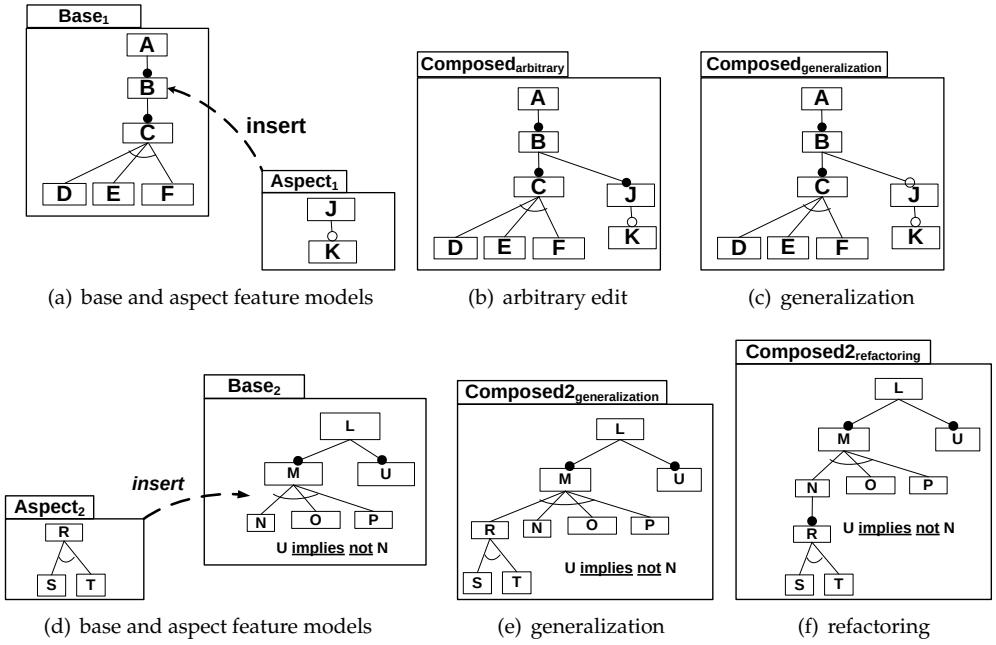
  The insertion of an aspect feature model into the root feature of the base feature model with the variability operator *Xor* or *Or* is not allowed (since the root feature of the base feature model has, by definition, no parent).

Semantic properties. How an aspect feature model is inserted has a direct impact on the set of configurations of the resulting feature model. The insert operator can be seen as an edit (or modification) of a feature model (i.e., the base feature model) into another feature model (i.e., the new composed feature model). In particular, an insertion may change the set of legal feature combinations of the base feature model. As a result, the relationship between the base feature model and the new composed feature model is of primary interest. It explains why we make the distinction between an aspect feature model and a base feature model – we are mostly interested in the *evolution* of the base feature model.

In the reminder, $Base_{FM}$ and $Base'_{FM}$ are two feature models and $[\![Base_{FM}]\!]$ and $[\![Base'_{FM}]\!]$ denote their respective set of configurations. $insert$ produces a feature model $Base'_{FM}$ given a base feature model $Base_{FM}$ and an aspect feature model $Aspect_{FM}$. The semantics of the operator $insert$ is expressed in terms of the relationship between the configuration sets of the input models ($Base_{FM}$ and $Aspect_{FM}$) and the resulting model $Base'_{FM}$, that this, the semantics of the operator $insert$ is defined in terms of the relationship between $[\![Base_{FM}]\!]$, $[\![Aspect_{FM}]\!]$ and $[\![Base'_{FM}]\!]$.

We now illustrate and analyze the properties of the insert operator, given several relevant examples. It is tempting to state that when an aspect feature model is added somewhere in a base feature model $Base_{FM}$, the set of configurations of $Base_{FM}$ "grows" necessarily, that is, new configurations are added to the original set of configuration (and thereby $Base'_{FM}$ is a *generalization* of $Base_{FM}$). This is not necessary the case ($Base'_{FM}$ can also be an arbitrary edit or a refactoring of $Base_{FM}$).

Arbitrary edit. Let us consider the feature models of Figure 5.2(a). We consider the case



(a) base and aspect feature models (b) arbitrary edit (c) generalization

(d) base and aspect feature models (e) generalization (f) refactoring

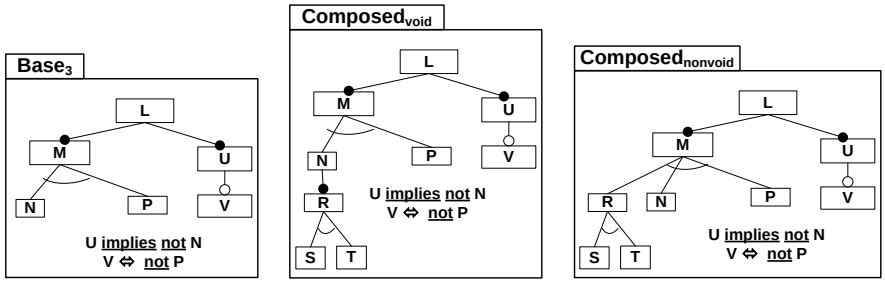
Figure 5.2: Four insertions with different variability operators (optional, mandatory, Xor) and relationships between $Base'_{FM}$ and $Base_{FM}$ (generalization, refactoring)

where the aspect feature model is inserted into the feature B with *Mandatory* as variability operator (see Figure 5.2(b)). The evolution of the base feature model of Figure 5.2(a) into a new composed feature model has altered its original set of configurations. We can notice that each valid configuration of the resulting composed feature model is no longer a valid configuration of the base feature model. The situation corresponds to an arbitrary edit (see Definition 10) between $Base_{FM}$ and $Base'_{FM}$. More precisely, the set of configurations of the composed feature model is equal to the reduced product (see Definition 11) of the two configuration sets, i.e., the following relation holds: $[\![Base_{FM}]\!] \otimes [\![Aspect_{FM}]\!] = [\![Base'_{FM}]\!]$

Generalization. $Base'_{FM}$ may be a generalization of $Base_{FM}$: In this case, new configurations, not valid in of $Base_{FM}$, are represented while all original configurations of $Base_{FM}$ are also valid in $Base'_{FM}$. An example is given in Figure 5.2(c) in which the aspect feature model has been inserted into the feature B with *optional* as variability operator. In this case, all configurations originally valid in the base feature model (see Figure 5.2(a)) are also valid in the composed feature model. The additional set of configurations can be characterized with the cross product. Formally:

$$([\![Base_{FM}]\!] \otimes [\![Aspect_{FM}]\!]) \cup [\![Base_{FM}]\!] = [\![Base'_{FM}]\!] \tag{5.1}$$

Another example of generalization for which the relation 5.1 truly holds is given in Figure 5.2(e): an aspect feature model $Aspect_2$ has been inserted into the feature N of the base feature model $Base_2$ with *Xor* as variability operator.



(a) void feature model (b) void feature model (c) generalization

Figure 5.3: Insertion and satifiability: the case of void feature model

Refactoring. In Figure 5.2(f), the aspect feature model FM_{MRI} is inserted into the feature CT with *mandatory* as variability operator. $Base_{FM}$ and $Base'_{FM}$ have exactly the same set of configurations (i.e., $Base'_{FM}$ is a refactoring of $Base_{FM}$). The reason is that the feature CT is originally a dead feature – a feature that no valid configuration can include (see Definition 8, page 26) – in $Base_{FM}$, and thus, all its child features are also dead features.

Another important facet of the semantics of the insert operator is related to the satisfiability of the feature models. We recall that a feature model is void (or unsatisfiable) if it represents no configurations (see Definition 7, page 26).

Satisfiability. Let us consider the base feature model of Figure 5.3(a). It may happen that the insertion of an aspect feature model does produce a void feature model in case the base feature model is void. For example, in Figure 5.3(b), the aspect feature model $Aspect_2$ is inserted into the feature N with *mandatory* as variability operator: The resulting feature model is also void. Intuitively, when a feature model is void, all features are dead. As a result, we are in the same situation as in Figure 5.2(f) – no new configurations have been added to the original empty set of configurations of the base feature model.

In addition, it is also possible that a base feature model whose set of configurations is empty (as the one of Figure 5.3(a)) becomes a non void feature model after an insertion. For example, in Figure 5.3(c), the aspect feature model $Aspect_2$ is inserted into the feature N with *Xor* as variability operator. The resulting feature model is no longer void since four configurations are now valid: $[\![Base'_{FM}]\!] = \{ \{ \mathsf{L, M, U, R, S} \}, \{ \mathsf{L, M, U, R, T,} \}, \{ \mathsf{L, M, U, R, S, V} \}, \{ \mathsf{L, M, U, R, T, V} \} \}$

Recaps. The various examples show that the kind of relationship between $Base_{FM}$ and $Base'_{FM}$ is dependent both on the original properties of $Base_{FM}$ and $Aspect_{FM}$, the variability operator and the joinpoint feature chosen. Table 5.1 recaps the kind of relationship and the relationship between $Base_{FM}$ and $Base'_{FM}$ in terms of sets of configurations using the examples of this section. Importantly, $Base'_{FM}$ being a specialization (see Definition 10) of $Base_{FM}$ is not possible, assuming that $Aspect_{FM}$ is not void.

Example	Relationship	Properties
Figure 5.2(b) and 5.3(c)	arbitrary edit	$[\![Base_{FM}]\!] \otimes [\![Aspect_{FM}]\!] = [\![Base'_{FM}]\!]$
Figure 5.2(c) and 5.2(e)	generalization	$[\![Base_{FM}]\!] \subset [\![Base'_{FM}]\!] \wedge$ $([\![Base_{FM}]\!] \otimes [\![Aspect_{FM}]\!]) \cup [\![Base_{FM}]\!] = [\![Base'_{FM}]\!]$
Figure 5.3(b) and 5.2(f)	refactoring	$[\![Base_{FM}]\!] = [\![Base'_{FM}]\!]$

Table 5.1: Insert operator: properties of $Base'_{FM}$ in terms of $Base_{FM}$ and $Aspect_{FM}$

Proof by contradiction. Suppose $Base'_{FM}$ is a specialization of $Base_{FM}$ when $Aspect_{FM}$ is inserted into $Base_{FM}$. In this case, all inserted features, including the root feature aFT of $Aspect_{FM}$, should be dead in $Base'_{FM}$ (otherwise there exists a configuration $c \in [\![Base'_{FM}]\!]$ such that $aFT \in c$ and $Base'_{FM}$ is an arbitrary edit or a generalization of $Base_{FM}$). The only situation in which the feature aFT can be dead is when it is inserted as a child feature of another dead feature of $Base_{FM}$. Therefore it does not change the set of configurations and $[\![Base_{FM}]\!] = [\![Base'_{FM}]\!]$. This contradicts our initial assumption, so we can conclude that $Base'_{FM}$ is not a specialization of $Base_{FM}$.

Similarly, we can demonstrate that an insertion of an aspect feature model into a satisfiable base feature model cannot produce a void feature model.

This simply follows the *meaning* of an insertion which is to add details and to populate the base feature model with additional information (as initially discussed in the beginning of this section). Due to the various kinds of relationships that may hold when performing an insertion, automated techniques are needed to reason about the properties of the composed feature model. This is further discussed in the next chapters.

5.3 Aggregate Operator

Another form of composition is to *aggregate* two (or more than two) feature models (e.g., see Figure 5.4(a)). The operator aims to inter-relate separated feature models through cross-tree constraints: Features in input feature models are related to each other through relations expressed in propositional logic.

5.3.1 Design Choices

To define the syntax or the semantic properties of the aggregate operator, we do not make the distinction between a base feature model and an aspect feature model. We rather consider that the feature models are equally important.

Syntactic definition. We syntactically define the aggregate operator as follows:

aggregate (sFM: set of FeatureModel, sCst: set of Constraint)

The aggregate operator takes as input a set of feature models (sFM), a set of propositional constraints ($sCst$) and produces a new feature model. The input feature models are

aggregated under a *synthetic* root $synthetic_{ft}$ so that the root features of input feature models are child-mandatory features of $synthetic_{ft}$. In addition, the propositional constraints are added in the resulting feature model.

Why a synthetic root? Instead of creating a new root feature in the resulting feature model, one can argue that one of the root features of the input feature models can play the role of the root feature. This solution may not properly restitute the meaning of the features and the way they are *conceptually* or *ontologically* related. For example, the feature A can be the root feature of the aggregated feature model while the feature M becomes its child-mandatory feature. However, the concept described by the feature A may not be composed-of or refined by the concept described by the feature M, i.e., the two concepts may be unrelated or the relation between A and M should actually be reversed. Another conceivale solution is to build a forest (i.e., a disjoint union of trees).

In line with the formalism used throughought the thesis, we consider that the synthetic root is only here to ensure the well-formedness of the hierarchy and is *not* part of the set of configurations of the aggregated feature model. Its particular status should be treated as such by a tool (e.g., when encoding the feature model into a propositional formula).

5.3.2 Semantic properties

From a semantical perspective, it is interesting to characterize the set of configurations of the aggregated feature model in terms of the set of configurations of the input feature models. The aggregate operator is applied four times using the same input feature models, FM_{c1} and FM_{c2}, but with a different set of constraints: The four resulting feature models, noted $FM1_{aggregated}$, $FM2_{aggregated}$, $FM3_{aggregated}$ and $FM4_{aggregated}$, are shown in Figure 5.4(a), 5.4(b), 5.4(c) and 5.4(d). We analyze their properties.

Reduced product. In Figure 5.4(a), the following relation holds:

$$[\![FM1_{aggregated}]\!] \subset ([\![FM_{c1}]\!] \otimes [\![FM_{c2}]\!])$$

Intuitively, some valid configurations of FM_{c1} and FM_{c2} are combined together to form new configurations in $FM1_{aggregated}$. Due to the constraints, not all combination of configurations of FM_{c1} and FM_{c2} are allowed in $FM1_{aggregated}$. For example, { A, B, E, S, M, N, Model } is not valid in $FM1_{aggregated}$.

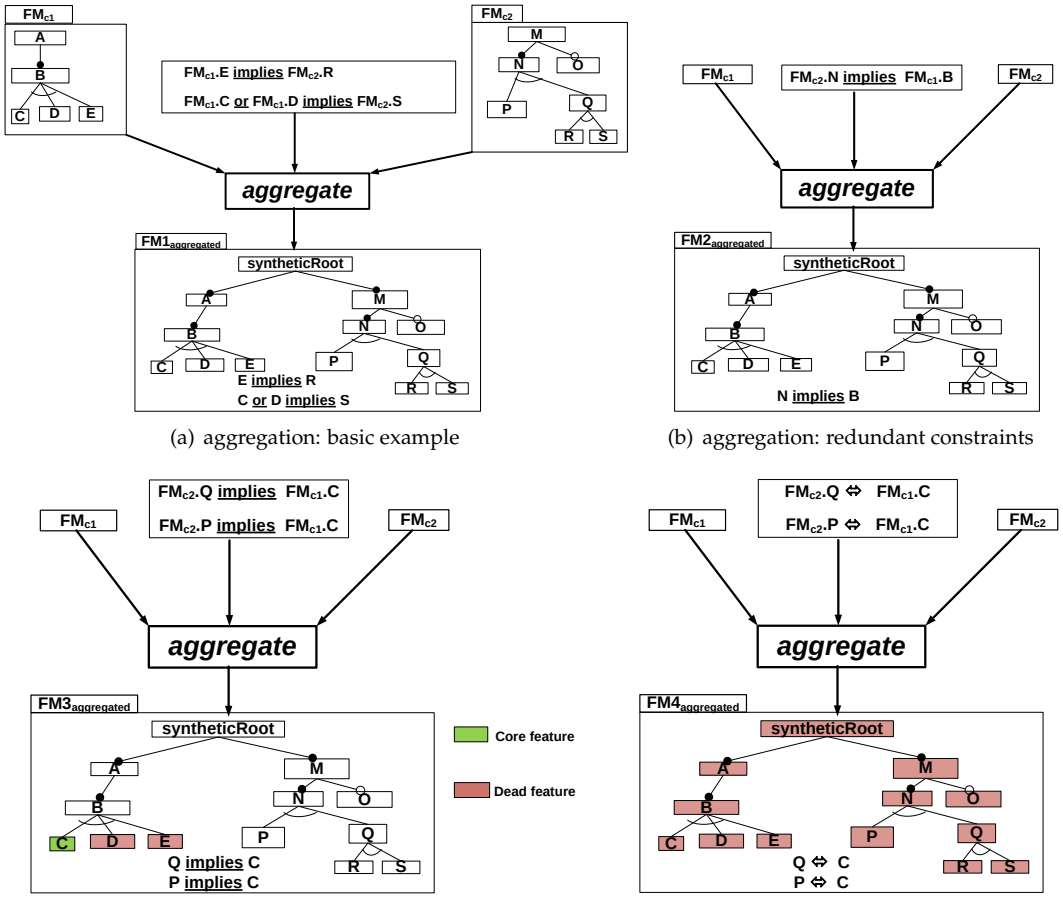
Redundant constraints. In Figure 5.4(b), the following relation holds:

$$([\![FM_{c1}]\!] \otimes [\![FM_{c2}]\!]) = [\![FM2_{aggregated}]\!]$$

Contrary to the previous example, the constraints do not reduce the set of configurations $([\![FM_{c1}]\!] \otimes [\![FM_{c2}]\!])$ and all combination of configurations of FM_{c1} and FM_{c2} are allowed in $FM2_{aggregated}$. As a result, the aggregated feature model of Figure 5.4(a) is logically equivalent to the aggregation of FM_{c1} and FM_{c2} without constraints. Intuitively, the constraint $N \Rightarrow B$ is logically entailed by the aggregation of FM_{c1} and FM_{c2} without constraints.

Dead and core features. In Figure 5.4(c), the following relation holds:

$$([\![FM_{c1}]\!] \otimes [\![FM_{c2}]\!]) \subset [\![FM3_{aggregated}]\!]$$



(a) aggregation: basic example

(b) aggregation: redundant constraints

(c) aggregation: dead and core features

(d) aggregation: void feature model

Figure 5.4: Aggregate: four cases

Due to the constraints, features D and E are now dead features while feature C is a core feature (see Definition 9, page 27) of $FM3_{aggregated}$.

Void feature model. In Figure 5.4(d), the following relation holds:

$$[\![FM4_{aggregated}]\!] = \emptyset$$

Hence the aggregation of two feature models together with constraints may produce a void feature model.

Recaps. The properties of the aggregated feature model heavily depends on the set of propositional constraints used during the aggregate. It may lead to situations where the aggregated feature model does not represent any valid configuration or include dead or core features. We consider that the aggregate operator is purely syntactical: Other automated techniques should be developed, for example, to simplify the aggregated feature model or to reason about its properties. This is further discussed in the next chapters.

5.4 MERGE OPERATOR

5.4.1 Design Choices

Motivation. Different views or perspectives are usually elaborated to describe a system, a concern of the system or simply a concept. The views or perspectives may exhibit variability and therefore several feature models may be developed. For example, the different feature models FM_{algo}, FM_{algo1} and FM_{algo2} of Figure 5.1, page 52 can be seen as different views of a medical imaging algorithm. In this case, it is likely that feature models use similar features to describe the system under study. As a result it is necessary to reason about their composition. In particular, one wants to *merge* the overlapping parts of the feature models to obtain a single model that presents an integrated view of the system. We thus design a merge operator that, given a set of input feature models, produces a new feature model, called merged feature model.

Assumptions. The goal of model merging is traditionally to group together (i.e., merge) model elements that describe the same concepts in the input models to be composed [Brunet et al. 2006]. In the context of feature models, we consider that model elements are features and that two features match (i.e., represent the same concepts) if they have the same name. For example, when merging the two feature models fm_{m3} and fm_{m4} (see Figure 5.6(a) and Figure 5.6(b)), we consider that the features A, B, C, F, E match. We assume that at least the root features of the input feature models match (i.e., have the same name).

Syntactic definition. At this step, a crucial issue is to determine which variability information should be attached to features in the merged feature model. We consider that there are different ways to perform the merging, i.e., different semantics of the merge operator can be chosen. For example, we might want to include all configurations of the input feature models ; we might have a more restrictive strategy in which only common configurations of the input feature models are retained, etc. In both cases, we consider that the key

element that determines the semantics' choice of an SPL practitioner is the set of configurations characterized by the merged feature model. We syntactically define the aggregate operator as follows:

merge (sFM: set of FeatureModel, mode: MergeMode)

We consider that the merge operator takes as input a set of feature models sFM, a merging mode $mode$ and produces a new feature model. As there are different ways to merge two or more than two feature models, several modes are defined for the merge operator. The merging mode can be either *union*, *strict union*, *intersection* or *diff*. The set of configurations expressed by the merge feature model depends on this merging mode.

5.4.2 Semantic properties

Configuration Semantics. Like for the operators insert or aggregate, the properties of a merged feature model produced by an application of the merge operator are formalized in terms of the sets of configurations of input feature models. We now described these properties for all modes of the merge operator.

Union mode. The union mode is the most inclusive option: the merged feature model includes all the valid configurations defined by the input feature models. Formally:

$$[\![FM_1]\!] \cup [\![FM_2]\!] \subseteq [\![FM_r]\!] \qquad (M_1)$$

The property M_1 and this mode are considered in [Segura et al. 2008, Acher et al. 2009b]. We will not use this mode in the context of the thesis since the property M_1 is too loose. In particular, some valid configurations of FM_r are neither valid in FM_1 nor in FM_2, leaving open to various interpretations. Therefore the configuration semantics of FM_r appears to be too vague for this mode.

Strict Union mode. In the strict[4] union mode, we want to obtain a merged feature model FM_r that represents exactly the union of the two sets of configurations of FM_1 and FM_2. In this mode, each valid configuration of FM_r is also valid either in FM_1 or FM_2 (or in both). Formally:

$$[\![FM_1]\!] \cup [\![FM_2]\!] = [\![FM_r]\!] \qquad (M_2)$$

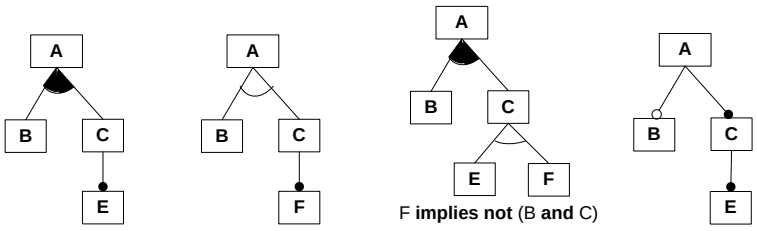
The merge operator in the strict union mode is denoted $FM_1 \oplus_{\cup_s} FM_2 = FM_r$.

An example is given in given in Figure 5.5: fm_{m56} (see Figure 5.5(c)) is the feature model resulting from the merge in intersection mode of fm_{m5} (see Figure 5.5(a)) and fm_{m6} (see Figure 5.5(b)).

Intersection mode. The intersection mode is the most restrictive option: the merged feature model, FM_r, expresses the common valid configurations of FM_1 and FM_2. The merge operator in the intersection mode is denoted as follows: $FM_1 \oplus_{\cap} FM_2 = FM_r$. The relationship between a merged feature model FM_r in intersection mode and two input feature models FM_1 and FM_2 can be expressed as follows:

$$[\![FM_1]\!] \cap [\![FM_2]\!] = [\![FM_r]\!] \qquad (M_3)$$

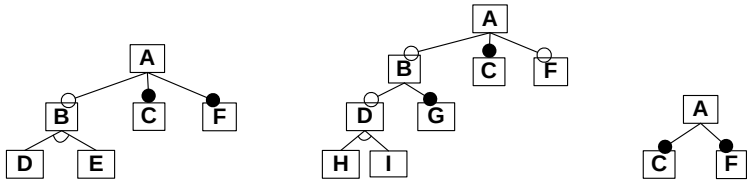
[4] In [Acher et al. 2009b], we use the expression *"strict union mode"* to clearly differentiate the property M_1 from the property M_2. We keep this terminology in this thesis.



(a) fm_{m5} (b) fm_{m6} (c) fm_{m56} (d) fm_{diff56}

Figure 5.5: Merging in strict union mode ($fm_{m5} \oplus_{\cup_s} fm_{m6} = fm_{m56}$) and diff mode ($fm_{m5} \oplus_{\setminus} fm_{m6} = fm_{diff56}$)

An example is given in given in Figure 5.6: fm_{m34} (see Figure 5.6(c)) is the feature model resulting from the merge in intersection mode of fm_{m3} (see Figure 5.6(a)) and fm_{m4} (see Figure 5.6(b)).



(a) fm_{m3} (b) fm_{m4} (c) fm_{m34}

Figure 5.6: Merging in intersection mode: ($fm_{m3} \oplus_{\cap} fm_{m4} = fm_{m34}$)

As we rely on set theory, the merge operators in union, strict union and intersection mode are associative and commutative w.r.t. the set of configurations. Therefore, the properties M_1, M_2 and M_3 defined for two feature models can be extended to n feature models, $n \geq 2$.

Diff mode. Another merge operator, called diff, is denoted as $FM_1 \oplus_{\setminus} FM_2 = FM_r$. The following defines the semantics of this operator:

$$[\![FM_1]\!] \setminus [\![FM_2]\!] = \{x \in [\![FM_1]\!] \mid x \notin [\![FM_2]\!]\} = [\![FM_r]\!] \qquad (M_3)$$

An example is given in given in Figure 5.5: fm_{diff56} (see Figure 5.5(d)) is the feature model resulting from the merge in diff mode of fm_{m5} (see Figure 5.5(a)) and fm_{m6} (see Figure 5.5(b)).

Hierarchy. Several feature models, with different hierarchies, can represent the same set of configurations (see Chapter 3, page 26). So in particular several merged feature models can be produced and consistently represent the expected set of configurations while having different hierarchies. From a user perspective, some hierarchies may decrease the maintainability or understandability of the resulting feature model. We consider that the

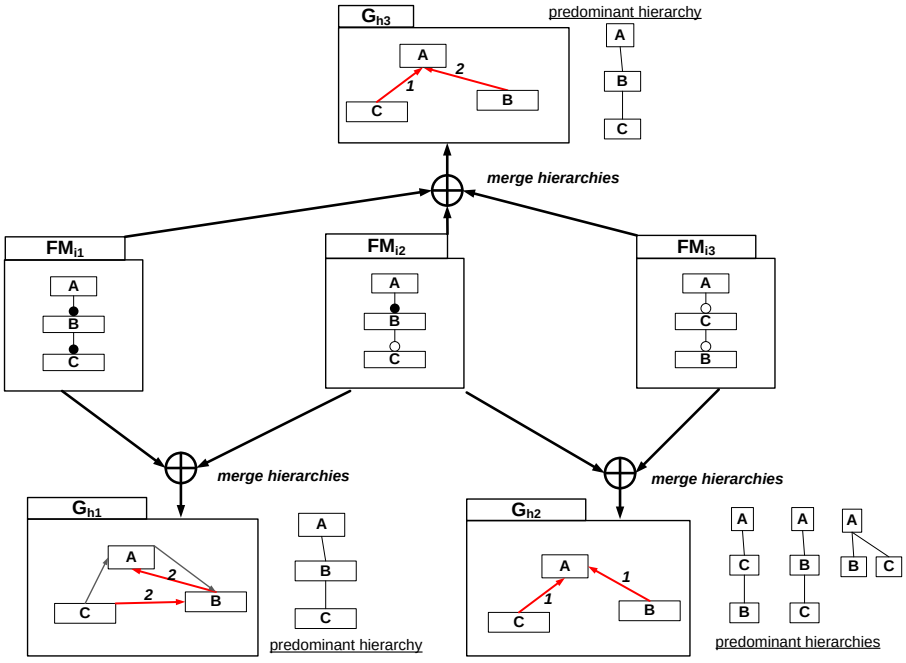


Figure 5.7: Hierarchy and merge operations (strict union mode)

hierarchy of the merged feature model should be part of the semantics of the merge operator.

Strict union mode. We first consider the merge in strict union mode. When the input feature models have the same parent-child relations w.r.t. the common features, the hierarchy of the merged feature model is immediate: It is simply the union of the set of features of the input feature models and the union of the set of edges of the input feature models. For example, we consider the merging of FM_{i1} and FM_{i2} (see Figure 5.7). The hierarchy is $G_{h12} = (V_{h12}, E_{h12}, r_{h12})$ with $V_{h12} = \mathcal{F}_{FM_{i1}} \cup \mathcal{F}_{FM_{i2}} = \{A, B, C\}$, $E_{h12} = E_{FM_{i1}} \cup E_{FM_{i2}} = \{(A, B), (B, C)\}$ and $r_{h12} = A$.

It is more complex when merging FM_{i2} and FM_{i3} (see Figure 5.7). The parent-child relationship between B and C is not in the same order in FM_{i2} or FM_{i3}. At first glance, it seems difficult to determine the most desirable hierarchy. Intuitively we want a *predominant hierarchy* that restitues as much as possible the parent-child relationships originally expressed in FM_{i2} and FM_{i3}. For this example, three predominant hierarchies are obtained. Two of the hierarchies, though, cannot be used to build the feature model. Let us investigate why.

The expected set of configurations of the merged feature model, denoted FM_{h123}:
$[\![FM_{i2}]\!] \cup [\![FM_{i3}]\!] = [\![FM_{h123}]\!] = \{\{A, B\}, \{A, B, C\}, \{A, C\}, \{A\}\}$

We can observe that the feature B cannot be a child feature of C (otherwise, the configuration $\{A, B\}$ is not valid). Similarly, the feature C cannot be a child feature of B (otherwise, the configuration $\{A, C\}$ is not valid). Fortunately the third predominant hierarchy, in which features B and C are sibling features, can be chosen.

Let us now consider the merging of FM_{i1}, FM_{i2} and FM_{i3} (see Figure 5.7). In this example, there is only one predominant hierarchy: The feature B is the parent feature of the feature C since it is also the case in the two feature models FM_{i1} and FM_{i2}. Unfortunately, this predominant hierarchy cannot be used to build the feature model.

The idea of predominant hierarchy, as currently exposed, is not correct. The reason is that the solution only considers one aspect of a feature model at a time – the hierarchy. We need to consider at the same time the configuration semantics and the hierarchy of the feature models. In particular, we have identified that, in some cases, the set of possible hierarchies is restricted by the expected set of configurations.

Hence, we propose to work directly at the level of configuration semantics. We first compute the implication graph I of the propositional formula ϕ_r, representing the set of configurations of FM_r, the merged feature model. I is a directed graph $I = (V_{imp}, E_{imp})$ formally defined as follows:

$$V_{imp} = \mathcal{F}_r \qquad E_{imp} = \{(f_i, f_j) \mid \phi_r \wedge f_i \Rightarrow f_j\} \tag{5.2}$$

Each binary, directed edge from feature f_i to feature f_j represents an implication. Binary edges in the implication graph are candidate as parent-child relationships in a feature model. However, the transitivity of implication results in many more binary edges. Hence we need to select and remove some edges until obtaining a comprehensive hierarchy (i.e., a tree). Intuitively, the more a parent-child relation occurs in the input feature models, the more an edge in the implication graph should be retained.

We formulate the problem of choosing a hierarchy from amongst a set of hierarchies as a *directed minimum spanning tree problem* [Edmonds 1967].

I is now considered as a directed, weighted graph. A function $w : E_{imp} \rightarrow \mathbb{N}$ assigns a weight $w(e)$ to each edge $e \in E_{imp}$ equal to $-n$, n being the number of times e occurs in the different hierarchies of the input feature models (see red arrows in Figure 5.7). The problem is then to find a rooted directed spanning tree, $T = (V_{imp}, S)$ where S is a subset of E_{imp} and such that the function $w(T) = \sum_{e \in T} w(e)$ is minimized. The rooted directed spanning tree is defined as a graph which connects, without any cycle, all nodes with $|V_{imp}| - 1$ edges.

Several algorithms have been developed for resolving the problem [Edmonds 1967, Tarjan 1977]. The order of complexity is similar to the algorithms developed for the classical problem of undirected minimum spanning tree. An important property is that there may be several directed minimum spanning trees of the same weight having a minimum number of edges. Without additional knowledge or clear criteria, the hierarchy can simply be arbitrary chosen.

Intersection and diff mode. We now consider the merge in intersection or diff mode. An important remark is that some features of the input feature models may not be part of any configuration of the merged feature model. For example, the merge in intersection mode of fm_{m3} (see Figure 5.6(a)) and fm_{m4} (see Figure 5.6(b)) produces a feature model fm_{m34} (see Figure 5.6(c)) in which the feature B is not present. As a result, similar techniques can be applied to determine the hierarchy of the merged feature model but dead features should be preliminarily removed.

5.5 Implementation of the Merge Operator

We have defined the properties of the composition operators – mainly in terms of set of configurations and hierarchy of the input and composed feature models. The challenge is now to develop automated techniques that compute the feature models while guaranteeing the expected properties. The implementation of the insert and aggregate operator is rather straightforward. In this section, we focus on the implementation of the merge operator that presents much more difficulties.

5.5.1 Propositional Logic Based Merging

The computation of the hierarchy of the merged feature model has indicated that we need to reason directly at the semantic level. The key ideas of the proposed algorithm are to *i)* compute the propositional formula representing the expected set of configurations and then *ii)* to apply propositional logic reasoning techniques to construct a feature model, including a hierarchy, feature groups and constraints, from the propositional formula.

Formula Computation. For each mode of the merge operator, a different propositional formula is computed.

Union. The strict union of two sets of configurations represented by two feature models, FM_1, and FM_2, is computed as follows. First, FM_1 (resp. FM_2) feature models are encoded into a propositional formula ϕ_{FM_1} (resp. ϕ_{FM_2}) as defined in [Batory 2005, Czarnecki and Wąsowski 2007]. Then, the following formula is computed:

$$\phi_{Result} = (\phi_{FM_1} \wedge not(\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1})) \vee (\phi_{FM_2} \wedge not(\mathcal{F}_{FM_1} \setminus \mathcal{F}_{FM_2}))$$

with $\mathcal{F}_{FM_1}$ (resp. $\mathcal{F}_{FM_2}$) the set of features of FM_1 (resp. FM_2) feature model. $\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1}$ denotes the complement or difference of $\mathcal{F}_{FM_2}$ with respect to $\mathcal{F}_{FM_1}$.

not is a function that, given a non-empty set of features, returns the Boolean conjunction of all negated variables corresponding to features:

$$not(\{f_1, f_2, ..., f_n\}) = \bigwedge_{i=1..n} \neg f_i$$

The presence of negated variables is needed since we need to emulate the deselection of features that are in FM_1 (resp. FM_2) but not in FM_2 (resp. FM_1). Otherwise, two features, say $f \in \mathcal{F}_{FM_1}$ and $g \in \mathcal{F}_{FM_2}$ such that $f \neq g$, can be combined to form a configuration, thereby violating the configuration semantics of the merge in strict union mode. The importance of negating features is illustrated in Appendix .1.

Intersection. Computing the intersection of two sets of configurations represented by two FMs, FM_1 and FM_2, follows the same principles and we obtain:

$$\phi_{Result} = (\phi_{FM_1} \wedge not(\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1})) \wedge (\phi_{FM_2} \wedge not(\mathcal{F}_{FM_1} \setminus \mathcal{F}_{FM_2}))$$

Diff. Similarly, computing the diff of two sets of configurations represented by two FMs, FM_1 and FM_2, is as follows:

$$\phi_{Result} = (\phi_{FM_1} \wedge not(\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1})) \wedge \neg(\phi_{FM_2} \wedge not(\mathcal{F}_{FM_1} \setminus \mathcal{F}_{FM_2}))$$

From formula to feature model. We reuse and adapt techniques presented in [Czarnecki

and Wąsowski 2007, She et al. 2011]. Czarnecki et al. propose an algorithm to construct a feature diagram from a Boolean formula. Propositional logics techniques are developed to detect features logically implied, *Xor-* and *Or-* groups using the method of prime implicants. Furthermore the authors propose a generalized notation, roughly, a directed acyclic graph with additional nodes for feature groups. A major difference is that, in our work, we already *know* the resulting hierarchy, i.e., a tree in which a feature has only one parent. We thus exploit this information to streamline the algorithm. It proceeds as follows:

① **Hierarchy.** We first compute G_{Result}, the hierarchy of the merged feature model, as previously explained. At this step, all features, except root, are optional (see Definition 1, page 23).

② **Mandatory and Feature Groups.** We compute the implication graph (see previous section), noted I_{Result}, of the formula ϕ_{Result} over $\mathcal{F}'_{Result} = \mathcal{F}_{Result} \setminus deads(\phi_{Result})$, $\mathcal{F}_{Result}$ being the set of features of FM_{Result}, $deads$ being a function that computes the set of dead features.

I_{Result} is a directed graph $G = (V, E)$ formally defined as:

$$V = \mathcal{F}'_{Result} \qquad E = \{(f_i, f_j) \mid \phi_{Result} \wedge f_i \Rightarrow f_j\}$$

We use I_{Result} to identify biimplications and thus set mandatory features together with their parents (i.e., setting $E_{Result_{MAND}}$). For feature groups, we reuse the prime implications method proposed in [Czarnecki and Wąsowski 2007] and thus set $\mathcal{F}_{Result_{XOR}}$ and $\mathcal{F}_{Result_{OR}}$. It must be noted that a feature may be candidate to several feature groups (which is not allowed by our formalism). For example, a feature may be candidate to be part of an Or-group and a Xor-group. Nevertheless, it is difficult to determine whether a feature group must predominate over another feature group. Moreover, it is out of the scope of the semantics of the merge operator. Currently, we use the information of the original input feature models to favor features that were initially grouped.

③ **Constraints.** The set of implies constraints, if any, can be deduced by removing edges of I_{Result} that are already expressed in the feature diagram (e.g., parent-child relations). Similarly, excludes constraints that were not chosen to be represented as an Xor-group are added. During the incremental adding of constraints, we control that the constraint is not already induced by the current feature diagram.

The feature diagram, including the implies/excludes constraints, may still be an over approximation of ϕ_{Result}. It is detected by checking the logical equality between ϕ_{Result} and $\phi_{Result_{diagram}}$, the encoding of the computed feature diagram as a propositional formula. The relative complement of ϕ_{Result} with respect to $\phi_{Result_{diagram}}$ corresponds to $\psi_{Result_{cst}}$ (see Definition 2, page 24) and can be computed using standard propositional logic techniques.

5.5.2 Comparison with Other Techniques

There are several existing techniques that are intuitive candidates to implement the merging operators. We briefly describe some of these techniques and report our experience with them (see [Acher et al. 2010a] for more details).

The overall idea of the technique described in [Schobbens et al. 2007] and [Heymans et al. 2008] is that intersection or (strict) union can be realized by maintaining *separate* input

feature models and inter-relating them with constraints. The major limitations are that *i)* the resulting merged feature model may contain anomalies (false optional features, dead features) and *ii)* the entire set of features of input feature models is included in the resulting feature model so that the number of features quickly increases and large feature models are produced. The authors recognize that *"the resulting feature model should probably be simplified for readability"*. The presented technique can be seen as a *reference*-based technique. In the next chapter, we will further study this class of techniques and show that they imply additional drawbacks. Furthermore, we will show how anomalies can be removed and thus how the resulting feature model can be simplified for readability in Chapter 7.

Alves et al. motivate the need to manage the evolution of feature models or more generally of an SPL and extend the notion of refactoring to feature models [Alves et al. 2006]. Although their work is focused on refactoring single feature models, they also suggest to use these rules to merge feature models. Inspired by the work of Alves et al., Segura et al. provide a catalogue of visual rules to describe how to merge feature models [Segura et al. 2008]. They propose to apply the catalog of rules using AGG technology [Taentzer 2004]. In AGG, a transformation rule is composed mainly of a source graph or Left-Hand Side (LHS) and a target graph or Right-Hand Side (RHS). For each merge rule of the catalogue, LHS consists of two input feature model patterns (pre-conditions) and RHS describes an output feature model pattern representing the merging result (post-conditions). Our experience is that the implementation turned out to be time-consuming and error-prone. More importantly, the confidence in the implementation appears to be too low: There is no proof of the completeness of the AGG rules so that the semantic properties cannot be guaranteed in all cases. Studying theorem provers and model checkers, as done in [Gheyi et al. 2006] *for* refactoring rules (the starting point of [Segura et al. 2008]), is still to be done and requires intensive research. The intersection mode remains particularly challenging to be implemented due to the presence of dead features.

We also considered the use of Kompose [Reddy et al. 2006, Fleurey et al. 2007] which implements a generic structural composition operator that can be specialized for a particular modeling language. In Kompose, the composition mechanism is structured in two major phases: (1) The *Matching* phase identifies model elements that describe the same concepts in the input models to be composed; (2) The *Merging* phase where matched elements are merged to create new elements in the resulting model. We reuse Kompose facilities to realize the merging strategy. The decision to merge features or not and the nature of the resulting operator depends on the intended semantic properties (e.g., the merging of an Or-group with an Or-Group gives an Or-Group in union mode), as described in [Acher et al. 2009b]. Our experience is that a compositional approach structured in two-stages (matching and merging) is too syntactical for reifying the variability information, especially in the presence of constraints.

In [Acher et al. 2009b], we assumed that input feature models do not contain cross-tree constraints. Nevertheless, the handling of constraints in feature models is useful and we raise this limitation in this chapter. Moreover, this assumption is not valid in the general case since the merge operator may produce constraints in the merged feature model. For example, if we consider the merge in strict union mode of fm_{m5} (see Figure 5.5(a)) and fm_{m6} (see Figure 5.5(b)), we can observe that the resulting feature fm_{m56} (see Figure 5.5(c)) does contain constraints.

[van den Broek et al. 2010] consider the problem of merging feature models which *"consist of trees with implies and excludes constraints"*. They propose an algorithm that guarantees

some properties (minimality, parent compatibility, commutativity, etc.) of the merged feature models. The formalism used exactly corresponds to the notion of feature diagram defined in Chapter 3. They also assume that the input feature models are parent-compatible. Therefore their algorithm should be adapted to handle arbitrary propositional constraints and deal with different hierarchies. This last limitation is also shared by the work of [Alves et al. 2006, Segura et al. 2008, Acher et al. 2009b].

To summarize, the use of propositional logic techniques for the implementation of the merge operators outperforms current solutions, raises previous limitations and notably preserves, by construction, the set of configurations.

Six

Merge Operator and Multiple Feature Models

This chapter shares material with the technical report "Managing Multiple Software Product Lines Using Merging Techniques" [Acher et al. 2010b]

In this chapter, we show *how merging techniques can be used to manage multiple feature models*. We illustrate how the merge operators can be applied to manage the variability of a set of SPLs (called multiple SPLs in the rest of the chapter) by producing feature models that support selection of products from among sets of competing products provided by different suppliers. We use the example of Chapter 4 that involves the building of a catalog of medical image analysis services. We show that the proposed technique results in more compact feature models that are easier to understand and analyse.

Context and Motivation. In some SPL environments, support for manipulating *multiple SPLs* may be needed. For example, in the consumer electronics domain, the reuse of software components from different SPLs is commonplace [van Ommering 2002]. Some of these SPLs may be developed and maintained by external suppliers, and some of the suppliers may compete to deliver similar products. The same observation can be made in the semiconductor industry where hardware components from several suppliers are integrated into a product [Hartmann and Trew 2008]. In the medical imaging analysis on the grid domain, catalogs of analysis services are built and reused to create new workflows with strong guarantees on highly variable functional and non functional properties. Across many fields, there is thus a need for SPL engineering approaches that support defining and managing variability across different SPLs [Pohl et al. 2005, Buhne et al. 2005].

Managing variabilities across multiple SPLs is especially challenging when the SPLs are owned by different companies [Pohl et al. 2005, Hartmann and Trew 2008, Hartmann et al. 2009, Bosch 2009]. Support for composing multiple feature models can help domain engineers produce coherent characterizations of valid combinations of features taken from multiple SPLs. Product (application) engineers also need support for producing valid product configurations that belong to one or several SPLs. In particular there is a need to determine which SPLs are able to provide a specific (combination of) feature(s) or not.

Merge vs Reference. Recently proposed techniques support automatic creation of feature models that integrate features from multiple SPLs with *references* to them [Schobbens et al. 2007, Hartmann et al. 2009, Reiser and Weber 2007]. Nevertheless, the number of features and cross-tree constraints in the resulting feature model quickly becomes too large to be understood by an SPL practitioner or to be analyzed by state-of-the-art reasoning tools. To

avoid this problem, another solution is to *merge* similar features (e.g., when features match and have typically the same name) of input feature models. This solution assumes that the SPLs to be managed share similar features, but it is often the case when the SPLs are competing in a specific domain.

In this chapter, we provide some evidence that these *reference-based techniques* are not scalable since the feature models they produce are difficult to understand and use in comparison with the *merging techniques* we present. Furthermore we show that the reuse of state-of-the-art reasoning operations for querying the feature model produced (e.g., for a comparison with another feature model) is directly applicable and does not require any adaptation.

Remainder. We first motivate the need for managing multiple SPLs using an example in which a catalog of image analysis services are built (Section 6.1). We introduce *competing* multiple SPLs, an important class of multiple SPLs, in which a valid combination of features corresponds to at least one product of constituent SPLs (Section 6.2).

To formally define competing multiple SPLs, we describe a semantic foundation that is expressed in terms of feature models and feature configurations. We show that the semantic foundation can be realized by merge operators, assuming that features match (Section 6.3). We demonstrate how the merge operators automatically assist stakeholders in *i)* developing an SPL from multiple SPLs, *ii)* ensuring availability of products across multiple SPLs or *iii)* evolving existing SPLs. Properties of the merge operators are compared to reference-based techniques (Section 6.4).

6.1 Back to the Running Example: Engineering Services as SPLs

The elaboration of a feature model to model the variability of SPLs (e.g., services) is an important engineering activity that involves different tasks. We illustrate them for a SPL of segmentation services, on Figure 6.1.

This figure contains snapshots of three typical stages of the life-cycle of an SPL : ① shows the variability requirements (described as a feature model) that corresponds to the range of segmentation services needed by domain users for building their medical workflows, ② represents the choices (a configuration of this feature model) made by one user for selecting the services that meet the requirements of a given workflow (the green check mark in the box states that a feature is selected, the red cross means that the feature is deselected and nothing in the box states that no choice has been made yet). ③ corresponds to the set of services (one or several feature models) that are effectively provided to this user for implementing the workflow. These services may be home-made or provided by external suppliers as it is the case in Figure 6.1, where services from different SPLs (from *Supplier 1* and *Supplier 2*) share common characteristics (e.g., MedicalImage, ModalityAcquisition and Format supported by a service) and can be organized within a catalog of services, referred as *multiple SPLs* in the next section.

Figure 6.1 shows that it is necessary to reason about the various interactions that exist between the many feature models and configurations described in these three snapshots. For instance, an important consistency check of the SPL is to determine whether all valid combinations of features offered to the medical imaging expert can be realized by existing services of at least one supplier [Metzger et al. 2007].

An interesting property is that the various kinds of reasoning activities are independent of the order chosen for obtaining the variability requirements and the catalog of services. There are typically three generic scenarios. The first one corresponds to the scenario where an organization only owns or has access to one or several catalogs of legacy services. Then the construction of the workflow is limited to the use of those services and the variability requirements of the SPL provided to the users are infered (extracted) from them. We refer to this situation as the *bottom-up*[1] scenario. This means that the feature model of ① is computed from the feature model(s) of ③. The extraction of feature models can be performed at the product level by building a feature model from existing products' specification (typically, a hierarchy of features *without* variability) or directly at the SPL level by considering feature models of different SPLs.

The second scenario, called the *top-down* scenario, consists of using feature models to perform domain analysis and scoping so that the variation to be supported in the production line is identified first. This is the specification of the valid combinations of features intended to be supported by the family of segmentation services. In such a scenario, variability requirements (①) is designed before the corresponding software assets to be reused (③) are implemented.

The third scenario, which can be qualified as *hybrid*, is as follows. The variability requirements are deduced from the catalog and then adapted to the needs of the user and, in the mean time, the user starts from scratch the requirements and confronts them to the catalog, adapting it if needed. This means than ① and ③ are built in parallel.

Depending on the scenario, the questions to be answered when reasoning are not the same but they rely on the same core operators.

6.2 COMPETING MULTIPLE SOFTWARE PRODUCT LINES

In order to meet the requirements determined above, we propose to use multiple SPLs. Informally, *a multiple SPL M_{SPL} is an SPL that manages a set of constituent SPLs $\{SPL_1, SPL_2, \ldots, SPL_n\}$ and its set of products is described by a feature model $FM_{M_{SPL}}$.*

At this stage, the semantics of a multiple SPL is left deliberately vague to support a variety of interpretations, i.e., there are still different ways of combining the constituent SPLs and their feature models. In particular, the informal definition does not characterize the combinations of features (i.e., configurations) supported by a multiple SPL M_{SPL} and allowed by its feature model $FM_{M_{SPL}}$.

For example, given a multiple SPL that manages three SPLs described by the three feature models FM_{supp_1}, FM_{supp_2} and FM_{supp_3} shown in Figure 6.2, a product developer may want to determine whether the following combination of features Medical Image , MRI, T1, Header and DICOM are allowed in the multiple SPL. On the one hand, we can observe that this combination of features does not correspond to any valid configuration of FM_{supp_1} (see Figure 6.2(a)), FM_{supp_2} (see Figure 6.2(b)) or FM_{supp_3} (see Figure 6.2(c)). On the other hand, we can argue that the combination of features can be realized by choosing features from different feature models, for example, by choosing features Medical Image, MRI, T1 in FM_{supp_1}, feature Header in FM_{supp_2}, and feature DICOM in FM_{supp_3}. In this case, it can be seen as a form of *compositional* multiple SPL, in which each part may correspond to several SPLs, built by different parties, which are then combined to form an

[1] We reuse the terminology presented in Section 2.1.3.

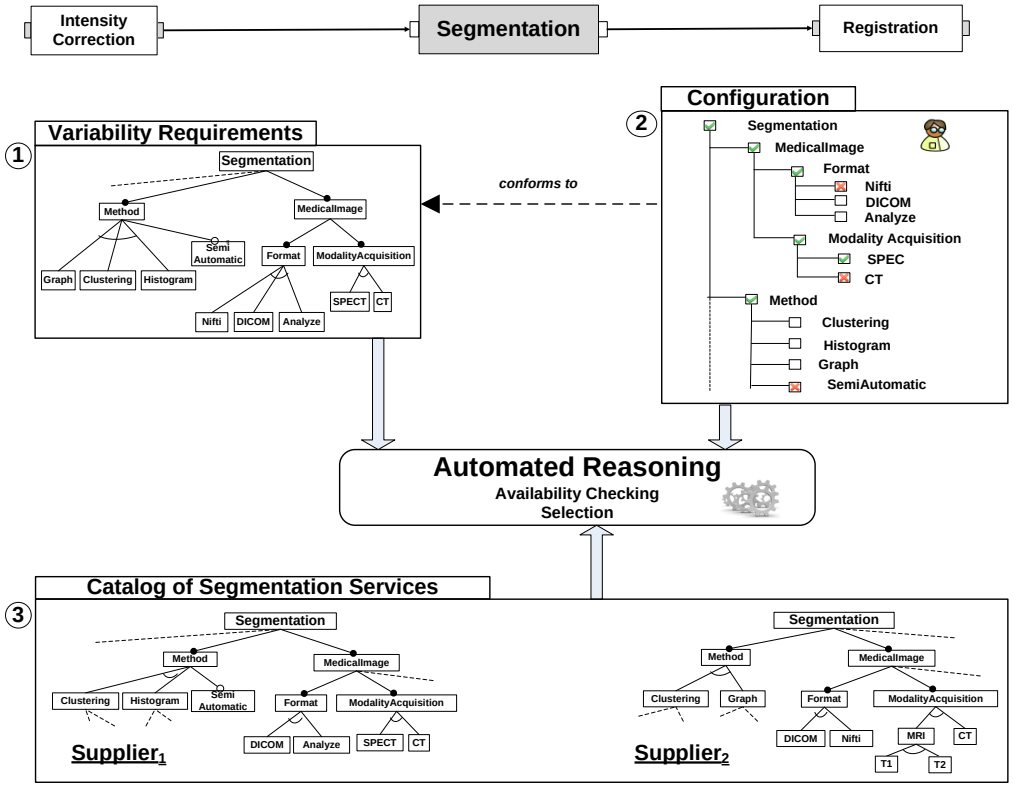
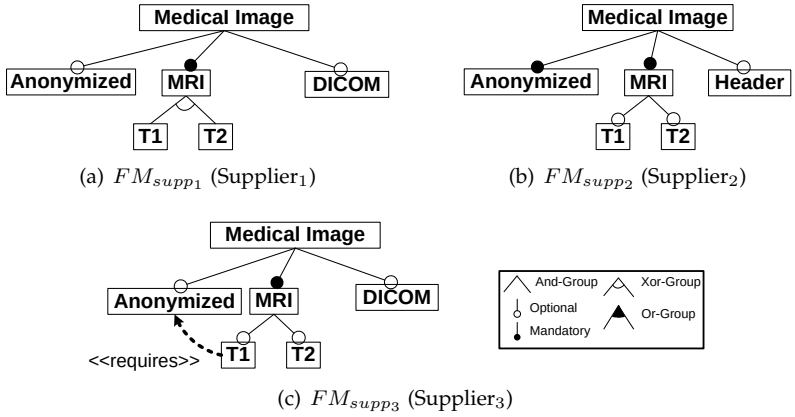


Figure 6.1: Handling multiple variability inputs (e.g., for segmentation services)

integrated software system. This form of multiple SPL raises important issues regarding its associated semantics, as when choosing one or another feature, one must also reason on the composition of the software artifacts represented by these features.

In this chapter, we focus on a specific class of multiple SPLs, i.e., *competing* multiple SPLs. In a competing multiple SPL, each constituent SPL describes a product that competes with products described in other constituent SPLs, for example, the two SPLs in Figure 8.3 provide competing Segmentation Services with different features. Each SPL in a competing multiple SPL defines a set of products offered by a competing *supplier* (e.g., a research group). The SPL of Supplier$_1$ is the only one to propose Segmentation Services that support Analyze format but cannot process MRI images while the SPL of Supplier$_2$ can process images. In a competing multiple SPL, each combination of features must correspond to an actual product of at least one SPL. A product is a combination of features in which all features are provided by one and only one supplier. Obviously, it is possible that for a given combination of features, more than one corresponding product exists. The configuration {Segmentation, Method, MedicalImage, Clustering, Format, ModalityAcquisition, DICOM, CT} corresponds to two services, one provided by Supplier$_1$ and the other by Supplier$_2$ (see Figure 8.3).

To formalize the concept of competing multiple SPLs, we define its semantics in terms



(a) FM_{supp_1} (Supplier$_1$)

(b) FM_{supp_2} (Supplier$_2$)

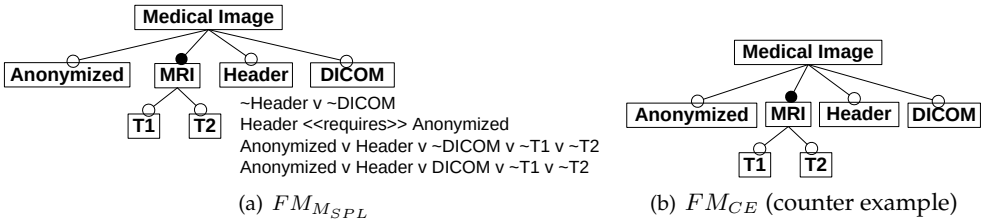
(c) FM_{supp_3} (Supplier$_3$)

Figure 6.2: Three feature models from different suppliers (adapted from [Hartmann et al. 2009])

of the set of products defined by its constituent SPLs, $SPL_1, SPL_2, \ldots, SPL_n$. FMs are used to describe the set of products of an SPL, and thus the semantics of a multiple SPL can be defined as a relationship between the FM of the multiple SPL, $FM_{M_{SPL}}$, and the FMs of the constituent SPLs, FM_1, FM_2, ..., FM_n. We formally define the semantics of competing multiple SPLs below.

Definition 12 (Competing Multiple SPL)**.** *Each product of a competing multiple SPL, M_{SPL}, is a product belonging to one of its constituent SPLs SPL_1, SPL_2, ..., or SPL_n. More precisely, any configuration of the feature model of M_{SPL}, $FM_{M_{SPL}}$, should correspond to at least one valid configuration of an FM describing the products of M_{SPL} constituent SPLs, that is, FM_1, FM_2, ..., FM_n. Formally: $\forall c \in [\![FM_{M_{SPL}}]\!] : c \in [\![FM_1]\!] \vee c \in [\![FM_2]\!] \vee \ldots \vee c \in [\![FM_n]\!]$*

The FM $FM_{M_{SPL}}$ of Figure 6.3(a) represents the sets of configuration of the competing multiple SPL that manages the set of SPLs represented by FMs FM_{supp_1}, FM_{supp_2} and FM_{supp_3} of figures 6.2(a), 6.2(b) and 6.2(c). This is not the case for the FM FM_{CE} of Figure 6.3(b). A counter-example is given by { Medical Image, MRI, Header, DICOM} which is a valid configuration of FM_{CE} but is not a valid configuration of either FM_{supp_1}, FM_{supp_2} or FM_{supp_3}.



(a) $FM_{M_{SPL}}$

(b) FM_{CE} (counter example)

Figure 6.3: FM of a competing multiple SPL

In a competing multiple SPL, some producers build individual, unique, and user-specific products that others do not. Therefore, it is useful to determine which products of a competing multiple SPL are unique.

Definition 13 (Product Uniqueness). *A product p of a competing multiple SPL M_{SPL} is unique if p belongs exclusively to either $SPL_1, SPL_2, \ldots,$ or SPL_n. Let p be a product of M_{SPL} described by a configuration c. p is unique if and only if $\exists i \in 1...n : \forall j \in 1...n, j \neq i \ \wedge \ c \in [\![FM_{M_{SPL}}]\!] \ \wedge \ c \in [\![FM_i]\!] \ \wedge \ c \notin [\![FM_j]\!]$. By extension, an SPL SPL_i of a competing multiple SPL M_{SPL} is unique if all products of SPL_i are unique: $\forall c \in [\![FM_i]\!], \forall j \in 1...n : j \neq i \wedge c \notin [\![FM_j]\!]$*

For example, SPL_1 of Figure 6.2(a) has two unique products: { Medical Image, MRI, T1} and { Medical Image, MRI, DICOM, T1} ; SPL_2 of Figure 6.2(b) has three unique products: { Medical Image, Anonymized, MRI, Header}, { Medical Image, Anonymized, MRI, Header, T1}, { Medical Image, Anonymized, MRI, Header, T2} ; SPL_3 of Figure 6.2(c) has four unique products: { Medical Image, MRI}, { Medical Image, MRI, DICOM}, { Medical Image, MRI, DICOM, Anonymized} and { Medical Image, Anonymized, MRI, DICOM, T1, T2}.

Similarly, we can identify common products in a competing multiple SPL, i.e., products that belong to every constituent SPL of a competing multiple SPL.

Definition 14 (Commonality of a Product). *A product p of a competing multiple SPL M_{SPL} is common if p belongs to $SPL_1, SPL_2, \ldots,$ and SPL_n. Let p a product of M_{SPL} described by a configuration c. p is common if and only if $c \in [\![FM_{M_{SPL}}]\!] \ \wedge \ \forall i \in 1...n, c \in [\![FM_i]\!]$*

In the example shown in Figure 6.2, there are two common products: { Medical Image, Anonymized, MRI, T1} and { Medical Image, Anonymized, MRI, T2}.

The following defines the number of products defined by a multiple SPL.

Definition 15 (Number of products). *The number of products of a multiple SPL M_{SPL} is denoted $|M_{SPL}|$ and is defined as the cardinality of its set of products, i.e., $|M_{SPL}| = |[\![FM_{M_{SPL}}]\!]|$.*

In a competing multiple SPL, $|[\![FM_{M_{SPL}}]\!]| \leq |[\![FM_1]\!]| + |[\![FM_2]\!]| + \ldots + |[\![FM_n]\!]|$, that is, the Inclusion-Exclusion principle of combinatorial mathematics applies. For example, $|[\![FM_{M_{SPL}}]\!]| = 18 \leq |[\![FM_{supp_1}]\!]| + |[\![FM_{supp_2}]\!]| + |[\![FM_{supp_3}]\!]| = 28$.

The reader can verify that the number of valid configurations of FM $FM_{M_{SPL}}$ depicted in Figure 6.3(a) is 18 whereas the number of valid configurations of FM FM_{CE} (cf. Figure 6.3(b)) is 32.

6.3 Merging Techniques to Manage Competing Multiple SPLs

The feature model of a competing multiple SPL must precisely characterize a set of valid configurations allowed by a set of feature models, no more, no less. Ensuring that a multiple SPL meets this requirements makes manual development and management of a multiple SPL difficult and error-prone. Let us consider Figure 6.3(b). FM_{CE} is an example of a feature model with an invalid set of configurations (w.r.t. the semantics of a competing multiple SPL). FM_{CE} is presented[2] in [Hartmann et al. 2009] and is a typical example of a feature model that could have been manually created. As we will see in Section 6.4,

[2]We just change the features' names.

this is the kind of feature model that a user visualizes and manipulates using reference-based techniques. Even on this small example, the error is quite large, with 43% of the FM_{CE} configurations that are not valid in FM_{supp_1}, FM_{supp_2}, or FM_{supp_3}. This is not acceptable for representing a competing multiple SPL.

Moreover a lot of cross-tree constraints between features may exist (e.g., in Figure 6.3(a), features DICOM and Header are mutually exclusive). It is particularly difficult to infer the constraints that relate features Anonymized, Header, DICOM, T1 and T2. A manual design of a feature model may lead to an under-approximation or over-approximation of the set of valid configurations (as an example, see Figure 6.3(b)). Therefore *automation* support is required for managing competing multiple SPLs and realizing the semantics previously defined.

Assumption. It is likely that a set of competing SPLs exhibits a large proportion of *similar* features – products offered by the competing SPLs target the same domain/market and thus they are likely to have similar characteristics. In this work, we assume that two features are similar if their names are strictly equal. The same working assumption has already been used by other SPL researchers, for example, to compare two feature models [Thüm et al. 2009], to superimpose feature structure trees [Apel et al. 2008] or to merge feature models [Alves et al. 2006, Segura et al. 2008, Schobbens et al. 2007]. The name of the feature is often the only discriminant information we can automatically exploit in a feature model. In certain cases, though, a different vocabulary may have been used during the elaboration of the different feature models ; the feature models may have different level of granularity (e.g., much more details in one of the feature models) ; the features may refer to different concepts. In this case, a strict equality based on names may be too restrictive. Basic edits to feature models (such as the renaming of features) can be performed or more advanced techniques to align feature models (as proposed in the next chapter) can be used. We will further discuss the alignment problem in the concluding part of the document. In our case study, the alignment effort is currently not significant since suppliers rely on a common ontology [Temal et al. 2008] and feature models are views on the ontology [Fagereng Johansen et al. 2010].

In the rest of this section, we will show how merge operators dedicated to feature models can provide automated support for managing competing multiple SPLs.

Using the Merge Operators. In Figure 6.4, we consider a representative application scenario of the management of competing multiple SPLs. A scientist designs a medical imaging processing chain and should obtain, at the end, a composition of several services that fit his/her requirements and that are provided by (different) suppliers in a catalog of services. We show how the merge operators defined above can be used to realize such a scenario.

6.3.1 Building a Catalog of Services

We first focus on the construction of a catalog of services (see Ⓐ in Figure 6.4). Several suppliers (e.g., research teams around the world) provide access to a set of (legacy) services implementing diverse medical imaging algorithms. The purpose is to provide a catalog of services describing the features of the services offered to scientists.

Modeling Variability of Services. The first step is to augment the service description, in-

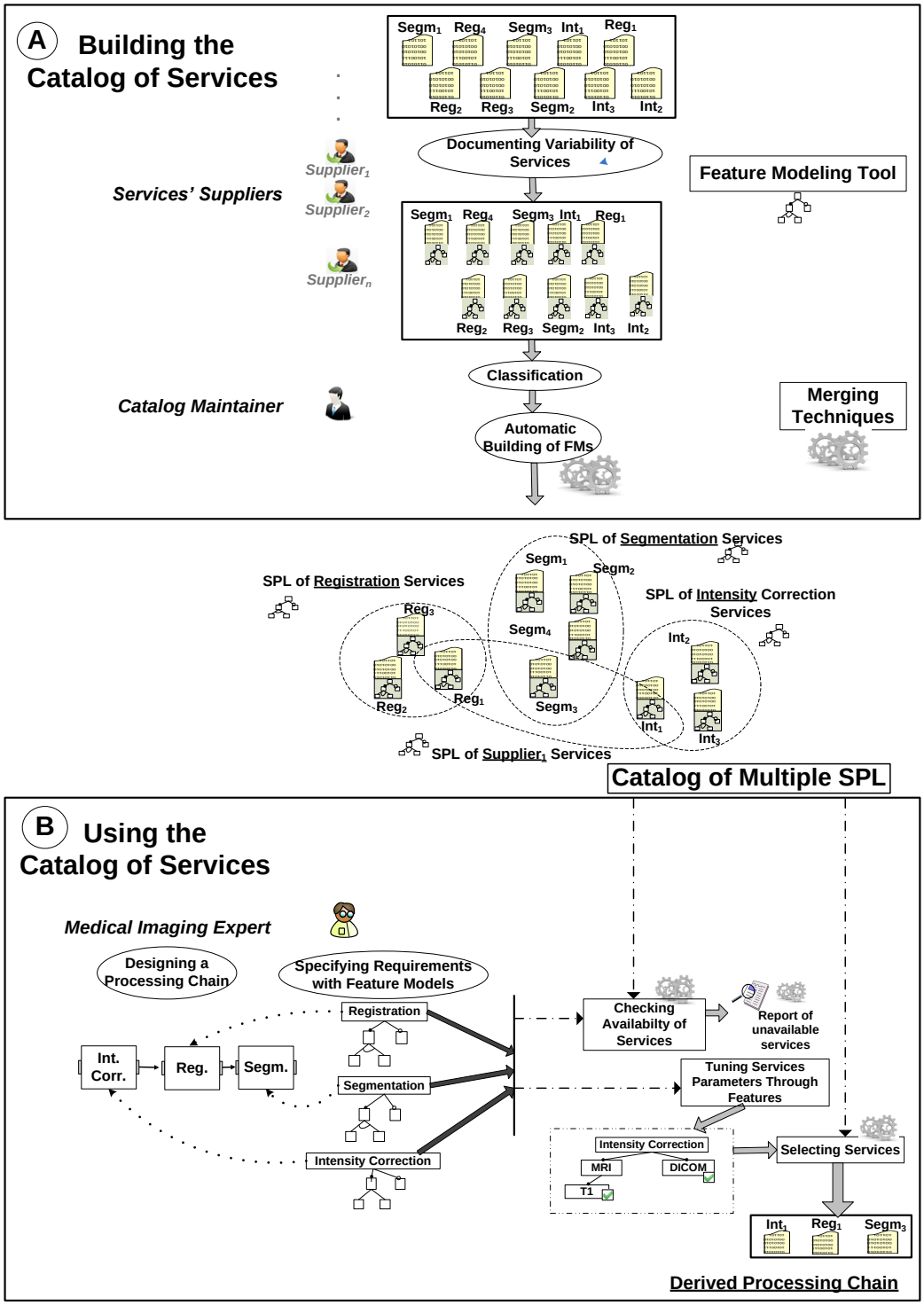


Figure 6.4: An application scenario

cluding general information about the supplier providing the service, the kind of algorithm implemented by the service. In addition, service suppliers document the variability information of each service, i.e., information on the service's ability to extend, change, customize or configure its behavior (e.g., support of network protocols or medical imaging formats, QoS properties provided in a particular context). The variability information is described through the set of combinations of features supported by the service and represented as a feature model. A service may exhibit no variability, e.g, a service may support only one medical image format and one network protocol. In this case, the service's features are represented as a feature model in which all features are mandatory.

Classification of Services. A catalog of services assists users in finding the most appropriate service according to their needs. Due to the large number of service features, there are various way to classify services. In Figure 6.4, the classification is based on the kind of algorithm (intensity correction, registration or segmentation) implemented by services. Three sets of services are grouped together and form three SPLs. Another classification criteria illustrated in Figure 6.4 is the suppliers names: $Segm_3$, Int_1 and Reg_1 form a fourth SPL in which all services are provided by Supplier$_1$.

Automatic building of feature models. In this activity, developers identify services that are to be managed through a unique SPL. Such an SPL should preserve the combinations of features provided by each service. This activity involves building a competing multiple SPL which manages a set of services corresponding to the classification that have been retained. For example, let us consider a competing multiple SPL M_{SPL} that manages a set of services $supp_1$, $supp_2$ and $supp_3$.

When considering feature models of Figure 6.2, we obtain $FM_{M_{SPL}}$ of Figure 6.3(a) by computing

$$FM_{M_{SPL}} = FM_{supp_1} \oplus_{\cup_s} FM_{supp_2} \oplus_{\cup_s} FM_{supp_3}$$

$FM_{M_{SPL}}$ of Figure 6.3(a) is synthesized using the following Boolean formula:

$$\phi_{result} = (\phi_{FM_{supp_1}} \wedge \neg Header) \vee (\phi_{FM_{supp_2}} \wedge \neg DICOM) \vee (\phi_{FM_{supp_3}} \wedge \neg Header)$$

Similarly, when services exhibit no variability, constructing an SPL from services description can be done by applying the merge operator in strict union mode. In this case, we represent each service description as a feature model with no variability.

Reasoning about the Catalog. Once the feature model has been constructed, reasoning operations can be performed. For example, a user may want to determine the number of products of a competing multiple SPL. According to Definition 15, it is not correct to compute the sum of number of products of each SPL. The correct way is to use the previous Boolean formula and count the number of satisfying variable assignments of ϕ_{result}.

Another interesting operation for users is to determine which services exhibit the same variability information. It involves determining the set of common products of a competing multiple SPL, corresponding to the set of configurations represented by FM_{common}. It can be computed as follows: $FM_1 \oplus_{\cap} FM_2 \oplus_{\cap} \ldots \oplus_{\cap} FM_n = FM_{common}$. Note that the set of common products can be rendered as a feature diagram.

Some suppliers propose services with unique combination of features. According to Definition 13, a unique product belongs necessary to only one SPL. The unique products

of an SPL_i corresponds to the set of configurations of FM_{unique_i} where

$$FM_{unique_i} = FM_i \oplus_{\setminus} (FM_1 \oplus_{\cup_s} \dots FM_{i-1} \oplus_{\cup_s} FM_{i+1} \oplus_{\cup_s} \dots \oplus_{\cup_s} FM_n)$$

The set of unique products can then be deduced by computing

$$FM_{unique_1} \oplus_{\cup_s} FM_{unique_2} \oplus_{\cup_s} \dots FM_{unique_n}$$

Evolution of the Catalog. A competing multiple SPL M_{SPL} can evolve over time. For example, the set of services provided by a new supplier can be added to a catalog of services. Instead of specifying from scratch the variability of a new service, we can rely on names and hierarchies already used in feature models from the current catalog services. This may notably reduce the alignment problem mentioned above and facilitate the elaboration of a new feature model trough the reuse of an existing one. When an SPL SPL_{n+1} is added into a competing multiple SPL M_{SPL}, we should obtain a new competing multiple SPL $M_{SPL'}$ such that

$$[\![FM_{n+1}]\!] \cup [\![FM_{M_{SPL}}]\!] = [\![FM_{M_{SPL'}}]\!]$$

Since the merge operator in strict union mode is commutative w.r.t. the set of configurations, the evolution of a competing multiple SPL can be accomplished in an incremental manner and there is no need to re-compute the merge of all feature models managed by M_{SPL}.

Interestingly, when FM_{n+1} is a *specialization* (resp. *generalization*) of $FM_{M_{SPL}}$ (see Definition 10, page 53), then $FM_{M_{SPL'}} = FM_{M_{SPL}}$ (resp. $FM_{M_{SPL'}} = FM_{n+1}$).

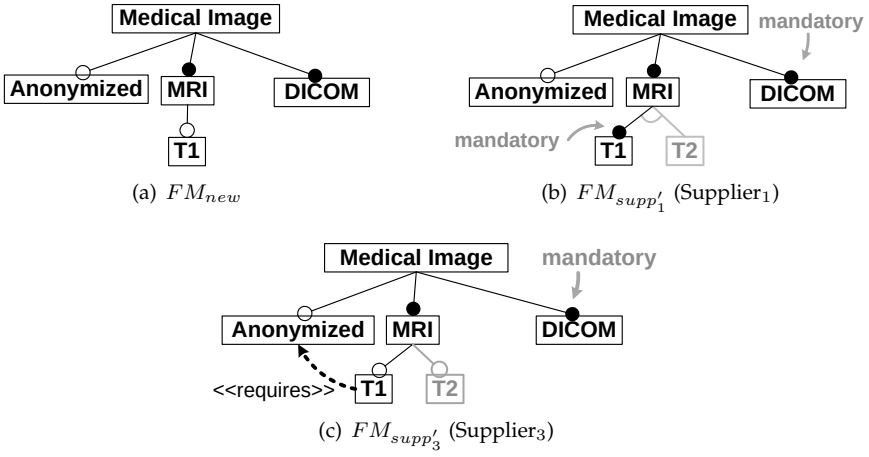
Proof. If FM_{n+1} is a *specialization* of $FM_{M_{SPL}}$, then $[\![FM_{n+1}]\!] \subset [\![FM_{M_{SPL}}]\!]$ and in particular we have $[\![FM_{n+1}]\!] \cup [\![FM_{M_{SPL}}]\!] = [\![FM_{M_{SPL}}]\!]$ according to set theory.

6.3.2 Using the Catalog of Services

In our scenario, scientists usually compose several services to process medical imaging data (see Ⓑ in Figure 6.4) and define an expected set of features which corresponds to the application requirements. The catalog of services offers to scientists a set of multiple competing SPL so that services can be selected and reused. We now explain how merging techniques can be used to derive a composition of fully parameterized services of the catalog.

Checking Availability of Services. For each service of the processing chain, a scientist specifies the desired feature combinations for a service. There is a need to determine which services of the catalog fit with the requirements of the scientist. For example, a scientist may want to determine which suppliers from among Supplier_1, Supplier_2 and Supplier_3 (see Figure 6.2) can provide a subset of the configurations represented by the feature model FM_{new} of Figure 6.5(a). In this case, the merge operator in intersection mode is applied.

For example, the computation of $FM_{new} \oplus_{\cap} FM_2$ gives the empty set so that we know Supplier_2 cannot provide any service. On the contrary, Supplier_1 can provide two services represented by the two following configurations: { Medical Image, MRI, DICOM, T1, Anonymized} and { Medical Image, MRI, DICOM, T1}. Supplier_3 can provide three services represented by the following configurations: { Medical Image, MRI, DICOM, T1, Anonymized}, { Medical Image, MRI, DICOM, Anonymized}, { Medical Image, MRI, DICOM}. As



(a) FM_{new} (b) $FM_{supp'_1}$ (Supplier$_1$)

(c) $FM_{supp'_3}$ (Supplier$_3$)

Figure 6.5: Checking availability: a new feature model and suppliers' feature model updates of Figure 6.2

a result, the competing multiple SPL is an SPL represented by FM_{new} and which manages the SPLs provided by Supplier$_1$ and Supplier$_3$. The set of products provided by Supplier$_1$ is described by $FM_{supp'_1} = FM_{new} \oplus_{\cap} FM_{supp_1}$ of Figure 6.5(b) in which the feature T2 is no longer proposed while DICOM and T1 are now mandatory features. Similarly, the set of services provided by Supplier$_3$ is described by $FM_{supp'_3} = FM_{new} \oplus_{\cap} FM_{supp_3}$ of Figure 6.5(c).

It is then necessary to determine whether the set of suppliers is able to provide *all* products of FM_{new}. We can compute the union of each set of products that belong to FM_{new} *and* that are provided by each supplier. In this case, the union set should be equal to the set of products of FM_{new} and the relation (1) should hold (see below). Using properties of the merge operators, the relation can be rewritten – see relation (2). We know that $FM_{supp_1} \oplus_{\cup_s} FM_{supp_2} \oplus_{\cup_s} FM_{supp_3} = FM_{M_{SPL}}$ so that the relation (3) holds:

$$FM_{new} = (FM_{new} \oplus_{\cap} FM_{supp_1}) \oplus_{\cup_s} (FM_{new} \oplus_{\cap} FM_{supp_2}) \oplus_{\cup_s} (FM_{new} \oplus_{\cap} FM_{supp_3}) \quad (6.1)$$

$$= FM_{new} \oplus_{\cap} (FM_{supp_1} \oplus_{\cup_s} FM_{supp_2} \oplus_{\cup_s} FM_{supp_3}) \quad (6.2)$$

$$= FM_{new} \oplus_{\cap} FM_{M_{SPL}} \quad (6.3)$$

Considering the relation (3), we know that

$$[\![FM_{new}]\!] = [\![FM_{new}]\!] \cap [\![FM_{M_{SPL}}]\!]$$

According to set theory, this is equivalent to

$$[\![FM_{new}]\!] \subseteq [\![FM_{M_{SPL}}]\!]$$

As a result, when $FM_{M_{SPL}}$is computed (see Figure 6.3(a)), ensuring that the set of products of FM_{new} is provided by all suppliers is equivalent to determine whether FM_{new}

is a *specialization* or a *refactoring* of $FM_{M_{SPL}}$. This can dramatically reduce the amount of time needed when the number of suppliers to consider is large, because there is no need to do pair-wise compatibility checks between each supplier and the new SPL specification.

Assisting Users. It may happen that some combinations of features are not supported by any SPL of the catalog of services. For example, when FM_{new} is a *generalization* of $FM_{supp_1} \oplus_{\cup_s} FM_{supp_2} \oplus_{\cup_s} FM_{supp_3}$, it means that some configurations of FM_{new} are not expressed by FM_{supp_1}, FM_{supp_2} nor FM_{supp_3}. The computation of missing configurations may assist the scientist in understanding which services' features are not supported in the catalog of services. $FM_{missing}$ represents the set of missing configurations and is obtained using the merge *diff* operator (see Section 5.4):

$$\begin{aligned} FM_{missing} &= FM_{new} \oplus_{\setminus} (FM_{supp_1} \oplus_{\cup_s} FM_{supp_2} \oplus_{\cup_s} FM_{supp_3}) \\ &= FM_{new} \oplus_{\setminus} FM_{M_{SPL}} \end{aligned}$$

Determining if FM_{new} is a *specialization* of $FM_{M_{SPL}}$ can be done by reusing the algorithm presented in [Thüm et al. 2009] or by using the merge in diff mode (see Lemma 1).

Lemma 1 (Merge Diff and Specialization/Refactoring)**.** *Let f and g be feature models. f is a specialization or a refactoring of g if* $(f \oplus_{\setminus} g)$ *has no valid configurations*

Proof. According to set theory, $[\![f]\!] \subseteq [\![g]\!]$ is equivalent to $[\![f]\!] \setminus [\![g]\!] = \emptyset$.

Configuration process. In application engineering, scientists require facilities for selecting/deselecting features. For example, when a scientist selects the feature Header of $FM_{M_{SPL}}$(see Figure 6.3(a)), the Boolean formula $\phi_{result} \wedge Header$ can be used to check the consistency of the feature selection or to deduce the possible values (i.e., selected/deselected) for features that have not been previously configured by the user. This is equivalent to setting the variable $Header$ to true in each Boolean formula $\phi_{FM_{supp_1}}$, $\phi_{FM_{supp_2}}$ and $\phi_{FM_{supp_3}}$ (corresponding resp. to FM_{supp_1}, FM_{supp_2} and FM_{supp_3}) and then computing the merge in strict union mode:

$$\begin{aligned} Header \wedge \phi_{result} = Header \wedge ((\phi_{FM_{supp_1}} \wedge \neg Header) \vee (\phi_{FM_{supp_2}} \wedge \neg DICOM) \\ \vee (\phi_{FM_{supp_3}} \wedge \neg Header)) \\ = ((\phi_{FM_{supp_1}} \wedge Header) \wedge \neg Header) \vee ((\phi_{FM_{supp_2}} \wedge Header) \wedge \\ \neg DICOM) \vee ((\phi_{FM_{supp_3}} \wedge Header) \wedge \neg Header) \\ = ((\phi_{FM_{supp_2}} \wedge Header) \wedge \neg DICOM) \end{aligned}$$

In the example, the selection of Header allows one to infer that $Supplier_2$ is the only supplier able to provide products that exhibit feature Header. An important property is that $FM_{M_{SPL}}$ can be used *independently* during configuration process, i.e., without considering $FM_1, FM_2, \ldots, FM_n$.

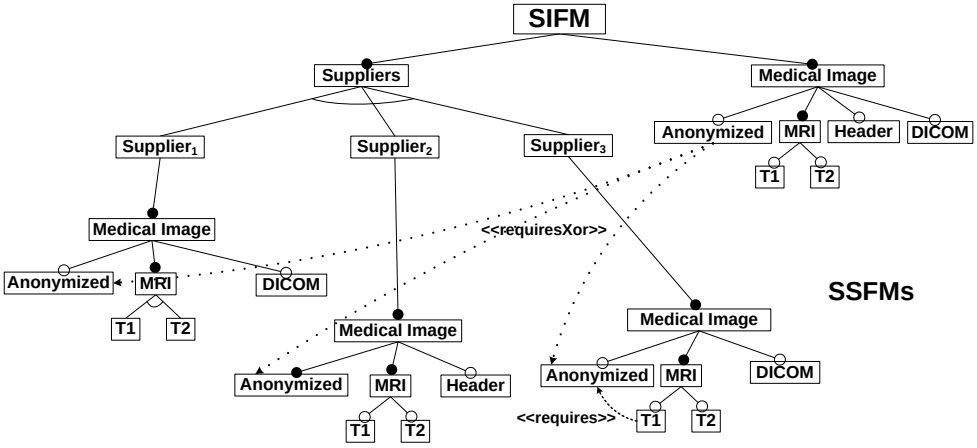
Selecting Services. The last step of the scenario is to obtain a composition of fully parameterized services. This activity involves mapping scientists' requirements (i.e., a set

of selected features) to the features offered by services of the catalog. It is possible that more than one service matches the requirements. Scientists then have to choose the most appropriate service. In the scenario illustrated in Figure 6.4, the selection criteria relies upon suppliers that provide the service: the services Int_1, Reg_1 and $Segm_3$ all belong to the same supplier (Supplier$_1$).

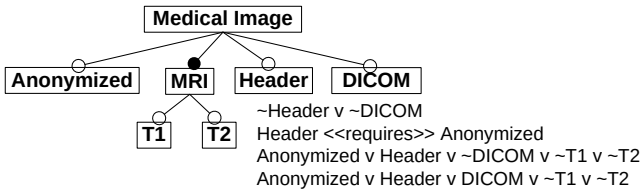
6.4 COMPARISON WITH THE REFERENCE-BASED TECHNIQUES

The goal of the merging techniques is to manage multiple feature models. Instead of *merging* multiple feature models, another conceivable technique, yet drastically different, is to *reference* those multiple feature models. Such reference-based techniques can then be applied to the application scenario (see Figure 6.4, Section 6.3). We now compare the two techniques, using the work from Hartmann et al. [Hartmann et al. 2009] for the reference-based ones.

This approach introduces the *Supplier Independent Feature Model* (SIFM) in order to select products among the set of products described by several *Supplier Specific Feature Models* (SSFM). Intuitively, the SIFM references several SSFMs thanks to constraints between features. In Figure 6.6(a), we show how the SIFM and SSFMs are built considering feature models of suppliers Supplier$_1$, Supplier$_2$ and Supplier$_3$ of Figure 6.2(a), 6.2(b) and 6.2(c).



(a) Reference-based technique (adapted from [Hartmann et al. 2009])



(b) Merging technique

Figure 6.6: Reference-based and merging techniques

The overall idea is that any feature of SIFM, say feature F, is then related to the features F of SSFMs using cross-tree constraints. Additional constraints between SSFMs are expressed so that features F of SSFMs cannot be selected at the same time. By defining such constraints between SIFM and SSFMs, Hartman et al. allow users to build a multiple SPL thanks to several suppliers' SPLs. The SIFM is designed as follows. First, the root feature SIFM has two subfeatures: the feature Suppliers and the common root feature of SSFMs (e.g., in Figure 6.6(a), the common root feature corresponds to Medical Image). Then, the feature Suppliers contains as many subfeatures as there are suppliers and those features are mutually exclusive (only one supplier must be selected). The SIFM contains the "super-set of the features" from all the suppliers and constraints are specified to inter relate features of SIFM and SSFMs. In addition, cross-tree constraints between features are specified such that each child feature F of Medical Image is related to the corresponding features F in each appropriate SSFM thanks to a constraint *requiresXor*. For example, in Figure 6.6(a), cross-tree constraints prevent the selection of more than one Anonymized feature of SSFMs. Finally, a feature Medical Image of an SSFM is a mandatory sub-feature[3] of either $Supplier_1$, $Supplier_2$ or $Supplier_3$.

According to the authors, the approach proposed has the advantage to be realizable by current feature modeling tools and techniques. We now compare our approach with their approach.

Semantics. The definition of a competing multiple SPL corresponds to the supplier independent problem informally described in [Hartmann et al. 2009].The semantics presented in Section 6.2 aims at providing a sound framework for the management of multiple SPL and opens new perspectives to handle new properties (e.g., uniqueness of a product) not defined in [Hartmann et al. 2009].

Complexity. The approach of Hartmann et al. leads to reasoning on a large set of features (i.e., all the features of SIFM and SSFMs') related by a large number of constraints. For example, when their approach is applied on the feature models of Figure 6.3, the number of variables of SIFM and SSFMs is equal to 12 + 6 + 6 + 6 = 30 features. The number of variables to be generated may become an issue in terms of computational or space complexity and hinder some automated analysis operations of feature models (see Chapter 3). For example, if the number of input feature models is equal to 200, each feature model including 200 features, there is need to consider $200 * 200 = 40000$ variables, together with a large number of cross-tree constraints. In the application scenario of Figure 6.4, this limitation have an impact on the construction *and* the use of the catalog of services, especially when a large number of services with a large number of features exist.

User perspective. As noticed in [Deelstra et al. 2005], one of the problems identified by organizations is the complexity of the product family in terms of number of variation points. Merging features reduce the number of features to consider and produce more compact feature models. As a result, the amount of time and effort needed during the configuration process can be reduced. For example, let us consider 10 input feature models with 100 common features per feature models while 20% of the total features are not common

[3] As in [Hartmann et al. 2009], the mandatory relation can be similarly expressed by a bi-implication between a feature Medical Image of an SSFM and one of the child features of Suppliers

(i.e., do not match). In this case, $(10 * 100) + ((10 * 100) * 0.2) = 1200$ features and 11 feature models have to be considered with the referenced-based technique against only one feature model and 300 features with the merging technique. For 10 feature models, 60 common features per feature models and with the rather low percentage 50% of non common features, there would still be $(10 * 60) + ((10 * 60) * 0.5) = 900$ features on the reference based side (against 360 features).

Stakeholders use feature model of a competing multiple SPL to design new SPLs and/or configure products. In such engineering activities, understanding the feature model is crucial. From a user perspective, the super-set of all supplier features contained in SIFM over approximates the sets of configurations and hides some constraints to the user (see Figure 6.6(a)). For example, users cannot understand that features Header and DICOM are mutually exclusive until considering constraints between SIFM and SSFMs. When SIFM is solely considered, a large amount of configurations (43%) does not correspond to any product. In our approach, the set of configurations represented by the merged feature model can be directly used. As previously, such limitations have an impact on the application scenario of Figure 6.4, i.e., on the catalog maintainer and the medical imaging expert activities. The catalog maintainer has to classify and maintain feature models: it is a cumbersome and error-prone activity due to the difficulty to read and understand the feature models generated by the reference-based techniques. For the same reasons, the medical imaging expert encounters difficulties when tuning services parameters of the catalog (i.e., when configuring feature models).

Incrementality and Modular Reasoning. In our approach, the feature model representing the set of configurations of a multiple SPL is independent from the other supplier feature models. This is not the case when using SIFM since when a feature is selected/deselected, reasoning tools have to consider every SSFM and all constraints before updating SIFM. Similarly, determining if a subset of products can be provided by suppliers cannot be done without considering all SSFMs. Moreover the evolution of the multiple SPL (e.g, when a new supplier is considered) does not imply to update both SIFM and cross tree constraints. As previously, such limitations have an impact on the application scenario of Figure 6.4, i.e., on the maintenance and evolution of the catalog of services as well as the selection from among sets of catalog services.

Seven

Decomposing Feature Models

This chapter shares material with the ASE'11 paper "Slicing Feature Models" [Acher et al. 2011e] and the AOSD'12 paper "Separation of Concerns in Feature Modeling: Support and Applications" [Acher et al. 2012] (currently under review)

.

The complexity of problems or systems can be greatly reduced and better managed when broken down into parts that are easier to conceive, understand, program, and maintain. "Divide and Conquer" is certainly the best illustration of this principle. It consists in recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

In the previous chapter, we have shown how sub-problems (i.e., feature models) can be combined. As important is the ability to *decompose* a problem (i.e., a feature model).

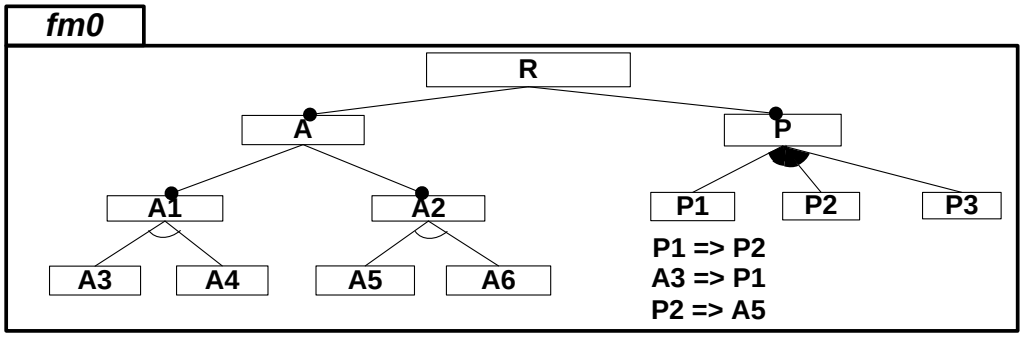
Various decomposition mechanisms have been developed in the software engineering community, for example, for breaking a large system down into progressively smaller routines, classes, objects, components or services. Though a multitude of approaches have been proposed for decomposing artifacts of various natures (textual documents, code, components, models, aspects, etc.), we should consider the specificities of the formalism of feature models.

We first wonder *what does decomposition mean in the context of feature models* (Section 7.1) . We show that a syntactical extraction has limitations and is thus inadequate. We present a *slicing* technique that produces semantically meaningful decompositions of feature models, given an arbitrary set of features considered to be pertinent by an SPL practitioner. We discuss and define its semantics in terms of set of configurations and hierarchy (Section 7.2). We show how to *automatically realize* the decomposition and we describe some interesting properties of the algorithm (Section 7.3).

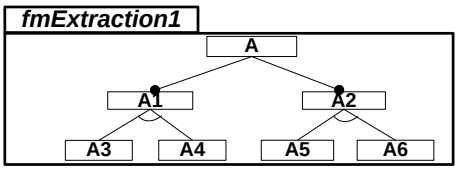
In Section 7.4, we illustrate *how decomposition techniques can be used to manage multiple feature models*. In particular, we show how the slicing operator can assist in tedious and error prone tasks such as automated correction of feature models' anomalies, update of feature model views, reconciliation of feature models or reasoning about properties of inter-related feature models.

7.1 Motivation and Principles

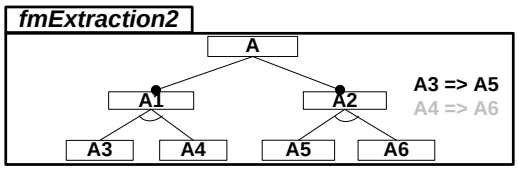
Let us consider the feature model shown in Figure 7.1. Though the feature model is relatively modest in size (12 features and 8 valid configurations) understanding the relations between features is not straightforward for a human due to the presence of propositional



(a) input feature model



(b) basic extraction



(c) semantics aware extraction

Figure 7.1: input feature model

constraints. It can be much worse for larger feature models (see, e.g., Figure 7.5 in page 95) or for large-scale feature models with hundreds or thousands of features that have been reported in the literature (see Section 4.2.3).

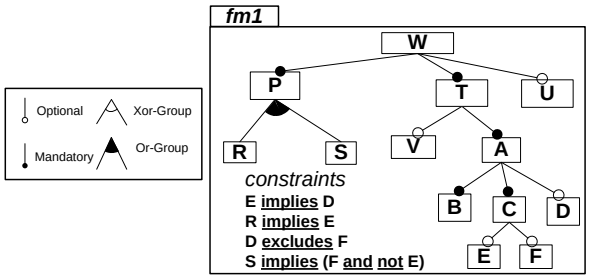
Managing a large number of features is obviously a problem per se for an SPL practitioner. It becomes even more complex when the legal combination of features are governed by many and often complex rules. For example, [Lotufo et al. 2010, Berger et al. 2010] found considerably many constraints involving more than one feature, with extreme cases of constraints containing up to 56 features in the Linux kernel feature model.

It is thus of crucial importance to be able to simplify and automate activities of SPL practitioners related to the understanding, elaboration, evolution and analysis of (large) feature models. Dividing feature models into localized and separated parts seem particularly well-suited solution to the problem, because SPL practitioners can focus their attention on one part at a time.

7.1.1 Why a basic extraction is not sufficient?

We consider the feature model $fm0$ of Figure 7.1(a). We want to decompose $fm0$ into separated feature models, for example, we want to focus on features A, A1, A2, A3, A4, A5, A6. It is tempting to "copy" the sub-tree rooted at feature A (see $fmExtraction1$ in Figure 7.1(b)).

The extraction is purely syntactical. The hierarchy and the variability information modeled in $fmExtraction1$ are restituted as in the original feature model $fm0$. In particular, as there is no cross-tree constraint in $fm0$ that directly relates features A, A1, A2, A3, A4, A5, A6, there is no cross-tree constraint in $fmExtraction1$. This basic strategy has an important limitation: It does not correctly restitute the legal combination of features as



(a) $fm1$, input feature model

$$[[fm1]] = \{$$
$$\{A, B, C, D, E, P, R, T, U, W\},$$
$$\{A, B, C, F, P, S, T, U, W\},$$
$$\{A, B, C, D, E, P, R, T, W\},$$
$$\{A, B, C, F, P, S, T, V, W\},$$
$$\{A, B, C, F, P, S, T, U, V, W\},$$
$$\{A, B, C, F, P, S, T, W\},$$
$$\{A, B, C, D, E, P, R, T, V, W\},$$
$$\{A, B, C, D, E, P, R, T, U, V, W\}$$
$$\}$$

(b) corresponding set of configurations

Figure 7.2: feature model and its set of configurations

originally expressed in $fm0$. For example, selecting the features { A, A1, A2, A3, A6 } is permitted by $fmExtraction1$ whereas it is not the case in $fm0$.

Intuitively, the basic extraction fails to infer the transitivity of implications (e.g., $A3 \Rightarrow A5$ or $A4 \Rightarrow A6$). A more accurate decomposition is shown in Figure 7.1(c) (see $fmExtraction2$): here selecting the features { A, A1, A2, A3, A6 } is not allowed.

Another important limit of the basic extraction technique is that features should belong to the same sub-tree. We do not want to restrict as such the decomposition technique.

To summarize, we need a more reliable and generic technique.

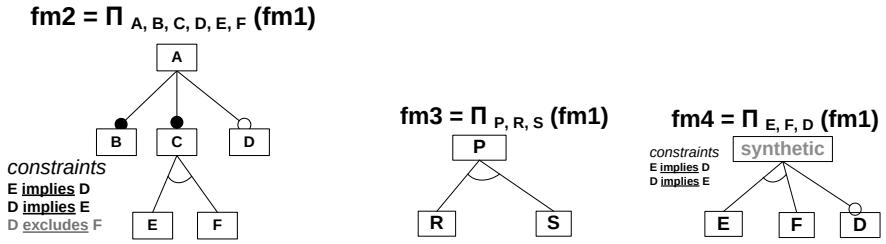
7.1.2 Slicing Technique

We propose an automated technique, called slicing, that produces a feature model that contains only a relevant subset of features.

The overall idea behind feature model slicing is similar to program slicing [Weiser 1981]. Program slicing has been successfully applied in computer programming to eliminate all parts from the program that are not currently of interest to the programmer. It aims at simplifying or abstracting programs by focusing on selected aspects of semantics. It has several practical applications in program understanding, maintenance, debugging, testing, differencing, specialization, reuse, and merging. Program slicing techniques proceed in two steps: the subset of elements of interest (e.g., a set of variables of interest and a program location), called the slicing *criterion*, is first identified ; then, a *slice* (e.g., a subset of the source code) is computed. In the context of feature models, we define the slicing criterion as a set of features considered to be pertinent by an SPL practitioner while the slice is a new feature model.

Examples of Slicing. In the rest of this section, we use the feature model shown in Figure 7.2(a) to illustrate the semantic properties of the slice operator.

A first example is given in Figure 7.3(a) where the slicing criterion corresponds to the set of features A, B, C, D, E and F. The hierarchy of fm_2 does not alter the structure (i.e., parent-child relationships) of the original feature model fm_1. It corresponds to a subtree of the tree of fm_1 whose root is feature A. The valid configurations characterized by fm_2 corresponds to the valid configurations of the original feature model fm_1, when looking



(a) [[[fm2]] = {{A, B, C, D, E},{A, B, C, F}} (b) [[[fm3]] = {{P, S}, {P, R}} (c) [[[fm4]]={{F}, {D, E}}

Figure 7.3: Example of slice operations applied on the feature model of Figure. 7.2(a).

only at the specific features of the criterion. It can be seen as a *projection* of the relational algebra on $[\![fm1]\!]$ (see Figure 7.2(b)) when the features not included in the criterion (W, P, ..., U) are discarded. The variability of fm_2 is then set to accurately represent $[\![fm_2]\!]$. We can observe that:

- features E and F form an Xor-group in fm_2 whereas they are optional features in fm_1. The reason is that features E and F *are* mutually exclusive in fm_1 but it is not restituted as such ;
- the constraint D implies E has been added to fm_2. The reason is that though the constraint is not part of the original feature model, it is logically entailed by fm_1 ;
- the constraint D excludes F is not added to fm_2 since the constraint is redundant (i.e., does not alter $[\![fm2]\!]$). As a result, the order in which constraints are added to fm_2 matters, i.e., different sets of propositional constraints can be added with a logically equivalent result.

A second example is shown in Figure 7.3(b). The resulting hierarchy of fm_3 preserves the structure of fm_1 and features R and S form an Xor-group whereas they originally form an Or-group in fm_1. $[\![fm_3]\!]$ is equal to the projection onto features R and S of $[\![fm_1]\!]$

The third example (see Figure 7.3(c)) is less straightforward as the slicing criteria involves features from different locations of the original feature model fm_1. The feature hierarchy tries to preserve as much as possible the original structure. An important remark is that, given the set of configurations of fm_4, neither feature D, E, nor F can be the root feature. Indeed, a root feature is by definition always part of any set of configuration and none of the features satisfy this condition (see $[\![fm_4]\!]$). As a result, we are obliged to add a synthetic root.

7.2 SEMANTICS

We define slicing as a unary operation on feature model, denoted $\Pi_{\mathcal{F}_{slice}}\ (FM)$ where $\mathcal{F}_{slice} = \{ft_1, ft_2, ..., ft_n\} \subseteq \mathcal{F}$ is a set of features.

Definition 16 (Slicing Properties)**.** *The result of the slicing operation is a new feature model, FM_{slice}, such that:*

- ***configuration semantics***
  $[\![FM_{slice}]\!] = \{\ x \in [\![FM]\!] \mid x\ \cap\ \mathcal{F}_{slice}\ \}$ *(called the projected set of configurations);*

- ***hierarchy***
  $G_{slice} = (\mathcal{F}_{FM_{slice}}, E_{slice})$ *with* $\mathcal{F}_{FM_{slice}} = ((\mathcal{F}_{slice} \setminus deads(FM)) \cup synthetic_s)$ *and* $E_{slice} \subseteq E$ *such that* $E_{slice} = \{e = (v, v') \mid e \in E' \wedge \nexists\, v'' \in E' : ((v, v'') \in E' \wedge (v', v'') \in E')\}$ *where* $G' = (\mathcal{F}', E')$ *is the transitive closure of* G_{slice} *;*

$deads(FM)$ computes the set of dead features of FM (see Definition 8, page 26).

About the synthetic root. We consider that the synthetic root is *not* part of $[\![FM_{slice}]\!]$ and is only here to ensure the well-formedness of the hierarchy[1].

The synthetic root can be removed from G_{slice} if and only if one or more than one of its child feature is a *core* (see Definition 9, page 27) feature:

- in case the synthetic root has two or more than two child features that are core features, a procedure should choose one – deciding which feature to choose from amongs the core features is left as open – to replace the synthetic root ;
- in case there is exactly one core child feature f_{core}, the root feature of FM_{slice} becomes f_{core}. For example, feature A is the root feature of the sliced feature model of Figure 7.3(a) and feature P is the root feature of the sliced feature model of Figure 7.3(b).

The synthetic root cannot be removed from G_{slice} and is necessary (e.g., for the purpose of visualization) if all its child features are not core features. Figure 7.3(c) gives an example.

Discussion about the semantics. We consider that the slicing operation is applied to the feature model fm_1 shown in Figure 7.2(a) using the set of features T, S, E, D as slicing criterion. Formally:

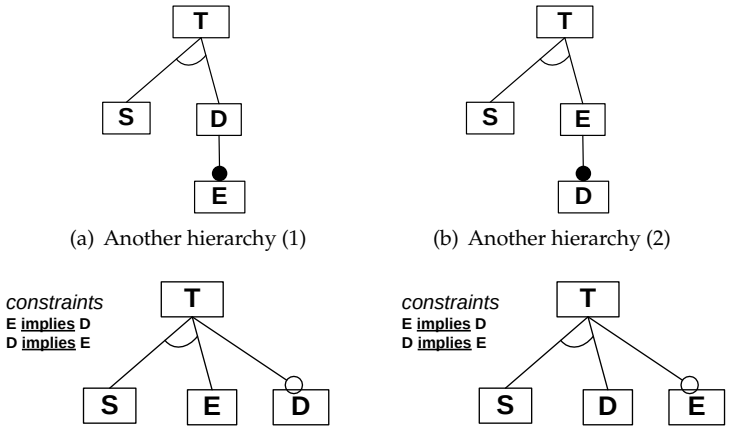
$$\Pi_{T,S,E,D}\ (fm_1)$$

As we did for the compositional operators, the semantics of the slicing operator has been defined in terms of set of configurations and hierarchy. The semantics controls that the feature models shown in Figure 7.4(a) and Figure 7.4(b) are not correct. Though the set of configurations is correctly restituted (*{ {D, E, T}, {S, T}}*), the hierarchy is not consistent with the properties exposed in Definition 16. From an ontological perspective, one may argue that the hierarchy does not correctly organize the features. For example, it may be preferable to move feature D as a child feature of feature E. However, due to the lack of knowledge about the actual meaning of features, it is out of the scope of the slicing operator. This is the role of an SPL practitioner to reorganize the hierarchy if needs be.

Another important remark is that there may exists more than one feature model that are consistent with the semantic properties of the slicing operator. The feature models shown in Figure 7.4(c) and Figure 7.4(d) correctly restitute the expected set of configurations and hierarchy but the feature groups differ. In the feature model of Figure 7.4(c), features S and E form a *Xor*-group whereas it is not the case in the feature model of Figure 7.4(d) (features S and D form a *Xor*-group). We do not find any criteria that can justify the choice of one feature group in preference to another.

Summary. We discuss through different examples the semantics of the slicing operator and

[1] Its particular status should be treated as such by a tool (e.g., when encoding the feature model into a propositional formula). This is similar to what have been discussed with the aggregate operator (see Section 5.3).



(a) Another hierarchy (1)

(b) Another hierarchy (2)

(c) A possible strategy for feature groups

(d) Another strategy for feature groups

Figure 7.4: Four possible feature models for the same slicing operation

we justify our different choices. The proposed semantics (see Definition 16) focuses on the primary aspects of feature models (set of configurations *plus* hierarchy). The semantics has been intentionally left open for some other aspects (e.g., how to group features? which constraints to add?).

7.3 ALGORITHM

Our previous experience in the merging of feature models has shown that *syntactical* strategies have severe limitations to accurately represent the expected set of configurations, especially in the presence of cross-tree constraints (see Chapter 5). The same observation applies for the slicing operation so that reasoning directly at the *semantic* level is required. The key ideas of the proposed algorithm are to *i)* compute the propositional formula representing the projected set of configurations and then *ii)* to apply propositional logic reasoning techniques to construct a feature model (including its hierarchy, variability information and cross-tree constraints) from the propositional formula.

7.3.1 Formula Computation

For a slicing $FM_{slice} = \Pi_{ft_1,ft_2,\ldots,ft_n}\ (FM)$, the propositional formula corresponding to FM_{slice} can be defined as follows:

$$\phi_{slice} \equiv \exists\ ftx_1, ftx_2, \ldots ftx_{m'}\ \phi$$

where $ftx_1, ftx_2, \ldots ftx_{m'} \in\ (\mathcal{F} \setminus \mathcal{F}_{slice}) =\ \mathcal{F}_{removed}$.

The propositional formula ϕ_{slice} is obtained from ϕ by *existentially quantifying* out variables in $\mathcal{F}_{removed}$ (see below for an illustration). Intuitively, all occurrences of features that are not present in any configuration of FM_{slice} are removed by existential quantification in ϕ. The slicing can be seen as a safe removal of a set of features as existential quantification

removes a variable from a propositional formula without affecting its satisfiability — in particular, dead features are removed.

Definition 17 (Existential Quantification)**.** *Let v be a Boolean variable occurring in ϕ. Then $\phi_{|_v}$ (resp. $\phi_{|_{\bar{v}}}$) is ϕ where variable v is assigned the value True (resp. False). Existential quantification is then defined as $\exists v\ \phi =_{def} \phi_{|_v} \vee \phi_{|_{\bar{v}}}$.*

An example of existential quantification is given in Appendix .2.

7.3.2 From formula to feature model

The translation of the propositional formula ϕ_{slice} to a feature model is largely similar to the technique described for the merge operator in Section 5.5. In the two cases, we have the propositional formula and we already *know* what the resulting hierarchy is. Therefore the algorithm remains the same. We first compute the hierarchy ①, we then set the variability information (mandatory/optional, Xor and Or-groups) ② and finally the constraints (implies/excludes/others) ③.

① **Hierarchy Computation.** Let G be the hierarchy of the input feature model to be sliced, FM. The synthetic root is added to G so that $synthetic_s$ is the new root of G. We obtain G_{slice}, the hierarchy of the resulting sliced feature model, by incrementally removing all features of $\mathcal{F}_{removed}$ in G. In case the feature is a leaf, the feature and its associated edge are simply removed. In case the feature is not a leaf, the feature and all associated edges are removed while its children are connected to its parent feature by adding new edges.

② **Mandatory and Feature Groups.** At this step, all features, except root, are currently optional (see Definition 1, page 23). We compute the implication graph, noted I_{slice}, of the formula ϕ_{slice} over $\mathcal{F}_{slice}$.

I_{slice} is a directed graph $G = (V, E)$ formally defined as:

$$V = \mathcal{F}_{slice} \qquad E = \{(f_i, f_j) \mid \phi_{slice} \wedge f_i \Rightarrow f_j\}$$

We use I_{slice} to identify biimplications and thus set mandatory features together with their parents (i.e., setting E_{MAND}). For feature groups, we reuse the prime implications method proposed in [Czarnecki and Wąsowski 2007] and thus set $\mathcal{F}_{XOR}$ and $\mathcal{F}_{OR}$. An important issue is that a feature may be candidate to several feature groups (which is not allowed by feature diagrams). We use information of the original feature model to favor features that were initially grouped.

③ **Constraints.** The set of implies constraints can be deduced by removing edges of I_{slice} that are already expressed in the feature diagram (e.g., parent-child relations). Similarly, excludes constraints that were not chosen to be represented as an Xor-group are added. When adding constraints, we control that the constraint is not already induced by the feature model. The feature diagram *plus* the implies/excludes constraints may still be an over approximation of ϕ_{slice}. The complement corresponds to $\psi_{slice_{cst}}$.

7.3.3 Some Properties of the Algorithm

Anomaly free. Benavides et al. identify different kinds of *anomalies* (also called errors) in feature models, for example, dead features, false optional features, or redundancies (see [Benavides et al. 2010] or Section 3.1.2). The slicing algorithm we propose ensures, by

construction, that there is no dead feature, correctly detect mandatory features and avoids redundancy in the representation (e.g., we add an implies/excludes constraint only if it is not already induced by the feature model). Hence, we guarantee that the resulting sliced feature model does not contain anomalies.

Non identity property. When we apply the slicing operator to a feature model using $\mathcal{F}$ as slicing criterion, the resulting feature model is logically equivalent to the original feature model but the variability information (e.g., $\mathcal{F}_{XOR}$) can be different. An example is given by the feature model of Figure 7.2(a) when some Or-groups would be Xor-groups in the sliced feature model.

Slice as a Corrective Operator. The anomaly free and non identity properties shows that the slice operator can be used as an *automated technique to correct anomalies* of feature models while preserving the original set of configurations and feature hierarchy. Moreover the corrective modifications applied to the original feature model can be detected and reported to an SPL developer so that he/she can understand the anomalies.

7.4 ILLUSTRATIONS

In this section the objective is to convince the reader that coupling the use of compositional operators with *slice* allows to address some issues that could not be easily achieved without it such as: updating views interacting one with the others after insertion of new constraints, extracting new views from an existing system, reconciling two views of a system or checking that the specification of a problem can be handled by current capabilities of the implementation platform. In the following, all views, system, problem specification or platform implementation are seen as one or several feature models.

7.4.1 Updating Views

In Figure 7.5, a registration service SPL is described through different views: information about the deployment on the grid, internal algorithms, the supported communication protocols and the type of handled medical images. These feature models can then be used to check consistency between composed services in the workflow and to facilitate their coherent configurations.

In practice, the different feature model views of a service are not independent. The workflow designer has to add constraints to enforce interactions between the feature model views (see bottom part of Figure 7.5). Determining the impact of these constraints on each feature model view cannot be done manually or even automatically with current techniques and tools. We rely on the corrective capabilities identified in previous section to perform the update of the different feature model views. Using the slice operator, it simply consists in *i)* aggregating the four feature models into a single one (*fmService*) with constraints mapped on it, *ii)* invoking *slice* four times producing as much sliced feature models, the slicing criteria being respectively the features of each of the four feature models.

As a result, figures 7.6(a) and 7.6(b) correspond to the sliced feature model with respectively the features of the feature models $FM_{MIsupport}$ and FM_{algo}. The two other feature models are not impacted by the constraints mapped on *fmService*.

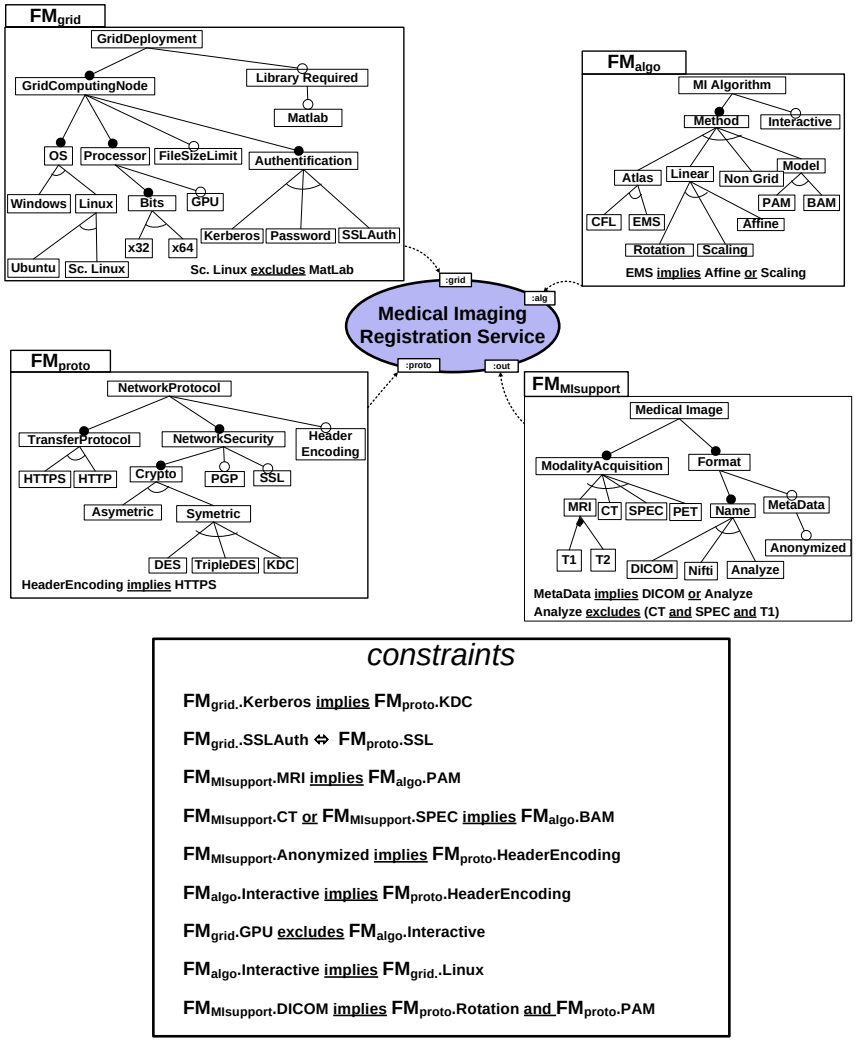
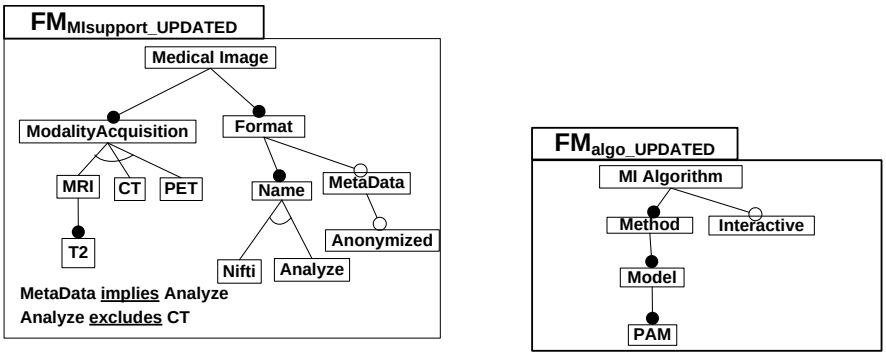


Figure 7.5: Medical Imaging Service: Variability and Concerns

7.4.2 Supporting Multiple Perspectives

On the same example, the slice operator can be used to extract other views (or *perspectives*) of a service. In Figure 7.7, we capture expertises related to security features or to the medical imaging domain. Two slice operations are applied below and compute two feature model views, stored into *fmViewMI* and *fmViewSecurity*. The slicing criterion used to compute *fmViewMI* (resp. *fmViewSecurity*) contains features from the feature models FM_{MI}, FM_{algo} and FM_{grid} (resp. FM_{MI}, FM_{proto} and FM_{grid}). The slice guarantees that all the interactions existing with other feature model views are still enforced.



(a) Update of medical image support feature model

(b) Update of algorithm feature model

Figure 7.6: Updating feature model views

7.4.3 Reconciling Feature Models

When managing a set of feature models, the different stakeholders involved in the SPL development may have to put together very similar variability information but with a different structure. For example, as already noted in the previous chapter, different suppliers in the medical imaging domain (scientists, research teams, companies, etc.) provide imaging services and may use different hierarchies, concepts, vocabulary, etc. when elaborating the feature models.

Let us consider two feature models, $fmMI1$ and $fmMI2$, in Figure 7.8.

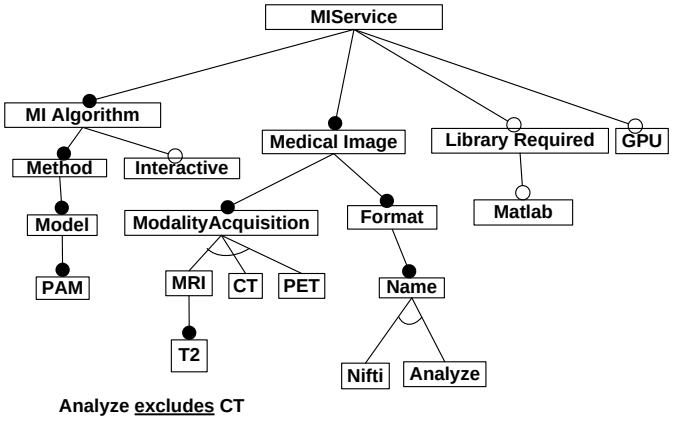
The two feature models differ. In particular, features Open, Proprietary, NiftiI, NiftiII are present in $fmMI1$ but not in $fmMI2$. Intuitively, more structure and details are modeled in $fmMI1$. As a result, a comparison (see Definition 10, page 53) or a merging of the two feature models leads to counter intuitive results, i.e., the intersection of the two configuration sets is empty. Looking at the two feature models, some configurations seem to correspond, for example, the valid configuration $\{MedicalImage, DICOM\}$ of $fmMI2$ with the configuration $\{MedicalImage, Open, DICOM\}$ of $fmMI1$. We thus need to *reconcile* (or align) the two feature models and allow an SPL practitioner to align in a coherent way information from $fmMI1$ and $fmMI2$.

Using the slice operator, we simply remove features of $fmMI1$ *i)* that structure the feature model (i.e., features Open, Proprietary) and *ii)* that can be abstracted by a single feature (i.e., Nifti abstracts features NiftiI and NiftiII). Then, the merge operator, for instance, can be used (see Figure 7.8).

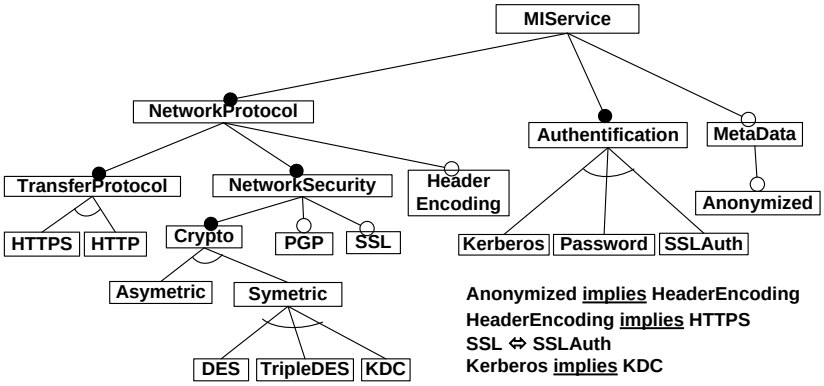
7.4.4 Reasoning about Two Kinds of Variability

In SPL engineering, two kinds of variability are usually distinguished (e.g., see Section 3.2.1 or [Pohl et al. 2005, Metzger et al. 2007]): software variability, hidden from customers (also called internal variability), as opposed to product line (PL) variability, visible to them (also called external variability).

Software variability and PL variability can be seen as two *concerns* of an SPL. Metzger et al. proposed a formal and concise approach for *separating* PL variability and software



(a) Medical Image expert view



(b) Security expert view

Figure 7.7: Another decomposition strategy and set of views

variability and enabling automatic analysis [Metzger et al. 2007]. The two concerns are modeled as two feature models and inter-related by constraints. The authors mention several properties that should be checked when reasoning about the two kinds of variability. We revisit here the approach defended in [Metzger et al. 2007]. We show how the operators can be combined to support separation of concerns in this context.

Realized-by property. An important property of an SPL is *realizability*, that is, whether the set of products that the PL management decides to offer is fully covered by the set of products that the software platform allows for building.

In Figure 7.9, we want to ensure that for each valid selection/deselection of features of fm_{PL} performed by a customer, there exists at least one corresponding software product described by $fm_{software}$. The PL variability is documented using fm_{PL}, the software variability is documented using another feature model (see $fm_{software}$) and the two feature models are related through constraints (see map_{SoftPL}). Note that the mapping between

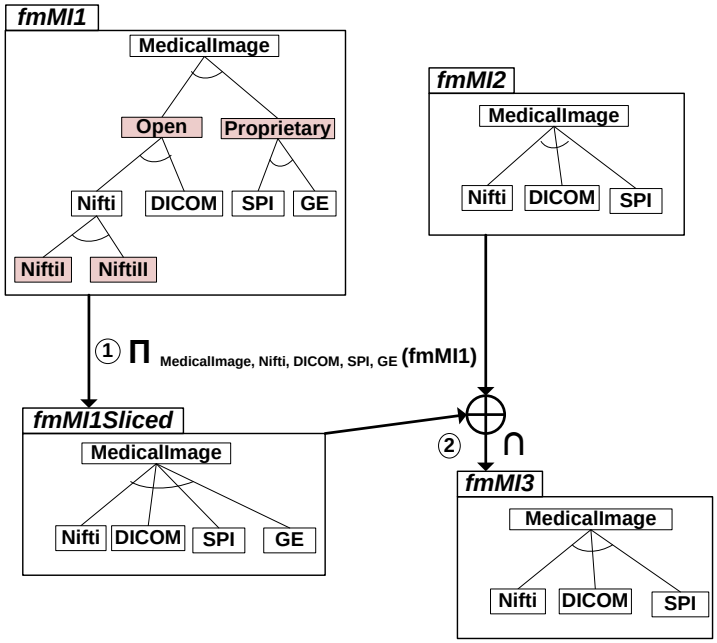


Figure 7.8: Slicing (①) to reconcile feature models and allow, e.g., merging (②)

features of fm_{PL} and $fm_{software}$ is not necessarily one-to-one.

To do so we first reason about the relationship between $fm_{software}$ and fm_{PL}. We compute fm_G, the *aggregation* of fm_{PL} and $fm_{software}$ together with the constraints map_{SoftPL}. In terms of feature models, the realizability property can be formally expressed as follows:

$$\forall cp \in [\![fm_{PL}]\!], cp \in [\![\Pi_{\mathcal{F}_{PL}}\ (fm_G)]\!] \tag{7.1}$$

with $\mathcal{F}_{PL}$ the set of features of fm_{PL}.

Intuitively, if the restriction of the PL features to $[\![fm_G]\!]$ is equivalent to the original $[\![fm_{PL}]\!]$, the constraints map_{SoftPL} has no effect on the PL part of fm_G and thus the realizability property holds. Otherwise some products cannot be realized in the platform. The property of Equation 7.1 can then be implemented with the slice operator and, if needs be, one can also enumerate all products that cannot be realized. More precisely, Equation 7.1 implies to check if fm_{PL} is a *refactoring* of $\Pi_{\mathcal{F}_{PL}}\ (fm_G)$.

Using the aggregate, slice and merge diff operators, we can automatically check this property. We first slice the aggregated feature model fm_G by only including $\mathcal{F}_{PL}$, the set of features of fm_{PL}. The slice produces a new feature model, denoted $fmPLPrime$. Formally:

$$fmPLPrime = \Pi_{\mathcal{F}_{PL}}\ (fm_G)$$

Then, we compare the resulting feature model, $fmPLPrime$, with the original PL feature model, $fmPL$. If $fmPLPrime$ is not a refactoring of $fmPL$, the realizability property

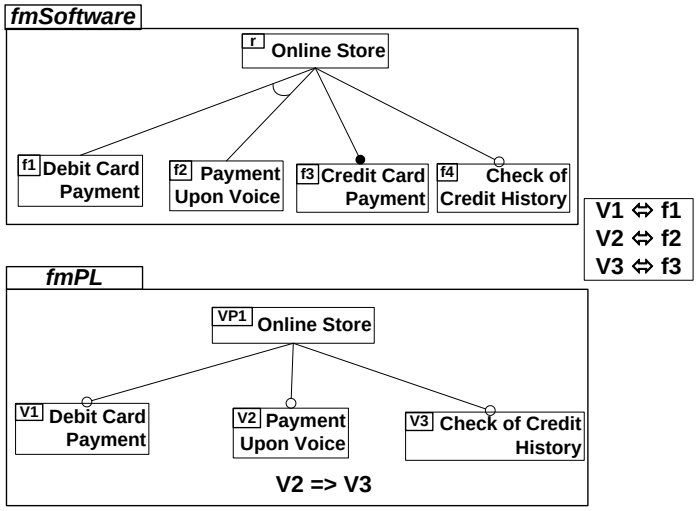


Figure 7.9: Software and PL Variability (adapted from [Metzger et al. 2007])

is violated since some existing products of $fmPL$ are removed in $fmPLPrime$ and no product is added.

Finally, we can compute the set of products that are in $fmPL$ but not in $fmPLPrime$ using the merge operator in *diff* mode. The merge operator produces fm_{PLDiff}

Back to the example of Figure 7.9, we obtain that the realizability property does not hold and that three products proposed to customers cannot be realized by the platform:

$[\![fm_{PLDiff}]\!] = \{\{V1, V3, V2, VP1\}, \{V1, VP1\}, \{V3, VP1\}\}$

The merge in intersection mode of $fmPL$ and $fmPLPrime$, denoted $fm_{PLInter}$, allow one to determine that only the following two products can be realized:

$[\![fm_{PLInter}]\!] = \{\{V1, V3, VP1\}, \{V2, V3, VP1\}\}$

Non useful products. A product is useful if it is a possible realization of a PL member. As argued in [Metzger et al. 2007], the list of non-useful products is a symptom of unused flexibility of the software platform. It can be on purpose, for example, justified by future marketing extensions. Formally, all products are useful if the following relation holds:

$$\forall cp \in [\![fm_{software}]\!], cp \in [\![\Pi_{\mathcal{F}_{software}}\ (fm_G)]\!]$$

The usefulness property can be seen as the "symmetric" of the realized-by property. Therefore similar techniques can be applied.

7.4.5 Slicing-based Merge Implementation

We show how the slicing operator can be used to provide an alternative way of implementing the merge operator dedicated to feature models described in Chapter 5.

We rely on the same example used in Section 6.4.

We want to merge in strict union mode the feature models FM_{supp_1}, FM_{supp_2} and FM_{supp_3} of Figure 6.2 (see page 75). There are identified as I_0, I_1 and I_2 in Figure 7.10. We

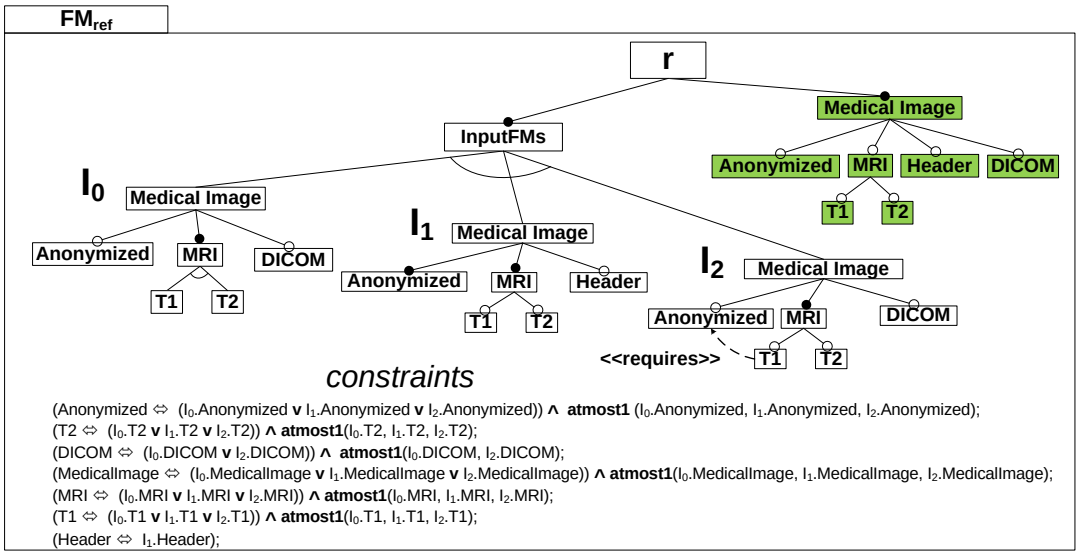


Figure 7.10: Merge of three feature models, I_0, I_1 and I_2 using the slicing operator

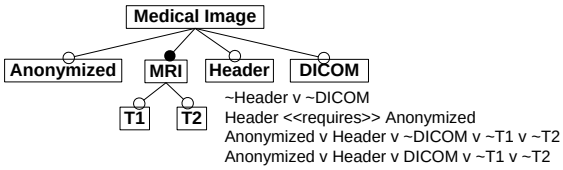


Figure 7.11: Merged feature model in strict union mode of the three feature models, I_0, I_1 and I_2

reuse the reference-based technique described in Section 6.4 and we obtain FM_{ref}. The constraints are specified as follows:

- the term atmost1 $(C_1, ..., C_n)$ is equivalent to $\wedge_{i<j}(\neg C_i \vee \neg C_j)$
- each feature name is prefixed and unique in FM_{ref} (e.g., I_1 . $Format$ correspond to the feature Format of the feature model I_1) but for the purpose of readability, the original name of features is depicted in the boxes.

The idea is to slice FM_{ref} using as a slicing criterion the set of features of the *reference* feature model (called SIFM in [Hartmann et al. 2009]), that contains the "super-set of the features" from all the input feature models (green features in Figure 7.10). The resulting sliced feature model is equal to the feature model produced by the merge operator (see Figure 7.11). Formally:

$$\Pi_{\,MedicalImage,Anonymized,MRI,Header,DICOM,T1,T2}\,(FM_{ref}) = I_0 \oplus_{\cup_s} I_1 \oplus_{\cup_s} I_2$$

7.5 Comparison with Other Solutions

We now briefly discuss how we extend previous approaches and how they can benefit from our slicing operator.

We apply the techniques described in Section 7.4.4 using the larger example described in [Metzger et al. 2007]. We successfully retrieved the same results, but our approach is more efficient since we do not enumerate configurations/products as they do. Moreover, high-level operators (slice, aggregate, compare, merge) facilitate the reasoning realization and offer a systematic solution for SPL practitioners when understanding and maintaining the two feature models.

Thüm et al. [Thüm et al. 2009] presented an automated and scalable technique to characterize the kinds of edit between two feature models. An original property of the technique is that they distinguish abstract features from concrete features when reasoning. Abstract features are, in their work, non-leaf features. We consider this is the role of an SPL practitioner to explicitly determine which features are abstract (as shown in the example of Section 7.4.3, abstract features are not necessary non-leaf features). Our technique is thus more general and realize the vision of [Schobbens et al. 2007] that makes the distinction between features that are of interest per se (i.e., that will influence the final product) and others.

In the context of feature-based configuration, several works proposed techniques to separate the configuration process in different steps or stages[Mendonca and Cowan 2010, Hubaux et al. 2009]. Our work is complementary since we propose techniques to decompose feature models according to different perspectives or role of stakeholders involved in the configuration process. Hubaux et al. provide view mechanisms to decompose a large feature model [Hubaux et al. 2010a]. However they do not propose a comprehensive solution when dealing with cross-tree constraints. They also consider that the root feature should always be included, which is a limitation not imposed by our approach.

Part III

FAMILIAR

In the previous part, we have laid the foundations of managing multiple feature models, but an operational solution is still needed to support SPL practitioners activities. Our proposal is a language dedicated to the management of feature models, called FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning).

In Chapter 8, we introduce FAMILIAR (rationale, syntax, set of operators, etc.) and we illustrate how the language can be used to realize a non trivial scenario in which multiple SPLs are managed.

In Chapter 9, we describe the language implementation and we evaluate the performance of two important operators, merge and slice.

Eight

A Domain-Specific Language for Managing Feature Models

This chapter shares material with the SAC'11 paper "A Domain-Specific Language for Managing Feature Models" [Acher et al. 2011b], the VaMoS'11 paper "Managing Feature Models with FAMILIAR: a Demonstration of the Language and its Tool Support" [Acher et al. 2011d] and the ASE'11 paper (tool demonstration) "Decomposing Feature Models: Language, Environment, and Applications" [Acher et al. 2011c]

In Part II, we have presented a set of *operators* (insert, aggregate, merge, slice) to support composition and decomposition of feature models. We have outlined how the operators can be combined to realize complex management tasks.

To support large scale *management* of feature models, a more comprehensive and operational solution, though, is needed. We mention at several places of Part II that reasoning facilities and other operators should be developed. For example:

- when an aspect feature model is inserted into a base feature model, one may want to reason about the new set of configurations ;
- when two feature models are aggregated, one may want to determine if the resulting feature model is not void ;
- before merging two or more than two feature models, one may need to reconcile them by renaming features or removing unnecessary details ;
- when a slice operation is performed on a feature model, one may want to configure the sliced feature model.

In addition, using these operators all alone is very restrictive: What is the use of aggregating two feature models if you cannot reason about the satisfiability of the composition? What is the use of merging two feature models if you cannot reason about the new set of configurations? We rather need to combine these operators and perform sequences of operations to realize complex reasoning tasks. Several examples described in Part II suggest that sequences of different manipulations are needed, for instance, when making evolve feature model structures and their configurations, inserting new features, renaming features, extracting sub-feature models while maintaining constraints, and reasoning about intermediate results during composition. There is thus a need to provide SPL practitioners with the means to control when and how (de-)composition and analysis mechanisms are applied, and with the capability to replay and reuse (de-)composition and analysis procedures.

To summarize, we need both a more comprehensive set of operators and practical means to perform sequences of operations.

In Section 8.1, we explain why we chose to design a *domain-specific language* (DSL) and we discuss other alternatives.

In Section 8.2, we give an overview of the DSL named FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning). We present the main constructs of the language and the new operators we have introduced in addition to the composition and decomposition mechanisms.

In Section 8.3, we describe an application of the DSL on a larger scale problem that involves manipulating multiple feature models with different scripts.

8.1 WHY A DOMAIN-SPECIFIC LANGUAGE?

There are at least three possible solutions to meet the requirements above. One approach is to reuse existing feature model development tools and editors. The other two involve using a language, either general-purpose or domain-specific.

8.1.1 Textual vs Graphical Approach

Several graphical feature model editors are currently available, and some do provide support for managing some aspects of feature model development like pure::variants [pure::variants 2006], FeatureIDE [Kästner et al. 2009a], SPLOT [Mendonca et al. 2009a], etc. (see Section 3.3). For large scale management, *pure::variants* is a commercial tool with good support for binding to other models and for code generation. *FeatureIDE* is a comprehensive environment that interconnects with different feature model management tools and has a Java API to manipulate feature models. Integration of reasoning tools is thus facilitated, for example, a tool for feature model edits [Thüm et al. 2009] has been integrated. Nevertheless, current tools do not fully support the composition or decomposition of several separated feature models. A conceivable solution would be to integrate our feature model composition operators (insert, aggregate and merge) as additional functionalities inside a mainstream graphical editor. Our example (and our case studies, see Part IV) indicate that manipulating several feature models with composition operators requires support for replaying sequences of operations, observing properties along its manipulations, and organizing all these actions as reusable parts. We thus identify this as a requirement for a textual, executable language, close to common scripting languages. As already mentioned, such a script language should provide *i)* basic sequencing of feature model operations, *ii)* access to feature model internals, *iii)* reasoning operations and *iv* composition and decomposition mechanisms. A textual script performs a sequence of operations on feature models. Such operations are *reproducible* and *reusable*. Obtaining the same properties in a graphical editor requires an additional effort, for example, the implementation of an undo/redo system and serialization of the sequence of operations. This is very close to what *GUI scripting* languages do with macro-commands. This could have been the only requirement of the FAMILIAR language, but using the language, we believe its textual form favors *readability* of the specified operations, and more *usability* and *productivity* when dealing with compositional operations on feature models. On the other hand, graphical visualization has proved to assist users: This does not avoid the possibility to also provide graphical counterparts built on top of the textual language, as in many other domains (see Chapter 9).

8.1.2 Domain-Specific Language vs General Purpose Language

As editors like FeatureIDE and frameworks such as FAMA (see Section 3.3) provide an API, another conceivable solution would be to build an API extension in a mainstream programming language in order to provide support for using composition operators and other feature model management operations. While this may be a feasible solution, it would imply many repetitive and error-prone actions, except if the API have been created thinking in terms of simplicity, readability and focusing on the domain of feature models. In this case, one can argue that the API is an *internal*[1] DSL, written on top of a host language (e.g., Java)

A DSL, being internal or external, should allow a feature model user to more quickly build the code they need to manipulate feature models.

In the previous chapters, we use a *mathematical notation* to combine operators. Appropriate or established domain-specific notations are usually beyond the limited user-definable operator notation offered by general purpose languages. An external DSL seems particularly adequate in this case, as it would provide only the necessary expressive power for anticipated feature model manipulations.

The facilities provided to the feature model users must allow the description of complex operations dedicated to feature models, in both a compact and readable way, while being understandable by an expert who may not necessarily be a software engineer. In addition, such an external DSL should be used more easily by feature model users as the learning curve is expected to be more favorable.

To conclude the discussions, the predominant idea is that we need a *textual* language dedicated to the *domain* of feature models. TVL [Classen et al. 2010a] (for Textual Variability Language) belongs to this category but its focus is to *specify* feature models and is thus inadequate regarding the requirements identified above.

8.1.3 The Domain of Propositional Feature Models

We have argued that a DSL seems particularly adequate to the needs of managing feature models. The decision to develop a new DSL (e.g., *when?*, *why?* and *how?*) is a difficult one as it involves both advantages/disadvantages or risks/opportunities [van Deursen and Klint 1998, Mernik et al. 2005, Fowler 2010]. Essential to this decision is the notion of *domain*. We further discuss the *domain* of feature models.

The domain of automated analysis. Since the introduction of feature models, the automated analysis of feature models is an active area of research and is gaining importance in both practitioners and researchers in the SPL community. The operators we have developed in Part II, and thus the language we want to create, are obviously in that domain. [Benavides et al. 2010] conduct a literature review and claim that:

> *"From the analysis of current solutions, we conclude that the analysis of feature models is maturing with an increasing number of contributions, operations, tools and empirical works."*

[1] The external/internal dichotomy is generally used to characterize DSLs. An *external* DSL is a completely separate language and has its own custom syntax. An *internal* DSL is more or less a set of APIs written on top of a host language (e.g., Java). Internal DSL is limited to the syntax and structure of its host language. Both internal and external DSLs have strengths and weaknesses (learning curve, cost of building, programmer familiarity, communication with domain experts, mixing in the host language, strong expressiveness boundary, etc.) [Fowler 2010]

Hence the development of a DSL integrating automated analysis techniques seems particularly adequate – as argued by [van Deursen and Klint 1998], "a prerequisite to developing a DSL is *mature* domain knowledge." Furthermore, DSLs are considered as "enablers of reuse" [Mernik et al. 2005] and there is a clear opportunity to *reuse operations* already defined in the domain.

Domain analysis. In the analysis phase of DSL development, the problem domain is identified (descriptions of domain concepts and domain terminology (vocabulary), domain definition defining the scope of the domain) and domain knowledge is gathered. The domain concepts we have identified are enumerated below:
feature model: the primary entities we want to manipulate and reason about ;
features: a feature model is composed of features ;
feature name: features are characterized by their names (string-based representation) ;
hierarchy: the essence of feature modeling is hierarchy and the way features are organized ;
variability information: a feature may be optional, mandatory or be part of a feature group (Or- or Xor-group) ;
constraint: features (and by extension feature models) may be related through constraints, expressed in propositional logic ;
configuration: a feature model controls the legal combinations of features, called feature configurations (or configurations for short). An SPL practitioner usually selects and deselects features during a configuration process until there is no choice to perform ;
propositional formula: to reason about properties of a propositional feature model, we need a logical representation ;
number: properties of a feature model include the number of valid configurations or some metrics (e.g., commonality) ;
We can notice that the number of concepts rather small. This is a good point for the DSL as it will focus on a limited set of concepts.

Scoping the domain. The term *propositional feature model* is used to restrict the scope of feature models. For example, some authors argue that a feature model should come with feature descriptions (including binding times, priorities, stakeholders, etc.) or with feature attributes (e.g., numerical attributes). When including attributes in feature models the analysis becomes more challenging because not only attribute-value pairs can be contemplated, but more complex constraints, beyond propositional logic, can be specified. Furthermore this type of relationships can affect operations of analysis and can include new ones. We do not consider feature attributes and restrict the scope of the domain to propositional constructions and operations. Similarly we do not consider cardinality-based feature models [Czarnecki et al. 2005a, Michel et al. 2011]. In SPL engineering, feature models are usually mapped to other artefacts of an SPL (see Chapter 2). We do not consider the relationship between feature models and other artefacts, i.e., the domain is restricted to feature models.

Summary. Though there is still an active research in feature modeling, the domain of propositional feature models is well established. There is an opportunity for the DSL to reuse operations already defined in the literature or developed in the context of this thesis. The DSL should provide a domain-specific notation to manipulate a restricted set of concepts as well as a support for the composition and decomposition operators.

8.2 LANGUAGE IN A NUTSHELL

The FAMILIAR DSL is an executable scripting language that supports manipulating and reasoning about feature models. The next subsections will detail and illustrate the main constructs of the language.

8.2.1 Types and Variables

FAMILIAR is a typed language that supports both complex and primitive types. The various types supported by the language have been chosen according to the domain concepts identified in the previous section. Variables representing complex types record a reference to the data whereas other variables record the data value itself. (The notions of *reference* and *value* are similar to the ones used in programming languages.) Complex types are *Feature Model*, *Configuration*, *Feature*, *Constraint*, etc. or generic *Set* which represents container values. Primitive types include *String* (e.g., feature names are strings), *Boolean*, *Enum*, *Integer* and *Real*. An example script that illustrates typing is given below:

```
1 mi1 = FM ( MedicalImage: Modality Format Anatomy [Anonymized];
2    Modality: (PET | CT); Format: (DICOM|Nifti); Anatomy: Brain;)
3 mi2 = FM ( MedicalImage: Modality Format Anatomy [Anonymized];
4    Modality: (PET | CT); Format: (DICOM|Nifti); Anatomy: Brain;)
5 b1 = mi1 eq mi2 // b1 is true
6 b2 = mi1 == mi2 // b2 is false
7 str = "PET" // str records the value "PET"
8 fmSet = { mi1 mi2 }  // fmSet records a reference to a set
```

Lines 1-4 define two variables of type *Feature Model*: *mi1* and *mi2*. For variables with complex types we may need to compare either the reference or the content of the recorded data so that we propose two operators. Lines 5 and 6 illustrate the use of content equality (**eq**) and reference equality (**==**) on complex types. Lines 7 and 8 show assignments to variables of type String (str) and Set ($fmSet$ is a set of feature models). Note that for the variable str, the use of content or reference equality would return the same result which corresponds to content equality. Types have accessors for observing the content of a variable. The example below illustrates the use of accessors :

```
1 mi3 = FM ( MedicalImage: Modality Format Anatomy Anonymized;
2    Modality: (MRI | CT); Format: Analyze; Anatomy: Kidney;)
3 f1 = parent MRI // f1 refers to feature named 'Modality' in mi3
4 f2 = root mi3 // f2 refers to feature named 'MedicalImage' in mi3
5 s1 = name f2 // s1 is a string "MedicalImage"
6 fs = children f1 // set of features named 'MRI' and 'CT' in mi3
7 nfs = size fs // 2
```

In line 3, variable *f1* records a reference to feature Modality (the parent of feature MRI). We consider that features are uniquely identified by their names in a feature model (as stated in Chapter 3). In line 4, variable *f2* contains a reference to the feature MedicalImage (the root of the feature model $mi3$). The remaining lines (5-7) illustrate operations returning *i)* the name of a feature, *ii)* the set of direct subfeatures of a given feature and *iii)* the number of elements of a set.

8.2.2 Importing and Exporting Feature Models

We provide multiple notations for specifying feature models (SPLOT, GUIDSL/FeatureIDE, a subset of TVL, etc.) A FAMILIAR user can:

- load feature models in those notations (using **FM** constructor) ;
- serialize the feature models in those notations (using **serialize**).

Internal notation. FAMILIAR also provides a concise notation, largely inspired from *FeatureIDE* and *feature-model-synthesis* (see Section 9.1.2):

```
1 fmFoo = FM (A : B C D; D : (E|F|G); C : (H|I|J)+ ; (C implies I) ; )
2 fmFoo2 = FM ("fmFoo2.tvl")
3 fmFoo3 = FM ("fmFoo3.m")
4 serialize fmFoo into SPLOT
```

In the example of line 1:

- A is the root ;
- B, C and D are child-features of A: B and C are mandatory while D is optional ;
- E, F and G form a Xor-group and are child-features of D ;
- H, I, and J form an Or-group and are child-features of C ;
- C implies I corresponds to a propositional constraint of the feature model.

Other notations. Feature models can also be imported using other notations and the same **FM** constructor (in line 2 we import a TVL model while in line 3 we import a feature model in FeatureIDE notation). Line 4 gives an example of a serialization.

8.2.3 Operations

Modifying feature models. The language provides some basic operators for renaming and removing features in feature models. Renaming can be useful when composing or comparing feature models that use different terminology for the same concept (i.e., feature). The following illustrates how features can be renamed:

```
1 oldFeature = parent Analyze // 'Format' feature of mi3
2 newName = strConcat "MI" (name oldFeature)
3 b1 = renameFeature oldFeature as newName // aligning terms
4 assert (b1 eq true) // assert (b1) is equivalent
```

In lines 1-3, the feature DICOM of feature model *mi3* is renamed to MIFormat by concatenating the string *MI* with the old name *Format*. The operator **assert** in line 4 stops the program with an appropriate error message if the renaming is not successful (i.e., *b1* is **false**). A renaming is not successful if the feature to be renamed (e.g., oldFeature in the previous example) does not exist. Similarly, the operator **removeFeature** takes a feature as an argument and removes the feature and its descendants from the feature model it belongs to (see Section 8.2.6 for an example). It also returns **true** or **false** depending whether it is successful or not.

Identifier resolution. In line 1 of the script above, the identifier Analyze is used. It corresponds to the feature Analyze of the feature model *mi3*. It could be expressed more explicitly as follows:

```
1 oldFeatureBis = parent mi3.Analyze // 'Format' feature of mi3
2 conflictingFeature = parent DICOM // feature present in mi1 and mi2
```

In FAMILIAR, an identifier may refer to a variable identifier (e.g., oldFeatureBis) or to a feature in a feature model. It may happen that two features with the same identifier are present in two different feature models (see line 2). In this case, the script stops and the error is reported to the FAMILIAR user.

Handling feature model configurations. The language also allows FAMILIAR users to create feature model configurations, and then **select**, **deselect**, or **unselect** a feature. To **select** a feature means that the configuration includes the feature. To **deselect** means that it will not be part of the configuration. To **unselect** means that no decision has been made: the feature is neither selected nor deselected. Each of these configuration manipulation operations returns a boolean value, i.e., true if the feature selected/deselected/unselected does exist.

An example usage of these operations is given below:

```
1 conf1 = configuration mi1 // create a configuration of mi1
2 b1 = select Anonymized in conf1 // feature Anonymized is selected
3 b2 = deselect Anonymized in conf1 // override the previous selection
4 b3 = unselect Anonymized in conf1 // neither selected nor deselected
```

In line 1, the operator **configuration** creates and initializes a configuration of the feature model *mi1*. Lines 2-4 provide examples of the configuration manipulations. In addition, the accessors **selectedF**, **deselectedF**, **unselectedF** return respectively the set of selected, deselected and unselected features of a configuration.

Reasoning about feature models and configurations. FAMILIAR provides several operators to support reasoning about feature models and configurations. The script below provides examples of the feature model manipulation and reasoning operators:

```
1 conf2 = copy conf1
2 nb = counting mi1 // number of valid configurations: 8
3 b1 = isValid conf1
4 b2 = (selectedF conf1) eq (selectedF conf2) // true
5 select Anonymized in conf1
6 confFM = asFM conf1 // convert a configuration into a feature model
7 cmp = compare m1 mi2 // refactoring
```

Line 2 computes the number of valid configurations of *mi1*. The **isValid** operator checks whether a configuration is not inconsistent (see line 3) according to its feature model. Indeed, a selection or deselection of feature may lead to an unvalid configuration (e.g., when two mutually exclusive features are selected).

The **isValid** operator can also perform on an feature model and determines its *satisfiability* (see Definition 7, page 26). FAMILIAR also provides an operator, called **isComplete**, that checks whether a configuration is *complete*, i.e., whether all features have been selected or deselected. The *Configuration* type provides three accessors that return the set of selected, deselected and unselected features: **selectedF**, **deselectedF** and **unselectedF**. Line 4 checks that the set of selected features in both *conf1* and *conf2* are equal (which is true simply because *conf2* is a copy of *conf1*).

Moreover, the operator **asFM** can convert a configuration, say c, into a feature model: for each selected feature of c, say f, we add a propositional constraint (i.e., a literal f) to the feature model of c and for each deselected feature of c, say g, we add a propositional constraint (i.e., a negative literal $\bar{g}$) to the feature model of c. For example, in line 5-6, *confFM* is the feature model *mi1 plus* the literal $Anonymized$. **asFM** is typically used when a configuration is not complete. The operator **asFM** guarantees that the resulting feature model does not contain *anomalies* (see Section 3.1.2).

The **compare** operation is used to determine whether an feature model is a refactoring, a generalization, a specialization or an arbitrary edit of another feature model. This operation is based on the algorithm and terminology used in [Thüm et al. 2009] (see also Definition 10, page 53). Line 7 illustrates comparison capabilities based on sets of configurations of feature models. *cmp* is an *Enum* type whose possible values can be **REFACTORING**, **SPECIALIZATION**, **GENERALIZATION** and **ARBITRARY**. In the example above, *cmp* has the value **REFACTORING** since *mi1* and *mi2* represent the same set of configurations.

Conditional and iterator constructs. The conditional construct is a classical **if then else** (see Section 8.3 for an illustration). FAMILIAR has also conditional and loop control structures. **foreach** loop (see lines 2 to 5 in the example script below) can be used to iterate over a set of variables (e.g., representing feature models, features, and configurations) and over a sequence of operations. FAMILIAR also allows a script writer to use a wildcard "*" to define a set of elements (e.g., feature models, features). It may be placed just after "." or anywhere within a variable or feature name. The following illustrates the use of the loop control structure and the wildcard:

```
1 varset = mi1.* // it can be written: varset = features mi1
2 foreach (f in varset) do
3   newName = strConcat "new_" f
4   renameFeature f as newName
5 end
```

Line 1 gathers in *varset* all the features of *mi1*. We then iterator over *varset* (which represents a set of features) so that each feature name becomes prefixed with "new_".

8.2.4 Decomposition

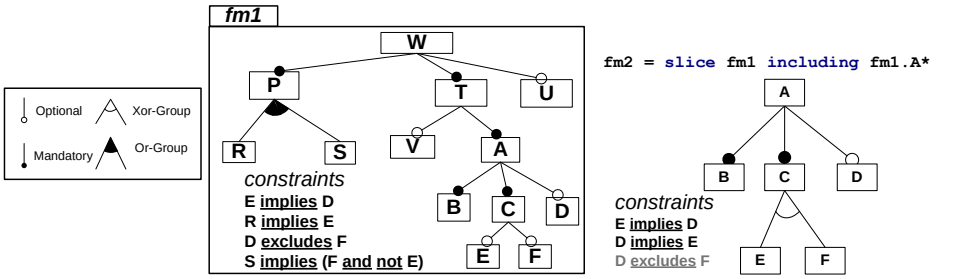
In Figure 8.1, we show three uses of the **slice** operator, which syntax is as follows:

```
fmS = slice anFM including | excluding setOfFeatures
```

anFM is the feature model that is sliced and *fmS* is the resulting one (*anFM* remains unchanged). *including*, resp. *excluding*, keeps only, resp. rejects, the *setOfFeatures* in parameter. This parameter may be obtained using set operations or FAMILIAR accessors – an XPath-like syntax is provided to address features within a feature model.

For example in Figure 8.1(b), *fm1.A**, corresponds to all features included in the sub-hierarchy *A* of *fm1* (including *A* itself). In Figure 8.1(d) *{E, D, F}* is a set of three features *E, D, F* of *fm1*. In the following example below, we use the operator *setUnion* to build a set which corresponds to all features included in the sub-hierarchies *A* and *P* of *fm1* (but not including *A* nor *P*).

```
fm5 = slice fm1 including setUnion fm1.A.* fm1.P.*
```



(a) $fm1$, input feature model

(b) [[*fm2*]] = {{*A, B, C, D, E*},{*A, B, C, F*}}

(c) [[*fm3*]] = {{*P, S*}, {*P, R*}}

(d) [[*fm4*]]={{*F*}, {*D, E*}}

Figure 8.1: Example of slice operations applied on the feature model of Figure. 8.1(a).

8.2.5 Composition

Inserting feature model. The **insert** operator produces an feature model by inserting a feature model into another base feature model. In line with Section 5.2, the operator takes three arguments: *i)* the aspect feature model to be inserted, *ii)* the feature in the base feature model where the insertion is to take place, and *iii)* the operator (e.g., Xor) that determines the form of the insertion. The insertion returns **false** if the two sets of features in the base and aspect feature model are not disjoint (we recall that each feature name must be unique in a feature model, see Section 3.1.2), if the feature in the base feature model does not exist or if the aspect feature model is inserted into the root feature of the base feature model with a Xor or Or-group. The base feature model is *modified* if the insertion succeeds. An example of a FAMILIAR script describing an insertion is given below:

```
1 base = FM (Format: [OpenSource] Header Name
2            [Anonymization] ; Name: (DICOM|Analyze); )
3 aspect1 = FM ( MetaData : [Secured] Patient [Provenance]; )
4 insert aspect1 into Anonymization with mand  // 'base' is modified
5 removeVariable aspect1 // no longer need 'aspect1' variable
6 fInt = parent MetaData // feature Anonymization is now in 'base' FM
7 assert ((name fInt) eq "Anonymization") // check it
8 // check feature MetaData in 'base' FM has still three child features
9 assert ((size (children MetaData)) eq 3)
```

In the example, the feature MetaData is inserted below the feature Anonymization (line 4): MetaData is a child feature of Anonymization with the mandatory status, i.e., the selection of Anonymization implies the selection of MetaData. The **assert** operations are performed to check that the insertion produced an feature model with correct properties.

Mode	Semantic properties	Mathematical notation	FAMILIAR notation
Intersection	$[\![FM_1]\!] \cap [\![FM_2]\!] = [\![FM_r]\!]$	$FM_1 \oplus_{\cap} FM_2 = FM_r$	**merge intersection** { fm1 fm2 }
Union	$[\![FM_1]\!] \cup [\![FM_2]\!] \subseteq [\![FM_r]\!]$	$FM_1 \oplus_{\cup} FM_2 = FM_r$	**merge union** { fm1 fm2 }
Strict Union	$[\![FM_1]\!] \cup [\![FM_2]\!] = [\![FM_r]\!]$	$FM_1 \oplus_{\cup_s} FM_2 = FM_r$	**merge sunion** { fm1 fm2 }
Diff	$\{x \in [\![FM_1]\!] \mid x \notin [\![FM_2]\!]\} = [\![FM_r]\!]$	$FM_1 \setminus FM_2 = FM_r$	**merge diff** { fm1 fm2 }

Table 8.1: Merge: semantic properties and notation

In line 5, the role of **removeVariable** is twofold. First, the variable identifier *aspect1* is no longer "active" in the rest of the script so that when the identifier *MetaData* is used in line 6, it is not ambiguous: it corresponds to the feature MetaData of *base*. Second, it releases the memory used by *aspect1*.

Merging feature models. The syntax of the **merge** operator is closed to the notation defined in Chapter 5 (see Table 8.1). The first parameter is the mode of the operator and the second parameter is a *Set* of feature models In Table 8.1, the properties of the merged feature model are summarized with respect to the sets of configurations of two input feature models and the mode.

In FAMILIAR, the merge operators act on (a set of) feature models[2] and produce feature models with semantic properties according to the mode specified by the programmer. Below is part of a script that uses a merge operator:

```
1 mi4 = FM ( MedicalImage: Modality Format Anatomy [Anonymized];
2          Modality: (v10.1|v10) ; Format: (NiftiII|Analyze) ; Anatomy: Brain;)
3 mi5 = FM ( MedicalImage: Modality Format Anatomy [Header];
4          Modality: (v10.1|v10|v9) ; Format: NiftiII ; Anatomy: (Kidney|Brain);)
5 mi_inter = merge intersection { mi4 mi5 }
6 mi_inter_expected = FM ( MedicalImage: Modality Format Anatomy ;
7          Modality: (v10.1|v10) ; Format: NiftiII ; Anatomy: Brain ;)
8 assert (mi_inter eq mi_inter_expected)
```

In line 5, the merge operator in intersection mode is applied on *mi4* and *mi5* and produces a new feature model that can be manipulated through the variable mi_inter. In line 8, we check that mi_inter is equal to $mi\_inter\_expected$. The binary operator **eq** is specific to variable complex types. In particular, two variables of feature model type are equal if *i)* they represent the same set of configurations, i.e., the **compare** operator applied to the two variables returns **REFACTORING** and *ii)* they have the same hierarchy.

```
9  mi_sunion = merge sunion { mi4 mi5 }
10 n_sunion = counting mi_sunion // number of valid configurations
11 n_expected = counting mi4 + counting mi5 - counting mi_inter
12 assert (n_sunion eq n_expected)
```

In line 9, the merge operator in strict union mode is applied on *mi4* and *mi5* and produces a new feature model that can be manipulated through the variable mi_sunion. In

[2]Configurations can be merged using **asFM**.

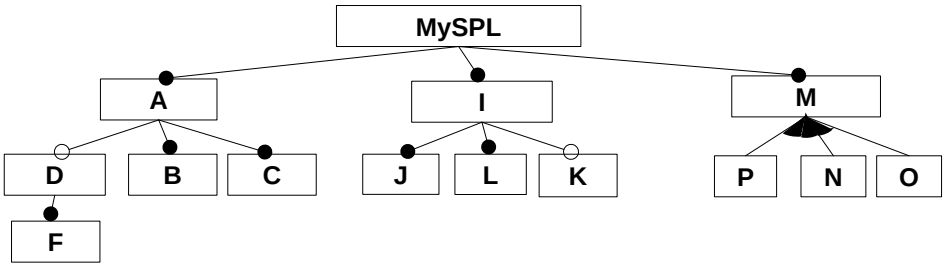


Figure 8.2: Aggregated feature model

lines 10-12, we check the following property:

$$[\![mi4]\!] \cup [\![mi5]\!]| = |[\![mi4]\!]| + |[\![mi5]\!]| - |[\![mi4 \cap mi5]\!]| = |[\![mi\_sunion]\!]|$$

using **counting** operations, i.e., the value of n is equal to $[\![mi\_sunion]\!]$.

Aggregating feature models. Another form of composition can be applied using cross-tree constraints between features so that separated feature models are inter-related. The operator **aggregate** is used for producing a new feature model in which a *synthetic* root relates a set of feature models and integrates a set of propositional constraints.

```
1  fm1 = FM (A : B [C] [D] ; D : (E|F) ; C -> !E; ) // E and F are mutually exclusive
2  fm2 = FM (I : J [K] L ; )        //
3  fm3 = FM (M : (N|O|P)+ ; ) // M, N, O, P form an Or-group
4  cst = constraints (J implies C ; )
5  fm4 = aggregate fm* withMapping cst // equivalent to aggregate { fm1 fm2 fm3 }
6
7  // simplify fm4 (e.g., by removing dead features)
8  fm5 = slice fm4 including fm4.*
9
10 op = operator fm5.F
11 assert (op eq mand) // mandatory status since E is no longer in the Xor-group
12
13  // renaming of the root feature
14 rfm5 = root fm5
15 renameFeature rfm5 as "MySPL"
```

All reasoning operations (e.g., **counting**, **isValid**) can be similarly performed on the new feature model resulting from the aggregation.

In particular, as we have seen in Section 5.3, when feature models are related through constraints, some features may be dead or core features. We use the corrective capabilities of the **slice** operator *i)* to remove dead features (here: `E` is removed due to the constraint $C \Rightarrow \neg E$) *ii)* to set the variability information (here: the feature `C` becomes mandatory due to the constraints). Figure 8.2 recaps the situation at the end of the script execution.

8.2.6 Modularization mechanisms

Statements are organized in scripts. FAMILIAR provides modularization mechanisms that allow for the creation and use of multiple scripts in a single SPL project, and that support the definition of reusable scripts.

Namespace and Script Calling. Variable name conflicts may occur, for example, when it is necessary to run the same script several times (see discussion on parameterized scripts below) or when features having the same name are used by feature models referred to by different variables. FAMILIAR relies on namespaces to allow disambiguation of variables having the same name. By default, a namespace is attached to each variable of type feature model so that it is possible to identify a feature by specifying the name of the variable of type feature model followed by "."

```
1 children mi1.Modality // explicit notation needed
2 mi2.MedicalImage // MedicalImage exists also in mi1 and mi3
3 parent MRI // non ambiguous: equivalent to mi3.MRI
```

Lines 1-3 illustrate the use of namespace: features Modality and MedicalImage are present in two feature models, *mi1* and *mi2*, and are identified thanks to the namespace. Note that MRI appears only in *mi3* and is non-ambiguous so that there is no need to explicitly use the namespace.

Namespaces are also used to logically group related variables of a script, making the development more modular. The example below is used to illustrate how FAMILIAR supports the reuse of existing scripts:

```
1 run "fooScript1" into script_declaration
2 varset = script_declaration.*
3 export varset
4 hide script_declaration.mi*
```

Line 1 shows how to run a script contained in the file *fooScript1* from the current script. The namespace $script\_declaration$ is an abstract container providing context for all the variables of the script *fooScript1*. In addition, FAMILIAR allows a script programmer to use a wildcard "*" to access a set of elements (e.g., feature models, features). It may be placed just after "." or anywhere within a variable or feature name. For example, line 2 (resp. 4) accesses the set of all variables of $script\_declaration$ (resp. all variables starting by mi in $script\_declaration$). By default, a script makes visible to other scripts all its variables. Using **export** with several variable names means that only those variables remain visible. Using **hide** instead means that all variables mentioned are not visible.

Parameterized Script. A script can be parameterized using an ordered list of **parameter**s (see lines 2-3 below). A parameter records a variable and, optionally, the type expected.

```
1 // fooParameterizedScript.fml : a parametrized script
2 parameter fm1 : FeatureModel
3 parameter fm2 : FeatureModel // type specification is optional
4
5 d1 = deads fm1
```

```
d2 = deads fm2
//... comparison of dead features
c1 = cores fm1
c2 = cores fm2
//... comparison of core features
```

The example below illustrates how the parameterized script can be called:

```
newMI = FM (MedicalImage: Anatomy [Header]; Anatomy: (Brain|n256);)
newMI2 = copy newMI
setMandatory newMI2.Header // editing facilities: Header is now a mandatory
//  feature in newMI2
run "fooParameterizedScript" { newMI newMI2 }
```

8.3 An Application to the Management of Multiple SPLs

In this section we illustrate how FAMILIAR can be used to support the management of multiple SPLs[3] and in particular provide a practical solution to the approach developed in Chapter 6.

8.3.1 First Management of the Multiple SPL

We recall that in a *competing* multiple SPL, each constituent SPL describes a different family of products (e.g., services) in the same market segment (e.g., medical imaging domain) produced by competing suppliers (e.g., research teams). The example scenario that will be used to illustrate how FAMILIAR can be used to manage multiple SPLs is presented in Figure 8.3. The scenario involves three steps. In the first step the medical imaging ex-

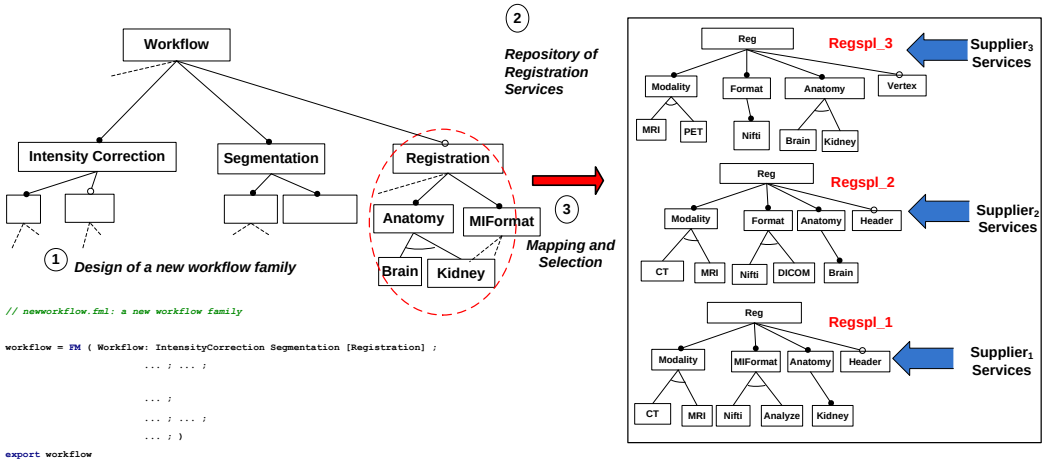


Figure 8.3: Managing Multiple SPLs

pert produces a feature model with no assumptions about the parts provided by external suppliers – see ①. In the next step, this family of workflow is viewed as an aggregation of

[3] In [Acher et al. 2011b], we have also shown how FAMILIAR can be applied using a "laptop" example.

several competing multiple SPLs for different parts of the workflow: for example, Intensity Correction, Segmentation, Registration. In the scenario, the need for an external supplier for a registration medical imaging service is identified. This requirement results in the generation of competing multiple SPLs that represents the offerings of the three suppliers for the Registration (see ②). The SPL of each supplier's service family is built from its services. After the requirements of the medical imaging expert is mapped with the one of suppliers (see ③), application-specific processing should be performed. The medical imaging expert can use competing multiple SPLs to *i)* determine whether the set of suppliers is able to provide the entire set of services and to cover all combinations of features or to *ii)* identify missing services, and to *iii)* eliminate the suppliers that do not provide the required services.

Building SPLs' Repositories. In the script below, $Supplier_1$ proposes eight registration services, each one being distinguished from the others by features. (Note that we can use the same name convention for all suppliers since namespaces are used to disambiguate names.) The set of registration services can then be organized (e.g., grouped together) by the medical imaging expert within an SPL.

We consider that services exhibit no variability: Each registration service description is represented as a feature model in which all features are mandatory.

```
1 // RegSupplier_1.fml: registration services' specification of Supplier_1
2 Regservice1 = FM ( Reg: Modality Format Anatomy ; Modality: MRI  ; ...)
3 Regservice2 = FM ( Reg: Modality Format Anatomy ; Modality: CT ; ...)
4 ...
5 Regservice8 = FM ( Reg: Modality Format Anatomy Anonymized ; Modality: CT ; ..)
```

Building an SPL from the set of existing services can be done by applying the merge operator in strict union mode on the set of corresponding feature models (line 3 in the script below). A feature model is then produced for each supplier (e.g., $Regspl\_1$ feature model for $Supplier_1$).

```
1 // repositoryReg.fml: repository of registration services
2 run "RegSupplier_1" into Regsupp1
3 Regspl_1 = merge sunion Regsupp1.*
4 run "RegSupplier_2" into Regsupp2
5 Regspl_2 = merge sunion Regsupp2.*
6 run "RegSupplier_3" into Regsupp3
7 Regspl_3 = merge sunion Regsupp3.*
8 renameFeature Regspl_3.MIFormat as "Format" // aligning terms
9 export Regspl_* // export the three suppliers' FMs/SPLs
```

A repository of registration services is now represented by a set of different feature models (one per supplier). Similarly, other repositories can be built for other kinds of services (intensity correction, segmentation, etc.) and imported in a script (see lines 3-5 below).

```
1 // workflowScenario.fml: implementation of suppliers' scenario
2 run "newworkflow" // new family of workflow firstly designed
3 // load repositories
```

```
4 run "repositoryReg" into Reg
5 run "repositorySegm" into Segm
6 run "repositoryIntensity" into intensity
```

Mapping Repositories to the SPL. The script below describes mappings from SPLs in repositories to the medical imaging expert's SPL. For each part of the family of workflow (handled by the script *newworkflow.fml* – see **run** command line 1 and Figure 8.3), it is necessary to determine which SPLs and suppliers are suitable to provide the given service. Indeed, the family of workflow specifies different alternatives for the choice of a registration service. In order to reason about the registration service of feature model $workflow$, the variability information of the registration service (sub-tree rooted at feature Registration) is first extracted. $originalReg$, resulting from the **slice** (line 10), is a copy of the sub-tree and can be manipulated as a feature model. In particular, each valid configuration of $originalReg$ should correspond to at least one service provided by a supplier and present in the registration service repository, i.e., the following relation (C_1) should hold:

$$[\![originalReg]\!] \cap ([\![Regspl\_1]\!] \cup [\![Regspl\_2]\!] \cup [\![Regspl\_3]\!]) = [\![originalReg]\!]$$

It may happen that such a property is not respected and two cases have to be considered. First, the intersection between the set of services of $originalReg$ and the set of suppliers' service may be empty (lines 9-16). Verifying this property can be done by first performing the merge operations on $originalReg$, $Regspl\_1$, $Regspl\_2$ and $Regspl\_3$ (line 16). This gives a new feature model mi_merged and its satisfiability can then be controlled, i.e., checking whether or not there is at least one valid configuration (lines 17-20).

Another possibility is that the family of workflow offers to medical imaging experts some services that cannot be entirely provided by suppliers (lines 22-27). In this case, $originalReg$ is a *generalization* or an *an arbitrary edit* of the union of set of suppliers' services (line 22). Performing a **merge diff** operation (see Table 8.1) assists users in understanding which set of services is missing (line 23-26).

```
7  // we map the service Reg description with the service registration repository
8  allServicesReg = merge sunion Reg.* // Regspl_1, Regspl_2, Regspl_3
9  originalReg = slice workflow.Registration* // extraction of the subtree
10
11 // alignment: renaming terms to be coherent with the repository
12 renameFeature originalReg.MIFormat as "Format"
13 renameFeature originalReg.Registration as "Reg" //...
14
15 /***** checking the availability of services *****/
16 mi_merged = merge intersection { originalReg allServicesReg }
17 if (not (isValid mi_merged)) then
18         print "No service can be provided"
19         exit // stop the program
20 end
21 cmp_mi = (compare originalReg allServicesReg)
22 if (cmp_mi eq GENERALIZATION || cmp_mi eq ARBITRARY) then
23   mi_losed = merge diff { originalReg allServicesReg } // missing services
24   s_losed = configs mi_losed // set of configurations of mi_losed
```

```
25    n_losed = counting mi_losed // number of configurations
26    println n_losed " service(s) cannot be provided: " s_losed
27  end
28  assert (cmp_mi eq REFACTORING || cmp_mi eq SPECIALIZATION)
```

At this step, *all* services of $originalReg$ can be provided by suppliers. For example, the relation holds considering the feature models depicted in Figure 8.4. Indeed, the set of configurations of $originalReg$ is included or equal to the union of set of suppliers' services since, according to set theory, the relation (C_1) is equivalent to $[\![originalReg]\!] \subseteq ([\![Regspl\_1]\!] \cup [\![Regspl\_2]\!] \cup [\![Regspl\_3]\!])$. We check this property in line 27.

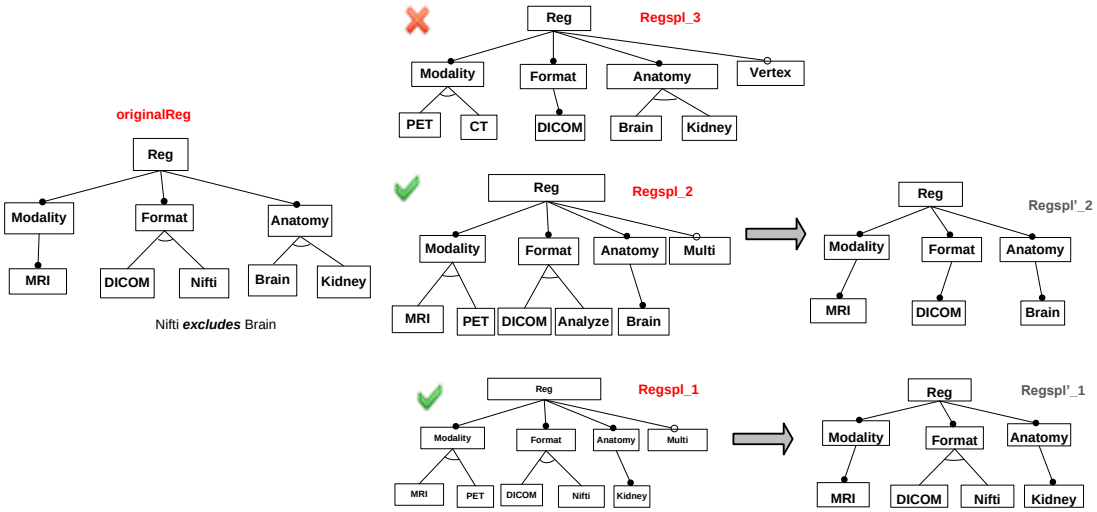


Figure 8.4: Available suppliers and services.

Selecting Suppliers. At this point, the medical imaging expert needs to determine which suppliers can provide a subset of the services of $originalReg$ (lines 28-40). Some suppliers cannot provide at least one service corresponding to any configuration of $originalReg$ (lines 31-33) and so should not be considered. Figure 8.4 illustrates the situation: $Supplier_3$ is no longer available since the intersection between $[\![originalReg]\!]$ and $[\![Regspl\_3]\!]$ is the empty set. Some suppliers offer services that correspond to a valid configuration of $originalReg$ but also offer out-of scope services. To remove these services, a merge in intersection mode is systematically performed to restrict attention to the set of relevant supplier services (line 31). For example, the feature Multi is no longer included in the set of services of $Supplier_1$ and $Supplier_2$ (see $Regspl'\_1$ and $Regspl'\_2$ in Figure 8.4) while $Supplier_2$ is now able to deliver only one service.

```
28  Reg_suppliers_in = setEmpty // create an empty set
29  foreach (supplReg in Reg.*) do
30          // checking each supplier providing Regs
31          fmReg_inter = merge intersection { originalReg supplReg }
32          bReginter = isValid fmReg_inter
```

```
33          if (not bReginter) then
34             println "The supplier is unable ...:\t" supplReg
35          else
36             setAdd Reg_suppliers_in fmReg_inter // add relevant FM
37          end
38  end
39  assert ((size Reg_suppliers_in) >= 1) // available supplier >= 1
```

At the end of the loop, $Reg\_suppliers\_in$ corresponds to a set of feature models, where each feature model represents a valuable service line of a supplier. Then, $originalReg$ can be configured: a similar sequence of merge operations on configurations can be executed until an unique service of a supplier is chosen.

```
41  miService = configuration originalReg // configuration process
42  select Brain in miService // ...
```

8.3.2 Towards Reusable Scripts

The FAMILIAR script *workflowScenario.fml* can fully realize the scenario of Figure 8.3. Nevertheless, this script has some limitations. First, when users select or deselect features, some suppliers may become unable to provide services corresponding to the new requirements. We want to perform validity checks at each step of the configuration process. An approach where sequences of code are copied from above and pasted in again is not desirable. Second, reasoning operations are planned to be performed on each part of the workflow (as done with the registration service). The current script does not allow a programmer to reuse the repetitive tasks performed on feature models, leading again to duplicate codes. We thus propose to apply modularization mechanisms provided by FAMILIAR to raise the limitations mentioned above.

Modularizing the code. The script *workflowScenario2.fml* (see below) is a refactoring of the code *workflowScenario.fml*. The code used in line 1-13 remains the same: repositories of services are built; the workflow feature model describing the valid combinations of features is specified; some alignment operations are performed so that one can reason about the workflow feature model and the feature models of the repositories. This time, the checking operations are performed in two parameterized scripts: *checkAvailability.fml* and *availableSuppliers.fml* (see below).

```
14  // workflowScenario2.fml: lines 1-13 remain the same
15  /***** checking and inform which available suppliers are *****/
16  run "checkAvailability" { originalReg allServicesReg } into
17  mi_availability
18  if (mi_availability.available) then
19          run "availableSuppliers" { Reg.* originalReg } into
20  mi_suppliers
21  end
```

The script *checkAvailability.fml* compares the set of configurations of two feature models, $originalFM$ and $allServices$, which are two parameters of the script. It controls whether or not each valid configuration of $originalFM$ is also valid in $allServices$. It also provides some feedback to users, for example, the number of configurations valid in $originalFM$

but invalid in $allServices$. When the set of configurations of $originalFM$ is fully covered by $allServices$, the boolean variable $available$ is set to **true** and exported. Then, this variable is accessible from the calling script, i.e., *workflowScenario2.fml* (see line 18).

```
1  // checkAvailability.fml : is the set of services fully covered
2  parameter originalFM : FeatureModel
3  parameter allServices : FeatureModel
4
5  fm_merged = merge intersection { originalFM allServices }
6  available = true
7  if (not (isValid fm_merged)) then
8          available = false
9          println "No service can be provided: "
10         exit // stop the script execution
11 end
12 cmp = compare originalFM allServices
13 if (cmp eq GENERALIZATION || cmp eq ARBITRARY) then
14    available = false
15    fm_losed = merge diff { originalFM allServices } // missing services
16    println (counting fm_losed) " service(s) cannot be provided: " ...
17 end
18 export available
```

In line 17 of *workflowScenario2.fml*, each combination of features of a service (e.g., registration service) corresponds to at least one service from the repository. We want to determine which specific suppliers are able to offer such services. A parameterized script *availableSuppliers.fml* is used to perform the needed operations. The first parameter, named $suppliers$, is a set of feature models and represents the suppliers' offer: a feature model of the set corresponds to the offer of one supplier. The second parameter, named $services$, is a feature model representing the service specification. For each feature model $suppl$ that belongs to the set $suppliers$, we determine whether or not some valid configurations of $services$ are valid in $suppl$. If this is not the case, this means the supplier is unable to provide any service and feedback is given to users. At the end, the script displays the available suppliers and the corresponding feature models are updated.

```
1  // availableSuppliers.fml: suppliers able to provide services
2  parameter suppliers : Set // set of feature models
3  parameter services : FeatureModel
4
5  suppliers_in = setEmpty
6  foreach (suppl in suppliers) do
7          // checking suppliers' offer
8          fminter = merge intersection { services suppl }
9          if (not (isValid fminter)) then
10           println "The supplier is unable to provide ...:\t" suppl
11         else
12           setAdd suppliers_in fminter // add relevant FM
13         end
14 end
15 nsuppliers = size suppliers_in
```

```
println nsuppliers " suppliers are available:"
foreach (supp in suppliers_in) do
        println supp
end
```

Resulting benefits. The FAMILIAR script code has been modularized. We can raise the limitations described above. At each step of the configuration process, users can have feedback from available suppliers (even if the set of features is not fully selected or deselected). In addition, for each service of the workflow (segmentation services, intensity correction services, etc.), similar checking operations can be performed by reusing parameterized scripts.

```
/**** workflowScenario2.fml: multi-step configuration ****/
miRegService = configuration originalReg
select DICOM in miRegService
miRegFM = asFM miRegService // convert a configuration to a FM
run "availableSuppliers" Reg.* miRegFM
// configuration continues until a supplier's service is chose
```

8.4 SUMMARY

In this chapter, we have discussed why a textual domain-specific language (DSL) seems particularly adequate to the domain of propositional feature models. The main features of FAMILIAR, a DSL to manage feature models, have been introduced. FAMILIAR provides support for our composition and decomposition operators, but not only: A set of complementary operators for editing and reasoning about feature models has also been presented. We have described how language constructs allow a feature model user *i)* to control when and how (de-)composition and analysis mechanisms are applied, *ii)* to replay and reuse (de-)composition and analysis procedures. We have illustrated the capabilities of FAMILIAR with a non trivial scenario in the medical imaging domain.

Nine

Implementation Details and Performance Evaluation

In this chapter, we briefly described the implementation of FAMILIAR (parsing process, architecture, connection with other APIs, environment) and we evaluate the performance of two important operators, *merge* and *slice*.

9.1 Implementation Details

9.1.1 Architecture Overview

FAMILIAR is developed in Java language using Xtext [Xtext 2011], a framework for the development of external DSLs. The term *language workbench* is sometimes used since Xtext supports DSL creation not just in terms of parsing and code generation but also in providing a better editing experience for DSL users (syntax highlighting, cross-reference resolving, validation, formatting, outline-view, code-completion, rename refactoring, etc.). Xtext is fully integrated to the Eclipse platform and builds on Eclipse Modeling Framework [1] (EMF). We use Xtext to parse FAMILIAR scripts, obtain a representation of an DSL script that we can analyze and interpret using the framework facilities of Xtext (see Figure 9.1). In Xtext, the representation of an DSL script is an EMF-based *model* (also called semantic model in [Fowler 2009; 2010]), that captures all the important semantic behavior.

A semantic model is usually distinguished from an abstract syntax tree (AST) because they serve separate purposes. A syntax tree corresponds to the structure of the DSL scripts. Although an AST may simplify and somewhat reorganize the input data, it still takes fundamentally the same form. The semantic model, however, is based on what will be done with the information from a DSL script. It often will be a substantially different structure, and usually not a tree structure. In Xtext, the semantic model and the AST are rather equivalent. The minor difference is that the semantic model is actually a graph rather than a tree, since it also contains crosslinks.

Xtext derives a EMF-based meta-model from the grammar definition (see Figure 9.1). As a result, each semantic model is an instance of EMF-based meta-model. Furthermore, Xtext generates an Antlr-Parser from the grammar definition and automatically converts the AST into the semantic model. In Xtext, the parser creates the EMF-based semantic model or the abstract syntax tree from the textual representation of the model. Many components, including our interpreter, work on this semantic model rather than directly on the textual representation.

[1] http://www.eclipse.org/modeling/emf/

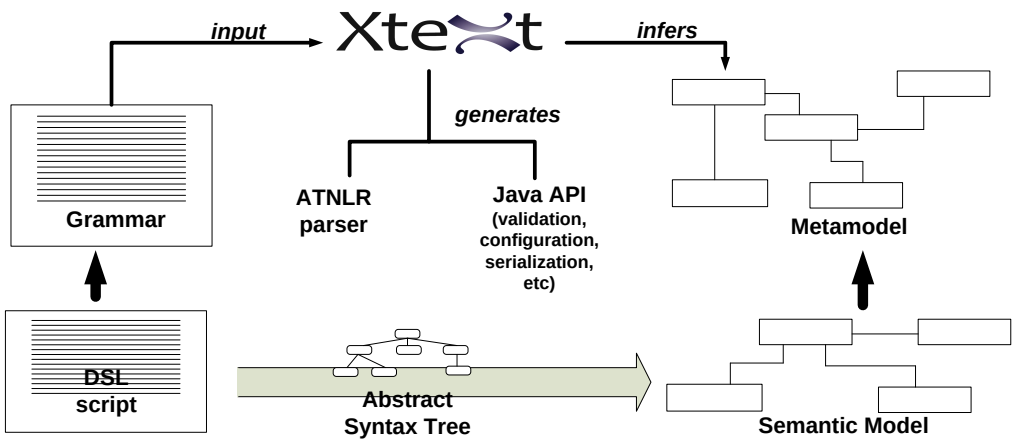


Figure 9.1: Architecture of DSL processing

9.1.2 Environment and Ecosystem

We provide an Eclipse text editor (including syntax highlighting, formatting, code-completion, etc.) and an interpreter that executes the various FAMILIAR scripts. An interactive toplevel is also available, connected with graphical editors.

Moreover several "bridges" have been developed to import and serialize feature models (see Figure 9.2). The connection with other tools and languages has several benefits.

From interoperability to Reuse. First, several notations can be used for specifying feature models (SPLOT [Mendonca et al. 2009a], GUIDSL/FeatureIDE [Batory 2005, Kästner et al. 2009a], a subset of TVL [Boucher et al. 2010], etc.). The proposed bridges allow users to import feature models or configurations from their own environments and encourage *interoperability* between feature modeling tools.

Second, as a feature model or a configuration can be exported (using the **save** operation), outputs generated by FAMILIAR can be processed by other third party tools, for example, modeling tools when we need to relate feature models to other models [Czarnecki and Antkiewicz 2005, Heidenreich et al. 2010].

Third, and perhaps more importantly, the support of different formats allows one to easily reuse state-of-the-art operations already implemented in existing tools. We reuse the technique to reason about edits described in [Thüm et al. 2009] and implemented in FeatureIDE [Kästner et al. 2009a] to implement the **compare** operator. Feature-model-synthesis [2] implements an algorithm to synthesize a feature model from a propositional formula (as described in [Czarnecki and Wąsowski 2007]). We reuse and adapt some techniques of the algorithm to implement the merge and the slice operators (see Chapter 5 and Chapter 7). Moreover, we reuse some heuristics developed in SPLOT [Mendonca et al. 2009a] to compile large feature models into BDDs.

[2] Feature-model-synthesis is developed by Steven She (University of Waterloo, Canada) and Andrzej Wąsowski (IT University of Copenhagen) and is available at `https://bitbucket.org/mintcoffee/feature-model-synthesis`. We take this opportunity to thank Steven She and Andrzej Wasowski for early sharing with us their implementation.

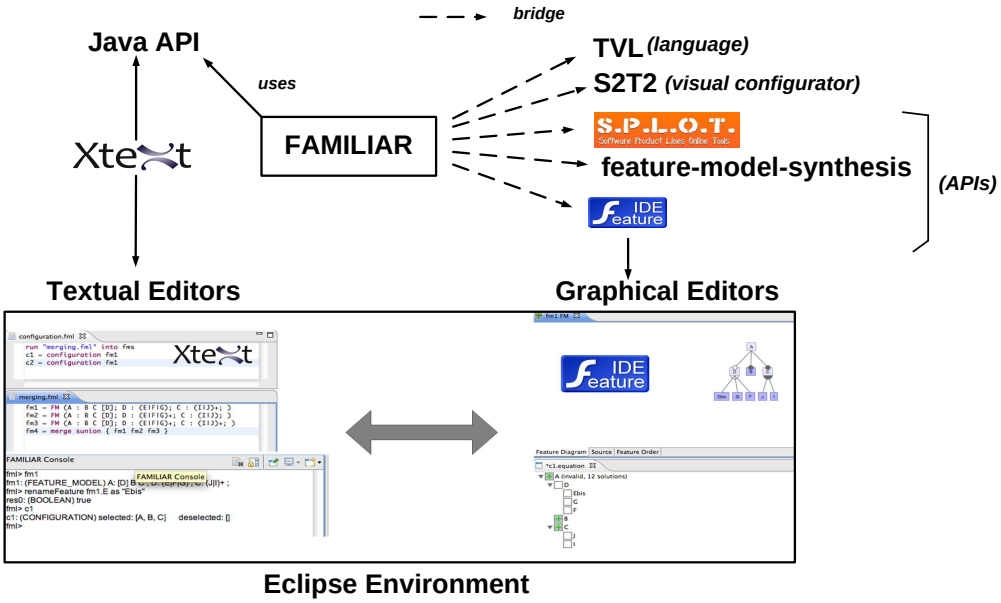


Figure 9.2: FAMILIAR Infrastructure and Ecosystem

Fourth, we reuse graphical facilities. The connection with the FeatureIDE framework allows us to reuse the graphical editors (for editing and configuring feature models). All graphical edits are synchronized with variables environment and all interactive commands are synchronized with the graphical editors. Similarly, we reuse SPLOT facilities to initiate the development of a web-based FAMILIAR environment. We also developed a bridge with S2T2[3], a configurator developed at Lero.

Cross-checking of operations. In order to test the correctness of the operator implementations, we develop a comprehensive set of unit tests, complemented by cross-checked testing with other operations provided by FAMILIAR. For example, many APIs, including ours, propose the counting operation (the APIs of TVL [Boucher et al. 2010], FeatureIDE and SPLOT). We verify that all implementations of the operator compute the same result.

9.1.3 Reasoning Backend

The set of configurations represented by a feature model can be described by a propositional formula defined over a set of Boolean variables, where each variable corresponds to a feature. From the propositional formula, various reasoning operations can be automated and performed. Two reasoning backends are internally used in FAMILIAR: SAT solvers and Binary Decision Diagrams.

SAT solvers. The satisfiability problem (SAT) is the problem of determining for a given for-

[3] http://download.lero.ie/spl/s2t2/

mula whether there is an assignment such that the formula evaluates to true or not. A SAT solver is a tool that decides the satisfiability problem. Most SAT solvers require the input formulas to be in Conjunctive Normal Form (CNF). If a SAT solver is given a satisfiable formula, the solver returns a satisfying assignment (a model) of the formula. If the given formula is unsatisfiable, the solver returns a proof for the unsatisfiability. Despite the NP-completeness of the satisfiability problem, nowadays SAT solvers are extremely efficient and are able to deal with formulas with thousands of variables and are still improving as illustrated by the yearly SAT Competition [http://www.satcompetition.org/ 2011].

Definition 18 (Formula, literal, clause, CNF)**.** *A propositional formula is called a literal iff it is a variable or a negated variable. A formula is called a clause iff it is a disjunction of zero of more literals. Each variable must appear at most once in a clause. A formula is in the conjunctive normal form (CNF) iff it is a conjunction of clauses.*

Binary Decision Diagrams (BDDs). BDDs are widely used in digital system design [Minato 1996], model checking and also in the feature modeling community (e.g., [Czarnecki and Wąsowski 2007, Mendonca et al. 2008, Benavides et al. 2010, Classen et al. 2011]). Efficient optimized implementations of these operations are provided by off-the-shelf BDD libraries (e.g., we use the open source JavaBDD library [JavaBDD 2007]).

A BDD is a rooted directed acyclic graph with nodes labeled by names of input variables, except for two special leaf nodes labeled 1 and 0, corresponding to True and False. A BDD is a compact representation of a Boolean function on a particular ordering of input variables. To obtain the value of the formula under a certain assignment, one follows the appropriate edges until reaching one of the leaves. In the rest of the thesis and in conformance with the literature, the term BDD always refers to Reduced Ordered Binary Decision Diagram (ROBDDs). ROBDDs are canonical: that is, for every Boolean function and fixed variable ordering, there is a unique logically equivalent ROBDD.

Polynomial time algorithms are available for computing the negation, conjunction, disjunction or existential quantification of BDDs [Brace et al. 1990], for computing valid domains during the configuration process or for checking equivalence between two BDDs. The response time of most of standard algorithms on BDDs requires time proportional to the *size of the BDD*. The size of a BDD is the number of nodes manipulated in the graph structure and depends on the binary function being represented as well as the chosen variable ordering.

A major drawback of BDDs is their high sensitivity to variable ordering, i.e., finding an optimal variable ordering during BDD construction is NP-hard [Brace et al. 1990, Bollig and Wegener 1996, Minato 1996]. This ordering has a dominant influence on the size of BDDs: A bad order can lead to excessive memory use, often beyond capabilities of typical technologies. Some heuristics have been developed and outperformed naive encoding of feature models into BDDs. It has been reported that these techniques successfully compile feature models to BDDs for a number of features up to 2000 [Mendonca et al. 2008]. We thus reuse the heuristics (i.e., Pre-CL-MinSpan) developed in [Mendonca et al. 2008] to reduce the size of BDDs.

BDD and/or SAT. Even though there has been an enormous increase in computing power in the last decade, the problems of feature combinatorics remain NP-hard and can take

a long time to solve. Not all representations will perform equally well [Darwiche and Marquis 2002]. Both BDDs or SAT solvers have been used to perform various reasoning operations on feature models, for example, to determine if a feature model contains at least one product or how many valid configurations it describes (see Chapter 3).

FAMILIAR provides, for most of the reasoning operations (**counting**, **isValid**, etc.), two implementations that rely either on BDDs or SAT solvers. For the implementation of the merge and the slice operators, our current effort is limited to an implementation based on BDDs. Challenges of an implementation based on SAT solvers of these operators are discussed at the end of the chapter.

9.2 EVALUATING THE PERFORMANCE OF OPERATORS

The investigation of practical performance of operators should allow for answering the following questions:

- which tools should be used and when?
- can the choice of which tools to use be automatically made to minimize the time to analyze feature models?
- is it necessary to integrate different solvers?

Benavides et al. analyzed the performance of Constraint Satisfaction Problem (CSP), SAT and BDD solvers in finding valid configuration given a feature model [Benavides et al. 2006]. For example, they show that BDDs are faster than CSP or SAT solvers, but with a ten times higher memory usage. We have retrieved the initial observations of Benavides et al. For example, we have experimented the **counting** operation of FAMILIAR using either SAT or BDD: SAT solver does not scale for counting all solutions of feature models (difficulties arise even for feature models with more than 200 features) whereas BDDs are much faster in counting all solutions and scale better.

9.2.1 Merge Operators

Implementation Details. We implemented the merging operators presented in Section 5.4.2 using BDDs. As mentioned above, computing the negation, conjunction or disjunction of BDDs can be performed in at most polynomial time with respect to the sizes of the BDDs involved. These logical operations are used extensively during the merging of several feature models.

Experimental Setup. We evaluate how the size of the models affects performance of merge operators that are implemented using BDDs. We use randomly generated feature models to produce inputs with variations on *i)* the number of constituent SPLs (noted nFM) in the multiple SPL, *ii)* the number of features *commonly* shared by SPLs (noted $nComm$), and *iii)* the percentage of features *commonly* shared by SPLs (noted per). The variation of $nComm$ has an impact on the total number of features *within* each feature model whereas per allows us to vary the features commonly shared *between* feature models. We make per vary between 100% (which corresponds to the case where all features are commonly shared) and 50%, so that we can analyze the merge behavior when all features are not necessarily shared by feature models. As we are dealing with competing SPLs, the percentage of shared features is intended to be rather high. For our setup, we consider 50% is

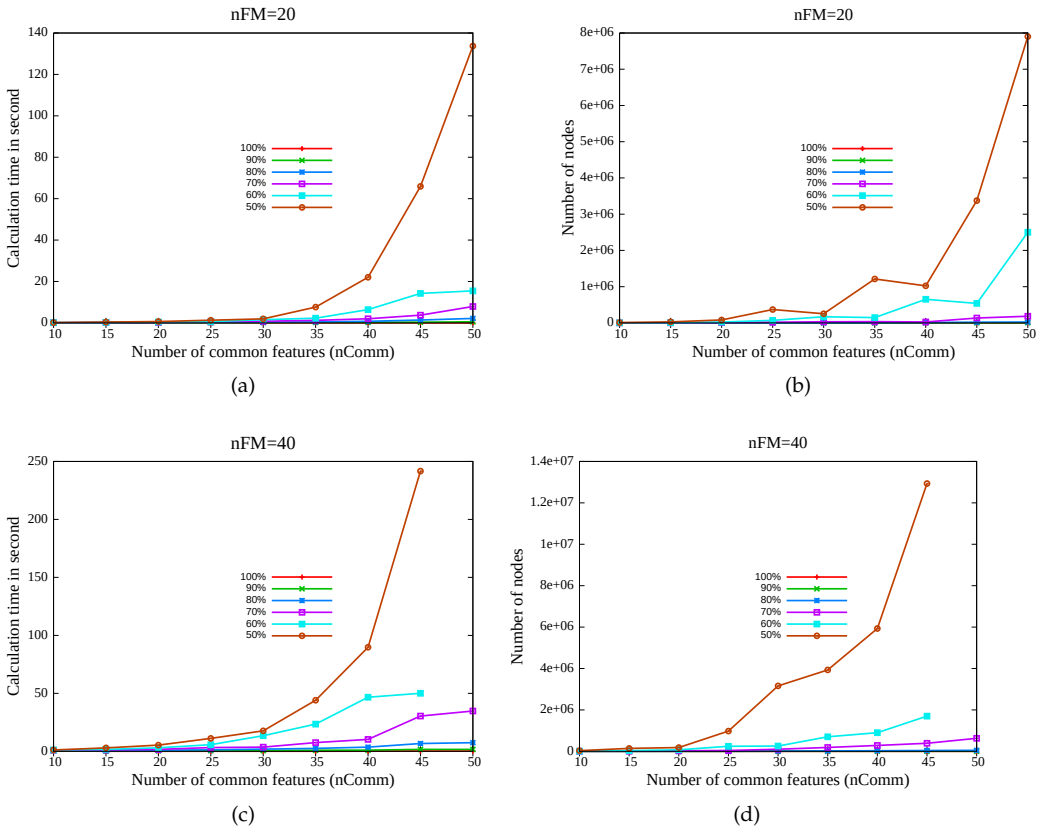


Figure 9.3: Calculation time and space complexity (strict union merge)

a reasonable[4] value under which a set of SPLs are not likely to be grouped together, especially for the case of competing multiple SPL described in Chapter 6. For each value of per, we experimented with different values of nFM, to determine how many feature models can be handled by the merge, and of $nComm$, to determine the manageable size of input feature models. We chose to focus on the common features of each feature model since the goal of the merge operators is to group two or $nComm$ features with the same name into one feature. We measured the calculation time needed to perform logical operations on BDDs. For input feature models, we randomly created an initial feature model and performed random edits (similarly as in [Thüm et al. 2009]) to obtain new input feature models. Then, features are randomly added to these feature models. This way, we can parametrically control nFM, $nComm$ and per.

Results. In Figure 9.3, we report our results when $nFM = 20$ and $nFM = 40$, with merging operations performed in strict union mode. We also performed experiments on

[4]It must be noted that this value is not backed by any empirical study. We only use it to restrict our experimentations to parts that look the more relevant to us.

the merge operator in intersection mode and we observed similar results as with the strict union mode. Only an excerpt of figures is given. All experimental results for $10 \leq nFM \leq 200$, $50 \leq per \leq 100$ and $10 \leq nComm \leq 200$ are available at https://nyx.unice.fr/projects/familiar/wiki/mofm.

Experiments show that when all features are commonly shared by SPLs ($per = 100\%$), the calculation time is almost linear to the number of features and the implementation scales even for nFM greater than 200 and for $nComm$ greater than 200. With $per \leq 60\%$ scalability issues occur when $nFM = 40$ and for $nComm \geq 45$ (see Figure 9.3). For these values, the logical operations fail since data overflows the main memory.

In addition to the calculation time, we measured the memory space, i.e., the size of the resulting BDD. As features with the same name are encoded into the same Boolean variable, the total number of Boolean variables, noted $nBooleanVariables$, can be deduced from $nComm$, nFM and per using the following relation: $nBooleanVariables = nComm + (nFM * nComm(\frac{1-per}{per}))$.

The experimental results show that *i)* the increase of the number of nodes is proportional to $nBooleanVariables$, and *ii)* the computation time is almost linear to the size of the BDD. As a result, the merge operations can efficiently manage a large number of SPLs even with a lot of features, especially when the SPLs share a large amount of common features (between 80% and 100%). In this case, the number of Boolean variables encoded in the resulting BDD appears to be tractable.

Bottleneck. In the presented experimental results, we only consider the use of logical operations on BDD, as it is a compact representation of the configuration set of the expected feature model. For a majority of operations on feature model (e.g., consistency checking), reasoning about the configuration set is sufficient, whereas the hierarchy is only needed from a user perspective (e.g., visualization or selection of features). We therefore do not take into consideration the time needed to reconstruct the hierarchy and the structuring information of the feature model. Experiments indicate that on typical Boolean formulas, the reconstruction algorithm presented in [Czarnecki and Wąsowski 2007] scales up to 800 variables[5], for example, the number of common features should not exceed 300 when $per = 100\%$.

9.2.2 Slice Operator

Complexity and Performance. The proposed *slice* algorithm relies on propositional formula and feature model hierarchy computations. Hierarchy computations use standard graph techniques and are not an issue, even for very large feature models. The handling of logical operations relies on Binary Decision Diagrams (BDDs) [Brace et al. 1990]. As mentioned above, an important property of BDDs is that computing their existential quantification (e.g., computing ϕ_{slice}) can be performed in at most polynomial time with respect to the sizes of the BDDs involved [Brace et al. 1990].

Genuine feature models. For a first evaluation, we used several small and medium-sized

[5]Janota et al. reported that the BDD-based algorithm proposed in [Czarnecki and Wąsowski 2007] scales up only for feature models with 300/400 features [Janota et al. 2008], but did not use the heuristics proposed in [Mendonca et al. 2008].

#features / CTCR	10	20	30	40	50	60	70	80	90	100
100	100	100	100	100	100	100	100	100	100	100
300	100	100	100	100	100	100	100	100	100	100
400	100	100	100	100	100	100	100	100	90	$\times_B$
500	100	100	100	100	100	100	90	80	70	$\times_B$
800	100	90	80	80	70	70	60	60	50	$\times_B$
1000	100	80	80	70	60	50	50	$\times_B$	$\times_B$	$\times_B$
2000	35	$\times_B$	$\times_B$	$\times_B$	$\times_B$	$\times_B$	$\times_B$	$\times_B$	$\times_B$	$\times_B$
> 2000	$\times_B$	$\times_B$	$\times_B$	$\times_B$	$\times_B$	$\times_B$	$\times_B$	$\times_B$	$\times_B$	$\times_B$

Figure 9.4: Scalability of the algorithm wrt. the number of features in a feature model and CTCR

feature models that were publicly available from SPLOT [Marcilio Mendonca 2009] repository as well as feature models from our case studies (see Part IV). We performed our experiments on more than 100 feature models, the bigger one having 290 features. An important parameter of the slicing operator is the number of features within the slicing criterion, noted n_{slice}. It varies from 1 to n (n is the number of features in the feature model). We note per_{slice} the ratio n_{slice} / n expressed as a percentage (e.g., 100% when the slicing criterion contains all features of the feature model). We found that computing the slice (including the feature diagram synthesis) is almost instantaneous in all cases (i.e., for $1 \leq per_{slice} \leq 100$).

Generated feature models. Even though industry reports feature models with several hundreds of features, we did not have access to such feature models. Therefore, we randomly generated them. We varied *i)* the number of features, noted $\#features$, from 100 to 2000 features (the practical limits of BDD) ; *ii)* the ratio of the number of features in the constraints to the number of features in the feature hierarchy expressed as percentage, noted $CTCR$, varies from 10% to 100%. We used the procedure described in [Marcilio Mendonca 2009] and publicly available to randomly generate the feature models. In each generated model, each type of mandatory, optional, Xor and Or-groups was added with equal probability. The cross-tree constraints were generated as a single Random 3-CNF formula. Results are shown in Figure 9.4. For each pair of value $(\#features, CTCR)$ the maximum value of per_{slice} for which the algorithm scales up is reported while $\times_B$ means that the BDD cannot be compiled. The results show that:

- whenever a feature model can be represented as a BDD, and for all values of per_{slice}, ϕ_{slice} can be computed. Hence the encoding of ϕ_{slice} can scale up to 2000 features with a CTCR of 10 ;
- the synthesis of the feature diagram has practical limits (up to 800 features in the slicing criterion). However it can scale even for a feature model with 2000 features if the percentage of features to slice is $\leq$ 35%. The reason is that the size of a BDD will always be smaller or at least unchanged after existential quantification ;
- the algorithm scales for a large CTCR and large number of features, which is important as we want to handle complex cross-tree constraints ;
- the primary limit of the BDD-based implementation lies in the difficulties to compile BDD from the original feature model.

9.2.3 Threats to Validity.

External threats. Threats to external validity are conditions that limit our ability to generalize the results of our experiments to industrial practice. Our primary concern is whether generated feature models and other variables of the experiments (i.e., slicing criteria, edits of feature models) are representative of industrial usage. Our case studies show that feature models with CTRC > 50 exist but currently we have not observed publicly available feature models with more than 300 features.

Internal threats. A first threat concerns the correctness of the slicing and merge operator implementation. In particular, the slicing operator is supposed to guarantee that some semantic properties are preserved. We develop a comprehensive set of unit tests, complemented by cross-checked testing with other operations provided by FAMILIAR. Among others, we control that:

- the **merge diff** operator is in line with the **compare** or the **isValid** operator ;
- the **configs** operator is in line with the **counting** operator ;
- the **merge** (**union** or **intersection**) of a feature model $fm1$ with itself gives a feature model logically equivalent to $fm1$ and having the same hierarchy of $fm1$;

We also manually verify a large number of slice and merge examples. Besides we observed that randomly generated feature models may contain a lot of anomalies (e.g., dead features). We used this opportunity to gain further confidence in our implementation. We thus applied our slicing technique to automatically correct anomalies in generated feature models. The implementation was in line with the properties presented in Section 7.2. For all cases, we checked that the original feature model and the slice (i.e., corrected) feature model were equivalent in terms of sets of configurations, that the slice feature model did not contain any error, and that the hierarchy was conform to the original feature model. Another threat is that we cannot guarantee that the slicing or merge operators do not depend on certain shapes of a feature model, or certain kinds of constraints. We reused an established technique from the state-of-the-art [Marcilio Mendonca 2009] that guarantees a random generation.

9.2.4 Summary

Limits. The main results show that:

- the synthesis of the feature diagram has practical limits (up to 800 features). This limit applies to the merge and slice operators. We observed that the slicing technique can scale even for a feature model with 2000 features if the percentage of features to slice is $\leq 35\%$. The reason is that the size of a BDD will always be smaller or at least unchanged after existential quantification.
- the primary limit of the BDD-based implementation lies in the difficulties to compile BDD from the original feature model. In particular, the total number of features in the resulting merged feature model should not be greater than 2000 features, otherwise it is impossible to having a BDD-representation of the merged formula. For the slice operator, whenever a feature model can be represented as a BDD, ϕ_{slice} can be computed. For example, the encoding of ϕ_{slice} can scale up to 2000 features with a CTCR of 10 ;
- the order of complexity of publicly available feature models can be handled.

Impact on FAMILIAR. *SAT or BDD.* For most of FAMILIAR operators, a SAT and/or BDD-based implementation is provided. Both have strengths and limits. The FAMILIAR framework on top of which the language is built allows one to choose an implementation for a given operation. We may provide to the FAMILIAR user a concrete syntactical mechanism (e.g., annotation in a script) to make his/her choice for each operator. Otherwise a default choice (e.g., the choice of BDD for **counting**) is applied. *Lazy Strategy.* For most of the operations, logic encoding prevails over diagrammatical representation. The propositional formula by itself is sufficient to perform reasoning operations (such as satisfiability checking, dead feature detection, etc.). Based on this observation, FAMILIAR implements a *lazy strategy*: for all merge, aggregate and slice operations, we only compute the propositional formula of the resulting feature model. The hierarchy of the feature model is constructed only when needs be, e.g., for the purpose of visualization or serialization. The lazy strategy is useful since reconstructing the hierarchy is costly.

SAT-based implementation. Recently, She et al. proposed techniques to reverse engineering very large feature models (i.e., with more than 5000 features). For this order of complexity, BDDs do not scale. The authors adapted their previous techniques and now rely on SAT solvers (rather than BDDs as in [Czarnecki and Wąsowski 2007]). They reported that the use of SAT solvers is significantly more scalable. In a way this is similar to what has been observed in [Marcilio Mendonca 2009, Janota 2010] for SAT-based analyses of feature models. Hence a SAT-based implementation of slicing and merging is an interesting perspective for future work.

We identify three challenges for a SAT-based implementation:

- the development of efficient techniques to existentially quantify variables from propositional formula for the slice operator ;
- the encoding of the merged formula in CNF so that the size of the formula remains tractable for the merge operator ;
- the support of Or-groups (the most expensive step in the BDD-based algorithm, and not taken into account in [She et al. 2011]) for slice and merge operators.

This would allow a systematic comparison of BDD and SAT-based implementations as well as an optimal support into FAMILIAR based on the study.

Part IV

Applications

In this part, we show how the proposed operators and FAMILIAR have been applied in different application domains (medical imaging, video surveillance) and for different purposes (scientific workflow design, variability modeling from requirements to runtime, reverse engineering feature models). Chapter 10 describes how we can combine multiple variability artifacts to assemble coherent workflows in the medical imaging domain. Chapter 11 reports how in dynamic adaptive systems, such as video surveillance systems, the variability requirements can be refined at design time so that the set of valid software configurations to be considered at runtime may be highly reduced. Chapter 12 illustrates the joint used of a set of operators to reverse engineer the feature model of FraSCAti, a large and highly configurable component and plugin based system.

Ten

Composing Multiple Variability Artifacts to Assemble Coherent Workflows

This chapter shares material with the SC'10 paper "Managing Variability in Workflow with Feature Model Composition Operators" [Acher et al. 2010c] and the SQJ'11 paper "Composing Multiple Variability Artifacts to Assemble Coherent Workflows" [Acher et al. 2011f].

Traditionally, variability management assumed that all artifact variants of a software system were provided by a single source. But now, in many SPL environments, including software systems, the amount of functionality that needs to be developed to satisfy customer needs is far larger than what can be built from scratch in a reasonable amount of time. To solve this problem and facilitate mass customization, it is necessary to take into account externally developed components or applications, themselves being highly variable.

Similar requirements occur during the building of workflow. In our experience, this is particularly the case during the construction of medical imaging workflows (see Chapter 4) where many different kinds of highly parameterized software services exist, provided by different suppliers. The tasks of identifying, tailoring and composing those services with their variability become tedious and error-prone. There is thus a strong need to manage the variability so that developers can more easily choose, configure and compose those services with automated consistency guarantees. To tackle this problem, our approach is to consider services as SPLs, as they are provided by different researchers or scientific teams, while the entire workflow is then seen as a multiple SPL in which the different service SPLs are composed.

We have shown in Chapter 5 and Chapter 6 how composition operators can be used to fulfill part of the requirements, but independently of the workflow. A comprehensive process – *from design to configuration of a workflow* – that allows a workflow designer to assemble the different variability sources (services, concerns, suppliers) is still to be proposed.

In this chapter, we show how to specify variability requirements over the workflow and how consistency is checked when available services in the catalog are composed, including constraints within or across services. We also describe how this part of the process can be automated in order to incrementally assist the user until deriving a consistent workflow product.

In Section 10.1, a typical usage scenario is unfolded from design to configuration of a workflow. In Section 10.2 we briefly describe the realization, how workflows are analyzed and how variability is described and reasoning made possible by generating appropriate code in FAMILIAR. In Section 10.3 we analyze our approach in terms of user assistance and

degree of automation. We also report a first user experiment on workflow building. In Section 10.4, we summarize our contribution.

10.1 From Design to Configuration of Workflow

The goal of our approach is to derive, from an high-level description augmented with variability requirements, a consistent workflow product composed of services offered by the catalog described in Chapter 6

10.1.1 Overview of the Process

Figure 10.1 gives an overview of the proposed multi-step process described in this chapter.

In step ① of the process, the workflow designer first develops a high-level description of the workflow that defines the computational steps (e.g., data analyses) that should take place as well as the dependencies between them. In Section 10.1.2, we will introduce the workflow description language (GWENDIA) we used in this study.

The workflow description is then augmented with rich representation of requirements in order to support discovery, creation and execution of services used to realize the computational steps. In step ②, the workflow designer identifies the variable concerns (e.g., medical image format, algorithm method) for each process of the scientific workflow. A feature model can be associated to a concern of a process, so that the variability of this concern is represented by it (Section 10.2 will discuss how this is implemented). Hence several feature models are woven at different, well-located places in each process (e.g., dataport, interface) for specifying the variability of application-specific requirements. We will present in Section 10.1.3 mechanisms to achieve separation of concerns and to reuse sub-parts of the catalog of feature models rather than developing from scratch feature models.

In the general case, some features of a concern may interact with one or more features of other concern(s). In step ③, some application-specific constraints within or across services are typically specified by the workflow designer to not permit some combinations of features. Similarly, some compatibility constraints (e.g., between dataports) can be deduced from the workflow structure and be activated or not by the workflow designer. We will describe in Section 10.1.4 the kinds of constraints that can be specified or deduced from the workflow structure.

In order to ensure that the variability requirements do match the combination of features offered by the catalog, the workflow designer compares, in step ④, the feature models woven in the workflow with the feature models in the catalog of legacy services. In Section 10.1.5, we will explain how the matching verification is performed for all services of the workflow and reduces the set of features to consider.

In step ⑤, we automatically reason about feature models and constraints specified by the workflow designer in step ① and ②. Constraints propagation and merging techniques are combined to reason about feature models and their compositions (see Section 10.1.6). This provides assistance to the workflow designer (detection of errors, automatic selection of features, etc.)

To complete the workflow configuration (step ⑥), the workflow designer has to resolve concern feature models where some variability still remains, and to perform select/deselect operations. The step ⑥ may be repeated as much as needed in order to allow

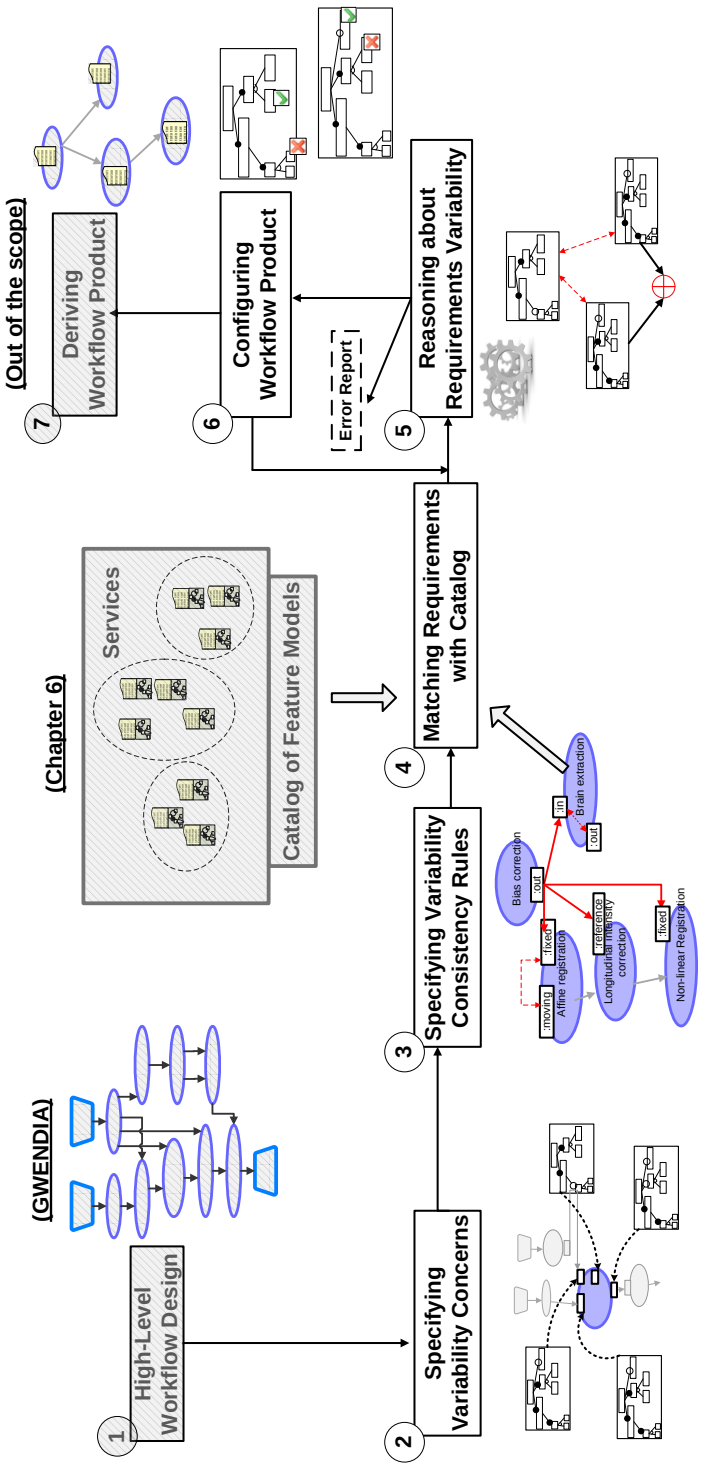


Figure 10.1: Overview of the approach: From design to configuration of the workflow. The step ⑦ is out the scope of this chapter while we rely on techniques described in Chapter 6 to build the catalog of feature models.

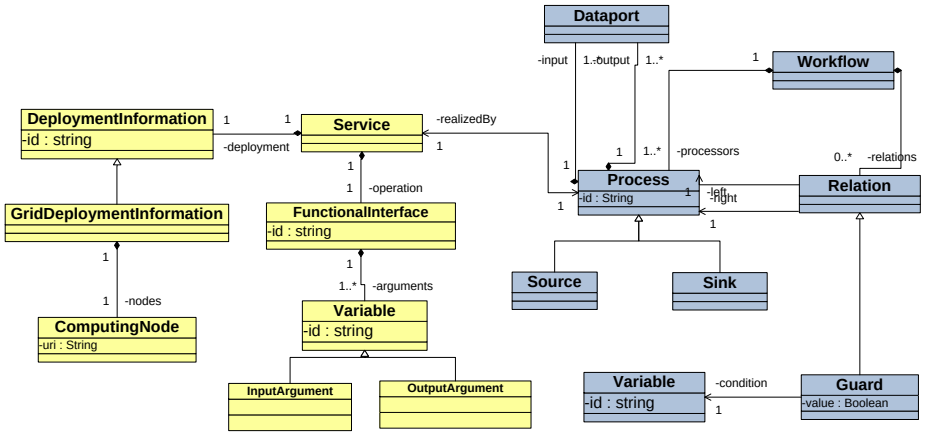


Figure 10.2: Excerpt of workflow and service metamodel.

the workflow designer to proceed incrementally. ⑤ should also be repeated in order to ensure workflow consistency is maintained. In step ⑦, the workflow designer uses the final workflow configuration to identify the services in the catalog that support the combination of features. If more than one service is identified, the workflow designer examines them to choose a best fit or an arbitrary configuration of legacy services. Then services are then combined to form the final workflow product. This step is out the scope of this chapter.

10.1.2 Workflow Design

To support the workflow modeling activity, we use the GWENDIA language [Montagnat et al. 2009]. GWENDIA specifically focuses on coarse grain data-intensive scientific applications and enables the description of massively data-parallel applications. Some workflow engines (e.g., MOTEUR [Glatard et al. 2008]) use the GWENDIA language to describe and deploy applications on grid[1] infrastructures. A GWENDIA workflow is notably composed of a series of processors connected to each other through their input and output ports. For the purpose of the paper, we consider that GWENDIA workflows can be represented using the metamodel described in Figure 10.2, referred hereafter as the GWENDIA metamodel. There are two main parts: the general description of a workflow (elements in blue color on the right) and the specification of a service (elements in yellow color on the left).

A workflow is a set of process which are connected by directed links *relation* through input and or output *dataports*. These links may correspond to operators for *i)* executing services in sequence, *ii)* parallel computations and *iii)* branching through if-then-else constructs. Some processes do not have inputs (*source*) while others do not have outputs (*sink*).

[1]The grid can be thought of as a large-scale distributed infrastructure that provides access to a coordinated set of resources (computing power, storage, data, etc.) [CoreGRID 2011, Foster et al. 2002].

The expected characteristics of the workflow services can be specified from different perspectives. This information can then be exploited in the workflow. For example, with descriptions of the data format, automatic reasoning techniques could check data interoperability between services connected in the workflow. In the GWENDIA metamodel of Figure 10.2, we have identified some abstraction capabilities that can be used to augment services' description. This includes the functional part of the service, in particular its input and output parameters, as well as extra-functional information that can be related to the grid platform in which the service is deployed. In our context, some variability information can be attached to services' elements, for example, to describe the variety of medical image formats supported as an input parameter. With regard to the metamodel, an instance of a service element is a joinpoint in which a feature model can be woven.

10.1.3 Separation of Concerns while Specifying Variability

There are at least two approaches that a workflow designer can use to define workflow service variabilities. One approach is to create from scratch feature models for each service with variable concerns (as in [Acher et al. 2010c]). This solution has two major drawbacks. First, the modeling effort tends to be important and time-consuming. Second, when the workflow designer wants to determine whether the specified combination of features is realized in the catalog, the feature models developed have to be compared with catalog feature models. There is a risk that vocabulary terms used for features' names, hierarchies to structure features as well as granularity detail largely differ, thus requiring an important *alignment* effort.

Another approach is to build feature models that are modified versions of catalog feature models, that is, closely matched catalog feature models are modified so that they include only the features that are needed in the workflow. Hence the modeling effort as well as the alignment effort are reduced through reuse.

Unfortunately, a feature model may represent the variability of *all* concerns within a service, including the features' constraints between concerns, whereas the workflow designer wants to focus on some specific views of the catalog of feature models. For example, in Figure 10.3, $FM_{CatalogAffineRegistration}$ describes four concerns: the two input medical images, the output medical image and the algorithm method, while there are some constraints between the feature Mono and the features Analyze[2]. The workflow designer may want to only consider the part of $FM_{CatalogAffineRegistration}$ corresponding to the algorithm method.

To resolve this issue, extractions, based on the slicing mechanism described in Chapter 7, can be performed and the original feature model can be split into smaller feature models, also called *variability concerns* in the remainder of the chapter. Once extracted, the workflow designer can weave the smaller feature models into specific places of a service to document its variability requirements.

For example, in Figure 10.3, two feature models $FM_{affmoving}$ and FM_{affop} are extracted from the feature model $FM_{CatalogAffineRegistration}$. These two feature models are then specialized (feature Rigid becomes mandatory in FM_{affop} and feature Nifti is no

[2]The two features Analyze have the same name but are different entities. To avoid ambiguity, we use a qualified feature name including the root feature when needs be (e.g., to distinguish the Analyze feature of MIMoving from the Analyze feature of MIFixed. In this specific case, the merge operator makes internally the distinction such that MIMoving.Analyze does not match with MIMoving.Analyze.

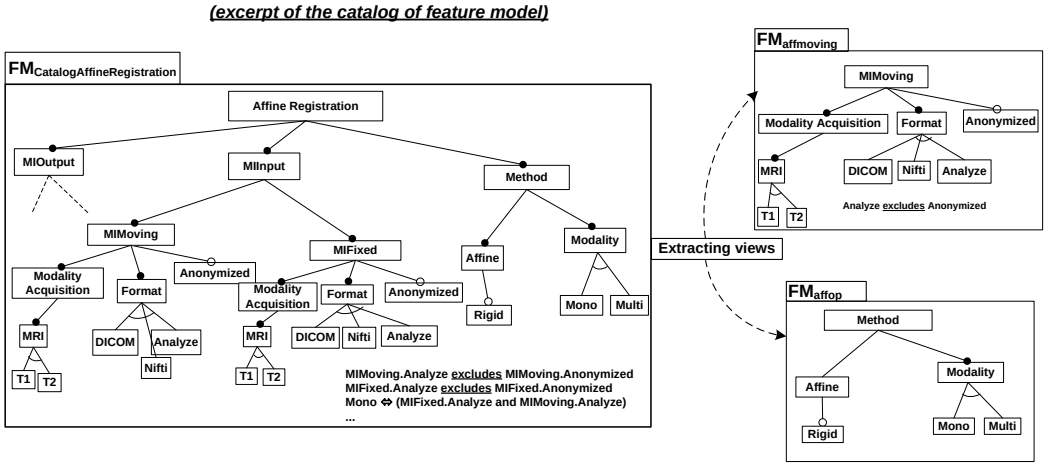


Figure 10.3: Extracting variability concerns from $FM_{CatalogAffineRegistration}$

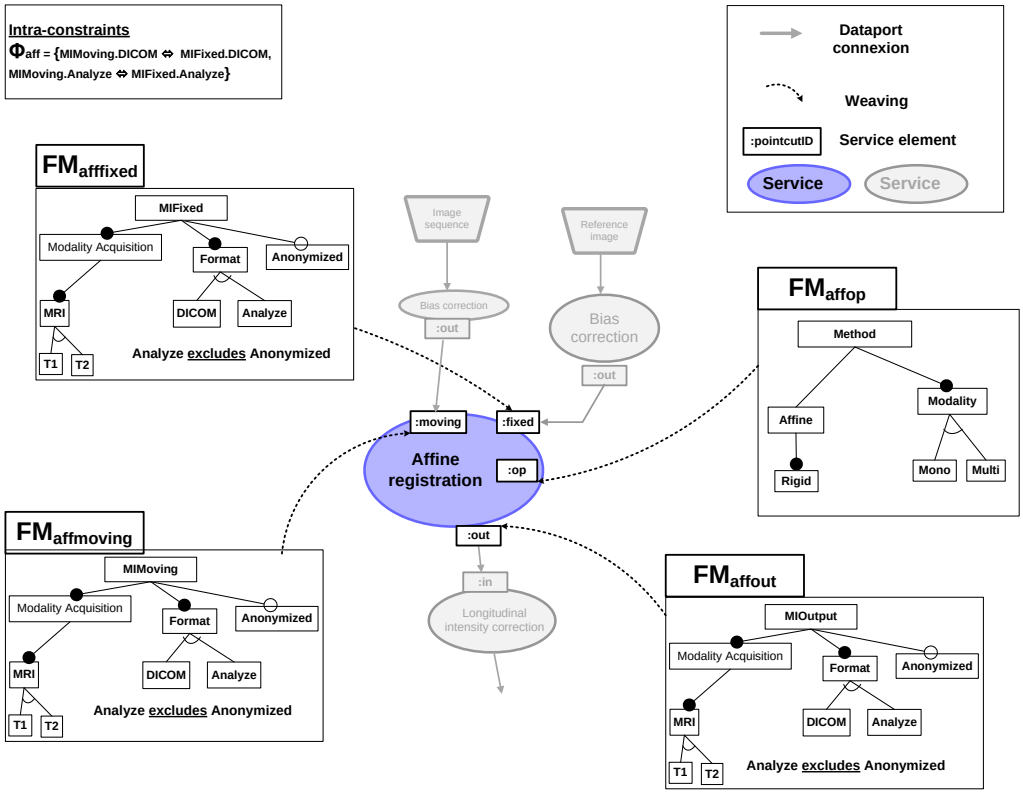


Figure 10.4: Weaving variability concerns into services

longer present in $FM_{affmoving}$) and finally woven into two joinpoints of an *Affine registration* service (see Figure 10.4). Four joinpoints are defined in *Affine registration*: *:moving* and *:fixed* are instances of type *InputArgument*, *:out* is an instance of type *OutputArgument* and *:op* is an instance of type *FunctionalInterface*. Four feature models, including the two feature models $FM_{affmoving}$ and FM_{affop}, are woven into the four joinpoints: three of them deal with medical image formats whereas the fourth feature model deals with the kind of algorithm used for processing the images (see Figure 10.4).

10.1.4 Variability Consistency Rules

The variability information attached to services authorizes numerous combinations of features (configurations) so that a final workflow product can be derived. Nevertheless, not all configurations are valid since variability concerns, documented as separated feature models, can be dependent on other variability concerns *within* services and *across* services. Furthermore some specific constraints may be specified by the workflow designer to restrict the set of valid configurations.

We define a classification of the various types of constraints and then we present how the specification and the verification of these constraints are integrated within the process shown in Figure 10.1.

Constraint Classification We identify four kinds of constraints:

Intra-services constraints. Variability concerns within a service may interact in two ways:

Catalog constraints. The specification of a workflow service should correspond to at least one service in the catalog. Therefore, variability concerns that describe a workflow service should match the constraints imposed by the catalog.

Application specific constraints. Intra-constraints may be specific to an application, for example, a user can require that the imaging formats supported as inputs must be the same for each input data port of the service. As a result, the feature models representing the different input images supported by a service are related to each other through constraints between features. For example, a user specifies in Figure 10.4 that the feature DICOM (resp. Analyze) of $FM_{affixed}$ implies the feature DICOM (resp. Analyze) of FM_{moving}.

Inter-services constraints. Variability concerns are likely to interact also across services. We identify two kinds of situations where the sets of valid combination of services' features may be further constrained:

Workflow Compatibility constraints. Due to the interconnection of services in the workflow, elements of services may be dependent. As a result, concerns attached to these elements may, in turn, be dependent on each other. This typically occurs when concerns are attached to input/output data port. For instance, the medical image output format of the service *Bias correction* is considered to be *compatible* with the medical image input format of the other connected services, i.e., *Affine registration*, *Longitudinal intensity correction*, *Brain extraction* and *Non-linear registration* in the workflow of Figure 10.5. The compatibility relation restricts the set of valid combination of features in each of those services (see next Section).

Application specific constraints. Two (resp. more than two) feature models attached to two (resp. more than two) different services may be related to each

<table>
<tr><th></th><th>Constraint Classification</th><th colspan="2">Constraint Specification</th><th>Contraint Checking</th></tr>
<tr><th></th><th></th><th>By who and how?</th><th>When</th><th></th></tr>
<tr><td rowspan="2">Intra-Service Constraints</td><td>Catalog constraints</td><td>The specification is implicit. But the association catalog/concern is performed explicitely by the workflow designer</td><td>Association is performed at step 3 of the process</td><td>It is performed at the step 4 of the process i.e. when the mapping catalog/specification is done (See Section 4.4.2)</td></tr>
<tr><td>Application-specific constraints</td><td rowspan="2">The workflow designer specifies it explicitely</td><td rowspan="2">At step 3 of the process</td><td rowspan="3">It is performed at the step 5 of the process i.e. when the consistency checking and variability propagation is done (See Section 4.4.3)</td></tr>
<tr><td rowspan="2">Inter-Service Constraints</td><td>Application-specific constraints</td></tr>
<tr><td>Workflow compatibility constraints</td><td>The specification is built-in. But the desactivation may be performed explicitely by the workflow designer</td><td>Possible deactivation is performed at step 3 of the process</td></tr>
</table>

Table 10.1: Specification and checking of constraints within the process.

other in some workflow applications. The user should have the ability to specify some specific constraints when he/she considers that services are tightly coupled. For example, it is required in the medical image domain that *registration* and *unbias* services, that are directly connected in the workflow, are using the same algorithm.

Integration of Constraints within the Process. Some constraints are manually specified by the user (e.g., inter-feature model constraints specific to an application) whereas some others can be detected from the workflow analysis (see Table 10.1). In particular, compatibility constraints between data ports can be deduced and then constraints are applied on feature models attached to data ports. Nevertheless, if the workflow designer is developing the workflow in an incremental manner, he/she may want to deactivate part of the compatibility checks according to the service and/or the concerns he/she focused on.

Dataport compatibility. The compatibility relation can be defined, at the feature model level, as follows: For at least one valid configuration of the feature model associated to the medical image output there is an equal configuration valid in the feature model(s) associated to the medical image input(s) (and vice-versa).

As shown in [Acher et al. 2010c], when n services are concurrently executed[3], it is not sufficient to reason about *pairs* of services independently when checking dataport compatibility:

[3]For other workflow constructs (e.g., if-then-else), properties in terms of sets of configuration have also been defined (see [Acher et al. 2010c] for more details).

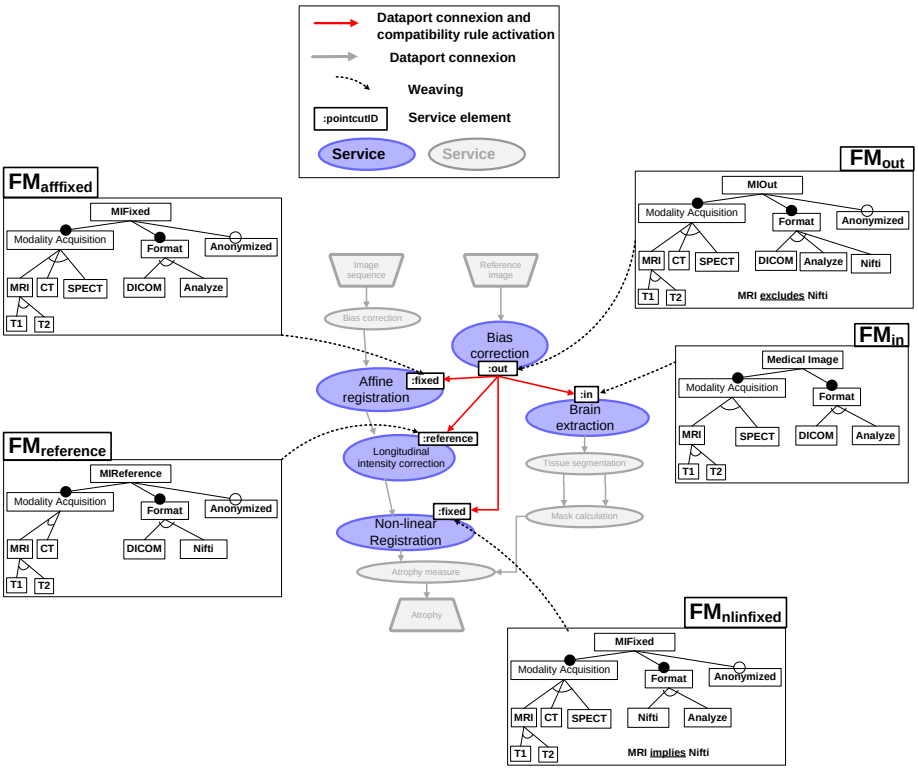


Figure 10.5: Data compatibility between services.

$$(FM_{out} \oplus_\cap FM_{afffixe} \neq nil) \wedge (FM_{out} \oplus_\cap FM_{reference} \neq nil) \wedge$$
$$(FM_{out} \oplus_\cap FM_{in} \neq nil) \wedge (FM_{out} \oplus_\cap FM_{nlinfixed} \neq nil)$$

since the merging (e.g., $(FM_{out} \oplus_\cap FM_{afffixe})$) has side effects on input feature models (e.g., FM_{out} and $FM_{afffixe}$).

We thus need to reason about *all* services at the same time:

$$(FM_{out} \oplus_\cap FM_{afffixe} \oplus_\cap FM_{reference} \oplus_\cap FM_{in} \oplus_\cap FM_{nlinfixed}) \neq nil \qquad (Cmp_1)$$

It can be generalized as follows: When the output dataport of a service $FService_1$ is connected to input data ports of a set of services $FService_2, \ldots, FService_n$, services are consistent according to feature models attached to dataports if the following relation holds:

$FM_{o1} \oplus_\cap FM_{i2} \oplus_\cap FM_{i3} \ldots \oplus_\cap FM_{in} \neq nil$

when FM_{o1} is the feature model attached to the output dataport of $FService1$ and $FM_{i2} \ldots FM_{in}$ are feature models attached to the input dataports of resp. $FService_2, \ldots, FService_n$.

10.1.5 Reasoning about Catalog and Requirements Variability

We have described and illustrated how a user can specify variability at different places as well as the kinds of constraints that may occur in a scientific workflow. We now show how to perform automated reasoning about the feature models and constraints.

Formalization. We first formalize the relationship between feature models, services and workflows as well as the notion of *validity* at the service and workflow levels. The formalization is used afterwards to describe the algorithms that realize reasoning operations.

Definition 19 (Service and Feature Models)**.** *A service $FService_i$ is described as*

- *a set of feature models, $VC_i = \{FM_{i,0}, FM_{i,1}, \ldots, FM_{i,n}\}$.*
- *a set of intra-constraints, Φ_i where each $\gamma \in \Phi_i$ is an arbitrary propositional constraint over the set of features of any feature model belonging to VC_i.*

Definition 20 (Service and Validity)**.** *Let Γ_{agg_i} be the aggregated feature model of $FService_i$ obtained by placing the feature models of VC_i under an And-decomposed synthetic root r and adding the conjunction of each constraint that belongs to Φ_i.*

A configuration c of a service $FService_i$ is a set of features selected where each feature of c is either a feature of $FM_{i,0}, FM_{i,1}, \ldots,$ or $FM_{i,n}$. The configuration c is valid iff $c \in (\llbracket \Gamma_{agg_i} \rrbracket \setminus r)$

For example, the service *Affine Registration* of Figure 10.4 is composed of four feature models, $FM_{affixed}$, $FM_{affmoving}$, FM_{affout} and FM_{affop}, and two intra-constraints. An example of a valid configuration of this service is given below:

{ MIFixed, MIFixed.Modality Acquisition, MIFixed.MRI, MIFixed.T1, MIFixed.Format, MIFixed.Analyze MIMoving, MIMoving.Modality Acquisition, MIMoving.MRI, MIMoving.T1, MIMoving.Format, MIMoving.Analyze MIOutput, MIOutput.Modality Acquisition, MIOutput.MRI, MIOutput.T1, MIOutput.Format, MIOutput.DICOM, MIOutput.Anonymized Method, Affine, Rigid, Modality, Mono }

Definition 21 (Workflow and Feature Models)**.** *A workflow is described as*

- *a set of services, $\epsilon_{services} = \{FService_0, FService_1, \ldots, FService_n\}$;*
- *a set of connections between those services, $\mathcal{C} \subseteq \epsilon_{services} \times \epsilon_{services}$;*
- *a set of inter-constraints, ζ where each $\eta \in \zeta$ is an arbitrary propositional constraint over the set of features of any feature model of $\epsilon_{services}$, i.e., $FM_{0,0}, FM_{0,1}, \ldots, FM_{0,m_0}, \ldots, FM_{i,0}, FM_{i,1}, \ldots$, or $FM_{i,m_i}, \ldots, FM_{n,m_n}$;*
- *a set of compatibility constraints μ over the set of feature model configurations of the workflow. The set can be deduced from the connections between workflow services or be deactivated/specified by a workflow designer.*

Definition 22 (Workflow and Validity)**.** *A configuration cw of a workflow is a set of features selected where each feature of cw is either a feature of $FM_{0,0}, FM_{0,1}, \ldots,$ or FM_{n,m_n}.*

Let Δ_{agg} be the aggregated feature model of a workflow obtained by placing the aggregated feature models of each service under an And-decomposed synthetic root r and adding the set of constraints ζ and μ.

The configuration c is valid iff $c \in (\llbracket \Delta_{agg} \rrbracket \setminus r)$

The approach we propose provides automated support for *i)* ensuring for each feature models associated with a service of the workflow, that only valid and consistent select/deselect decisions are made, *ii)* propagating the decisions so that the workflow designer is only required to answer questions needing human intervention (the answers to the other questions are inferred automatically).

We illustrate how we can automate consistency checking and reduction of variability using the *Affine registration* service as an example. According to the semantics defined above, the following three conditions should not be violated in the *Affine registration* service:

- **(a)** at least one configuration of the workflow service should correspond to another configuration of an existing service in the catalog. Formally:
  Let $\Gamma_{agg_{aff}}$ be the aggregated feature model of service *Affine registration* and $\Gamma_{catalog_{aff}}$ be the feature model of the corresponding family of service in the catalog. Then, the following relation holds: $[\![\Gamma_{agg_{aff}}]\!] \cap [\![\Gamma_{catalog_{aff}}]\!] \neq \emptyset$;
- **(b)** the compatibility constraints between *Affine registration* and other connected services are enforced. Formally: the relation (Cmp_1) holds (see page 149);
- **(c)** FM_{affout}, FM_{op}, $FM_{affixed}$ and $FM_{affmoving}$ are to be consistent. Formally:
  Let $\Gamma_{agg_{aff}}$ be the aggregated feature model of service *Affine registration*.
  $\exists c_{out} \in [\![FM_{affout}]\!], c_{op} \in [\![FM_{op}]\!], c_{fixed} \in [\![FM_{affixed}]\!]$,
  $c_{moving} \in [\![FM_{affmoving}]\!]$ $s.t.$ $(c_{out} \cup c_{op} \cup c_{fixed} \cup c_{moving}) \in [\![\Gamma_{agg_{aff}}]\!]$;

Catalog Mapping. The reasoning process starts by ensuring that the catalog can provide, for all services in the workflow, at least one corresponding feature model that matches its variability requirements. We consider that each workflow service may be mapped to a catalog of feature models[4]. The availability is checked for all services that are mapped to a catalog. The reasoning process has also the capability to identify variability choices that are no longer available in the catalog of feature models. In Figure. 10.6, we illustrate how the mapping between *Affine registration* and the catalog of Figure 10.3 is realized. Some variability choices have been inferred, for example, feature Multi is no longer present and thus the feature Mono is now a core feature. The intra-constraints have been reinforced. For example, configurations of the catalog that include the feature Nifti are not considered because the variability requirements of the service *Affine registration* do not include the feature Nifti. Such reasoning can be automated using the merging techniques described in Chapter 6.

The key idea is to assemble all feature models of the service into an aggregated feature model and then query the catalog of feature model. We use the Algorithm 3 (see Appendix .3) that describes how the catalog is queried. First, the feature models of each service are aggregated together with their intra-constraints. The merge in intersection mode is then performed[5] and finally the feature models of the workflow service, as well as the intra-constraints, are updated after *slicing* the merged feature model.

[4]The mapping between a service of the workflow and the catalog is specified by the user using a domain-specific language, see Section 10.2

[5]Some alignment issues may occur when merging two feature models. For example, a naive aggregation of feature models can lead to an aggregated feature model without the structuring feature MIInput, and thus disturbs the merging process. We provide to the user the ability to specify some pre-directives before merging feature models. The feature model alignment problem is more general and further discussed in Chapter 14.

For instance, $FM_{affixed}$ at the bottom of Figure 10.6 is sliced from the aggregated feature model and includes the constraints that involve its features Anonymized and Analyze.

In this step, every workflow service is consistently mapped to a catalog of feature models. There is no longer need to query again the catalog. The restrictions on the sets of configurations, compactly represented by the merged feature model, guarantee the existence of at least one corresponding service in the catalog.

10.1.6 Consistency Checking and Variability Propagation

Dataport compatibility. The reasoning process continues by ensuring that the compatibility constraints between dataports are enforced (see ① of Figure 10.7). At the feature model level, the merge intersection is performed between FM_{out}, $FM_{affixed}$, $FM_{reference}$, $FM_{linfixed}$ and FM_{in}. The root features of the different input feature models fed to the merge operator may have different names[6]. Such features disturb the merging process so that, theoretically, $[\![FM_{merged}]\!]$ is empty. For practical reasons, we automatically rename each root feature of the feature models involved in the merging with the same temporary name (e.g., Medical Image), if needs be. When the relation (Cmp_1) defined page 149 does not hold, the sources of the error (i.e., the identification of the services that are not compatible to each other) are reported to the user. Otherwise, a valid feature model is computed: FM_{merged}.

The resulting feature model produced by the merge operator, FM_{merged}, is then used to update (i.e., replace) *all* the feature models involved in the compatibility relation. For example, a new feature model, called $FM_{affixed'}$, is now associated to the pointcut *:fixed* of *Affine registration* (see ② of Figure 10.7) and is equal to FM_{merged}. Hence the features DICOM and Anonymized of $FM_{affixed}$ are no longer present. Algorithm 4 (see Appendix .3) recaps the situation.

Propagating constraints within a service. The intra-constraints of the service *Affine registration* may further reduce the set of valid combinations of features in other feature models of the service. When the feature model involved in the dataport compatibility has been modified, as it is the case for $FM_{affixed}$, intra-constraints have to be considered for checking validity or propagating choices within a service[7]. For example, the feature Analyze of $FM_{affixed'}$ implies the feature Analyze of $FM_{affmoving}$ (see ③ of Figure 10.7). A reasoning backend can then infer that the feature DICOM is no longer included in any configuration of $FM_{affmoving}$.

Furthermore, it may happen that the compatibility relation involving $FM_{affixed}$ truly holds but that the service is not valid due to intra-constraints.

The approach consists in aggregating the four feature models $FM_{affixed'}$, $FM_{affmoving}$, FM_{affout}, FM_{op}, FM_{aff} together with the constraints Φ_{aff} of the service *Affine registration*. The resulting feature model is denoted FM_{all}. FM_{all} is then being analyzed for various purposes:

[6]It may be for practical reasons (e.g., convention) or for better characterizing the high-level concept for which the feature model applies. In the example, rather than always using Medical Image, users prefer to be more precise for describing the kind of medical image associated to a feature model. This issue is an instance of the feature model alignment problem.

[7]Note that Algorithm 4 (see Appendix .3) does propagate constraints only for services whose feature models have been modified during the compatibility checking.

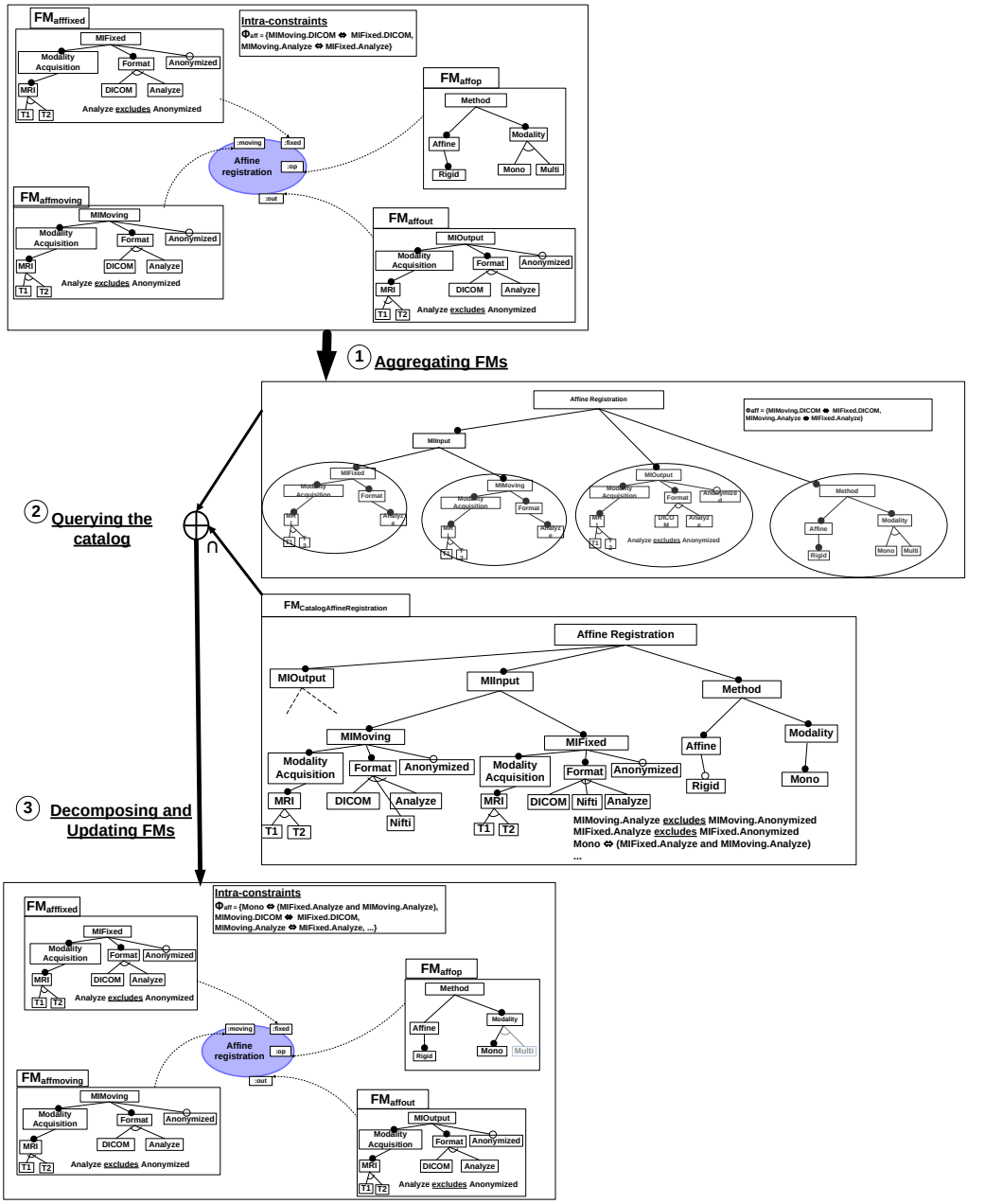


Figure 10.6: *Affine registration* is mapped to the catalog of Figure 10.3 and its four FMs are updated.

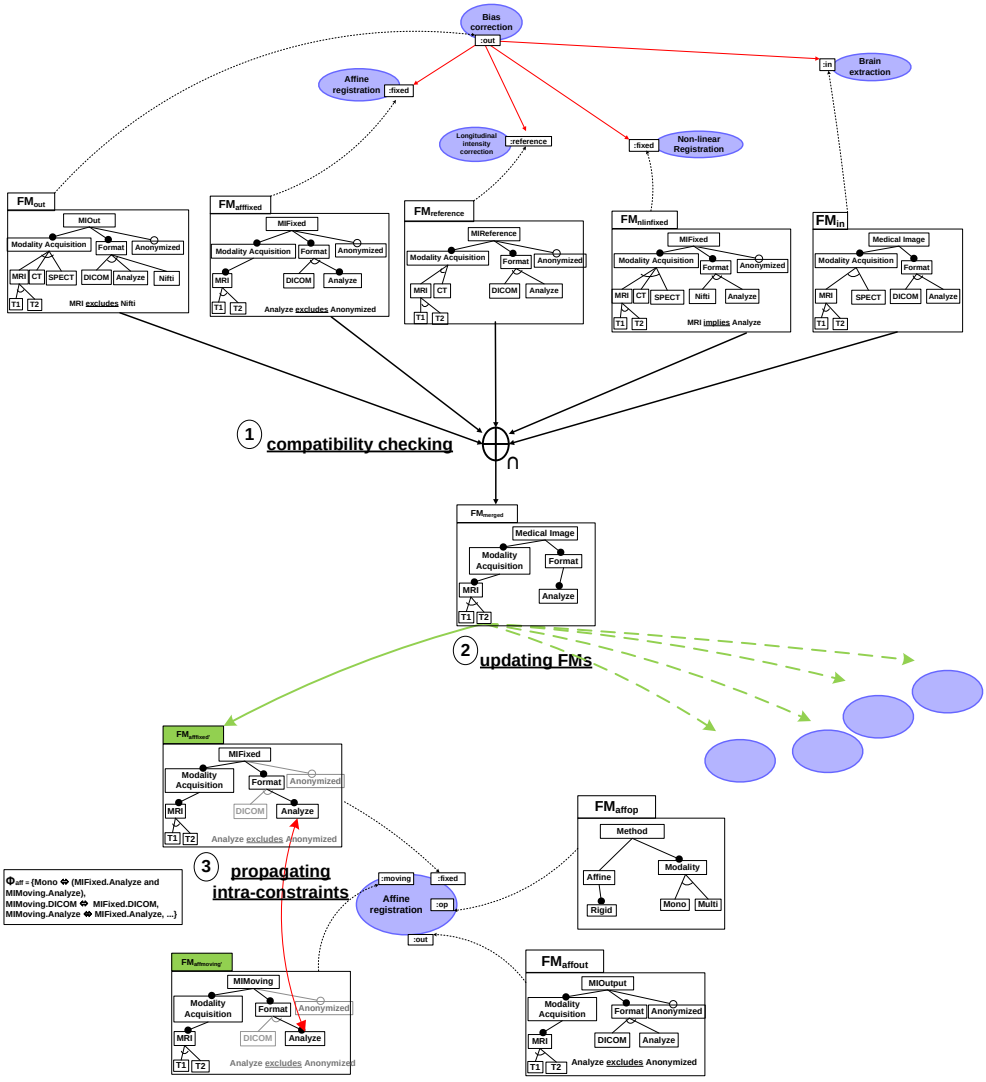


Figure 10.7: Reasoning process: for each connected dataports in the workflow, we propagate variability choices within each service involved in the compatibility checking.

- consistency of the service *Affine registration* can be decided by checking the satisfiability of FM_{all} ;
- we can detect dead and core features and report back to the workflow designer ;
- the corrective capabilities of the slicing technique can be applied to simplify and update the different feature models of the service *Affine registration* (removal of features when features are known to be dead, setting the mandatory status to some core features, etc.) ;

Reiterating the reasoning process. It may happen that the inference of variability choices through intra-constraints propagation leads to the modification of a feature model involved in a dataport compatibility. (It is not the case for *Affine registration*.)

Let us consider the example given in Figure 10.8 where services *FService1* and *FService2* are sequentially connected. Compatibility checking between their dataports *:out* and *:in* is first performed (see ①) such that features E, D are no longer present in FM_{output} of *FService1* while feature F is no longer present in FM_{input} of *FService2*. Then, constraint propagation is performed in *FService1* and *FService2* (see ②). No variability choices can be inferred in *FService2*. In *FService1*, feature Z2 has been removed due to the intra-constraint $B \vee C \Rightarrow \neg Z2$. In turn, feature C has been removed due to the intra-constraint $Z2 \iff C$. Hence, the feature model FM_{output} involved in a compatibility relation between dataports has been modified and compatibility checking should be reiterated (see ③). It modifies FM_{input} and, after constraint propagation, FM_Y in *FService2* (see ④). The reasoning process stops since no variability choices can be inferred in *FService1* and *FService2* and the compatibility checking between FM_{output} and FM_{input} has no effect.

Algorithm 6 of Appendix .3 describes the reasoning process. Compatibility checking is performed if and only if a feature model attached to a dataport connected to other dataports has been modified during constraint propagation (and thus *marked* during the execution of the Algorithm 5 of Appendix .3) . If the set of configuration of a feature model remains the same, the algorithm terminates. An important property of the merging operator in intersection mode is that each input feature model is either a refactoring or a generalization of the merged feature model. As a result, each time the compatibility checking is performed, feature models involved are either specialized or not impacted. Hence, the set of configurations is either the same or is *decreased* until being a singleton. This property guarantees that the algorithm necessary terminates when no variability choices can be deduced.

Multi-step Configuration of the Workflow. All feature models of the workflow can be partially configured or *specialized* [Czarnecki et al. 2005b, Thüm et al. 2009]. The specialization activity includes the selection/removal of some features, the adding of some constraints within a feature model, etc. Reasoning operations, as described above, can be similarly performed at each step to ensure consistency of the whole workflow and propagate variability choices (see ⑥ of Figure 10.1).

Once the specializations of all feature models are complete, we know by construction that it corresponds to services in the catalog. It may happen that given a configuration of a service, there is more than one service that corresponds in the catalog (since there exists services that support the same combinations of features). In this case, the user has to arbitrary choose which services he/she wants to include in the final workflow product.

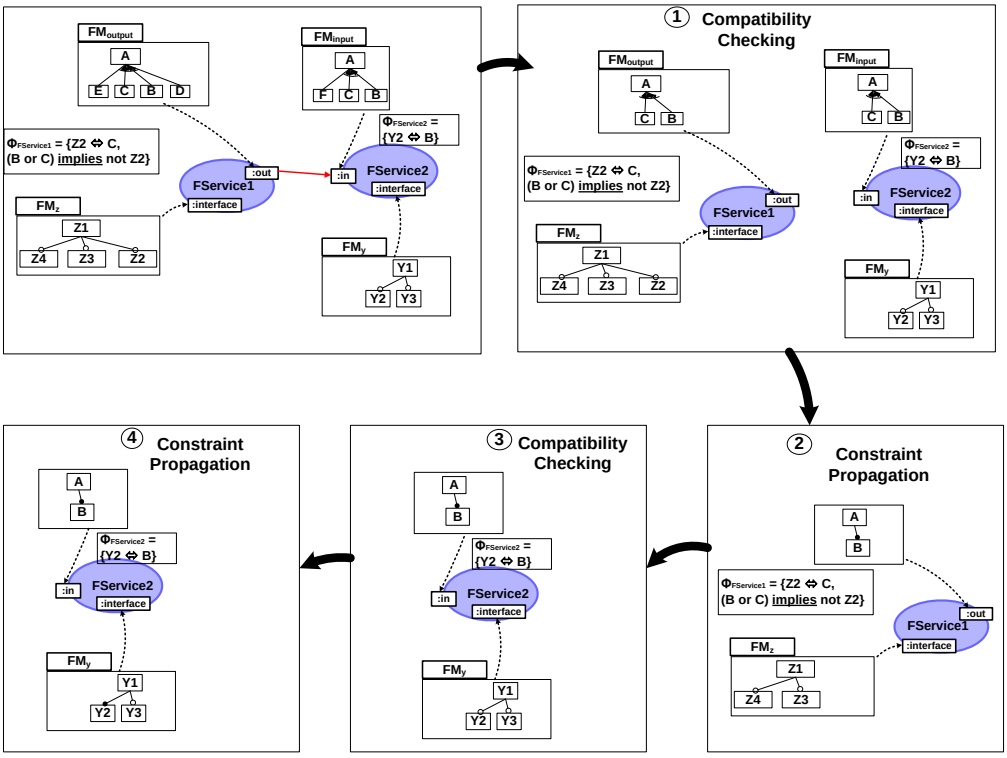


Figure 10.8: An example: reiterating compatibility checking and constraints propagation.

10.2 Realization and Tool Support

The approach proposed is comprehensively supported by a combination of dedicated tools and DSLs. The goal is to assist users at each step of the process – from workflow design to configuration of each of its constituent parts – described in Figure 10.1.

10.2.1 Workflow Modeling

The first activity is to design a workflow (see ① of Figure 10.1). We rely on the GWENDIA language (see Section 10.1.2), which proposes two concrete syntax, a graphical representation and a textual representation, and supports all the workflow constructions mentioned in Figure 10.2. Using GWENDIA, scientists can specify a workflow including all data connections (see left upper part of Figure 10.9).

To specify the variability requirements, we choose to develop a simple and dedicated formalism to relate feature models to services and workflows. The DSL, called *Wfamily* (see right upper part of Figure 10.9) enables one to:

- *import* feature models from external files while performing some high-level operations (extraction, renaming/removal of features, etc.). For example, the user can load an existing feature model from a catalog, then extracts the sub-parts that are of interest and finally specialize the different feature models ;

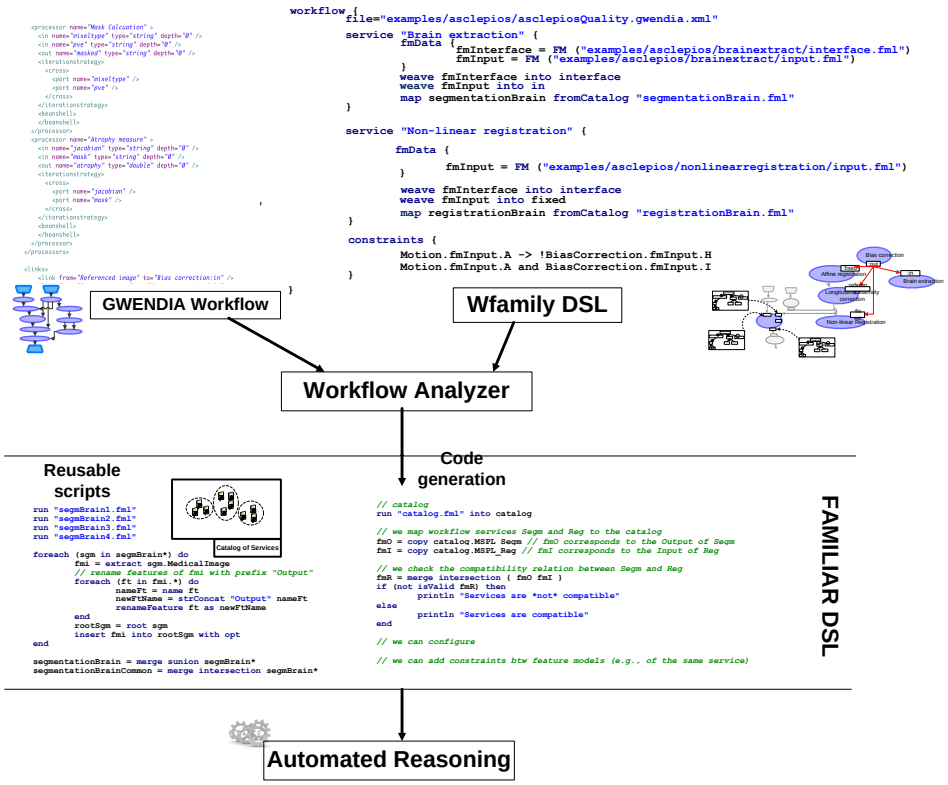


Figure 10.9: Tool support and Domain-Specific Languages.

- *weave* feature models to specific places of the workflow. We reify the concept of *pointcut*, which have an unique identifier within a service. Hence users can specify for which specific pointcut of a service a feature model is attached to ;
- *constrain* feature models within and across services by specifying propositional constraints. Each feature model that have been woven has an unique identifier and can be related with each other through cross-tree constraints.

10.2.2 Reasoning about Variability with FAMILIAR

FAMILIAR is also used in the following situations:

- to specify variability requirements within services (cf. ② and ③ of Figure 10.1): FAMILIAR is *embedded* into the DSL *Wfamily* described above. As a result, extracting a sub-feature model from a feature model in a catalog essentially consists in reusing FAMILIAR operators ;
- to build catalogs of services, organized as reusable scripts: feature models that document variability of services are merged together ; similarly, querying a catalog of services is realized using FAMILIAR scripts (cf. ④ of Figure 10.1) ;
- as a target language. FAMILIAR code is *generated* from the workflow analysis, for example, to reason about dataport compatibility (cf. ⑤ of Figure 10.1).

The FAMILIAR code is then interpreted to check the consistency of the whole workflow, to report errors to users as well as to automatically propagate choices (cf. ② and ③ of Figure 10.1). Users can incrementally configure, using graphical facilities provided by FeatureIDE editors, the various feature models of the workflow (cf. ⑥ of Figure 10.1). Finally, in order to derive a final workflow product, competing services can be chosen from among sets of services in the catalog using FAMILIAR reusable scripts (cf. ⑦ of Figure 10.1).

10.3 Discussion and Experiments

10.3.1 User Assistance and Degree of Automation

Organizing the workflow construction with SPL engineering techniques leads to a shift in the process. The activities are then well targeted for each kind of stakeholder. The service provider documents variability once and for all, the catalog maintainer handles all available services over time and the workflow designer can focus on its construction activity. It must also be noted that our proposed approach is heavily relying on the service catalog, so that the effort of building it is compensated. The catalog is indeed used both to assist the expert, when determining his relevant concerns — which is more error-prone when specified from scratch —, and to incrementally enforce consistency.

A first assistance to the user is provided when he/she can select an appropriate service from among sets of existing services. He/she may also want to search services matching several criteria to determine whether at least one service can fulfill a specific feature or a set of features. During the process, if the variability manipulated by the user leads to some inconsistency but is considered to be more important than the workflow structure, the user has to correct the workflow itself. Using our approach, such inconsistencies are automatically and systematically detected and several correction strategies can be applied. The separation of concerns provides the ability to precisely locate the source of errors and to give information to assist users in correcting the workflow: choose another service, correct an applied concern, relax some variability description, configure differently some services.

Properties of the merge operator can then be exploited. The various compositions of feature models may be performed in any order because of the associativity property of the merge operator. Heuristics, such as merging larger feature models first, can thus be planned to detect an earlier source of errors. The merging between feature models contributes to decrease the number of remaining variability choices presented to the user. Indeed an additional property of the merge in intersection mode is that the number of features of the resulting feature model is lesser than or equal to the number of features commonly shared by input feature models. This property can dramatically reduce the set of configurations to be considered by the user during workflow configuration. As a result, it is likely that the amount of time and effort needed during the configuration process can be reduced (see experimental results below).

As for the process automation, it ranges from the catalog building to the resulting workflow product. First, taking all service variability descriptions, the catalog of services is automatically generated. During the workflow construction and configuration, all assisted steps discussed before are coupled with incremental and automatic consistency checking. The specified concerns over services are extracted from the catalog with the guarantee to be consistent subsets. After having been woven to services, their consistency according

the workflow is automatically checked. When the resulting workflow is configured, the automatic propagation of constraints among feature models representing the concerns is conducted, ensuring again consistency while reducing the user burden. Finally, as the variability of services may evolve over time, the complete process can be easily replayed to check again the consistency with additional or modified concerns.

10.3.2 Experiments

Application to Three Real Workflows. Using the tool support described in Section 10.2, we have applied the proposed approach to three real medical imaging workflows, the Alzheimer's disease workflow [Lorenzi et al. 2010], a cardiac analysis workflow [Maheshwari et al. 2009], and a workflow for determining the quality of a segmentation algorithm [Pernod et al. 2008]. The number of services that constitute the three workflows varies from 9 to 24 (see #services in Table 10.2), so that experiments are conducted on different scales[8]. We consider scenarios in which the workflow designer augments the workflow description with feature models and constraints. As we want to determine the ability of our approach to handle compatibility constraints between services, we count the number of *active* dependencies (see #active). A dependency between two connected services is active when there are feature models related to the same concern on both sides, so that these feature models have to be merged. This number is lesser or equal than the number of data dependencies (see #dependencies) since feature models are not necessary attached to all dataports.

In Table 10.2, the total number of feature models (see #FMs), features (see #features), core features (see #cores), variation points (see #VP) and configurations[9] (see #configurations) in the initial workflow description are reported for the three workflows. Core features refer to features necessary included in any configuration whereas variation points refer to features whose selection/deselection still needs to be fixed. In order to determine how the proposed automated reasoning reduces the number of variation points (thus the number of valid configurations) and possibly facilitates the decision-making process, the same metrics are reported after the reasoning mechanism.

The first experiment concerns the workflow described in [Lorenzi et al. 2010]. This workflow is rather small (composed of 9 services) and 16 dependencies between data ports are present. We wove 12 feature models (using the catalog of feature models) into workflow services but not into *Atrophy measure* and *Mask Calculation* services. As a result, 9 compatibility constraints were detected (see #active). We did not edit the feature models and we only specified some intra-constraints services. Applying the reasoning mechanisms significantly reduced the number of variation points (from 79 to 32) and the number of configurations (from 10^{12} to 10^{4}).

For the second experiment, we used the cardiac analysis workflow described in [Maheshwari et al. 2009]. The management of variability was focused on data pre-treatments so that we only wove 8 feature models and we deliberately did not consider other parts of

[8]The size of *scientific* workflows varies depending on the domain (e.g., bioinformatics, medical imaging). In the medical imaging domain, the presence of 24 services can be considered as a large workflow, even though larger workflows have been developed.

[9]The number of initial configurations is computed by considering feature models without inter-/compatibility constraints.

		1. Alzheimer's disease	2. Cardiac analysis	3. Segmentation evaluation
Input workflow	#services	9	14	24
	#dependencies	16	20	41
	#active	9	6	19
Initial specification	#FMs	12	8	25
	#features	131	97	286
	#cores	52	43	110
	#VPs	79	54	176
	#configurations	10^{12}	10^{9}	10^{25}
After reasoning	#features	104	79	213
	#cores	72	48	146
	#VPs	32	31	67
	#configurations	10^{4}	10^{5}	10^{9}

Table 10.2: Experimental results on three scientific workflows.

the workflow. We specialized the format of the image sources before propagating choices. Again we observe a noticeable reduction of variability points.

For the third experiment, we used a larger workflow (cf. [Pernod et al. 2008]) in which 24 services are combined to evaluate segmentation. A noticeable property of this workflow is that 6 registration services and 5 normalization services are used. We thus made an extensive use of the catalog. Again, we did not edit the feature models and we only specified some intra-constraints services. The reduction of variation points is even more significant (from 176 to 67), mainly because of the large number of data dependencies (41) that are automatically handled by our approach.

As a result, these experiments show that the reasoning mechanisms developed for supporting consistent composition of multiple SPLs significantly reduces the high complexity to be managed by the workflow designer. Larger experimental validations should confirm these first results.

Practical Experience. We design an experiment in which different users from the NeuroLOG project should design and configure a workflow *without* our techniques and then, for the sake of comparison, *with* our techniques. The input of the experiment is as follows:

- a medium-size, GWENDIA workflow (see Figure 4.1, page 37) that consists in 9 processes (only 5 processes, *Affine Registration*, *Brain Extraction*, *Longitudinal intensity correction*, *Tissue segmentation* and *Non-linear registration*, have to be configured in the experiment) ;
- a description of about 80 existing services according to different criteria. We consider two categories of criteria: the first category concerns *medical images* (e.g., medical image format supported as input/output), the second category concerns *medical imaging algorithm* (e.g., affine or non affine transformation). For this experiment, the average number of features to consider per service is around 30. Semi-structured, tabular data are used for the description of variability services in terms of features and stored in the CSV (comma-separated values) format. To facilitate the identification of services, we group together similar services that are candidate to implement

a process ;
- a document describing in natural language the requirements of the application and the constraints of the workflow. Three scenarios are described in the document: the first scenario simply consists in selecting five services while ensuring data compatibility ; the second scenario is similar to the first scenario except that some requirements on the anonymization of images are added ; the third scenario involves more constraints (e.g., formats of images that can be used are restricted to two predefined alternatives and no interactive algorithm can be used).

The challenge for users is to have, at the end of the experiment, a workflow in which appropriate services are consistently combined. We report below our observations and lessons learned.

Effort and Time Needed. In the experiment, users have to consider a large number of candidate services (80) for a large number of features per service (30), so that the total number of distinct features[10] to consider is more than 200. At this scale, the configuration process turned out to be impractical without adequate support. In particular, a manual configuration process (e.g., based on "trial and error" strategy) should be avoided as it is both error-prone, laborious and time-consuming. The observation applies to two specific tasks of the workflow design. Firstly, when users have to select a service from among the set of existing services, the main difficulty comes from the fact that some features from different concerns interact (e.g., the selection of the format DICOM may imply the selection of an Interactive algorithm), which is not straightforward to identify. In addition, users complain from the lack of *querying* operations, for example, to filter the set of services that fulfills specific requirements. Secondly, when users have to ensure that services are data compatible, multiple variability descriptions are to be considered for resolving complex features' interactions. The difficulties we observed are not surprising. The satisfiability of a feature model is known to be a difficult computational problem, i.e., NP-complete [Schobbens et al. 2007] and in our case, not only one feature model has to be considered. Several authors claim that in real software projects, there can be thousands of features whose legal combinations are governed by many and often complex rules [Mendonça 2009, Mendonca and Cowan 2010, Hubaux et al. 2010b, Janota 2010] – the design of scientific workflows exhibits similar complexity. As argued by the same authors, it is thus of crucial importance to be able to simplify and automate the decision-making process as much as possible [Mendonça 2009, Mendonca and Cowan 2010, Hubaux et al. 2010b, Janota 2010]. Our observations and case study reinforce this requirement. Our tool-supported approach does assist the user when he/she can select an appropriate service from among sets of existing services (thanks to the construction of catalog of feature models) and propagate variability choices (thanks to automated reasoning techniques). Once the catalog of feature models has been built and the variability has been specified at the workflow level, the time and effort needed to complete the configuration process are significantly reduced so that their activities become manageable.

In addition, the three scenarios of the experiments can be realized in a similar fashion. We just *reuse* the catalog of feature models and modify the original script *Wfamily* developed for the first scenario to fulfill the new requirements. The costly process (in time and in user effort) is related to the construction of catalog of feature models and development

[10]Two features are considered distinct if their names are distinct.

of *Wfamily* script. For this experiment, the construction of catalog of feature models was highly facilitated by *i)* the identification and grouping of similar services, *ii)* the use of a common terminology and hierarchy of features to described services. The development of *Wfamily* script was more laborious due to the costs of *i)* of learning a new language and *ii)* understand the ideas behind the approach.

User Assistance and Correctness. A manual attempt for configuring the workflow leads to several errors that must be corrected, without adequate assistance. In addition, a manual checking that determines whether a selection/deselection of features is correct (i.e., does not violate any constraint of the workflow and corresponds to at least one existing service) was proved to be not satisfying (i.e., confidence about the solution was too low). An important benefit of using automated techniques based on a sound basis is that we can assist users at each step of the configuration process while guaranteeing properties of the designed workflow. FAMILIAR (and thus the underlying implementation of the approach) rely on SAT solvers or BDD. It has been shown that both SAT solvers and BDD can be used to implement an *interactive* configuration process, where the computer provides information about validity of choices each time the user makes a new choice – this feedback typically takes the form of graying out the possibilities that are no longer valid [Mendonça 2009, Janota 2010]. Moreover a configurator can *infer* which choices are valid and which are not at each step of the process.

Flexibility. Another drawback of a non tooled-approach was the lack of *flexibility* in selecting an appropriate service. Although finding one appropriate service can be sufficient, it is more preferable to have the choice between several candidate services, for example, to favor services that have been developed by a specific research team. Advanced querying operations were thus identified as important when designing the workflow. Our tool-supported approach supports a proper management of variability and the ability to infer which existing services are no longer able to fulfill the requirements.

10.4 SUMMARY

Building service-oriented scientific workflows mainly consists in first selecting some appropriate services from all available parameterized services, then configuring and assembling them in a consistent way. In many application domains, these activities are cumbersome and error-prone, and this hampers current development efforts in computational science. In this chapter, we introduced a rigorous and tooled approach that extends current SPL engineering techniques to facilitate and automate consistent workflow construction. This approach assumes that a variability-aware catalog was originally built to organize highly-parameterized services provided by different suppliers (as we described in Chapter 6).

The following contributions were presented:

- A multi-step process to obtain a consistent workflow was also detailed. Taking a basic description of the workflow, it consists in specifying different variable concerns (ranging from functional parameters to non-functional properties or deployment specificities) on one or more services. Constraints within or between concerns can be added and all these elements are incrementally checked for consistency against the service catalog. The workflow is then seen as a multiple SPL which composes the

SPLs of services. Configuration is assisted, consistency checking and propagation are incremental and automated, so that a consistent workflow product is obtained. Evolution of both the services and their variable parts is also supported by the approach. Moreover this process completely rests on a sound formal basis realized by feature model management operators and FAMILIAR, so that generic parts of the process can be more easily reused in other context.

- An overview of the implementation and user-oriented tool support has also been given. Besides illustrations were provided using a non trivial example in the representative domain of medical image analysis. Additionally, first experimental results have been discussed in terms of user assistance and degree of automation. These experiments show that the reasoning mechanisms developed for supporting consistent composition of multiple SPLs reduces the high complexity to be managed by the workflow designer. Nevertheless, a more comprehensive tool support, including graphical facilities, is needed as well as the conduct of further experiments to confirm these first results.

Eleven

Modeling Variability From Requirements to Runtime

This chapter shares material with the ICECCS'11 paper "Modeling Variability from Requirements to Runtime" [Acher et al. 2011g], the ICVS'2011 paper "Run Time Adaptation of Video-Surveillance Systems: A Software Modeling Approach" [Moisan et al. 2011], the Models@run.time'09 paper "Modeling Context and Dynamic Adaptations with Feature Models" [Acher et al. 2009a] and the MiSE'09 paper "Tackling High Variability in Video Surveillance Systems through a Model Transformation Approach" [Acher et al. 2009c]. This work is based on a collaboration with Jean-Paul Rigault and Sabine Moisan (PULSAR project-team, INRIA).

11.1 Variability of Dynamic, Complex Software Systems

Problem and Overview of the Approach. Video-surveillance processing chains are complex software systems, exhibiting high degrees of variability along several dimensions. From a technical and software perspective, the number of components, their variations due to choices among possible algorithms, the different ways to assemble them, the number of tunable parameters, etc. make the processing chain configuration rather challenging. Moreover, the number of different applications that video-surveillance covers, the environments and contexts where they run, the quality of service that they require increase the difficulty. Finally the context of an application may (and does) change in real time, at runtime, requiring dynamic reconfiguration of the chain. To make things even more complex, these variability factors are not independent: they are related by a tangled set of strong constraints.

This huge variability raises problems at design time (finding the configurations needed by the chain, foreseeing the different possible contexts), at deployment time (selecting the initial configuration), and at run time (switching configurations to react to context changes). In order to deploy the software solution, the engineering effort can last from one week to several months. There are several reasons. First, reasoning directly at the level of software components is far from obvious, even for an expert, since selecting among the software choices must respect a large number of constraints (e.g., night surveillance in natural light necessitates specific algorithm). This can be seen as an instance of the traditional gap that exists between the problem space and the solution space (see Chapter 2). Second, an unique characteristic of dynamic, self-adaptive software systems, such as video surveil-

lance systems, is that, at *runtime*, only a small part of the model related to requirements has to be kept – for example, features related to the context – so that this sub-model can be efficiently used to pilot self-adaptation mechanisms. Although technical know-how may help to reduce possible choices, it is difficult to determine which combination of artifacts remain valid at runtime according to the various possible *contexts* (e.g., lighting conditions, information on the objects to recognize).

In this chapter, we focus on the problem at design and deployment time with the purpose of *modeling* variability. The realization of variability or how configurations are changed at runtime to react to context events are out of the scope of this chapter. We consider that approaches and techniques to support product derivation at runtime can be applied in the context of this contribution, for instance, using the results of the DiVA[1] project which intends to manage dynamic variability in adaptive systems with the combined use of aspect-oriented and model-driven techniques [Morin et al. 2009b;a].

To tackle our problem, we follow a model-based approach in which both variability spaces are described through two feature models ; the first one describes the domain and the related requirements, while the other one is an abstract representation of the code. The relationships between variants are described as propositional rules relating features either in the same model or across models. Such an approach is somehow similar to other related work that promotes the (systematic) use of feature models [Kang et al. 1998, Hartmann and Trew 2008, Metzger et al. 2007, Tun et al. 2009].

We propose here a *modeling process* in which the feature models are systematically used and step-wise specialized, from requirements to runtime. Consequently, from a domain expert specification, the possible configurations space is highly reduced, as no more valid platform configurations can be automatically removed by transformation. We present techniques, based on propositional logic and fully supported by FAMILIAR, to assist SPL practitioners in the deployment of video surveillance processing chains while ensuring the end-to-end consistency of the manipulated models.

Video Surveillance Systems. Taking Video Surveillance (VS) as a representative domain of dynamic, adaptive systems, we now determine our motivating issues. The purpose of VS is to analyze image sequences to detect interesting situations or events. Depending on the application, the corresponding results may be stored for future processing or may raise alerts to human observers. There are several kinds of VS tasks according to the situations to be recognized: detecting intrusion, counting objects or events, tracking people, animals or vehicle, recognizing specific scenarios, etc. Apart from these functional characteristics, a VS task also sports non functional properties, such as quality of service: typical criteria are robustness, characterized by the number of false positive and negative detections, response time, or recognition accuracy. As a matter of examples, intrusion detection may accept some false positives, especially if human operators are monitoring the system; counting requires a precise object classification; recognizing dangerous behavior must be performed within a short delay.

Moreover, each kind of task has to be executed in a particular context. This context includes many different elements: information on the objects to recognize (size, color, texture...), description and topography of the scene under surveillance, nature and position of the sensors (especially video cameras), lighting conditions... These elements may be

[1] http://www.ict-diva.eu/

related together, for example, an indoor scene implies a particular lighting. They are also loosely related to the task to perform since different contexts are possible for the same functionality. For instance, intrusion detection may concern people entering a warehouse as well as pests landing on crop leaves.

The number of different tasks, the complexity of contextual information, and the relationships among them induce many possible variants at the specification level, especially on the context side. The first activity of a video surveillance application designer is to sort out these variants to precisely specify the function to realize and its context. Then the designer has to map this specification to software components that implement the needed algorithms.

At the implementation level, a typical VS processing chain (Figure 11.1) starts with image acquisition, then segmentation of the acquired images, clustering, to group image regions into blobs, classification of possible objects, and tracking these objects from one frame to the other. The final steps depend on the precise task. Additional steps may be introduced, such as reference image updating (if segmentation steps need it), data fusion (in case of multiple cameras) or even scenario recognition. All steps correspond to software components that the designer must correctly assemble to obtain a processing chain. Moreover, for each step, many variants exist, along different dimensions. For instance, there are various classification algorithms with different ranges of parameters, using different geometrical models of physical objects, with different merge and split strategies to identify relevant image blobs. The situation is similar for the other algorithms.

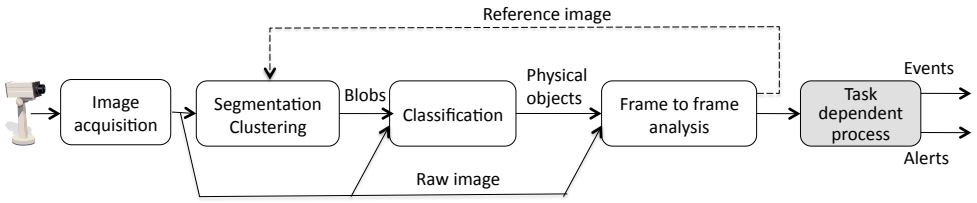


Figure 11.1: A simplified video surveillance processing chain

The domain of VS is now mature enough to provide components covering all classical steps and to provide unifying and stable frameworks to compose them, such as the platform used in this case study [Avanzi et al. 2005]. However these frameworks can only be mastered by video analysis specialists. The specification phase is not supported by tools, and component assembly seldom is. Thus designers work directly at the component assembly level, following an informal view of the specification. As a consequence, there is no guarantee that specification and implementation are consistent and that internal dependencies are respected, at specification as well as implementation level; moreover, tracing between specification and implementation is hard. Hence, designing a video surveillance system faces typical difficulties of current information systems in which domain and application engineering must be cleverly combined. Indeed this requires to cope with multiple sources of variability, both on the task specification side and on the implementation one. To bridge this gap, designers must have an extensive experience of the application domain as well as a deep knowledge of the software components. Yet the process remains tedious, error-prone, and possibly long.

11.2 MODELING VARIABILITY

Domain and Software Variability. In the VS case, we have multiple variability factors, both for specifying an application and for describing the software platform. On the platform side, stakeholders manipulate solution-oriented artifacts. There, variability is entirely related to the software and refers to "the ability of a system to be efficiently extended, changed, customized or configured for use in a particular context" [Svahnberg et al. 2005]. The software variability is expressed in a dedicated FM, called the *PlatForm Configuration* (PFC) model representing a view of implementation modules provided by the software platform. For deriving an actual product, we advocate the use of domain knowledge which contains relevant information to reason about and to select among variants at a higher level of abstraction. The domain variability is expressed through the *Video Surveillance Application Requirement* (VSAR) model which comprises, in our case, the task specification, the scene context, the object of interests, the Quality of Service (QoS).

Separation of Concerns. One option would be to transfer VSAR variability (e.g., all the possible variations in the context) into the PFC model, thus concentrating variability in one unique model. A major drawback is then to swamp domain variability concerns with the mass of platform details. Our approach is rather to separate variability concerns into two interrelated FMs: the VSAR FM represents the experts' business knowledge while the PFC one deals with the platform implementation. This separation of concerns offers several benefits. Each variability model addresses a different level of expertise. On the requirement side, users can follow their usual practices, use the domain vocabulary, and the specific definition of variation types. Confining the variability in a dedicated *space* thus improves the modeling process. During the application engineering process, users can take decisions only related to their know-how and domain. The impact of a modification in the platform model (e.g., code module added) or in the VSAR model (e.g., object of interest added) is clearly localized. Co-evolution and maintenance of both variabilities are facilitated.

We now describe more precisely the two VSAR and PFC models, together with transformation rules relating features of the two FMs.

VSAR *model.* Figure 11.2 (upper part) shows an excerpt of the FM corresponding to the VSAR side. This model describes the relevant concepts and features from stakeholders' point of view, in a way that is natural in the VS domain. To enforce separation of concerns, we identified four top level features. The Task feature expresses the precise function to perform. QoS corresponds to the non-functional requirements, especially those related to quality of service. Then, we need to define the Objects of interest to be detected, together with their properties. Finally, Scene context is the feature with the largest sub-tree; it describes the scene itself (its topography, the nature and location of sensors) and many other environmental properties (only some of them are shown on the figure). In this model, the (sub-)features are not independent, for example, selecting a feature may impact other choices. Thus, we have enriched the feature models by adding internal constraints to cope with relations local to a model. So far, we have identified two kinds of constraints. Choosing one feature may **imply** or **exclude** to select another specific one. For example, if feature Counting is selected, this implies high precision and thus a Field Of View which is a Large

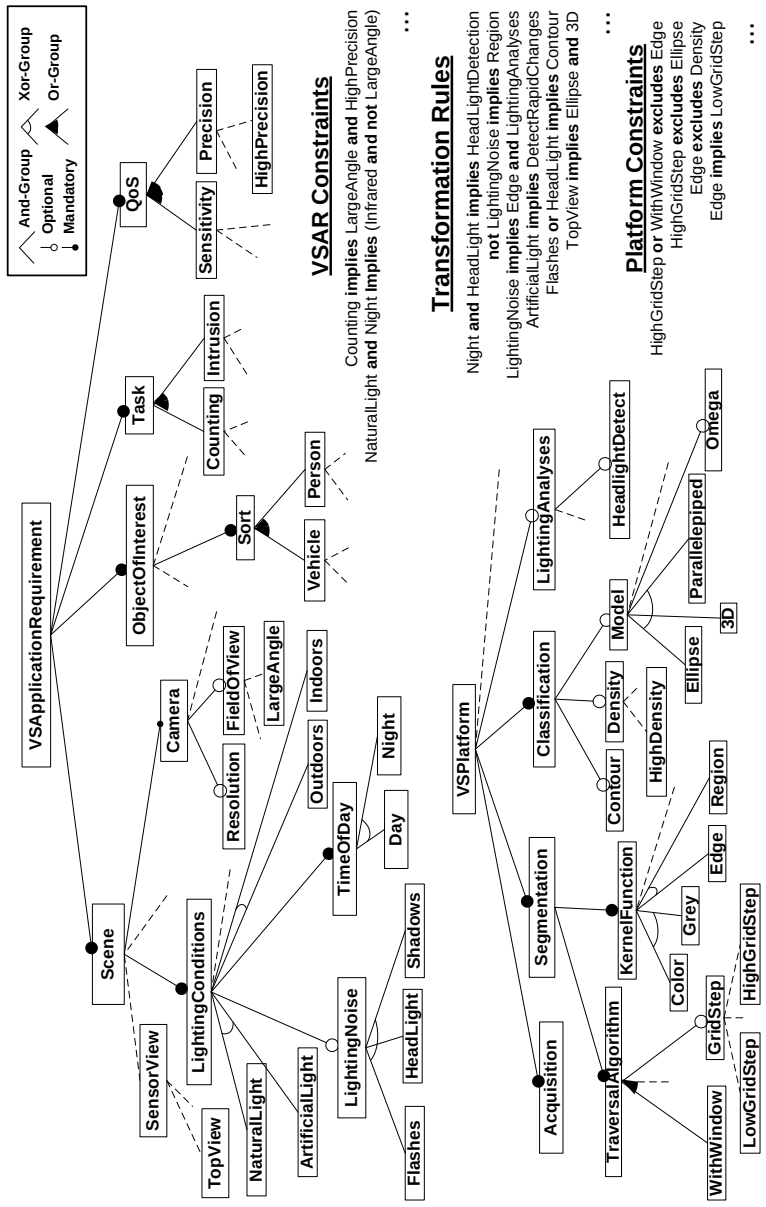


Figure 11.2: VSAR, PFC feature models and transformation rules

Angle (among others). The intra-constraints in the VSAR model show a strong dependency between the four top level features which confirms the need for integration into the same variability model.

PFC *model.* All steps of the VS processing chain correspond to software components that the designer must correctly assemble to obtain a coherent processing chain. A mandatory task is to acquire images. Then, for each step, many variants (e.g., alternative algorithms) exist. The platform FM describes the different software components of the platform, their parameters and their assembly constraints. Figure 11.2 displays a highly simplified form of the corresponding model. As shown, the top level features mainly correspond to the different steps of the processing chain. The figure focuses on the segmentation step. Similarly to the previous model, we also need to introduce internal constraints. They have the same form as before. For instance, Edge segmentation **implies** a thin image discretization, thus a Low Grid Step (see constraints related to PFC model in Figure 11.2).

Transformation Rules. Besides internal constraints, there are other constraints across models. These inter-model constraints make dependencies and interactions between features explicit by mapping the task specification to the software implementation. More importantly, such constraints prevent forbidden combinations of features, thus dramatically reducing the configuration sets. In our approach, these constraints correspond to model transformations from the VSAR model to the PFC model. They allow deriving automatically, or semi-automatically, a suitable processing chain from an application specification (see next Section). For instance, a Top View **implies** the use of both an Ellipse and a 3D model to describe persons (see transformation rules in Figure 11.2).

11.3 From Requirements to Deployment and Runtime

The platform feature model compactly represents the set of software configurations available for each category of components that can be activated to achieve the tasks of the processing chain. This is highly desirable for VS applications to cope with possible runtime change of implementation triggered by context variations. The goal is at runtime to deploy a VS processing chain able to adapt its configuration according to a valid contextual information. A key idea is that only a *specialized* feature model of VSAR is needed since the context features that may influence the runtime execution of the system is most of the time only a part of the VSAR feature model. We recall that a feature model f is a specialization of another feature model g iff $[\![f]\!] \subset [\![g]\!]$. If f has been produced from g and f is a specialization of g, we say that f is a specialized feature model of g.

11.3.1 Process

In Figure 11.3 we present a process together with sound formal basis to support rigorous reasoning and assist stakeholders until the application is deployed. In this Figure, we show also the behaviour when adaptations are performed at runtime and the operations required to ensure systematic consistency and end-to-end transformation. The two stakeholders (VS expert and software application engineer) of the approach interact with the feature models during modeling (see ❶), specialization (see ❸ and ❹) and transformation (see ❺).

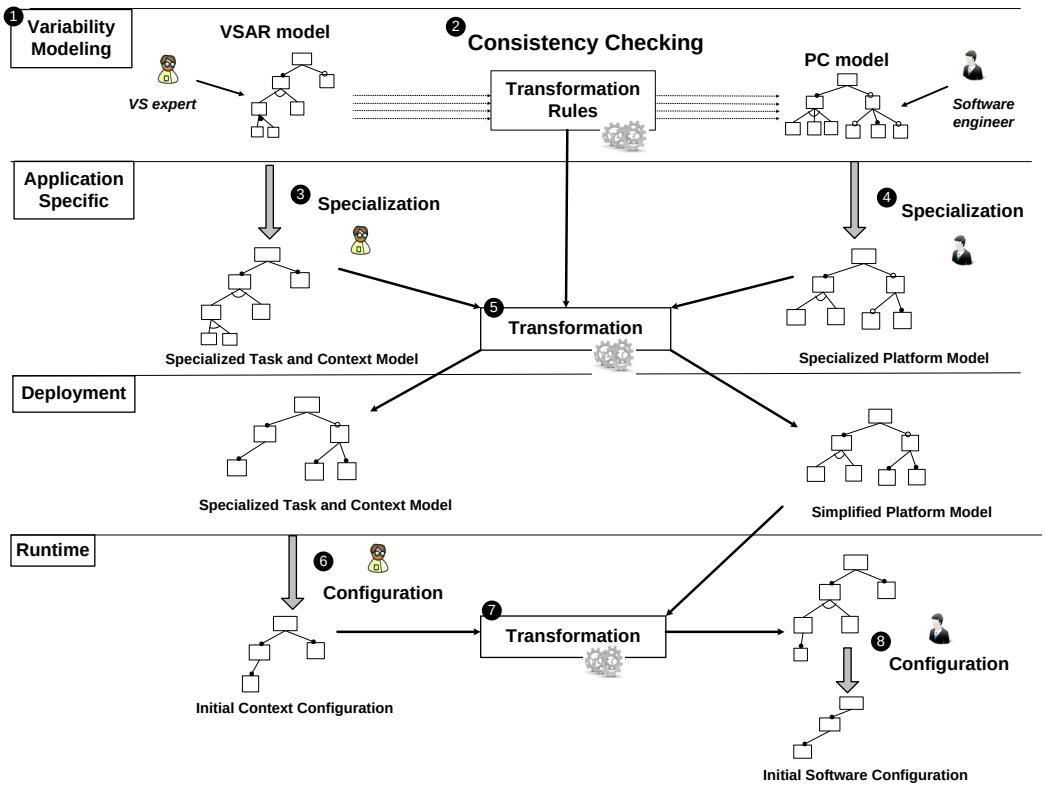


Figure 11.3: From Requirements to Deployment and Runtime: Process

During the elaboration activity, the two feature models and rules are designed by the VS expert and the software application engineer. A precondition to the configuration process is that feature models and rules are both consistent, individually as well as considered altogether ❷. It means that each feature model contains at least one valid configuration and that transformation rules have no contradictions. It is possible that each feature model is consistent individually whereas putting them together and applying rules leads to inconsistencies.

Once checked consistent, the two feature models are then edited, independently, during the *specialization* process. The specialization process conducted by the VS expert consists in a sequence of edit operations on the VSAR model, for example, the removal of an optional feature. The VS expert produces a new VSAR model, say Y, so that Y is a specialized feature model of VSAR.

The VS expert reasons over a complex set of constraints and typically removes the desired features over a series of steps, rather than in a single iteration. It is thus important, not to say mandatory, to ensure that each edit operation does specialize VSAR (❸).

Once checked, the specialization edits achieved on VSAR or PFC can lead to the automatic simplification of both feature models. For instance, when users decide that a feature

of an Xor-group is necessary included in any configuration of a feature model, the other features of the Xor-group cannot be selected and are no longer included in the feature model. An automated support for *i)* systematic specialization checking and *ii)* propagation of edits allows the VS expert to prevent unauthorized choices and better understand consequences of his/her decision at each step. At the end of the specialization process, the VS expert gets a specialized VSAR feature model where some parts still exhibit some variability (e.g., contextual information) and some other parts have been fully configured (e.g., object of interest).

From now, the VS expert can trigger the automatic transformation ❺. We propose a transformation operator **transform** that takes as input VSAR, PFC and the transformation rules, $rules$.

$$\textbf{transform} : \text{VSAR} \times \text{PFC} \times rules \rightarrow \begin{cases} \text{VSAR}_{spe} \times \text{PFC}_{spe} \\ \bot \end{cases}$$

Let VS_{full} be the aggregation of VSAR, PFC and $rules$, $\mathcal{F}_{\text{VSAR}}$ the set of features of VSAR and $\mathcal{F}_{\text{PFC}}$ the set of features of PFC. The operator computes two feature models, VSAR_{spe} and PFC_{spe}, such that

$$[\![\text{VSAR}_{spe}]\!] = \{\ x \in [\![VS_{full}]\!] \mid x \cap \mathcal{F}_{\text{VSAR}}\ \}$$

and

$$[\![\text{PFC}_{spe}]\!] = \{\ x \in [\![VS_{full}]\!] \mid x \cap \mathcal{F}_{\text{PFC}}\ \}$$

By construction, VSAR_{spe} (resp. PFC_{spe}) is a refactoring or a specialization of VSAR (resp. PFC). The operator is a partial function since the two feature models together with rules can be inconsistent. The transformational intent is to *propagate* edits to related feature models. The transformation operator reasons at the configuration level and tries to select or eliminate automatically undecided variability choices in VSAR and PFC. This is similar to what is presented in Section 7.4.1 where we show how to update feature model views using the corrective capabilities of the slice operator.

One of the result of the transformation, PFC_{spe}, is a specialization of the platform feature model. A set of variability choices still needs to be resolved in this model before being deployed ❸. In some deployment scenarios, the PFC model of the VS processing chain has also to be manually fine-tuned by the software engineer, during a specialization process ❹, as similarly done by the VS expert. The specialization processes can be reiterated by the two stakeholders.

When the application is run for the first time, the variable contextual information is set and an initial configuration of the VSAR model is produced (❻). This leads to the generation of an initial configuration for the PFC model thanks to a transformation (❼), and the corresponding software platform configuration is finally generated (❽). At runtime, the VSAR contextual information can be changed according to modifications of the real environment, enabling dynamic self-adaptation of the running system configuration. As stated early, this step is out of the scope of this chapter.

11.3.2 Supporting the Process

At different steps of the modeling process, we need to *reason* about the VSAR feature model, the PFC feature model, and both of them together with the transformation rules. Manually checking properties of the feature models is clearly not an option. The automated techniques that have already been described and developed in the previous chapters can be directly reused. Naturally we rely on FAMILIAR to support the proposed process.

Modeling variability. First, VSAR and PFC feature models are specified separately, using either the FAMILIAR notation or by importing feature models from different formats (e.g., FeatureIDE, TVL). Internal constraints are specified either during the specification or, in a modular way, using the **aggregate** operator. We only show here the VSAR model (handling of the PFC model is similar):

```
scene_fm = FM (Scene :  AprioriKnowledge Environment;
     Environment: [Noise] LightingConditions ;
     Noise: [BackgroundMovement] [LightingVariation] ;
     ... ; LightingConditions: (Indoors |
     Outdoors | LightingType)+ ; ... ; )
ooi_fm = FM (ObjectOfInterest: Cardinality ... ;
     Cardinality: (SingleObject | GroupOfObjects) ; )
QoS_fm = FM (QualityOfService: (ComputerLoad
     | ResponseTime |Quality)+ ; ... ; )
task_fm = FM (Task : (Counting | BehaviourRecognition
     | Tracking |...); ... ; )
VSARrules = constraints ( Counting -> Large & Precision
     & VingtCinqFrSec; NaturalLight & Night ->
     Infrared & !Large ; ... ; )
VSAR_fm = aggregate { task_fm ooi_fm scene_fm QoS_fm }
                          withMapping VSARrules
```

Consistency Checking. Once the two feature models have been developed, transformation rules are specified and then mapped to features using a generic script, also implemented using the modular capabilities of FAMILIAR. The script produces a new feature model that can be further analyzed. During the *elaboration* of the feature models (see ① in Figure 11.3), the presence of dead features can be considered as an error since it introduces an incorrect definition of relationships that does not match the intention of the feature models' developers. As a result, the script *detects* the errors and reports back to the user.

```
VSAR_fm = FM ("VSAR.fml")
PC_fm = FM ("PFC.fml")
// transformation rules
trRules = constraints (Counting ->  ReferenceImageUpdating
     & LowGridStep; Night & HeadLight -> HeadLightDetection ;
     Flashes | HeadLight -> Contour ; LessPrecision ->
     (GridStep | WithWindow) & !SegmFineTune;
     ArtificialLight <-> ScenarioRecognition ; ... )
run "generic_transfo" { VSAR PFC trRules }
assert (size (deads deployment_fm) == 0)
```

The generic script follows: It checks that there exists at least one valid configuration to ensure that the transformation rules do not lead to contradictory relations and detects dead features (if any).

```
// generic_transfo.fml
parameter fm1 : FeatureModel
parameter fm2 : FeatureModel
parameter rules : Set
global_fm = aggregate { fm1 fm2 }
                              withMapping rules
assert (isValid global_fm)
deads_global = deads global_fm
if size deads_global > 0 then
                     println "Dead features detected''
                     // we can enumerate the dead features
                     // ...
end
export global_fm
```

Reachability Checking. Before the execution of a system, feature models are used to verify important properties. Among others, we want to guarantee the *reachability* property, i.e., that for *all* valid specifications and contexts, there exists at least one valid software configuration. In terms of feature models, the reachability property can be formally expressed as follows:

$$\forall c \in [\![\text{VSAR}]\!], c \in [\![\Pi_{\mathcal{F}_{\text{VSAR}}}\ (VS_{full})]\!] \tag{11.1}$$

Π is the *slicing* operator described in Chapter 7. The reachability property 11.1 is similar to the realized-by property defined in Section 7.4.4 and the same techniques are thus reused:

```
VSFull_fm = aggregate { VSAR_fm PC_fm } withMapping trRules
VSARp_fm = slice VSFull_fm including VSAR_fm.*
cmpVSAR = compare VSARp_fm VSAR_fm
if (cmpVSAR neq REFACTORING) then
      fmVSARDiff = merge diff { VSAR_fm VSARp_fm }
      unrVSARContexts = configs fmVSARDiff
      //...
else
 //... all contexts are useful
end
```

A brute force strategy which consists in enumerating all possible specifications and then checking the existence of a software configuration would be clearly inappropriate, especially in our case where we have more than 10^8 valid specifications and more than 10^6 software configurations. As we have shown in Chapter 7, the combined use of **slice**, **compare** and **merge diff** are a much more scalable technique. Without these capabilities, this kind of reasoning would not be made possible for this order of complexity.

Specialization Checking. When the VS expert (resp. software engineer) edits the VSAR

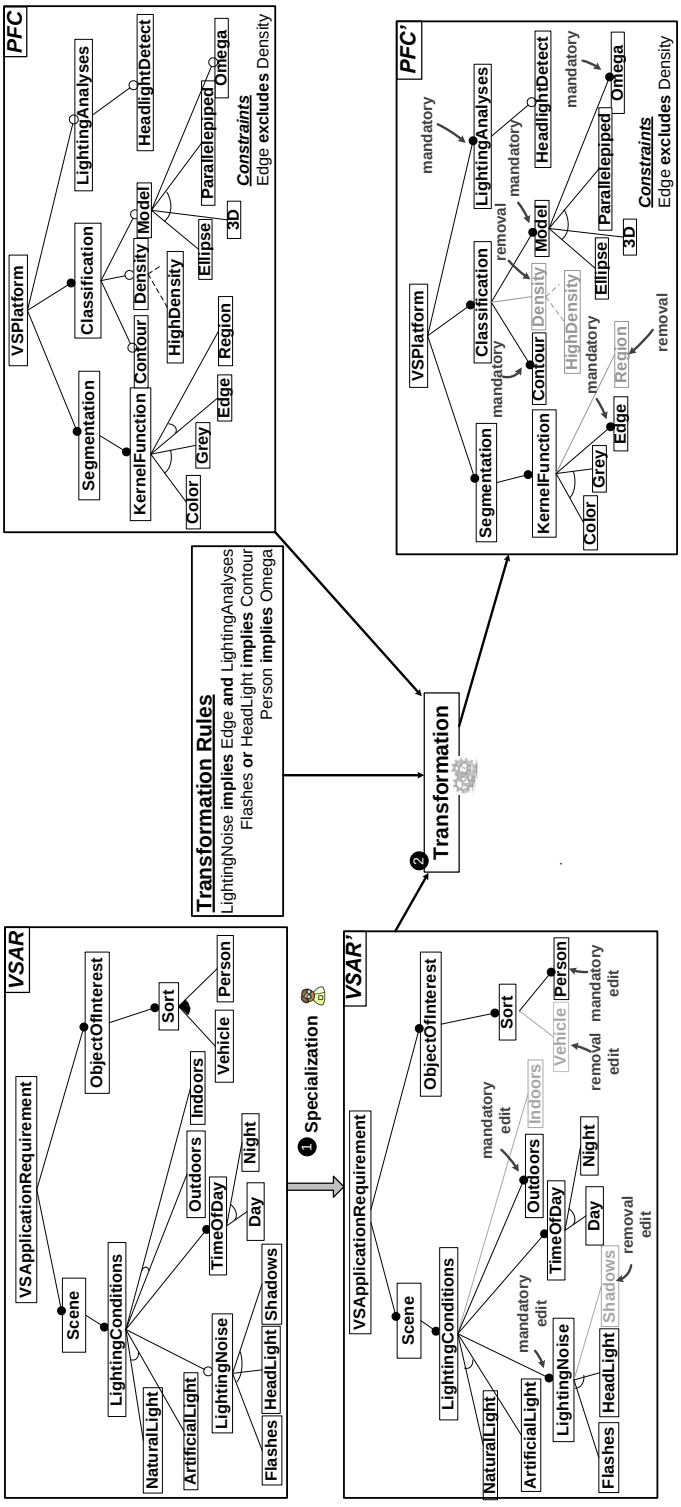


Figure 11.4: An example of specialization and transformation.

(resp. PFC) model, we control that edit operations performed do specialize the models, since some edits to VSAR (resp. PFC) may lead to an *arbitrary edit* (e.g, when a core feature is removed). The VS expert (resp. the software engineer) can use different modifier commands to specialize the VSAR (resp. PFC) model (e.g., **removeFeature**, **setMandatory** when translated into FAMILIAR):

```
// automatically generated from the interactive session
VSAR_spe = copy VSAR_fm
removeFeature VSAR_spe.LightingVariation
removeFeature VSAR_spe.Outdoors
removeFeature VSAR_spe.Counting
setMandatory VSAR_spe.Indoors
setMandatory VSAR_spe.Tracking
// ...
assert ((compare VSAR_spe VSAR_fm) eq "SPECIALIZATION")
```

Specialization and Transformation Example. Specialization choices in the VSAR feature model have possible complex consequences on features of the PFC model (and vice versa). In Figure 11.4, we illustrate the specialization activity and the automatic transformation using an excerpt of VSAR and PFC feature models and some transformation rules. First, a video surveillance expert *edits* VSAR feature model (see ①) by removing the feature Shadows and setting the mandatory status to the features Outdoors, Lighting Noise and Person. As a result, some features no longer appear in VSAR feature model (grey features in Figure 11.4). Then, the transformation operator takes the new specialized VSAR feature model, VSAR', the transformation rules and PFC feature model[2] (see ②). Hence several choices are automatically deduced in the platform feature model: the features Density, Region and High Density are removed while the features Contour, Lighting Analyses, Omega and Model become core features.

In FAMILIAR, once the two feature models VSAR and PFC have been edited, leading to VSAR_{spe}, PFC_{spe}, the **aggregate** and **slice** operators are used to propagate choices.

```
VSFull_update = aggregate { VSAR_spe PC_spe } withMapping trRules
VSAR_spe_update = slice VSFull_update including VSAR_spe.*
PC_spe_update = slice VSFull_update including PC_spe.*
// ...
```

Finally, the specialization/transformation process can be reiterated if needs be.

11.4 Case Study and Experiments

To gather some validation elements, we experimented the proposed modeling process on six deployment scenarios (detecting intrusion into building and a car park, counting persons, etc.) conducted by a VS expert using the FAMILIAR environment. For the experiments, we only considered the specialization of the VSAR feature model by the VS expert and we do not take the *direct* specialization of the PFC feature model into account. The VSAR feature model used for the experiments had 77 features and its number of valid configurations

[2]The transformation really produces two feature models but we only depict PFC' for the sake of brevity.

Scenario	#edits	#configurations	#cores	#removes	#VPs
1	13	48384	19	4	28
2	18	106560	18	2	31
3	12	24192	18	4	29
4	18	118656	18	1	32
5	16	32256	24	4	23
6	15	22608	21	2	28
PFC	-	$\geq 10^6$	13	-	38

Table 11.1: Measurements on the application of the process

was more than 10^8. The PFC feature model used for the experiments had 51 features and the number of valid configurations was more than 10^6. 22 transformation rules were considered. We checked that the *reachability* property described above truly holds.

For each scenario, we used the automated techniques described in this chapter to transform PFC feature model into PFC_{spe} given a sequence of edits performed in the VSAR model. In Table 11.1, we report for each scenario:

- the number of edits specialization performed in VSAR (#edits) ;
- the number of valid configurations in PFC_{spe} (#configurations) ;
- the number of features not present in PFC_{spe} but originally present in PFC feature model (#removes) ;
- the number of core features present in PFC_{spe} (#cores) ;
- the number of features that remain to be chosen at runtime in PFC_{spe} (#VPs). Note that $\#VPs = 51 - (\#cores + \#removes)$.

Results show that for all scenarios, after the specialization and transformation, the number of valid configurations in PFC_{spe} is significantly less important than the number of valid configurations in PFC, at least of an order of magnitude. Other criteria give more precise information about the reduction of variability. In the initial PFC model, there are 13 core features and #VPs=38. The transformation process allows for infering new core features (see #cores) and removing some others (see #removes).

#removes varies from 4 (Scenario 4 and 5) to 1 (Scenario 4). Therefore only a few features are no longer of interest at runtime. #cores varies from 18 (Scenario 2 and 3) to 24 (Scenario 4). It means that at least 5 features and at most 11 features have been bound at design time. Compared to #removes, the benefits are more significant. #VPs vary from 23 (Scenario 5) to 31 (Scenario 2), that is, at least 7 choices and at most 15 choices are no longer of interest at runtime.

One of the goal of the modeling process is to reduce the number of features that remain to be chosen at runtime. With respect to this purpose, a reduction of variability in the software part has been clearly observed for all scenarios.

Another property of the modeling process is the use of two *separated* feature models VSAR and PFC. In all scenarios, several choices have been inferred in the software part starting from a specification of the VS expert. The use of the VSAR feature model thus facilitates the specialization of the PFC feature model using high-level, domain-specific concepts. Nevertheless the exclusive use of the VSAR feature model to specialize the PFC model is not conceivable. The intervention of the software engineer is somehow required

(e.g., to fine tuned some parameters). Thanks to the reduction of variability and thanks to the separation of concerns, he/she can concentrate on his/her domain of expertise while irrelevant details have been removed.

11.5 SUMMARY

In this chapter, we have presented a process, supported by FAMILIAR, to model the variability of a dynamic adaptive system from requirements to runtime. The proposed approach distinguishes between requirements variability and software variability and explicitly breaks the variability spaces into two feature models. In order to take into account the interactions between specification and implementation choices the feature models are interrelated with propositional constraints. Starting from a high level specification (including execution environment and context), the set of valid software configurations to be considered at runtime can be automatically reduced. Before the deployment of the system, automated techniques can also be used to control that, for all valid execution contexts, there exists at least one valid software configuration.

The presented contributions are validated on a video surveillance SPL. First experiments show that, following our approach, the configuration spaces are reduced by an order of magnitude, whereas it would not have been possible without. We expect to fully integrate FAMILIAR in a run time adaptation architecture so that an end-to-end engineering of the video surveillance SPL can be made possible [Moisan et al. 2011].

Twelve

Reverse Engineering Architectural Feature Models

This chapter shares material with the ECSA'11 paper "Reverse Engineering Architectural Feature Models" [Acher et al. 2011a]. This work is based on a collaboration with Anthony Cleve (University of Namur, Belgium), Philippe Merle, the software architect of FraSCAti, and Laurence Duchien head of ADAM project-team (INRIA - LIFL CNRS/University of Lille 1).

In this chapter, we show how the operators of FAMILIAR have been applied to reverse engineer the variability model of FraSCAti[1], a large and highly configurable component and plugin based system. We develop automated techniques to extract and combine different variability descriptions of an architecture. Then, alignment and reasoning techniques are applied to integrate the architect knowledge and reinforce the extracted feature model. We also report on our experience.

12.1 Reverse Engineering Variability of FraSCAti

When SPL engineering principles are followed from the start, it is feasible to manage variability through one or more architectural feature models and then associate them to the architecture (e.g., as shown in [Parra et al. 2010; 2011]). The major architectural variations are mapped to given features, allowing for automated composition of the architecture when features are selected. In many cases, however, one has to deal with (legacy) software systems, that were not initially designed as SPLs. When such a system, like FraSCAti, offers a large number of variants, with many configuration and extension points, its variability should be properly managed. A first and essential step is to explicitly identify and represent its variability, for instance using a feature model.

Reverse engineering the feature model of an existing system is a challenging activity. The architect knowledge is essential to identify features and to explicit interactions or constraints between them. But the manual creation of feature models is both time-consuming and error-prone. On a large scale, it is very difficult for an architect to guarantee that the resulting feature model ensures a safe composition of the architectural elements when some features are selected. Both automatic extraction from existing parts and the architect knowledge should be ideally combined to achieve this goal.

In this chapter, we illustrate the problem, our proposal and report on experiments with a case study on the FraSCAti platform [FraSCAti 2011].

[1] FAMILIAR is now used to manage the development and releases of FraSCAti (see http://frascati.ow2.org/doc/1.4/ch12.html).

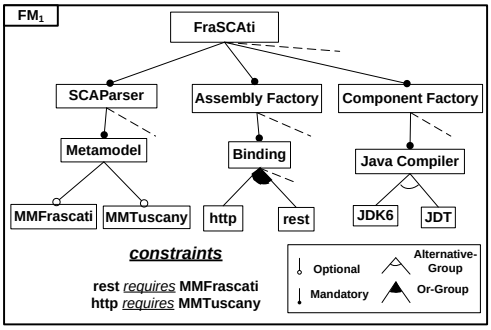


Figure 12.1: An excerpt of a possible architectural feature model

12.1.1 FraSCAti: the Need for Handling Variability

FraSCAti is an open-source implementation of the Service Component Architecture (SCA) standard [SCA standard 2007], which allows for building hierarchical component architectures with the potential support of many component and service technologies.

Started three years ago, the development of the FraSCAti platform begun with a framework, first validated by a basic implementation of the standard, and then incrementally enhanced. After four major releases, it now supports several SCA specifications (Assembly Model, Transaction Policy, Java Common Annotations and APIs, Java Component Implementation, Spring Component Implementation, BPEL Client and Implementation, Web Services Binding, JMS Binding), and provides a set of extensions to the standard, including binding implementation types (Java RMI, SOAP, REST, JSON-RPC, JNA, UPnP, etc.), component implementation types (Java, OSGi, Java supported scripting languages, Scala, Fractal), interface description types (Java, C headers, WSDL, UPnP), runtime API for assembly and component introspection/reconfiguration [Seinturier et al. 2009]. As its capabilities grew, FraSCAti has also been refactored and completely architected itself with SCA components.

With all these capabilities, the platform has become highly (re-)configurable in many parts of its own architecture. It notably exposes a larger number of extensions that can be activated throughout the platform, creating numerous variants of a FraSCAti deployment. For example, some variations consist in one or more specific components bound to many other mandatory or optional parts of the platform architecture. It then became obvious to FraSCAti technical leads that the variability of the platform should be managed to pilot and control its evolution as an SPL. Additionally, the benefits of factoring out variability should allow FraSCAti to be *"efficiently extended, changed, customized or configured for use in a particular context"* [Svahnberg et al. 2005].

12.1.2 Reverse Engineering FraSCAti as an SPL

In order to manage the FraSCAti platform as an SPL, we needed to capture its variability from the existing architecture. We rely on feature models to describe the variability of FraSCAti (see Figure 12.1 for an excerpt of an *architectural feature model*[2] of the FraSCAti

[2]It must be noted that the feature model that we discuss all along this chapter are "architectural" in that they focus on the variation points of the architecture.

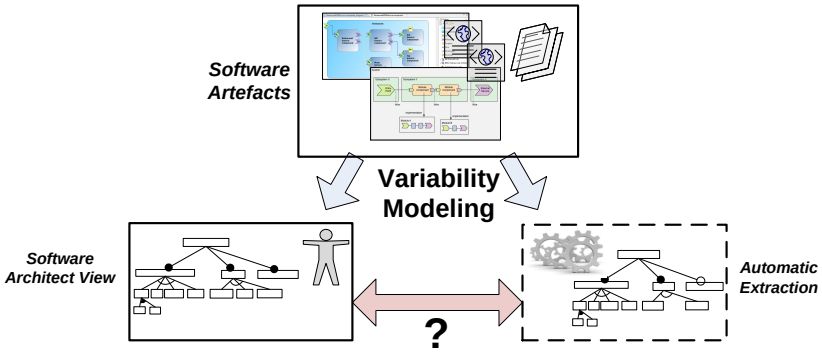


Figure 12.2: Variability Modeling from Software Artifacts

platform).

Several software artifacts (SCA composite files, Maven descriptors, informal documents) describe FraSCAti architecture, but variability, though, is not explicitly represented. As the FraSCAti main *software architect* (SA), Philippe Merle, had an extensive expertise in the architecture and in its evolution, it was decided to make him model the architecture he has in mind with variation points (see left part of Figure 12.2). As a domain expert, he had the ability to elicit the architectural variation points and explain rationale behind these decisions. To follow separation of concerns principles, it was also decided to separate the variability description from the architectural model itself. The principle is to model the variation points of the architecture, to represent them as features in an architectural feature model, and finally to describe the links between the features and the architectural elements. An important property is then to ensure consistency between feature model and architectures [Lopez-Herrejon and Egyed 2010], even if not all variability elements can be captured in a feature model.

This task resulted in a manually created feature model and it was clearly daunting, time-consuming and error-prone, requiring substantial effort from the SA. In this case as in all large scale architectures, it is very difficult to guarantee that the resulting feature model ensures a safe composition of the architectural elements when some features are selected. Another approach thus relies on an automated extraction, so that an architectural feature model that represents variability of the architecture is automatically extracted from the appropriate artifacts (see right part of Figure 12.2).This operation clearly saves time and reduces accidental complexity, but the accuracy of the results directly depends on the quality of the available documents and of the extraction procedure. This approach is notably followed in recent reverse engineering that which is doing large scale variability extraction from the Linux kernel [She et al. 2011].

The main challenge is then to reconcile these two architectural feature models into a final feature model being compatible with both the SA view and the actual architecture. It must also be noted that we could have tried to somehow *integrate* the SA knowledge in the extraction process or to let him edit an extracted feature model, but we argue that keeping the first two activities separated was better. It lets a highly experienced SA focus on its own variability scoping, and compare it afterwards to the extracted version. Moreover,

this allows for explicitly separating the required variability of the SA from the supported variability of the actual software system, as advocated in [Metzger et al. 2007].

In the next section, we describe the automated extraction process that we have applied to FraSCAti. We then show how the process is completed by refinement steps that enables the architect to compare and integrate his/her knowledge, so that a consistent architectural feature model is obtained (Section 12.3). This process is validated by experiments on the FraSCAti architecture and some lessons learned are briefly discussed.

12.2 Automatic Extraction of Architectural Feature Model

Overview. Figure 12.3 summarizes the steps needed to realize the process. First, a raw *architectural feature model*, noted $FM_{Arch_{150}}$, is extracted from a *150% architecture* of the system (see ①). The latter consists of the composition of the architecture fragments of *all* the system plugins. We call it a 150% architecture because it is not likely that the system may contain them all. Consequently, $FM_{Arch_{150}}$ does include all the *features* provided by the system, but it still constitutes an over approximation of the set of *valid combinations* of features of the system family. Indeed, some features may actually *require* or *exclude* other features, which is not always detectable in the architecture, hence the need for considering an additional source of information. We therefore also analyze the specification of the system plugins and the dependencies declared between them, with the ultimate goal of deriving inter-feature constraints from inter-plugin constraints. To this end, we extract a *plugin feature model* FM_{Plug}, that represents the system plugins and their dependencies (see ②). Then, we automatically reconstruct the bidirectional mapping that holds between the features of FM_{Plug} and those of $FM_{Arch_{150}}$ (see ③). The result of the mapping is FM_{Full}. Finally, we exploit this mapping as a basis to derive a richer architectural feature model, noted FM_{Arch}, where additional feature constraints have been added. As compared to $FM_{Arch_{150}}$, FM_{Arch} more accurately represents the architectural variability provided by the system.

12.2.1 Extracting $FM_{Arch_{150}}$

The architectural feature model extraction process starts from a set of n system plugins (or *modules*), each defining an architecture fragment. In order to extract an architectural feature model representing the entire product family, we need to consider *all* the system plugins at the same time. We therefore produce a *150% architecture* of the system, noted $Arch_{150}$. It consists of a hierarchy of components. In the SCA vocabulary, each component may be a composite, itself further decomposed into other components. Each component may provide a set of *services*, and may specify a set of *references* to other services. Services and references having compatible *interfaces* may be bound together via *wires*. Each wire has a reference as *source* and a service as *target*. Each reference r has a *multiplicity*, specifying the minimal and maximal number of services that can be bound to r. A reference having a $0..1$ or $0..N$ multiplicity is *optional*.

Note that $Arch_{150}$ may not correspond to the architecture of a *legal* product in the system family. For instance, several components may exclude each other because they all define a service matching the same $0..1$ reference r. In this case, the composition algorithm binds only one service to r, while the other ones are left unbound in the architecture.

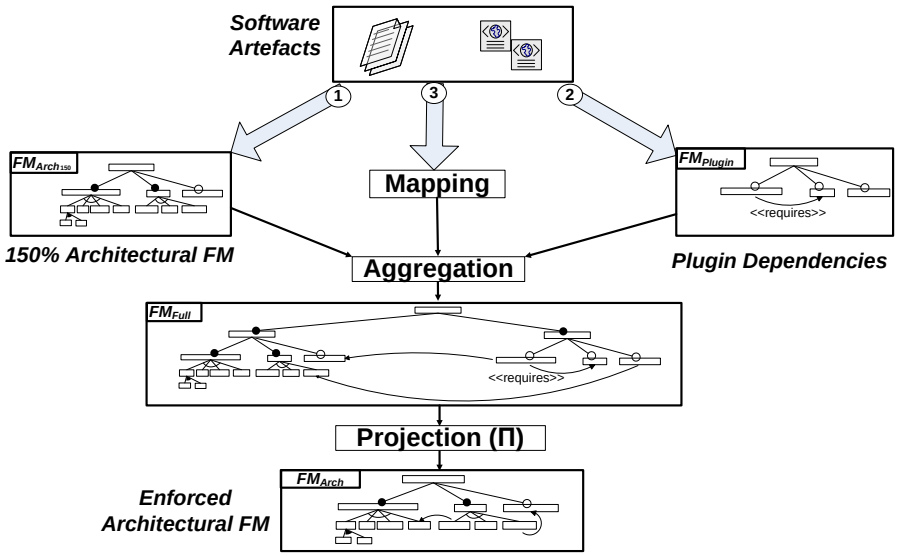


Figure 12.3: Process for Extracting FM_{Arch}

Since the extracted architectural feature model should represent the *variability* of the system of interest, we focus on its *extension points*, typically materialized by *optional* references. Algorithm 1 summarizes the behavior of the feature model extractor. The root

Algorithm 1 $ExtractArchitecturalFM_{150}(Arch_{150})$

Require: A 150% architecture of the plugin-based system ($Arch_{150}$).
Ensure: A feature model approximating the system family ($FM_{Arch_{150}}$).
1: $root \leftarrow MainComposite(Arch_{150})$
2: $f_{root} \leftarrow CreateFeature(root)$
3: $FM_{Arch_{150}} \leftarrow SetRootFeature(FM_{Arch_{150}}, f_{root})$
4: **for all** $c \in FirstLevelComponents(root)$ **do**
5: $f_c \leftarrow CreateFeature(c)$
6: $FM_{Arch_{150}} \leftarrow AddMandatoryChildFeature(FM_{Arch_{150}}, f_{root}, f_c)$
7: $FM_{Arch_{150}} \leftarrow AddChildFeatures(FM_{Arch_{150}}, c, f_c, Arch_{150})$
8: **end for**

feature of the extracted feature model (f_{root}) corresponds to the main composite ($root$) of $Arch_{150}$. The child features of f_{root} are the first-level components of $root$, the latter being considered as the main system features. The lower-level child features are produced by the $AddChildFeatures$ function (Algorithm 2). This recursive function looks for all the optional references r of component c and, for each of them, creates an optional child feature f_r, itself further decomposed through a *XOR* or an *OR* group (depending on the multiplicity of r). The child features f_{c_s} of the group correspond to all components c_s providing a service compatible with r.

Algorithm 2 $AddChildFeatures(FM, c, f_p, Arch_{150})$

Require: A feature model (FM), a component (c), a parent feature (f_p), a 150% architecture ($Arch_{150}$).
Ensure: FM enriched with the child features of f_p, if any.

```
1:  for all r ∈ OptionalReferences(c) do
2:    MC ← FindMatchingComponents(Arch150, r)
3:    if MC ≠ ∅ then
4:      f_r ← CreateFeature(r)
5:      FM ← AddOptionalChildFeature(FM, f_p, f_r)
6:      if Multiplicy(r) = 0..1 then
7:        g ← CreateXORGroup()
8:      else if Multiplicy(r) = 0..N then
9:        g ← CreateORGroup()
10:     end if
11:     FM ← AddGroup(FM, f_r, g)
12:     for all c_s ∈ MC do
13:       f_{c_s} ← CreateFeature(c_s)
14:       FM ← AddChildFeatureOfGroup(FM, g, f_{c_s})
15:       FM ← AddChildFeatures(FM, c_s, f_{c_s}, Arch150)
16:     end for
17:   end if
18: end for
```

12.2.2 Extracting FM_{Plug}

The extraction of the plugin feature model FM_{Plug} starts from the set of plugins $P = \{p_1, p_2, \ldots, p_n\}$ composing the system. This extraction is straightforward: each plugin p_i becomes a feature f_{p_i} of FM_{Plug}. If a plugin p_i is part of the system core, f_{p_i} is a mandatory feature, otherwise it is an optional feature. Each dependency of the form p_i *depends on* p_j is translated as an inter-feature dependency $f_{p_i} requires f_{p_j}$. Similarly, each p_i *excludes* p_j constraint is rewritten as an *excludes* dependency between f_{p_i} and f_{p_j}.

12.2.3 Mapping $FM_{Arch_{150}}$ and FM_{Plug}

When producing $Arch_{150}$, we keep track of the relationship between the input plugins and the architectural elements they define, and vice versa. On this basis, we specify a bidirectional mapping between the features of $FM_{Arch_{150}}$ and those of FM_{Plug} by means of *requires* constraints. This mapping allows us to determine (1) which plugin provides a given architectural feature, and (2) which architectural features are provided by a given plugin.

12.2.4 Deriving FM_{Arch}

We now illustrate how we derive FM_{Arch} using $FM_{Arch_{150}}$, FM_{Plug}, the mapping between FM_{Plug} and $FM_{Arch_{150}}$, and the *slice* operation using the example of Figure 12.4.

First FM_{Plug} and $FM_{Arch_{150}}$ are *aggregated* under a synthetic root $FtAggregation$ so that root features of input feature models are mandatory child features of $FtAggregation$. The aggregation operation corresponds to the one described in Chapter 5 and produces a

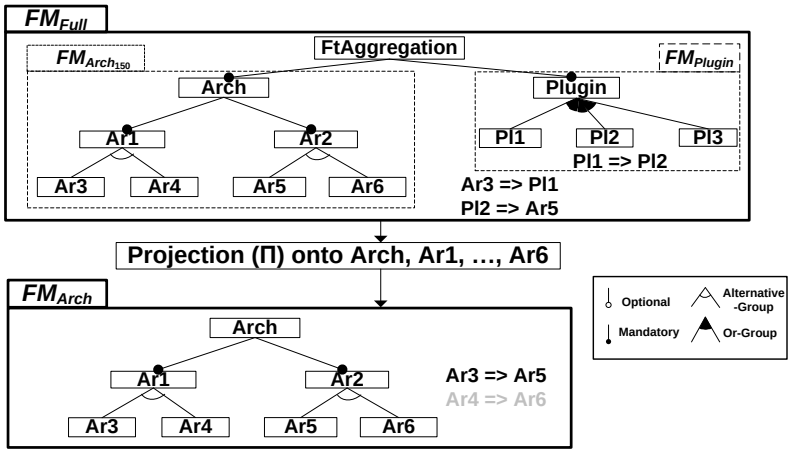


Figure 12.4: Enforcing architectural FM using aggregation and slicing.

new feature model, called FM_{Full} (see Figure 12.4). The propositional constraints relating features of FM_{Plug} to features of $FM_{Arch_{150}}$ are also added to FM_{Full}.

Second, we compute the *slice* of FM_{Full} onto the set of features of $FM_{Arch_{150}}$ (i.e., $\mathcal{F}_{FM_{Arch_{150}}} = \{Arch, Ar1, \ldots, Ar6\}$). We recall that the slice operation, given an input feature model FM_i and a set of features $ft_1, ft_2, ..., ft_n \subseteq \mathcal{F}_{FM_i}$, produces a new feature model, FM_{slice}, such that: $[\![FM_{slice}]\!] = \{\ x \in [\![FM_i]\!] \mid x \cap \{ft_1, ft_2, ..., ft_n\}\ \}$

In our case, the feature model produced by the slice operator is FM_{Arch} (see Figure 12.4). Formally:

$$\Pi_{\mathcal{F}_{FM_{Arch_{150}}}}(FM_{Full}) = FM_{Arch}$$

In the example of Figure 12.4, the relationship between $[\![FM_{Full}]\!]$ and $[\![FM_{Arch}]\!]$ is as expected. We can notice that one configuration of the original $FM_{Arch_{150}}$ has been removed, i.e., $[\![FM_{Arch_{150}}]\!] \setminus [\![FM_{Arch}]\!] = \{Ar1, Ar2, Ar3, Ar6, Arch\}$. Indeed the resulting feature model FM_{Arch} contains an additional constraint $Ar3 \Rightarrow Ar5$, that was not present in $FM_{Arch_{150}}$. Similarly, the constraint $Ar4 \Rightarrow Ar6$ (grey tint in Figure 12.4) can been derived but is redundant with $Ar3 \Rightarrow Ar5$. As we will see below, such constraint derivation can dramatically reduce the set of configurations of $FM_{Arch_{150}}$.

12.2.5 Practical Realization in FAMILIAR

Using FAMILIAR, we can realize the extraction process described in Figure 12.3. The following FAMILIAR script has been developed to illustrate the example of Figure 12.4.

First, we perform the aggregation of $FM_{Arch_{150}}$ and FM_{Plug} together with constraints (see lines 1-3). Then the slice is performed (line 5) and we obtain the enforced architectural feature model FM_{Arch}. Finally, we can compare FM_{Arch} and $FM_{Arch_{150}}$ (line 7) to determine whether some configurations have been removed. If it is the case, we can compute the difference between the two set of configurations $[\![FM_{Arch}]\!]$ and $[\![FM_{Arch_{150}}]\!]$ (lines 10-15).

```
1 fmArch150 = FM ( Arch : Ar1 Ar2; Ar1: (Ar3|Ar4); Ar2 : (Ar5|Ar6); )
2 fmPlugin = FM ( Plugin : (Pl1|Pl2|Pl3)+ ; Pl1 -> Pl2 ; )
```

```
fmFull = aggregate { fmArch150 fmPlugin } withMapping
                                  constraints (Ar3 -> Pl1 ; Pl2 -> Ar5; )
fmArch = slice fmFull including fmArch150.* // enforced architectural FM

if (compare fmArch fmArch150 eq SPECIALIZATION) then
            // we now compute the difference bewteen the set of configurations
            //of fmArch150 and fmArch
            // fmArch150Removal represents this difference
             fmArch150Removal = merge diff { fmArch150 fmArch }
                println "configurations removed from fmArch150="
                smissing = configs fmArch150Removal
                foreach (s in smissing) do
                        println s
                end
else
        // refactoring
    println "configurations of fmArch150 have not been modified"
end
```

12.3 Refining the Architectural Feature Model: Application

We conduct a study to *i)* determine if the architectural feature model designed by the SA[3], noted FM_{SA}, is *consistent* with the extracted feature model FM_{Arch} (and vice-versa) ; *ii)* step-wise refine FM_{SA} based on the previous observations. We describe the techniques developed for the case study and analyze the results[4].

12.3.1 Tool Support

We use FAMILIAR for two main purposes. Firstly, the extraction procedure *generates* FAMILIAR script to compute FM_{Arch}, similarly as the one illustrated above on a small scale. Secondly, FAMILIAR provides the SA with a dedicated approach for easily manipulating feature models during the *refinement* process.

12.3.2 Results

Automatic Extraction. In our case study, the $FM_{Arch_{150}}$ produced by the extraction procedure contains 50 features while the FM_{Plug} contains 41 features. The aggregated feature model, FM_{Full}, resulting from $FM_{Arch_{150}}$, FM_{Plug} and the bidirectional mapping contains 92 features and 158 cross-tree constraints. We first verify some properties of FM_{Full}. By construction, we know that the slicing of FM_{Full} onto $\mathcal{F}_{FM_{Arch_{150}}}$ is either a refactoring or a specialization of $FM_{Arch_{150}}$.

We observe that FM_{Arch} is a specialization of $FM_{Arch_{150}}$. More precisely, $FM_{Arch_{150}}$ admits 13.958.643.712 possible configurations ($\approx 10^{11}$), whereas FM_{Arch} represents 936.576 distinct products ($\approx 10^6$). As expected, the slicing technique significantly reduces the over approximation of $FM_{Arch_{150}}$.

[3] P. Merle, principal FraSCAti architect, plays the role of the SA in this study.

[4] See https://nyx.unice.fr/projects/familiar/wiki/ArchFm for further details about the case study.

To improve the understanding of the difference between two feature models, we use the merge *diff* operator described in Chapter 5 and denoted as $FM_1 \oplus_{\setminus} FM_2 = FM_r$. We recall that a feature model f is a *specialization* or a *refactoring* of g if $(f \oplus_{\setminus} g)$ has no valid configuration (see Lemma 1, page 82). Determining the kind of relationship (e.g., refactoring, specialization) between two feature models can thus be done by reusing the algorithm presented in [Thüm et al. 2009] or by using the merge diff operator. Besides, the diff operator can compute the difference (if any) between two feature models in terms of set of configurations. In particular, we can compute the cardinality of this set. For example, we correctly check the following relationship using the tool support: $|FM_{Arch_{150}}|$ - $\left|FM_{Arch_{150}} \oplus_{\setminus} FM_{Arch}\right| = |FM_{Arch}|$ where $|FM_i|$ denotes the number of configurations of FM_i, i.e., $|FM_i| = |[\![FM_i]\!]|$.

Refining Architectural feature models. The goal of the reverse engineering process is to elaborate a feature model that *accurately* represents the valid combinations of features of the SPL architecture. The absence of a *ground truth* feature model (i.e., a feature model for which we are certain that each combination of features is supported by the SPL architecture) makes uncertain the accuracy of variability specification expressed in FM_{Arch} as well as in FM_{SA}. It is the role of the SA to determine if the variability choices in FM_{SA} (resp. FM_{Arch}) are coherent regarding FM_{Arch} (resp. FM_{SA}). In case variability choices are conflicting, the SA can *refine*[5] the architectural feature model.

We now report the problems encountered when reasoning about the relationship between FM_{Arch} and FM_{SA}. We also describe the advanced techniques to assist the SA.

Reconciling FM_{Arch} and FM_{SA}. A first obstacle is related to the need of *reconciling* FM_{Arch} and FM_{SA} (see Figure 12.5). Both feature models come from difference sources and a preliminary work is needed before reasoning about their relationship. Firstly, the *vocabulary* (i.e., names of features) used differs in both feature models and should be aligned consequently. To solve this issue, we rely on string matching techniques (i.e., Levenshtein distance)[6] to automatically identify features of FM_{Arch} that correspond to features of FM_{SA}. Then a renaming is applied on all corresponding features in FM_{Arch}. As an example, "MMFraSCAti" of FM_{SA} has been identified to correspond to "sca_metamodel_frascati" of FM_{Arch} and after the renaming FM_{Arch} contains the feature "MMFraSCAti". We automatically detect 32 features. The SA manually specifies the mapping for 5 features in which the automated detection does not succeed (e.g., "MembraneFactory" corresponding to "fractal_bootstrap_class_providers"). Secondly, *granularity* details differ, (i.e., some features in one feature model are not present in the other feature model): FM_{SA} only contains 39 features whereas FM_{Arch} contains 50 features.

In FM_{SA} but not in FM_{Arch}. Two exclusive features $Felix$ and $Equinox$ are present in FM_{SA} but not in FM_{Arch}. We also observed that the two features are present in FM_{Plug} but not in $FM_{Arch_{150}}$ (and hence not in FM_{Arch}). A discussion with the SA reveals that these two plugins do not explicitly define architecture fragments in SCA. As a conse-

[5]Here *refine* does not necessary mean *specialize*. It can be any edit of a feature model, for example, the removal or addition of a feature or a constraint. Therefore the refinement process can produce a new feature model whose set of configurations is not a subset of the original feature model.

[6]Note that FAMILIAR does not include built-in mechanisms to support this kind of operation (this is further discussed in Chapter 14). This step can be seen as a pre-processing step realized by a third party tool.

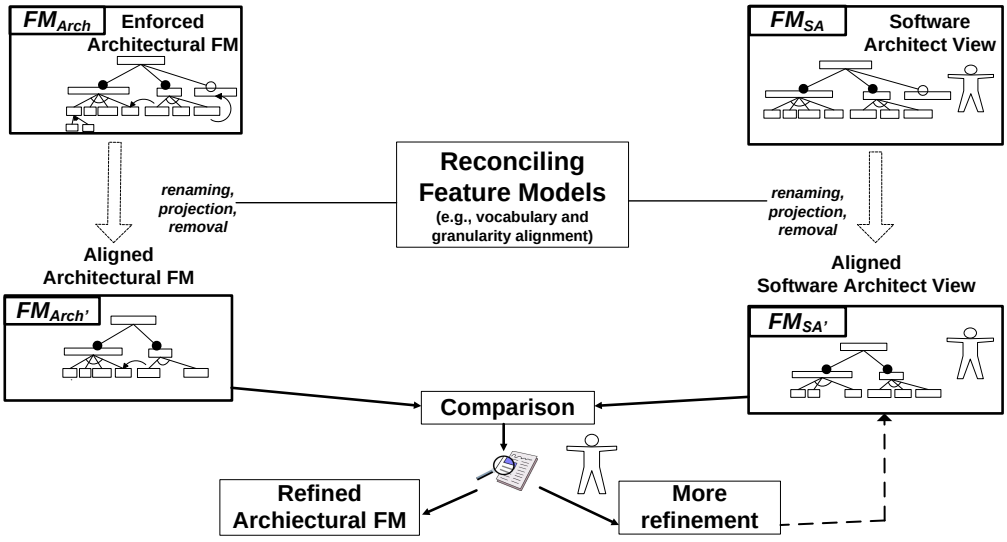


Figure 12.5: Process for Refining FM_{Arch}

quence, this variability point can simply not be identified in the architecture by the automatic extraction procedure.

In FM_{Arch} ***but not in*** FM_{SA}**.** We identified 13 features that are present in FM_{Arch} but not in FM_{SA}. Among others, two metamodels used by the SCA parser, three Bindings, two SCA properties, two Implementations and one Interface were missing. Given the complexity of the FraSCAti project, this is not surprising that the SA has forgotten some features. Hence, for most of the features, the SA considers the missing features as relevant and thus adds them in FM_{SA}. For one of the missing feature, "sca_interface_java", the SA reveals that he *intentionally* ignored it in FM_{SA}, arguing that it is a mandatory feature (i.e., every FraSCAti configuration has a Java interface) and that his focus was on variability rather than commonality. We indeed verify the mandatory nature of "sca_interface_java" in FM_{Arch}. Nevertheless, the SA decides to add "sca_interface_java" in FM_{SA}. Similarly, two first-level mandatory features, "binding_factory" and "services", were missing in FM_{SA}. The SA intentionally did not include the two features since they do not convey any further variation points, but he decides to edit FM_{SA} by adding those features. Another example concerns a feature of FM_{Arch}, "juliac", that adds unnecessary details (so that the way features are organized in FM_{SA} and FM_{Arch} slightly differ). Here the SA decides to remove "juliac" by projection.

Reasoning about FM_{Arch} ***and*** FM_{SA}**.** At this step, we can *compare* FM_{Arch} and FM_{SA}. A first comparison is to determine the kind of relationship between FM_{Arch} and FM_{SA}. We obtain an arbitrary edit, that is, some configurations of FM_{Arch} are not valid in FM_{SA} (and vice-versa). To go further, we use the merge diff operator and the merge in intersection mode (see [Acher et al. 2010a]). We enumerate and count the unique configurations of FM_{Arch} and FM_{SA} as well as the common configurations of FM_{Arch} and FM_{SA}.

Nevertheless, the techniques appear to be insufficient to really understand the difference between the two feature models. Intuitively, we need to identify more *local* differences. A first technique is to syntactically compare the variability associated to features of FM_{Arch} and FM_{SA} that have the same name. In particular, we detect that *i)* four features are optional in FM_{Arch} but mandatory in FM_{SA} and *ii)* three sets of features belong to Or-groups in FM_{Arch} whereas in FM_{SA}, the features are all optional. A second technique is to compute the intersection and the difference of the sets of requires constraints in FM_{Arch} and FM_{SA} based on their *implication graphs* (see page 65).

Step-wise Refinement of FM_{SA}**.** The comparison techniques have been reiterated until having a satisfactory architectural feature model. Based on the comparison results, the SA had several attitudes. Firstly, he used FM_{Arch} to *verify* the coherence of his original variability specification in FM_{SA}. Secondly, he considered that some variability decisions in FM_{SA} are correct despite their differences with FM_{Arch}. The SA imposed five variability decisions not identified by the extraction procedure. Thirdly, he edits FM_{SA}, for example, by adding some constraints only present in FM_{Arch} or by setting optional a feature originally mandatory. The extracted feature model notably identifies nine "obvious" constraints not expressed in FM_{SA} and allows the SA to incrementally correct FM_{Arch}.

12.3.3 Lessons Learned

The FraSCAti case study provides us with interesting insights into the reverse engineering of architectural feature models. First, the gap between FM_{SA} and FM_{Arch} appears to be manageable, due to an important similarity between the two feature models. However, it remains helpful to assist the SA with automated support, in particular, to establish correspondences between features of the two feature models. The most time-consuming task was to reconcile the granularity levels of both feature models. For this specific activity, tool supported, advanced techniques, such as the safe removal of a feature by projection, are not desirable but mandatory, basic manual edits of feature models are not sufficient.

Second, our extraction procedure (Section 12.2) yields very promising results. It recovers most of the variability expressed in FM_{SA} and encourages the SA to correct his initial model. A manual checking of the five variability decisions imposed by the SA shows that the extraction is not faulty. It correctly reproduces the information as described in the software artefacts of the project.

Third, the SA *knowledge* is also required *i)* to scope the SPL architecture (e.g., by restricting the set of configurations of the extracted feature model), especially when software artefacts do not correctly document the variability of the system and *ii)* to control the accuracy of the automated procedure. An open issue is then to provide a mechanism and a systematic process to reuse the SA knowledge, for example, for another version of the architectural feature model of FraSCAti.

12.4 RELATED WORK

In their recent work, She et al. [She et al. 2011] proposed a reverse engineering approach combining two distinct sources of information: textual feature descriptions and feature dependencies. Our approach also benefits from the combination of two (other) sources of information, namely plugin dependencies and architecture fragments. She et al. mostly focus on the retrieval of the feature diagram (heuristics for identifying the most likely

parent feature candidates of each feature, group detection, etc.) and assume that the set of valid configurations is correctly restituted, which is clearly not the case in our work. We also support the identification of feature groups (based on architectural extension points), of the right parent feature of each feature (based on architectural hierarchy) and of inter-feature dependencies (through projection of plugin dependencies).

Metzger et al. [Metzger et al. 2007] proposed an approach to cross-check *product-line* variability and *software* variability models, thus assuming that such models (or views) are available. Our approach is complementary since it allows to recover the *actually supported* variability of a software system, and since it involves the cross-analysis of architectural and plugin feature model. One of the key component and original contribution of our work is the combined used of aggregate and slicing operators.

Thüm et al. [Thüm et al. 2009] reasoned on the nature of feature model edits, and provided a classification that we rely on when comparing the extracted feature model with the software architect view. As we have shown, reasoning about the relationship of two feature models is inappropriate until feature models are not reconciled, i.e., pre-directives (e.g., safe removal of unnecessary details) have to be applied before. Moreover, the comparison operator developed in [Thüm et al. 2009] considers only the *concrete* features (i.e., leaves) but our experience shows that it is important to consider *all* features of the two feature models (e.g., a non leaf feature can be concrete).

12.5 SUMMARY

In this chapter, we presented a tool-supported approach to reverse engineer software variability from an architectural perspective. The reverse engineering process involves the automatically supported extraction, aggregation, slicing, alignment and comparison of feature models. It has the merit of combining several sources of information, namely software architecture, plugin dependencies and software architect knowledge. We successfully evaluated the proposed approach when applied to FraSCAti, a large and highly configurable plugin-based system. We showed that our automated procedures allow for producing both correct and useful results, thereby significantly reducing manual effort.

We learned, however, that fully automating the process is not realistic nor desirable, since the intervention of the software architect remains highly beneficial. The ongoing evolution of the FraSCAti project will bring us an opportunity to study how to reuse the accumulated knowledge of the software architect. As the validation was only conducted on a single case study, we need, on the long term, to adapt the proposed process to show its applicability to other forms of architecture (e.g., OSGi) and other architectural concepts. This should make significant steps to the provision of a validated, systematic process for extracting architectural feature models.

Part V

Conclusion and Perspectives

Thirteen

Conclusion

Feature models are a fundamental formalism for specifying and reasoning about commonality and variability of software product lines (SPLs). Feature models are becoming increasingly complex, handled by several stakeholders or organizations, used to describe features at various levels of abstraction and related in a variety of ways. In different contexts and application domains, maintaining a single large feature model is neither feasible nor desirable. Instead, multiple feature models are now used. In Part I of this document, we discussed this increasing complexity in feature modeling and the need of managing multiple feature models. We based our study on numerous examples in the literature as well as on our own experience (scientific workflow design in the medical imaging domain, variability modeling from requirements to runtime for the development of video surveillance systems, reverse engineering architectural feature models of component and plugin based system). In Chapter 4, we identified that a comprehensive support for separation of concerns (SoC) as well as automated reasoning techniques are needed to manage feature models on a large scale.

In this thesis, we developed theoretical foundations and practical support for managing multiple feature models and we described their use in different application domains.

In Part II, we applied the principles of SoC to feature modeling. We designed and developed a set of composition and decomposition operators (aggregate, merge, slice) dedicated to the formalism of feature model. The merge operator produces more compact feature models than existing techniques and thus eases the management, understanding and analysis of sets of feature models. The slice operator allows SPL practitioners to find semantically meaningful decompositions of a feature model. The implementation of the two operators guarantees that the set of configurations and the hierarchy of the produced feature model are consistent with the semantics we have defined for these operators. Furthermore, we showed how these operators enable automated reasoning and form a consistent and powerful support for SoC in feature modeling. Among others, we illustrated how these operators can be applied to manage a catalog of legacy medical imaging services, to update and extract feature model views, to reconcile feature models or to reason about two kinds of variability.

In Part III, we introduced FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning), a textual and executable domain-specific language that provides a practical support for managing multiple feature models. We gave an overview of syntactic facilities as well as operators provided in FAMILIAR so that feature model users can import, export, edit, configure, compose, decompose, configure and reason about feature models. FAMILIAR has been and is currently used in the different case studies described in this thesis. We illustrated how a catalog of medical imaging services can be practically

managed using reusable FAMILIAR scripts and the automated reasoning techniques provided in FAMILIAR. We described some details of the implementation (comprehensive environment, connection with other feature modeling frameworks, internal use of solvers) and we evaluated the performance of two important operators (merge and slice) of the language. The experiments showed that the order of complexity of publicly available feature models can be easily handled, but also that the current practical limits of the operators open new perspectives, especially for managing very large feature models.

In Part IV, we showed how the operators and FAMILIAR have been applied in different application domains. In the medical imaging domain, we proposed a comprehensive modeling process and tooling support (including FAMILIAR and a set of domain specific languages) for combining multiple variability artifacts with the purpose of assembling coherent scientific workflows. Separated feature models are used to describe the variability of the different artifacts. At each step of the workflow design, automated reasoning techniques assist medical imaging experts in selecting services from among sets of competing services organized in a catalog while guaranteeing that the composition of services does not violate important constraints. Our first experimental results showed that the overall approach offers an adequate user assistance and degree of automation for managing the large number of features and feature models, thereby decreasing the effort and time needed.

In another case study related to video surveillance systems, we proposed a modeling process to model the variability from requirements to runtime. The process distinguishes between domain variability and software variability and explicitly breaks these variability spaces into two feature models. We developed the idea that in dynamic, self-adaptive software systems (such as video surveillance systems) only a small part of the feature model related to requirements has to be kept, for example, features related to the context. The resulting specialized model can be efficiently used to pilot self-adaptation mechanisms. Using automated techniques, FAMILIAR and its environment as tool support, we described how the variability requirements can be expressed and then refined at design time so that the set of valid software configurations to be considered at runtime can be highly reduced. Furthermore, we showed that the proposed techniques are more scalable than existing ones, up to the point that checking some important properties (e.g., realizability) would not be possible without them. First experiments show that, following our approach, the configuration spaces is reduced by an order of magnitude for different deployment scenarios of video surveillance systems.

The last application concerns the reverse engineering of the variability of FraSCAti, a large and highly configurable component and plugin based system. In this context we developed automated techniques to extract and combine different variability descriptions of an architecture. Alignment and reasoning techniques have been applied to integrate the architect knowledge and reinforced the extracted feature model. The reverse engineering process has been made possible by the combined use of FAMILIAR operators – advanced techniques, such as the safe removal of a feature by slicing, are required. Our experience in the context of FraSCAti shows that the automated procedures produce both correct and useful results, thereby significantly reducing manual effort.

Both the operators and the FAMILIAR language bring new capabilities to the feature model users. Without these capabilities, some analysis and reasoning operations would not be made possible in the different case studies. We also showed, throughout this document, how existing approaches and scenarios for managing feature models can benefit

from our techniques: We revisited and reimplemented some of them and we observed better results in terms of scalability, reasoning capabilities, readability and evolution of feature models, or SoC support.

The results of this research look promising to us. Though the operators, language and its environment have been used in different application domains, by different people, sometimes external to our team, a wide adoption is still missing. Important questions arise and further research is needed. The performance and the adequacy of the operators should be analyzed, i.e., is the set of operators comprehensive enough to meet the requirements of feature models users? Furthermore, the quality of the FAMILIAR environment must be improved so that user experiments and qualitative assessments of the language can be conducted qualitatively. We also need to investigate further the applicability of our techniques (e.g., in other domains). The presented results also rely on the assumptions that the manipulated feature models are propositional feature models, without any extensions. Some extensions, such as feature attributes, are gaining importance in the SPL field and should be considered. We hope this additional research effort will help us to gather more validation elements.

Concerning our case studies:

- We need to reiterate the reverse engineering process on the ongoing evolution of FraSCAti and show its applicability to other forms of architecture (e.g., OSGi) ;
- FAMILIAR should be fully integrated in a run time adaptation architecture so that an end-to-end engineering of the video surveillance SPL can be made possible ;
- In the design of medical imaging workflows, a more comprehensive tool support, including graphical facilities, is needed as well as the conduct of further experiments.

In this thesis, we have considered the problem of engineering SPLs from a feature modeling perspective. We need to consider the problem from a more general perspective, including the different artifacts (e.g., source code, models) of an SPL. In particular, an important research direction is to investigate, at the theoretical and practical level, the relationship between multiple feature models and other artefacts (i.e., models) of an SPL.

Fourteen

Perspectives

We now discuss various perspectives of this research. Firstly, we focus on research questions that arise from the current results and related to the operators, the language FAMILIAR and the applicability of our techniques. This line of research will aim at consolidating our work and gathering more validation elements. Secondly, although feature models and the management of multiple feature models are important for SPL engineering, we need to further investigate the relationship between feature models and other artifacts.

14.1 Towards a Comprehensive Set of Operators

A large set of operators is provided in FAMILIAR. Nevertheless, we cannot guarantee that the operators are comprehensive enough to meet the requirements of feature models users. New techniques may be needed and emerge. We discuss here two kinds of techniques we have identified to be relevant (from our experience in two case studies).

Alignment (or Reconciliation) of Feature Models. In an open, distributed environment, suppliers can use different hierarchies, concepts, vocabulary, etc. when elaborating the feature models. It may become an issue when merging feature models but also when comparing two feature models. In such circumstances, the need for feature model *alignment* (or reconciliation) arose out of the need to integrate several feature models from different, independent sources. As previously mentioned, the alignment effort is not significant in the case study related to the medical imaging domain since suppliers rely on a common ontology [Temal et al. 2008] while feature models are views on such an ontology [Fagereng Johansen et al. 2010]. Moreover, a supplier may rely on an existing merged feature model and then edit the feature model. In this case, a supplier reuses the same hierarchy, vocabulary, etc. and alignment issues are avoided.

Nevertheless, the feature model alignment problem may occur more often in other applications or domains. To handle such situations, our techniques have to be extended, beyond manually restructuring the hierarchy and renaming or removing features. The techniques proposed in Chapter 7 are a first step. They consist in removing unnecessary details present in one feature model but not in another. In Chapter 12, we have practically applied these techniques to reconcile the feature model designed by the software architect and the feature model produced by the extraction procedure – and we obtained promising results. Nevertheless, we should apply these techniques on a larger scale and establish whether they are sufficient.

In addition, we think the techniques can be more automated. In particular, an error-prone and time-consuming task is to identify concepts (i.e., features) that are equivalent.

Matching techniques can be reused in this context (e.g., see [Euzenat and Shvaiko 2007]). Of course, the alignment process cannot be fully automated and the user intervention will be necessary: Tools should support the interactive process.

Diff of Feature Models. Comparison capabilities can help an SPL practitioner to understand and locate differences between two feature models. The need to compare and to produce *diffs* of two feature models may happen when a feature model evolves over time or when two feature models represent the same concept but with a different viewpoint. In the context of the reverse engineering of FraSCAti, for instance, we needed to understand and locate the differences between the feature model designed by the software architect and the feature model produced by the extraction procedure.

Comparing two feature models is not a trivial task, since the number of legal configurations in the two models can be very important. Currently, two techniques are available in FAMILIAR: *i)* an operator that determines the kind of relationship (refactoring, generalization, specialization, arbitrary edit) between two feature models ; *ii)* a merge diff that computes the difference of two sets of configurations. These techniques are, from our experience with FraSCAti, not sufficient. First, they should be coupled with syntactic comparisons and we developed some FAMILIAR scripts to compare the variability operators attached to features with the same name. Second, we developed techniques to compare the differences between the set of implies/excludes constraints of both models based on the implication/exclusion graph.

The process, though, may not be trivial in a larger scale (e.g., for larger feature models than the one currently developed in the FraSCAti project) since the information gathered may not be pertinent or too large to be understood by an SPL practitioner (e.g., a software architect). It should be noted that comparing two feature models is inadequate until the two feature models are not properly reconciled. Hence the diff and the alignment are two related problems. Our research agenda is to investigate further in this research direction. Generic techniques may emerge, soundly validated in different case studies (including FraSCAti) and possibly integrated in the FAMILIAR language.

Scalability. As we have done with the merge and slice operators, the performance of the diff and the alignment techniques should be evaluated. As mentioned at the end of Chapter 9, we plan to investigate the use of SAT solvers in the context of the merge and slice operators. Depending on the results, we may reuse it for the diff and the alignment techniques. Increasing the scalability is interesting since we plan to apply the operators and the techniques on a larger scale: *i)* feature models used in our different applications are likely to increase in size and complexity ; *ii)* we observe that large feature models with thousands of features are now automatically extracted from large implemented software systems (see, e.g., [Lotufo et al. 2010, She et al. 2011]).

14.2 Increasing the Adoption of the Language

Both engineering and research efforts are needed to increase the adoption of FAMILIAR by SPL practitioners.

Enhancing FAMILIAR. We first need to increase the quality of the implementation and the stability of the tooling. FAMILIAR is currently connected to FeatureIDE graphical editors.

There are some interoperability issues since the formalism used in FeatureIDE does not support more than one feature group by parent feature. Moreover we are unaware of any tool with native support for the edition of multiple feature models. Such a tool should come in complement to our composition and decomposition operators. Provided graphical facilities are likely to ease the understanding of large feature models, their decompositions in smaller feature models, their configurations, their comparisons or their alignments. The need for an advanced graphical support comes from our case studies and we expect to evaluate this new support in these different contexts.

Evaluation of the Language. The language can be evaluated from several perspectives. Firstly, in terms of *learnability*, SPL practitioners have to learn an extra language, which takes time and effort. The learning curve of FAMILIAR is expected to be favorable since a restricted set of concepts is manipulated. Secondly, in terms of *expressiveness*, as FAMILIAR is a DSL, the language is specific to that domain and limits the possible scenarios that can be expressed. We need to assess when these limitations arise and whether they are problematic. Thirdly, in terms of *productivity*, the operators are directly provided. This may lower development costs and effort of FAMILIAR scripts. Fourthly, in terms of *usability*, tools and methods supporting FAMILIAR should be easy and convenient to use. Finally, in terms of *reusability*, modular mechanisms and parameterized scripts offer solutions that should be assessed. For all criteria, we hope the graphical support may help to learn the language, improve the productivity or usability. As a result, we plan to conduct experiments once the tooling support (including the graphical support) and the language itself are considered to be mature and stable enough. Initially, participants involved in the experiments will be undergraduate and graduate students, researchers and some domain experts not necessary familiar with feature models. We then hope to find industrial partners to conduct experiments on a larger scale.

Integration of the Language. In the development of video surveillance systems, FAMILIAR is used to model and specialize the variability of the software platform and the context. Once the modeling process is completed, the variability at runtime should be used to pilot self-adaptation mechanisms. In the run time adaptation architecture we present in [Moisan et al. 2011], a run time component manager (RTCM) captures low level events manifesting context changes (e.g., lighting changes); it forwards them to a FAMILIAR interpreter which returns a new feature configuration; the RTCM is then responsible for applying this configuration, that is to tune, add, remove, or replace components, and possibly to change the workflow itself. We plan to generate, from a FAMILIAR script, the code needed to reason about feature configurations. At the end, we expect to fully integrate our proposal in the end-to-end engineering of the video surveillance systems.

In the medical imaging domain, our experience with the tool-supported approach reveals that an advanced user interface should be developed to facilitate the modeling of variability and the configuration process. Indeed the *Wfamily* DSL (we recall that FAMILIAR is embedded in *Wfamily*) suffers from a lack of integration with the workflow editor of GWENDIA such that it is difficult for users to specify the weaving of feature models and the mapping with the catalog. The development of new visualization techniques, for example, to facilitate the connection understanding between different feature models of the workflow is an interesting perspective to consider. It should be coupled with the graphical support for multiple feature models currently developed.

14.3 DEMONSTRATING THE APPLICABILITY

The results of this research are potentially applicable to different scenarios involving feature models. We now discuss some of these works.

Feature Models. Thompson and Heimdahl argue that a product family is multi-dimensional if a hierarchical decomposition is not sufficient to capture its structure. In other terms, it needs to be specified from n perspectives: one family-hierarchy per view such as software or hardware [Thompson and Heimdahl 2003]. It is a form of separation of concerns. A set-theoretic foundation is proposed and can be expressed using feature models. In this case, the different views correspond to different feature models. [Höfner et al. 2011] describe an algebra of product families and revisit the approach given in [Thompson and Heimdahl 2003]. The proposed algebra enables algebraic manipulations of families of specifications (expressed as a feature model). They notably propose techniques to resolve conflicts among views, i.e., when features in one view description are linked to other features of another view description. We think our operators can be applied in this context: the aggregate operator can be used to link the different views while the slicing operator can be used to resolve conflicts.

More generally, existing works briefly discussed in Chapter 4 can be revisited. We already produced this effort for the work presented in [Metzger et al. 2007] (see Chapter 7), in [Hartmann et al. 2009] (see Chapter 6) and in [Thüm et al. 2009] (see Chapter 7). Though some of these works differ from ours (e.g., regarding the formalism used), we think the approach and/or the scenarios presented in [Kang et al. 1998], [Czarnecki et al. 2005b], [Tun et al. 2009], [Hartmann and Trew 2008], [Reiser and Weber 2007], [Lee and Kang 2010], [Zaid et al. 2011] can benefit from our techniques and support. We do not have evidence of that yet and further research is needed.

Multiple SPL. In the thesis, we developed the concept of multiple SPL, i.e., an SPL that manages a set of constituent SPLs. For example, in the case study related to the medical imaging domain, services can be seen as SPLs provided by different researchers or scientific teams. The entire workflow is then a multiple SPL in which different SPLs are composed. Applying the same approach to another domain is possible and many automatic parts of the process can be reused.

Let us consider that the other domain would manipulate connected components, possibly hierarchically composed so that one would face different software artifacts with a different composition techniques. This case is comparable to the variable components proposed in [van der Storm 2004]. The process would then remain similar to the one we propose. The variability would have to be extracted from components and expressed as feature models, then organized in a new catalog, reusing the FAMILIAR script. A new DSL for weaving concerns on relevant point-cuts of components would have to be designed. Interpretation for this DSL would need to be developed, so that either reusable FAMILIAR scripts can be called or FAMILIAR code can be generated in order to provide automate propagation and checking as in our workflow illustration. Consequently, the application of our approach would only necessitate to focus on the definition and weaving of feature model concerns, whereas the main difficulties of handling feature model composition would be automated. Nevertheless, this needs to be done in future work.

Feature-based Configurations. In the medical imaging domain, a lot of variability choices are inferred to speed up the configuration process. In some cases, the user wants to know why a certain feature was automatically selected or eliminated (i.e., he/she wants explanations). Though some techniques exist (see, e.g., [Janota 2010]), it should be integrated and adapted in our context since several feature models are potentially impacted. In the context of feature-based configuration, several works proposed techniques to separate the configuration process in different steps or stages [Czarnecki et al. 2005b, Hubaux et al. 2009, Mendonca and Cowan 2010]. In the context of staged configuration, White et al. propose a method to detect conflicts in a given configuration and propose changes in the configuration in terms of features to be selected or deselected to remedy the problem [White et al. 2008]. Again, such techniques should be considered in the context of our work where multiple feature models are used.

14.4 BEYOND PROPOSITIONAL FEATURE MODELS

In this thesis, we relied on propositional (also called *basic*) feature models. Though the strength of propositional feature models is their simplicity and intuitiveness, the expressive power is rather limited in comparison with some extensions that have been proposed (e.g., feature attributes, rich modeling constructs) [Czarnecki et al. 2006]. Therefore the choice of a richer formalism can be justified in some domains, but it should be carefully be studied given the expected performance and needed operations (e.g., increasing the expressive power means that reasoning techniques are more costly), i.e., there is a tradeoff between performance and expressive power.

The possible use of feature attributes has been identified in our different case studies.

In the context of video surveillance applications, the formalism of propositional feature models has been considered to be simple enough, yet expressive, to be used by video surveillance experts and it can be given a formal semantics with useful outcomes. The number of features is currently not a barrier to scalability, even at runtime. Numerical constraints may be interesting (e.g., see [Acher et al. 2009a]). In the context of the reverse engineering of architectural feature models, we plan to use feature attributes to model quality attributes of the FraSCAti architecture [Acher et al. 2011a]. Finally, in the medical imaging domain, there exists some services that support the same combination of features (e.g., same format, same algorithm method) so that even at the end of the configuration process, more than two services are still adequate. More information can be included to describe and select services, including quality attributes attached to features in feature models.

Though the use of feature attributes have been suggested early [Czarnecki et al. 2002] or is now part of some languages [Classen et al. 2010a, Bąk et al. 2011], [Benavides et al. 2010] recognize that "extended feature models where numerical attributes are included, miss further coverage" (benchmarks, comprehensive tool support, etc.).

Using extended feature models may lead to similar requirements to those identified for propositional feature models. Therefore the ideas developed in the context of this thesis (use of multiple extended feature models, development of automated reasoning techniques, support for separation of concerns) may be considered. For example, we can imagine a similar modeling process for modeling variability of video surveillance systems with the benefits of handling quantitative constraints. Nevertheless the semantics of extended feature models should be taken into account, for example, what does it mean to merge or

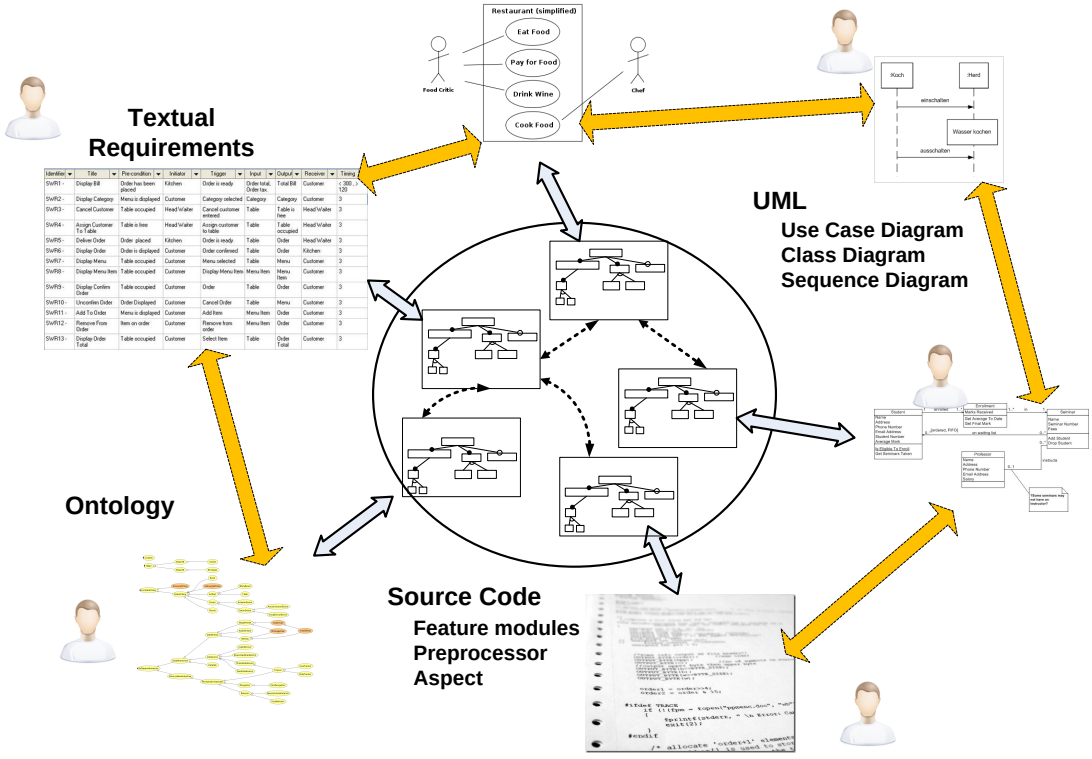


Figure 14.1: Relationships between Models and Feature Models

slice extended feature models?. It is also not clear whether a language should be designed or whether FAMILIAR should be extended and in which proportions. We consider that the domain of extended feature models is, in its current form, not mature enough. Further research is needed both at the theoretical and practical level.

14.5 On Feature Models and Other Models

We defend an approach that puts feature models at the center of SPL engineering (see Figure 14.1). The overall problem lies in the relationship between feature models and other models[1] of an SPL. A first issue is related to the extraction of feature models to abstract the variability of other models. A second issue concerns the realization of variability in other models. Finally, we plan to investigate how feature models can participate in a compositional approach involving a large number of inter-related models.

14.5.1 Extracting Feature Models From Other Models

In current practice, feature models are written in a top-down fashion through analysis of domain artifacts such as documentation and existing applications. It is a manual task

[1]Here, model is used in a broad sense and can be textual requirements, UML diagrams, source code, etc.

requiring expert analysis of multiple information sources. In particular, creating a feature model for an existing project is time-consuming and requires substantial effort from a modeler. Therefore, automated extraction techniques would significantly help an SPL practitioner by making the process less error-prone and more reproducible. The problem of reverse engineering the variability of existing models has definitely not received sufficient attention from the research community.

Several artifacts sources of informations and kinds of models are potentially candidate: system documentation [John 2006], textual requirements [Weston et al. 2009], ontology [Czarnecki et al. 2006, Fagereng Johansen et al. 2010], feature/plugin dependencies [She et al. 2011, Acher et al. 2011a]. In addition, artifacts that may not be commonly considered as feature models can arguably be considered as feature models *in disguise* [Czarnecki et al. 2006]. For example, products specification stored in tabular data (e.g., using the CSV (comma-separated values) format) are good candidate. Even though they have no explicit representation of variability or hierarchy, an expert can obtained feature models (we have performed preliminary experiments, see [Vanbeneden 2011]). The idea of extracting a feature model from models faces several challenges:

- automation of the extraction: the user effort, the time needed and the error-proneness should be reduced when building a feature model ;
- putting the user in the extraction process: though the procedure might be fully automated, the user should be able to play a part in the extraction (knowledge, scoping, etc.) ;
- quality of the extracted feature models: the extracted feature model should accurately represent the valid combination of features supported by artifacts of the SPL. A coherent hierarchy is also expected ;
- scalability of the extraction: the extraction should scale up for a large set of data. Theoretical or practical limits, if any, should be identified.

In this context, the new trend that consists in combining multiple information sources (including the expert knowledge, as we have done within the FraSCAti project) should be investigated further and in a larger context.

An orthogonal research question is to determine whether the concepts of feature modeling are sufficient in their current state so that the semantic gap with other, potentially richer, formalisms remains manageable. For example, we investigated the gap between propositional feature models and OWL-based ontologies [Fagereng Johansen et al. 2010]. Our original motivation was to reuse the domain knowledge provided by the ontologies developed in the context of Neurolog (a research project that aims at integrating data and knowledge in medical imaging). We primarily investigated how to reason about the consistency of a feature model in regards to an existing ontology.

This work, as well as other challenges identified, needs further research.

14.5.2 From Feature Models to Models

Although a feature model represents commonalities and variabilities in a very concise hierarchical form, features in a feature model are merely "names" [Czarnecki and Antkiewicz 2005, Classen et al. 2008]. By extension, the description of software products at the feature model level is restricted to a combination of syntactical symbols. In order to give semantics to features, we need to explicitly connect features to *base* models (such as behavioral, data specifications, UML models or even code representations). The overall challenge is to integrate feature models into a model-driven engineering (MDE) approach and to use them

to automatically generate (a set of) models corresponding to particular products from the SPL.

This is an important problem for several reasons. First and foremost, giving features semantics by linking them to base models is a crucial means to detect errors in the feature model and the base models. In this context, the main challenge is to assure safe composition [Czarnecki and Pietroszek 2006, Thaker et al. 2007, Classen et al. 2010b; 2011], that is, to make sure that every configuration allowed by the feature model leads to a correct model. Correctness can be defined in several ways, e.g., type safety in code [Thaker et al. 2007], structural validity in models [Czarnecki and Pietroszek 2006], satisfaction of specifications [Classen et al. 2010b; 2011] and so on. Whenever the safe composition property is guaranteed, we should be able to automatically derive a consistent and comprehensive model from all valid configurations of a feature model.

In this context, two research directions have been identified: the realization of variability at the model level and the need to develop compositional approaches for model-based SPL.

Variability in Base Models: Annotative vs. Compositional Approaches. In the literature, there are two very different approaches that have been proposed to implement variability in SPLs, both at the code or the model level: annotative and compositional approaches. In addition, several languages and tools have been developed to support the two approaches (e.g., CIDE and FeatureHouse [Kästner 2010], FeatureMapper [Heidenreich et al. 2008], CVL [Svendsen et al. 2010] or SmartAdapters [Perrouin et al. 2010]). Currently, the research effort in the SPL community has mainly focused on developing, comparing and even integrating annotative and compositional approaches at the code level (see, e.g., [Kästner 2010]). Though these techniques can certainly be reused or adapted at the model level, there is a need to better understand both approaches within the context of model-driven SPL engineering. Two main research questions can be formulated:

RQ1 *What are the current strengths and limitations of compositional and annotative approaches?* In particular, the approaches should be evaluated in terms of modularity, traceability, language integration, safe composition, granularity, feature interaction management, adoption, etc. Questions like "do annotative or compositional approaches offer adequate variability notation in base models?", "are the current automations and reasoning operations appropriate and sufficient?" will have to be covered.

RQ2 *What are the modeling techniques needed to improve both compositional and annotative approaches?* Based on the results obtained in **RQ1**, we need to investigate means to improve and extend existing approaches. Preliminary investigations show that (1) most existing work does not cover semantic issues and focus too much on syntactic properties (e.g., by only verifying syntactical correctness of a model product) ; (2) there is a lack of formalization so that semantic analyses can be hardly defined and automated ; (3) some challenges remain (e.g., Saval et al. identify three challenges in compositional approaches [Saval et al. 2009]).

In particular, the integration of feature models with other kinds of models involves *formalizing* the links between those notations, i.e., providing integrated formal syntax and semantics. Based on a formal semantics, syntactic and semantic analyzes of model-based SPLs can be properly defined and then automated. In addition the complexity of its associated decision procedures can be evaluated. In this context, the role of languages such

as Clafer [Bąk et al. 2011], TVL [Classen et al. 2010a, Michel et al. 2011] or CVL [Svendsen et al. 2010] should be carefully studied.

The Shift to Compositional Approaches. Some SPL engineering approaches now support defining and managing variability across different SPLs [Pohl et al. 2005, Buhne et al. 2005, Reiser and Weber 2007]. This "shift from variation to composition" and support for managing *multiple SPLs* (a.k.a. product populations [van Ommering and Bosch 2002] or software ecosystems [Bosch 2009]) are increasingly needed. Bosch proposes a paradigm shift from the traditional, integration-oriented approach of SPLs and suggest that SPLs are now multiple and *compositional* [Bosch 2010]. This is motivated by several reasons: intra-organizational and management issues (e.g., teams can be external, teams can be more independent and smaller), development of open ecosystems in which components are provided by third parties (including open source software) and in which customers can compose their own products, etc.

In this thesis, we have considered the problem of composing and managing multiple SPLs from a feature modeling perspective. This is a first step but we need to consider the problem from a more general perspective, including the different artifacts (e.g., source code, models) of SPLs. Compositional SPL approaches have a role to play. In particular, common mainstream modelling techniques advocate the use of different yet related models to represent the different stakeholders' needs - a practice known as Aspect-Oriented Modeling (AOM) or Multi-View Modeling (MVM). UML is an example of MVM where the different types of diagrams can represent distinct views of the same system.

Our vision is that a large number of models together with their feature models has to be used when describing a large, complex SPL. These models may be expressed in different formalisms, designed by different teams or stakeholders and used at different level of abstractions. Leveraging their use in compositional SPL approaches does pose a major challenge: how to extend them for handling consistency in and amongst the encapsulated multi-view variable parts while considering their composition. This both involves managing multiple feature models as we have proposed in thesis, investigating the relationship between feature models and models as described above, while managing multiple models as proposed by MDE, AOM or MVM approaches.

Appendices

.1 MERGE OPERATOR: WHY IS IT IMPORTANT TO NEGATE FEATURES?

Let us consider the merging of FM_1 and FM_2 (see Figure .2) in strict union mode. The resulting merged feature model, FM_r, is also depicted in Figure .2. The following relation truly holds:

$$[\![FM_1]\!] \cup [\![FM_2]\!] = [\![FM_r]\!]$$

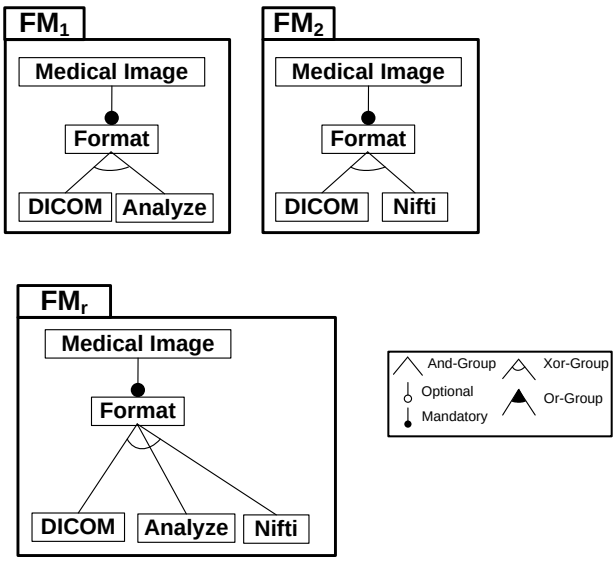


Figure .2: Merging of feature models: a simple example

Let ϕ_1 the propositional formula of FM_1.
Let ϕ_2 the propositional formula of FM_2.
Let ϕ_r the propositional formula of FM_r.

$$\begin{aligned}
\phi_1 \;=\; & MedicalImage \wedge MedicaImage \Leftrightarrow Format \\
& \wedge Analyze \Rightarrow Format \;\wedge DICOM \Rightarrow Format \\
& \wedge Format \Rightarrow (DICOM \;\vee Analyze) \wedge (\neg DICOM \vee \neg Analyze) \\
\phi_2 \;=\; & MedicalImage \wedge MedicaImage \Leftrightarrow Format \\
& \wedge Nifti \Rightarrow Format \;\wedge DICOM \Rightarrow Format \\
& \wedge Format \Rightarrow (DICOM \;\vee Nifti) \wedge (\neg DICOM \vee \neg Nifti)
\end{aligned}$$

An intuitive but incorrect encoding of ϕ_r is

$$\phi_r \;=\; \phi_1 \;\vee \phi_2$$

Indeed, $MedicalImage = true$, $Format = true$, $Nifti = true$, $Analyze = true$ and $DICOM = false$ is an assignment that satisfies ϕ_r. Unfortunately

$\{MedicalImage, Format, Nifti, Analyze\}$ corresponds to no valid configuration of neither FM_1 nor FM_2. Similarly, $\{MedicalImage, Format, DICOM, Analyze\}$ and $\{MedicalImage, Format, DICOM, Nifti\}$ correspond to no valid configuration of FM_1 or FM_2 (but are satisfying models of ϕ_r).

As we have previously defined, a correct encoding of ϕ_r is as follows:

$$\phi_r \ = \ (\phi_1 \ \wedge \neg Nifti) \vee (\phi_2 \ \wedge \neg Analyze)$$

Intuitively, we need to emulate the deselection of features that are in FM_1 (resp. FM_2) but not in FM_2 (resp. FM_1). In particular the assignment $MedicalImage = true$, $Format = true$, $Nifti = true$, $Analyze = true$ and $DICOM = false$ is no longer a satisfying model of ϕ_r (it is the same for the two other counter examples given above).

.2 Existential Quantification: An Example

Let

$$\phi = (f1 \vee \neg f3) \wedge (f1 \vee f2) \wedge (f2 \vee \neg f4 \vee f3) \wedge (\neg f1 \vee f4 \vee \neg f3)$$

The set of valid models of ϕ is enumerated below:

$$\{\{f1, f2\}, \{f1, f3, f4\}, \{f1, f3, f2, f4\}, \{f1, f2, f4\}, \{f2, f4\}, \{f1\}, \{f2\}\}$$

$$\begin{aligned}
\phi_{ex} &= \exists f1, f2 \ \phi \\
&= \exists f2 \ (\phi \wedge (f1 = 0)) \vee (\phi \wedge (f1 = 1)) \\
&= (\phi \wedge (f1 = 0) \wedge (f2 = 0)) \vee (\phi \wedge (f1 = 0) \wedge (f2 = 1)) \vee (\phi \wedge (f1 = 1) \wedge (f2 = 0)) \vee (\phi \wedge (f1 = 1) \wedge (f2 = 1)) \\
&= (\phi \wedge (f1 = 0) \wedge (f2 = 1)) \vee (\phi \wedge (f1 = 1) \wedge (f2 = 0)) \vee (\phi \wedge (f1 = 1) \wedge (f2 = 1)) \\
&= (\neg f3 \wedge (f4 \vee \neg f3)) \vee (\phi \wedge (f1 = 1) \wedge (f2 = 0)) \vee (\phi \wedge (f1 = 1) \wedge (f2 = 1)) \\
&= (\neg f3 \wedge (f4 \vee \neg f3)) \vee ((\neg f4 \vee f3) \wedge (f4 \vee \neg f3)) \vee (\phi \wedge (f1 = 1) \wedge (f2 = 1)) \\
&= (\neg f3 \wedge (f4 \vee \neg f3)) \vee ((\neg f4 \vee f3) \wedge (f4 \vee \neg f3)) \vee (f4 \vee \neg f3)
\end{aligned}$$

The set of valid models of ϕ_{ex} is enumerated below:

$$\{\{f4, f3\}, \{f4\}, \{\}\}$$

.3 Algorithms for the Assembly of Coherent Workflows

Algorithm 3 Querying the catalog

Require: the set of services, $FServices$, of the workflow that are mapped to a catalog
Ensure: services requirements match at least one service in the catalog and are updated.

```
for all FService_i ∈ FServices do
  Γ_i ← build the aggregated FM of FService_i
  FM_mapped ← the associated FM in the catalog
  if [[Γ_i]] ∩ [[FM_mapped]] = ∅ then
    print "Unable to find a corresponding service in the catalog"
  else
    FM_merged ← Γ_i ⊕∩ FM_mapped
    fms_decomposed ← decompose FM_merged
    for all FM_vc ∈ fms_decomposed do
      update FService_i with FM_vc
    end for
    Φ_new ← extract intra-constraints from fms_decomposed
    Φ_i ← Φ_new {update intra-constraints}
  end if
end for
```

Algorithm 4 Updating FMs after compatibility checking

Require: a set of dataport connections, $connection_{dps}$, where a dataport connection is represented as a set of dataports. $connection_{dps}$ is typically obtained through workflow analysis.

for all $connection \in connection_{dps}$ **do**
  $fmsToMerge \leftarrow \{\}$
  for all $dp \in connection$ **do**
    if dp has no FM attached **then**
      print "Warning: unable to find an FM in dataport"
    else
      $fmDP \leftarrow$ retrieve the FM attached to dp
      $fmsToMerge \leftarrow fmsToMerge \cup fmDP$
    end if
  end for
  $mergedFM \leftarrow fms_1 \oplus_\cap fms_2 \oplus_\cap \ldots fms_n$ where $fms_1, fms_2, \ldots, fms_n \in fmsToMerge$
  if $mergedFM$ not valid **then**
    print "Error: dataports of $connection$ are not compatible" {diff operations can be performed to provide fine-grained explanations}
  else
    $services_{modified} \leftarrow \{\}$ {services that have been impacted}
    for all $dp \in connection$ **do**
      $fm_{dp} \leftarrow$ retrieve the FM attached to dp
      if $mergedFM$ is a specialization of fm_{dp} **then**
        $service_{dp} \leftarrow$ retrieve the service of dp
        $services_{modified} \leftarrow services_{modified} \cup service_{dp}$ {propagation is needed}
      end if
      $fm_{dp} \leftarrow mergedFM$
    end for
  end if
end for
propagating choices on $services_{modified}$

Algorithm 5 Consistency checking and constraint propagation

Require: a set of services, $FServices$
Ensure: requirements variability are consistent and updated within each service
for all $FService_i \in FServices$ **do**
  $\Gamma_i \leftarrow$ build the aggregated FM of $FService_i$
  if $[\![\Gamma_i]\!] = \emptyset$ **then**
    print "The service $FService_i$ is not consistent"
  else
    propagate choices in Γ_i
    $fms_{decomposed} \leftarrow$ decompose Γ_i
    for all $FM_{vc} \in fms_{decomposed}$ **do**
      $FM_{corr} \leftarrow$ retrieve the original FM that corresponds to FM_{vc} in $FService_i$
      update $FService_i$ with FM_{vc}
      if FM_{vc} is attached to a dataport and FM_{vc} is a specialization of FM_{corr} **then**
        mark FM_{vc} {compatibility checking should be reiterated}
      end if
    end for
  end if
end for

Algorithm 6 Reiterating the reasoning process

Require: the set of services, $FServices$, of the workflow ; the set of dataport connections, $connection_{dps}$, of the workflow
for all $FService_i \in FServices$ **do**
  for all $vc \in VC_i$ **do**
    if vc is marked **then**
      print "Info: compatibility checking should be reiterated"
      $dp_{vc} \leftarrow$ retrieve dataport of vc
      $connection_{dp_{vc}} \leftarrow \{conn \in connection_{dps} \wedge dp_{vc} \in conn\}$
      unmark vc
      perform compatibility checking on $connection_{dp_{vc}}$
    end if
  end for
end for

Bibliography

Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., and Lahire, P. (2011a). Reverse Engineering Architectural Feature Models. In *5th European Conference on Software Architecture (ECSA'11) AR=22% , long paper*, LNCS, page 16, Essen (Germany). Springer. 4, 179, 201, 203

Acher, M., Collet, P., Fleurey, F., Lahire, P., Moisan, S., and Rigault, J.-P. (2009a). Modeling Context and Dynamic Adaptations with Feature Models. In *4th International Workshop Models@run.time at Models 2009 (MRT'09)*. 5, 165, 201

Acher, M., Collet, P., and Lahire, P. (2008a). Issues in Managing Variability of Medical Imaging Grid Services. In [Silvia et al. 2008]. 5, 35

Acher, M., Collet, P., Lahire, P., and France, R. (2009b). Composing Feature Models. In *2nd International Conference on Software Language Engineering (SLE'09) AR=19%*, LNCS, pages 62–81. Springer. 4, 51, 53, 62, 68, 69

Acher, M., Collet, P., Lahire, P., and France, R. (2010a). Comparing Approaches to Implement Feature Model Composition. In *6th European Conference on Modelling Foundations and Applications (ECMFA) AR=31%*, volume 6138 of *LNCS*, pages 3–19. Springer. 4, 51, 67, 188

Acher, M., Collet, P., Lahire, P., and France, R. (2010b). Managing Multiple Software Product Lines Using Merging Techniques. Technical report, University of Nice Sophia Antipolis, I3S CNRS, Sophia Antipolis, France. 53, 71

Acher, M., Collet, P., Lahire, P., and France, R. (2010c). Managing Variability in Worklow with Feature Model Composition Operators. In *9th International Conference on Software Composition (SC'10) AR=28%*, volume 6144 of *LNCS*, pages 17–33. Springer. 4, 141, 145, 148

Acher, M., Collet, P., Lahire, P., and France, R. (2011b). A Domain-Specific Language for Managing Feature Models. In *Symposium on Applied Computing (SAC) AR=33%*, , Taiwan. Programming Languages Track, ACM. 4, 107, 119

Acher, M., Collet, P., Lahire, P., and France, R. (2011c). Decomposing Feature Models: Language, Environment, and Applications. In *Automated Software Engineering (ASE'11), short paper: demonstration track*, , Lawrence, USA. IEEE/ACM. 4, 107

Acher, M., Collet, P., Lahire, P., and France, R. (2011d). Managing Feature Models with FAMILIAR: a Demonstration of the Language and its Tool Support. In *Fifth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'11) AR=55%*, VaMoS, Namur, Belgium. ACM. 5, 107

Acher, M., Collet, P., Lahire, P., and France, R. (2011e). Slicing Feature Models. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE'11), AR=22%, short paper*, , Lawrence, USA. IEEE/ACM. 4, 87

Acher, M., Collet, P., Lahire, P., and France, R. (2012). Separation of Concerns in Feature Modeling: Support and Applications. In *11th International Conference on Aspect-Oriented Software Development (AOSD'12)*, , Hasso-Plattner-Institut Potsdam, Germany. ACM. submitted, currently under review. 5, 87

Acher, M., Collet, P., Lahire, P., Gaignard, A., France, R., and Montagnat, J. (2011f). Composing Multiple Variability Artifacts to Assemble Coherent Workflows. *Software Quality Journal (Special issue on Quality Engineering for SPLs)*. 4, 141

Acher, M., Collet, P., Lahire, P., Moisan, S., and Rigault, J.-P. (2011g). Modeling Variability from Requirements to Runtime. In *16th International Conference on Engineering of Complex Computer Systems (ICECCS'11) AR=31%*, , Las Vegas. IEEE. 4, 165

Acher, M., Collet, P., Lahire, P., and Montagnat, J. (2008b). Imaging Services on the Grid as a Product Line: Requirements and Architecture. In *Service-Oriented Architectures and Software Product Lines - Putting Both Together (SOAPL'08)*. (associated workshop issue of SPLC 2008), IEEE Computer Society. 5, 35

Acher, M., Lahire, P., Moisan, S., and Rigault, J.-P. (2009c). Tackling High Variability in Video Surveillance Systems through a Model Transformation Approach. In *MiSE '09: Proceedings of the 2009 international workshop on Modeling in software engineering at ICSE 2009 AR=44%*. IEEE Computer Society. 5, 165

Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., and Lucena, C. (2006). Refactoring product lines. In *Proc. of GPCE'2006*, pages 201–210. ACM. 31, 43, 68, 69, 77

Apel, S., Janda, F., Trujillo, S., and Kästner, C. (2009). Model superimposition in software product lines. In Paige, R. F., editor, *ICMT*, volume 5563 of *Lecture Notes in Computer Science*, pages 4–19. Springer. 19

Apel, S. and Kästner, C. (2009). An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84. 13, 29

Apel, S., Lengauer, C., Moller, B., and Kastner, C. (2008). An algebra for features and feature composition. In *12th International Conference on Algebraic Methodology and Software Technology (AMAST '08)*, volume 5140 of *LNCS*, pages 36–50. Springer. 77

Asikainen, T., Männistö, T., and Soininen, T. (2004). Using a configurator for modelling and configuring software product lines based on feature models. In *Proceedings of the Workshop on Software Variability Management for Product Derivation, Software Product Line Conference (SPLC3)*. 18

Asikainen, T., Mannisto, T., and Soininen, T. (2006). A unified conceptual foundation for feature modelling. In *Proc. of SPLC'2006*, pages 31–40. IEEE. 18

Asikainen, T., Männistö, T., and Soininen, T. (2007). Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21(1):23–40. 18

Aspect-Oriented Modeling Workshop Series (2001/2011). http://www.aspect-modeling.org/. 16

Atkinson, C., Bayer, J., and Muthig, D. (2000). Component-based product line development: the kobra approach. In Donohoe, P., editor, *SPLC*, pages 289–310. Kluwer. 18

Avanzi, A., Brémond, F., Tornieri, C., and Thonnat, M. (2005). Design and assessment of an intelligent activity monitoring platform. *EURASIP Journal on Applied Signal Processing*, 14(8):2359–2374. 167

Bachmann, F. and Bass, L. (2001). Managing variability in software architectures. *SIGSOFT Softw. Eng. Notes*, 26:126–132. 17, 29

Bass, L., Clements, P., , and Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley. 9, 10, 11

Batory, D., Liu, J., and Sarvela, J. N. (2003). Refinements and multi-dimensional separation of concerns. In *ESEC'03: Proceedings of the 9th European software engineering conference*, pages 48–57, Helsinki, Finland. ACM. 29

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371. 19, 32

Batory, D. S. (2005). Feature models, grammars, and propositional formulas. In *9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *LNCS*, pages 7–20. 25, 27, 28, 29, 32, 66, 128

Benavides, D., Metzger, A., and Eisenecker, U. W., editors (2009). *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, 2009. Proceedings*, volume 29 of *ICB Research Report*. Universität Duisburg-Essen. 224, 229

Benavides, D., Ruiz-Cortés, A., and Trinidad, P. (2005). Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503. 28

Benavides, D., Segura, S., and Ruiz-Cortes, A. (2010). Automated Analysis of Feature Models 20 years Later: a Literature Review. *Information Systems*. 25, 26, 27, 28, 43, 93, 109, 130, 201

Benavides, D., Segura, S., Trinidad, P., and Ruiz–Cortés, A. (2006). A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*. 131

Berger, T., She, S., Lotufo, R., Wasowski, A., and Czarnecki, K. (2010). Variability modeling in the real: a perspective from the operating systems domain. In Pecheur, C., Andrews, J., and Nitto, E. D., editors, *ASE*, pages 73–82. ACM. 42, 88

Beuche, D. (2008). Modeling and building software product lines with Pure::Variants. In *2008 12th International Software Product Line Conference*, pages 358–358, Limerick, Ireland. 32

Beuche, D., Papajewski, H., and Schrader-Preikschat, W. (2004). Variability management with feature models. *Science of Computer Programming*, 53(3):333–352. 11, 18

Bézivin, J. (2005). On the unification power of models. *Software and System Modeling*, 4(2):171–188. 15

BigLever – Gears (2006). http://www.biglever.com/solution/product.html. 32

Bąk, K., Czarnecki, K., and Wąsowski, A. (2011). Feature and meta-models in clafer: Mixed, specialized, and coupled. In Malloy, B., Staab, S., and van den Brand, M., editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 102–122. Springer Berlin / Heidelberg. 18, 32, 201, 205

Böckle, G., Clements, P. C., McGregor, J. D., Muthig, D., and Schmid, K. (2004). Calculating roi for software product lines. *IEEE Software*, 21:23–31. 12

Bollig, B. and Wegener, I. (1996). Improving the variable ordering of obdds is np-complete. *IEEE Trans. Comput.*, 45(9):993–1002. 130

Bosch, J. (2000). *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. 29

Bosch, J. (2009). From software product lines to software ecosystems. In *Proc. of SPLC'2009*, volume 446 of *ICPS*, pages 111–119. ACM. 43, 71, 205

Bosch, J. (2010). Toward compositional software product lines. *IEEE Software*, 27:29–34. 205

Bosch, J. and Lee, J., editors (2010). *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, volume 6287 of *Lecture Notes in Computer Science*. Springer. 226, 230

Botterweck, G., Pleuss, A., Dhungana, D., Polzer, A., and Kowalewski, S. (2010). Evofm: feature-driven planning of product-line evolution. In *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering*, PLEASE '10, pages 24–31, New York, NY, USA. ACM. 43

Boucher, Q., Classen, A., Faber, P., and Heymans, P. (2010). Introducing TVL, a text-based feature modelling language. In *VaMoS'10*, pages 159–162. 32, 128, 129

Brace, K. S., Rudell, R. L., and Bryant, R. E. (1990). Efficient implementation of a bdd package. In *DAC'90*, pages 40–45. ACM. 130, 133

Brooks, Jr., F. P. (1987). No silver bullet essence and accidents of software engineering. *Computer*, 20:10–19. 14

Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., and Sabetzadeh, M. (2006). A manifesto for model merging. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12, New York, NY, USA. ACM. 61

Buhne, S., Lauenroth, K., and Pohl, K. (2005). Modelling requirements variability across product lines. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 41–52, Washington, DC, USA. IEEE Computer Society. 71, 205

Chen, K., Zhang, W., Zhao, H., and Mei, H. (2005). An approach to constructing feature models based on requirements clustering. In *RE*, pages 31–40. IEEE Computer Society. 29

Chen, L., Babar, M. A., and Ali, N. (2009). Variability management in software product lines: a systematic review. In Muthig, D. and McGregor, J. D., editors, *SPLC*, volume 446 of *ACM International Conference Proceeding Series*, pages 81–90. ACM. 18

Classen, A., Boucher, Q., and Heymans, P. (2010a). A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming, Special Issue on Software Evolution*. 32, 109, 201, 205

Classen, A., Heymans, P., and Schobbens, P.-Y. (2008). What's in a feature : A requirements engineering perspective. *Fundamental Approaches to Software Engineering*, pages 16–30. 29, 203

Classen, A., Heymans, P., Schobbens, P.-Y., and Legay, A. (2011). Symbolic model checking of software product lines. In Taylor, R. N., Gall, H., and Medvidovic, N., editors, *ICSE'11*, pages 321–330. ACM. 19, 130, 204

Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., and Raskin, J.-F. (2010b). Model checking lots of systems: efficient verification of temporal properties in software product lines. In Kramer, J., Bishop, J., Devanbu, P. T., and Uchitel, S., editors, *ICSE (1)*, pages 335–344. ACM. 19, 204

Clements, P. and Northrop, L. M. (2001). *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional. 9, 10, 12

Clements, P. C. and Krueger, C. (2002). Point – counterpoint: Being proactive pays off – eliminating the adoption. *IEEE Software*, 19(4):28–31. 11

CoreGRID (2011). `http://www.coregrid.net/`, european research network on foundations, software infrastructures and applications for large scale distributed, grid and peer-to-peer technologies. 144

Czarnecki, K. and Antkiewicz, M. (2005). Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, volume 3676 of *LNCS*, pages 422–437. 18, 19, 30, 128, 203

Czarnecki, K., Bednasch, T., Unger, P., and Eisenecker, U. (2002). *Generative Programming for Embedded Software: An Industrial Experience Report*, pages 156–172. LNCS. 28, 201

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley. 12, 14, 16, 21, 29, 32

Czarnecki, K. and Eisenecker, U. W. (1999). Components and generative programming. In Nierstrasz, O. and Lemoine, M., editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 2–19. Springer. 23

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645. 15

Czarnecki, K., Helsen, S., and Eisenecker, U. (2004). Staged configuration using feature models. In *Software Product Lines*, volume 3154/2004 of *Lecture Notes in Computer Science*, pages 266–283. Springer Berlin / Heidelberg. 32

Czarnecki, K., Helsen, S., and Eisenecker, U. (2005a). Formalizing Cardinality-based Feature Models and their Specialization. In *Software Process Improvement and Practice*, pages 7–29. 110

Czarnecki, K., Helsen, S., and Eisenecker, U. (2005b). Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169. 18, 28, 31, 43, 44, 155, 200, 201

Czarnecki, K., Kim, C. H. P., and Kalleberg, K. T. (2006). Feature models are views on ontologies. In *SPLC '06: Proceedings of the 10th International on Software Product Line Conference*, pages 41–51, Washington, DC, USA. IEEE Computer Society. 23, 28, 201, 203

Czarnecki, K. and Pietroszek, K. (2006). Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE'06*, pages 211–220. ACM. 19, 204

Czarnecki, K. and Wąsowski, A. (2007). Feature diagrams and logics: There and back again. In *SPLC'07*, pages 23–34. 25, 26, 28, 66, 67, 93, 128, 130, 133, 136

Darwiche, A. and Marquis, P. (2002). A knowledge compilation map. *J. Artif. Intell. Res. (JAIR)*, 17:229–264. 131

Deelstra, S., Sinnema, M., and Bosch, J. (2004). *Experiences in Software Product Families: Problems and Issues During Product Derivation*, pages 165–182. LNCS. xii, 12, 13

Deelstra, S., Sinnema, M., and Bosch, J. (2005). Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194. 12, 84

Deursen, A. v. and Klint, P. (2002). Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17. 32

Dhungana, D., Grünbacher, P., Rabiser, R., and Neumayer, T. (2010). Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software*, 83(7):1108–1122. 31, 44

Edmonds, J. (1967). Optimum branchings. *Journal of Research of the National Bureau of Standards*, 71B:233–240. 65

Erwig, M. (2010). A language for software variation research. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 3–12, New York, NY, USA. ACM. 17

Erwig, M. and Walkingshaw, E. (2011). The choice calculus: A representation for software variation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*. to appear. 17

Euzenat, J. and Shvaiko, P. (2007). *Ontology matching*. Springer-Verlag, Heidelberg (DE). 198

Fagereng Johansen, M., Fleurey, F., Acher, M., Collet, P., and Lahire, P. (2010). Exploring the Synergies Between Feature Models and Ontologies. In *International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010)*, volume 2 of *SPLC '10*, pages 163–171, Jeju Island, South Korea. Lancester University. 5, 77, 197, 203

FaMa (2008). http://www.isa.us.es/fama/. 32

FeatureMapper (2008). http://featuremapper.org. 18

Fleurey, F., Baudry, B., France, R. B., and Ghosh, S. (2007). A generic approach for automatic model composition. In Giese, H., editor, *MoDELS Workshops*, pages 7–15. Springer. 68

Foster, I., Kesselman, C., Nick, J., and Tuecke, S. (2002). The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG, GGF. 36, 144

Fowler, M. (2009). A pedagogical framework for domain-specific languages. *IEEE Software*, 26:13–14. 127

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional. 109, 127

FraSCAti (2011). https://wiki.ow2.org/frascati/Wiki.jsp?page=Main. 179

Génova, G., Valiente, M. C., and Marrero, M. (2009). On the difference between analysis and design, and why it is relevant for the interpretation of models in model driven engineering. *Journal of Object Technology*, 8(1):107–127. 14

Gheyi, R., Massoni, T., and Borba, P. (2006). A theory for feature models in alloy. In *Proceedings of First Alloy Workshop*, pages 71–80. 68

Gil, Y., Deelman, E., Ellisman, M. H., Fahringer, T., Fox, G., Gannon, D., Goble, C. A., Livny, M., Moreau, L., and Myers, J. (2007). Examining the challenges of scientific workflows. *IEEE Computer*, 40(12):24–32. 35

Glass, R. L. (2001). Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 18:112,110–111. 10

Glatard, T., Montagnat, J., Lingrand, D., and Pennec, X. (2008). Flexible and efficient workflow deployment of data-intensive applications on grids with moteur. *Int. J. High Perform. Comput. Appl.*, 22:347–360. 144

Gomaa, H. (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA. 18

Griss, M. L., Favaro, J., and d' Alessandro, M. (1998). Integrating feature modeling with the rseb. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 76, Washington, DC, USA. IEEE. 18, 32

Grünbacher, P., Rabiser, R., Dhungana, D., and Lehofer, M. (2009). Structuring the product line modeling space: Strategies and examples. In [Benavides et al. 2009], pages 77–82. 44

Halmans, G. and Pohl, K. (2003). Communicating the variability of a software-product family to customers. *Software and System Modeling*, 2(1):15–36. 17, 30

Harel, D. and Rumpe, B. (2004). Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72. 16

Hartmann, H. and Trew, T. (2008). Using feature diagrams with context variability to model multiple product lines for software supply chains. In *12th International Software Product Line Conference (SPLC'08)*, pages 12–21, Washington, DC, USA. IEEE Computer Society. 31, 43, 44, 71, 166, 200

Hartmann, H., Trew, T., and Matsinger, A. (2009). Supplier independent feature modelling. In *SPLC'09*, pages 191–200. IEEE. xiii, 31, 43, 44, 71, 75, 76, 83, 84, 100, 200

Haugen, O., Moller-Pedersen, B., Oldevik, J., Olsen, G., and Svendsen, A. (2008). Adding standardized variability to domain specific languages. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 139 –148. 18

Heidenreich, F., Kopcsek, J., and Wende, C. (2008). FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 943–944, New York, NY, USA. ACM. 18, 19, 204

Heidenreich, F., Sanchez, P., Santos, J., Zschaler, S., Alferez, M., Araujo, J., Fuentes, L., amd Ana Moreira, U. K., and Rashid, A. (2010). Relating feature models to other models of a software product line: A comparative study of featuremapper and vml*. *Transactions on Aspect-Oriented Software Development VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling*, 6210:69–114. 18, 19, 30, 128

Heymans, P., Czarnecki, K., and Eisenecker, U. W., editors (2011). *Fifth International Workshop on Variability Modelling of Software-Intensive Systems, Namur, Belgium, January 27-29, 2011. Proceedings*, ACM International Conference Proceedings Series. ACM. 227, 232

Heymans, P., Schobbens, P.-Y., Trigaux, J.-C., Bontemps, Y., Matulevicius, R., and Classen, A. (2008). Evaluating formal properties of feature diagram languages. *Software, IET*, 2(3):281–302. 67

Höfner, P., Khedri, R., and Möller, B. (2011). An algebra of product families. *Software and Systems Modeling*, 10:161–182. 200

http://www.satcompetition.org/ (2011). http://www.satcompetition.org/. 130

Hubaux, A., Classen, A., and Heymans, P. (2009). Formal modelling of feature configuration workflows. In *SPLC'09*, pages 221–230. IEEE. 31, 43, 44, 101, 201

Hubaux, A., Heymans, P., and Schobbens, P.-Y. (2010a). Supporting multiple perspectives in feature-based configuration: Foundations. Technical Report P-CS-TR PPFD-000001, University of Namur PReCISE Research Centre, Faculty of Computer Science Namur Belgium. 44, 101

Hubaux, A., Heymans, P., Schobbens, P.-Y., and Deridder, D. (2010b). Towards multi-view feature-based configuration. In Wieringa, R. and Persson, A., editors, *REFSQ*, volume 6182 of *Lecture Notes in Computer Science*, pages 106–112. Springer. 43, 44, 161

Jacobson, I., Griss, M., and Jonsson, P. (1997). *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. 10, 17

Janota, M. (2010). *SAT Solving in Interactive Configuration*. PhD thesis, Department of Computer Science at University College Dublin. 28, 136, 161, 162, 201

Janota, M. and Kiniry, J. (2007). Reasoning about feature models in higher-order logic. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 13–22. IEEE. 28

Janota, M., Kuzina, V., and Wąsowski, A. (2008). Model construction with external constraints: An interactive journey from semantics to syntax. In *MoDELS'08*, volume 5301 of *LNCS*, pages 431–445. Springer. 133

JavaBDD (2007). http://javabdd.sourceforge.net/index.html. 130

Jeanneret, C., France, R., and Baudry, B. (2008). A reference process for model composition. In *AOM '08: Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling*, pages 1–6, New York, NY, USA. ACM. 54

John, I. (2006). Capturing product line information from legacy user documentation. In *Software Product Lines*, pages 127–159. Springer. 203

John, I. and Eisenbarth, M. (2009). A decade of scoping: a survey. In *Proc. of SPLC'2009*, volume 446 of *ICPS*, pages 31–40. ACM. 17

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, S. (1990). Feature-Oriented Domain Analysis (FODA). Technical Report CMU/SEI-90-TR-21, SEI. 2, 21, 22, 23, 28, 29, 32, 51

Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). FORM: a feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168. 18, 29, 30, 32, 43, 44, 166, 200

Kästner, C. (2010). *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg. Logos Verlag Berlin, isbn 978-3-8325-2527-9. 19, 42, 204

Kästner, C., Thüm, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., and Apel, S. (2009a). Featureide: A tool framework for feature-oriented software development. In *31st International Conference on Software Engineering (ICSE'09), Tool demonstration*, pages 611–614. 19, 32, 108, 128

Kästner, C., Thüm, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., and Apel, S. (2009b). Featureide: A tool framework for feature-oriented software development. In *31st International Conference on Software Engineering (ICSE'09), Tool demonstration*, pages 611–614. 32

Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley . 16

Kramer, J. (2007). Is abstraction the key to computing? *Commun. ACM*, 50(4):36–42. 15

Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv.*, 24:131–183. 10

Krueger, C. W. (2001). Easing the transition to software mass customization. In *Software Product-Family Engineering, 4th International Workshop (PFE 2001)*, pages 282–293. 11

Krueger, C. W. (2006). New methods in software product line development. In *10th Int. Software Product Line Conf.*, pages 95–102, Los Alamitos, CA, USA. IEEE Computer Society. 11

Lauenroth, K., Pohl, K., and Toehning, S. (2009). Model checking of domain artifacts in product line engineering. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 269–280, Washington, DC, USA. IEEE Computer Society. 19

Lee, K. and Kang, K. C. (2010). Usage context as key driver for feature selection. In [Bosch and Lee 2010], pages 32–46. 31, 44, 200

Lopez-Herrejon, R. E. and Egyed, A. (2010). On the need of safe software product line architectures. In *ECSA'10*, volume 6285 of *LNCS*, pages 493–496. Springer. 181

Lorenzi, M., Ayache, N., Frisoni, G., and Pennec, X. (2010). 4d registration of serial brain mr's images: a robust measure of changes applied to alzheimer's disease. In *Miccai Workshop on Spatio-Temporal Image Analysis for Longitudinal and Time-Series Image Data. First prize award for the "Best Oral Presentation"*, Beijing, China. 36, 159

Lotufo, R., She, S., Berger, T., Czarnecki, K., and Wasowski, A. (2010). Evolution of the linux kernel variability model. In [Bosch and Lee 2010], pages 136–150. 88, 198

Maccari, A. and Heie, A. (2005). Managing infinite variability in mobile terminal software: Research articles. *Softw. Pract. Exper.*, 35(6):513–537. 10

Maheshwari, K., Glatard, T., Schaerer, J., Delhay, B., Camarasu, S., Clarysse, P., and Montagnat, J. (2009). Towards Production-level Cardiac Image Analysis with Grids. In *HealthGrid'09*, , Berlin. 159

Mannion, M. (2002). Using first-order logic for product line model validation. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 176–187, London, UK. Springer-Verlag. 27

Marcilio Mendonca, Andrzej Wąsowski, K. C. (2009). Sat-based analysis of feature models is easy. In *SPLC'09*, pages 231–241. IEEE. 28, 134, 135, 136

McIlroy, M. D. (1968). Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*. 1, 10

McPhillips, T., Bowers, S., Zinn, D., and Ludäscher, B. (2009). Scientific workflow design for mere mortals. *Future Gener. Comput. Syst.*, 25:541–551. 35

Mendonça, M. (2009). Efficient reasoning techniques for large scale feature models. Master's thesis, University of Waterloo, Waterloo. 161, 162

Mendonca, M., Branco, M., and Cowan, D. (2009a). S.p.l.o.t.: software product lines online tools. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762, New York, NY, USA. ACM. 32, 108, 128

Mendonca, M., Branco, M., and Cowan, D. (2009b). S.p.l.o.t.: software product lines online tools. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762, New York, NY, USA. ACM. 32

Mendonca, M. and Cowan, D. (2010). Decision-making coordination and efficient reasoning techniques for feature-based configuration. *Science of Computer Programming*, 75(5):311 – 332. Coordination Models, Languages and Applications (SAC'08). 28, 101, 161, 201

Mendonca, M., Wąsowski, A., Czarnecki, K., and Cowan, D. (2008). Efficient compilation techniques for large scale feature models. In *GPCE '08*, pages 13–22. ACM. 28, 130, 133

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344. 109, 110

Metzger, A., Pohl, K., Heymans, P., Schobbens, P.-Y., and Saval, G. (2007). Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *15th IEEE International Conference on Requirements Engineering (RE '07)*, pages 243–253. xiv, 17, 18, 30, 44, 72, 96, 97, 99, 101, 166, 182, 190, 200

Mezini, M. and Ostermann, K. (2004). Variability management with feature-oriented programming and aspects. *SIGSOFT Softw. Eng. Notes*, 29(6):127–136. 19

Michel, R., Classen, A., Hubaux, A., and Boucher, Q. (2011). A formal semantics for feature cardinalities in feature diagrams. In [Heymans et al. 2011], pages 82–89. 110, 205

Minato, S.-i. (1996). *Binary decision diagrams and applications for VLSI CAD*. Kluwer Academic Publishers, Norwell, MA, USA. 130

Moisan, S., Rigault, J.-P., Acher, M., Collet, P., and Lahire, P. (2011). Run Time Adaptation of Video-Surveillance Systems: A Software Modeling Approach. In *8th International Conference on Computer Vision Systems (ICVS'2011), AR=37%*, LNCS. Springer. 4, 165, 178, 199

Montagnat, J., Isnard, B., Glatard, T., Maheshwari, K., and Blay-Fornarino, M. (2009). A data-driven workflow language for grids based on array programming principles. In *Workshop on Workflows in Support of Large-Scale Science(WORKS'09)*, , pages 1–10, Portland, USA. ACM. 144

Morin, B., Barais, O., Jézéquel, J.-M., Fleurey, F., and Solberg, A. (2009a). Models at runtime to support dynamic adaptation. *IEEE Computer*, pages 46–53. 166

Morin, B., Barais, O., Nain, G., and Jézéquel, J.-M. (2009b). Taming dynamically adaptive systems using models and aspects. In *31st International Conference on Software Engineering (ICSE'09)*, pages 122–132. 18, 166

Morin, B., Vanwormhoudt, G., Lahire, P., Gaignard, A., Barais, O., and Jézéquel, J.-M. (2008). Managing variability complexity in aspect-oriented modeling. *Model Driven Engineering Languages and Systems*, pages 797–812. 19

Morin, Brice and Barais, Olivier and Jézéquel, Jean-Marc and Ramos, Rodrigo (2007). Towards a generic aspect-oriented modeling framework. In *Models and Aspects workshop, at ECOOP 2007*. 19

Muller, P.-A., Fondement, F., and Baudry, B. (2009). Modeling modeling. In Schürr, A. and Selic, B., editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 2–16. Springer. 15

Northrop, L., Feiler, P., Gabriel, R. P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., and et al. (2006). Ultra-large-scale systems - the software challenge of the future. *Technical report Software Engineering Institute Carnegie Mellon University ISBN*, 0. 1

Northrop, L. M. (2002). Sei's software product line tenets. *IEEE Softw.*, 19:32–40. 11, 12

Parnas, D. L. (1976). On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2(1):1–9. 10

Parra, C. A., Blanc, X., Cleve, A., and Duchien, L. (2011). Unifying design and runtime software adaptation using aspect models. *Sci. Comput. Program.*, 76(12):1247–1260. 179

Parra, C. A., Cleve, A., Blanc, X., and Duchien, L. (2010). Feature-based composition of software architectures. In *ECSA'10*, volume 6285 of *LNCS*, pages 230–245. Springer. 179

Pernod, E., Souplet, J.-C., Rojas Balderrama, J., Lingrand, D., and Pennec, X. (2008). Multiple Sclerosis Brain MRI Segmentation Workflow Deployment On The EGEE Grid. In [Silvia et al. 2008], pages 55–64. 159, 160

Perrouin, G., Klein, J., Guelfi, N., and Jézéquel, J.-M. (2008). Reconciling automation and flexibility in product derivation. In *SPLC'08*, pages 339–348. IEEE. 19

Perrouin, G., Vanwormhoudt, G., Lahire, P., Morin, B., Barais, O., and Jézéquel, J.-M. (2010). Weaving Variability into Domain Metamodels. *Software and Systems Modeling Special issue*, page 22. 18, 204

Pine, B. J. (1999). *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press. 9

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. 9, 10, 12, 17, 18, 30, 44, 71, 96, 205

pure::variants (2006). http://www.pure-systems.com/pure_variants.49.0.html. 32, 108

Rabiser, R., Grunbacher, P., and Dhungana, D. (2007). Supporting product derivation by adapting and augmenting variability models. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 141–150. IEEE. 18

Reddy, Y. R., Ghosh, S., France, R. B., Straw, G., Bieman, J. M., McEachen, N., Song, E., and Georg, G. (2006). Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development*, 3880:75–105. 68

Refstrup, J. G. (2009). Adapting to change: Architecture, processes and tools: A closer look at hp's experience in evolving the owen software product line. In *Proceedings of International Software Product Line Conference (SPLC'09)*. Keynote presentation. 42

Reiser, M.-O. and Weber, M. (2006). Managing highly complex product families with multi-level feature trees. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, pages 146–155, Washington, DC, USA. IEEE. 44

Reiser, M.-O. and Weber, M. (2007). Multi-level feature trees: A pragmatic approach to managing highly complex product families. *Requir. Eng.*, 12(2):57–75. 31, 43, 44, 71, 200, 205

Riebisch, M. (2003). Towards a more precise definition of feature models. In Riebisch, M., Coplien, J. O., and Streitferdt, D., editors, *Modelling Variability for Object-Oriented Product Lines*, pages 64–76. BookOnDemand Publ. Co, Norderstedt. 18

Riebisch, M., Böllert, K., Streitferdt, D., and Philippow, I. (2002). Extending feature diagrams with uml multiplicities. In *6th World Conference on Integrated Design & Process Technology (IDPT2002)*. 23

Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education. 16

Sanchez, P., Loughran, N., Fuentes, L., and Garcia, A. (2008). Engineering languages for specifying Product-Derivation processes in software product lines. In *1st International Conference on Software Language Engineering (SLE'08)*, LNCS, pages 188–207. Springer. 30

Saval, G., Puissant, J. P., Heymans, P., and Mens, T. (2009). Some challenges of feature-based merging of class diagrams. In [Benavides et al. 2009], pages 127–136. 19, 204

SCA standard (2007). http://www.osoa.org/. 180

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer*, 39:25–31. 16

Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., and Bontemps, Y. (2007). Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479. 18, 23, 28, 53, 67, 71, 77, 101, 161

Schwanninger, C., Groher, I., Elsner, C., and Lehofer, M. (2009). Variability modelling throughout the product line lifecycle. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 685–689, Berlin, Heidelberg. Springer-Verlag. 31

Segura, S., Benavides, D., Ruiz-Cortes, A., and Trinidad, P. (2008). Automated merging of feature models using graph transformations. *Post-proceedings of the Second Summer School on GTTSE*, 5235:489–505. 31, 43, 62, 68, 69, 77

SEI (2011). A framework for software product line practice, version 5.0 http://www.sei.cmu.edu/productlines/frame_report/index.html. 12

Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., and Stefani, J.-B. (2009). Reconfigurable SCA Applications with the FraSCAti Platform. In *SCC'09*, pages 268–275. IEEE. 180

She, S., Lotufo, R., Berger, T., Wąsowski, A., and Czarnecki, K. (2011). Reverse engineering feature models. In *ICSE'11*. to appear. 23, 26, 42, 67, 136, 181, 189, 198, 203

Silvia, O., Diane, L., and Johan, M., editors (2008). *MICCAI-Grid Workshop*, New York, NY, USA. 217, 228

Sinnema, M. and Deelstra, S. (2008). Industrial validation of covamof. *Journal of Systems and Software*, 81(4):584–600. 18

Sinnema, M., Deelstra, S., and Hoekstra, P. (2006). The covamof derivation process. In Morisio, M., editor, *ICSR*, volume 4039 of *Lecture Notes in Computer Science*, pages 101–114. Springer. 18

Steger, M., Tischer, C., Boss, B., Müller, A., Pertler, O., Stolz, W., and Ferber, S. (2004). Introducing pla at bosch gasoline systems: Experiences and practices. In Nord, R. L., editor, *SPLC*, volume 3154 of *Lecture Notes in Computer Science*, pages 34–50. Springer. 42

Svahnberg, M., van Gurp, J., and Bosch, J. (2005). A taxonomy of variability realization techniques: Research articles. *Software Practice and Experience*, 35(8):705–754. 17, 18, 30, 168, 180

Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Haugen, Ø., Møller-Pedersen, B., and Olsen, G. K. (2010). Developing a software product line for train control: A case study of cvl. In [Bosch and Lee 2010], pages 106–120. 19, 204, 205

Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley. 10

Taentzer, G. (2004). AGG: a graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations with Industrial Relevance*, pages 446–453. Springer. 68

Tarjan, R. E. (1977). Finding optimum branchings. *Networks*, 7:25 – 35. 65

Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr., S. M. (1999). N degrees of separation: multi-dimensional separation of concerns. In *ICSE'99*, pages 107–119. ACM. 40

Temal, L., Dojat, M., Kassel, G., and Gibaud, B. (2008). Towards an ontology for sharing medical images and regions of interest in neuroimaging. *Journal of Biomedical Informatics*. To appear. 77, 197

Thaker, S., Batory, D., Kitchin, D., and Cook, W. (2007). Safe composition of product lines. In *GPCE '07*, pages 95–104, New York, NY, USA. ACM. 204

Thompson, J. M. and Heimdahl, M. P. E. (2003). Structuring product family requirements for n-dimensional and hierarchical product lines. *Requirements Engineering*, 8(1):42–54. 200

Thüm, T., Batory, D., and Kästner, C. (2009). Reasoning about edits to feature models. In *ICSE'09*, pages 254–264. IEEE. 23, 28, 43, 53, 77, 82, 101, 108, 114, 128, 132, 155, 187, 190, 200

Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A., and Toro, M. (2008a). Automated error analysis for the agilization of feature modeling. *J. Syst. Softw.*, 81(6):883–896. 27, 28

Trinidad, P., Benavides, D., Ruiz-Cortés, A., Segura, S., and Jimenez, A. (2008b). Fama framework. In *Proceedings of the 2008 12th International Software Product Line Conference*, pages 359–, Washington, DC, USA. IEEE Computer Society. 32

Trujillo, S., Batory, D., and Diaz, O. (2006). Feature refactoring a multi-representation program into a product line. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 191–200, New York, NY, USA. ACM. 42

Tun, T. T., Boucher, Q., Classen, A., Hubaux, A., and Heymans, P. (2009). Relating requirements and feature configurations: A systematic approach. In *SPLC'09*, pages 201–210. IEEE. 28, 30, 31, 44, 166, 200

Tun, T. T. and Heymans, P. (2009). Concerns and their separation in feature diagram languages - an informal survey. In *Proceedings of the Workshop on Scalable Modelling Techniques for Software Product Lines (SCALE@SPLC'09)*, pages 107–110. 42

van den Broek, P. M., Galvao Lourenco da Silva, I., and Noppen, J. A. R. (2010). Merging feature models. In *14th International Software Product Line Conference, Volume 2, Jeju Island, South Korea*, pages 83–89, Lancaster, UK. Lancaster University, Lancaster, UK. 68

van der Linden, F. (2002). Software Product Families in Europe: The Esaps & Cafe Projects. *IEEE Software*, 19:41–49. 12

van der Storm, T. (2004). *Variability and Component Composition*, pages 157–166. Software Reuse: Methods, Techniques and Tools. 31, 200

van Deursen, A. and Klint, P. (1998). Little languages: little maintenance. *Journal of Software Maintenance*, 10:75–92. 109, 110

van Ommering, R. (2002). Building product populations with software components. In *ICSE '02*, pages 255–265. ACM. 71

van Ommering, R. and Bosch, J. (2002). Widening the scope of software product lines - from variation to composition. In *Software Product Lines*, pages 31–52. LNCS. 205

Vanbeneden, C. (2011). Extraction automatisée d'un Feature Model à partir d'un Catalogue de Produits (MSc Thesis). Master's thesis, University of Namur, Namur, Belgium. 203

Voelter, M. and Groher, I. (2007). Product line implementation using aspect-oriented and model-driven software development. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 233–242, Washington, DC, USA. IEEE Computer Society. 19, 30

Weiser, M. (1981). Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA. IEEE Press. 89

Weiss, D. M. and Lai, C. T. R. (1999). *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 12, 17

Weston, N., Chitchyan, R., and Rashid, A. (2009). A framework for constructing semantically composable feature models from natural language requirements. In *SPLC'09*, volume 446 of *ICPS*, pages 211–220. ACM. 44, 203

White, J., Benavides, D., Schmidt, D. C., Trinidad, P., and Ruiz-Cortés, A. (2008). Automated diagnosis of product-line configuration errors in feature models. In *SPLC'08*. IEEE. 28, 201

Xtext (2009/2011). http://www.eclipse.org/Xtext/. 127

Zaid, L. A., Kleinermann, F., and Troyer, O. D. (2010). Feature assembly: A new feature modeling technique. In Parsons, J., Saeki, M., Shoval, P., Woo, C. C., and Wand, Y., editors, *ER*, volume 6412 of *Lecture Notes in Computer Science*, pages 233–246. Springer. 31, 44

Zaid, L. A., Kleinermann, F., and Troyer, O. D. (2011). Feature assembly framework: towards scalable and reusable feature models. In [Heymans et al. 2011], pages 1–9. 31, 44, 200

Zave, P. and Jackson, M. (1997). Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30. 29

Zhang, H. and Jarzabek, S. (2004). Xvcl: a mechanism for handling variants in software product lines. *Sci. Comput. Program.*, 53(3):381–407. 19

Ziadi, T. and Jézéquel, J.-M. (2006). *Product Line Engineering with the UML: Deriving Products*, chapter 15, pages 557–586. Number 978-3-540-33252-7 in Software Product Lines: Reasearch Issues in Engineering and Management. Springer Verlag. 18, 30

Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araujo, J., and Kulesza, U. (2009). Vml* – a family of languages for variability management in software product lines. In *Software Language Engineering (SLE'09)*, volume 5969 of *LNCS*, pages 82–102. Springer. 18