

Using multiple Feature Models of Domains and Regulations to develop Configuration Systems

A Dissertation
by
JAIME CHAVARRIAGA

Submitted to the Faculty of Engineering of the
UNIVERSIDAD DE LOS ANDES
in partial fulfillment for the requirements for the Degree of
DOCTOR IN ENGINEERING

and
Submitted to the Faculty of Sciences of the
VRIJE UNIVERSITEIT BRUSSEL
in partial fulfillment for the requirements for the Degree of
DOCTOR IN SCIENCES

Promotors: Prof. Dr. Rubby Casallas
Prof. Dr. Viviane Jonckers

Jury: Prof. Dr. Mathieu Acher
Prof. Dr. David Benavides
Prof. Dr. Philippe Cara
Prof. Dr. Dario Correal
Prof. Dr. Wolfgang De Meuter
Prof. Dr. Kelly Garces

February 2017

Abstract

A *Configuration System* or *Configurator* is a software tool that guides users to specify products by selecting options and preventing conflicts among these choices. Configuration Systems for complex products such as Cars or Electrical Transformers are usually hard to model and create. First, these products comprise multiple domains of the concern of different domain experts. In a car, for instance, there are subsystems for the motor, the power train, the sensors for the car periphery, and the lighting. Second, these domains very often have interactions among them, e.g., modern lighting such as the automated leveled lights and the adaptive cornering lights of cars require specialized sensors. Third, which features in each domain must be present or not may be determined by standards and regulations that vary from one country to the other. For instance, the mentioned automated leveled lights are required in all the new cars in Europe but required only in cars equipped with high intensity headlights in the USA. Finally, these domains and standards may be shared across multiple product families. Trying to use a single model that represents all the domains, standards and families results in a model with a large number of elements, that is consequently, hard to create, process, and maintain.

Feature-based Configuration uses *Feature Models* to represent the configuration options and constraints. Feature models have been found easy to understand by non-technical users and easy to analyze using automated tools and solvers. A complex product may be modeled using a single feature model for all the domains, or a different model for each domain and concern. There are several operations to process and merge these models when multiple models are used. Regretfully, these approaches and operations aim to support (almost) orthogonal feature models that represent different domains, even in the presence of complex interactions, but are not intended to support crosscutting concerns such as feature models representing standards and regulations comprising mainly constraints on features that already exist in the other domains.

This dissertation aims to overcome some limitations of the existing approaches for modeling and creating configuration systems in the presence of multiple domains and standards:

Modeling: First, we propose an approach where the diverse domains and standards are modeled using multiple artifacts: *feature models for each domain*, *constraint sets* representing the interactions among the domains, and *feature models for the standards*.

A standard may restrict which features can be selected or not in the other domains, can define valid combinations of features and specify concrete feature values. Using a different feature model, each standard can be modeled by experts on the standard using the easy-to-understand structure of the model instead of a set of constraints. We have defined a systematic process to create and review the models with different experts. These models can be reused and combined to support diverse product families.

Model Processing: Second, we propose a new set of operations for feature models mainly aimed to combine feature models for domains and for standards. Considering that standards are crosscutting concerns that can be enforced or not depending on the customer requirements, we define operations for *Conditional Intersection Merge*, that can enforce a standard only when it is selected, and for *Partial Conditional Intersection Merge*, that can enforce only the part of a standard related to a domain. We define an operation for *Combination* that performs multiple operations to combine properly the diverse models for the domains and the standards of a product family.

Configuration Systems Derivation: Third, we propose a strategy to derive Configuration Systems. Our approach comprises, on one hand, a set of automated model transformations that takes feature models of a product family and specifications of the desired user interface to produce the corresponding configuration system. On the other hand, it comprises a set of runtime components that processes the user decisions and updates the user interface consequently. In contrast to other proposals, our user-decision processor exploits the previous decisions of the user to perform tasks such as *undoing* and *changing decisions* more efficiently.

Experimental Implementation: In addition, we build a complete implementation of tools that: (1) help users to combine the models and analyze the feature models for domains and standards, (2) derive automatically configuration systems from those models, and (3) process decisions during the configuration process.

Finally, we have applied our approach in an industrial case study. We worked in a joint research project between Siemens Colombia and Universidad de los Andes to model and create multiple configuration systems for Electrical Transformers using our approach and tools.

The main contributions of this dissertation are: (1) An approach to model features of products involving multiple technical domains and standards, (2) a set of automated operations to combine multiple models representing diverse domains and standards, (3) a model-driven approach to derive configuration systems from these models, (4) a complete implementation of all the techniques we propose, and (5) a case study on modeling Configuration Options of Electrical Transformers.

Keywords: Configuration Systems, Variability Management, Feature Models, Model-driven engineering.

Samenvatting

Een *Configuratiesysteem* of *Configurator* is een software tool dat gebruikers helpt om producten te specificeren door opties te selecteren en daarbij conflicten tussen gemaakte keuzes te vermijden. Configuratiesystemen voor complexe producten zoals auto's of transformatoren zijn doorgaans zeer moeilijk te modelleren en creëren. Vooreerst bestrijken deze producten in het algemeen meerdere domeinen waarvan de bekommernissen typisch door verschillende domein experts worden afgedekt. In een auto bijvoorbeeld zijn er subsystemen voor de motor, de aandrijving, de omgevingssensoren, de verlichtingssystemen, etc. Daarnaast interageren deze domeinen dikwijls. Bijvoorbeeld moderne koplampen die zichzelf automatisch uitlijnen of meedraaide hoeklichten vereisen dat gespecialiseerde sensoren aanwezig zijn. Daarbovenop kunnen standaarden en regelgeving, die van land tot land kunnen verschillen, verplichtingen of beperkingen opleggen aan welke opties aanwezig moeten of kunnen zijn. Bijvoorbeeld de zelf-uitlijnende koplampen zijn binnen Europa verplicht op alle nieuwe wagens terwijl ze in de USA maar verplicht zijn als de wagen is uitgerust met lampen met hoge intensiteit. En tenslotte moeten deze domeinen en standaarden kunnen gedeeld worden over verschillende productfamilies. Proberen om al deze domeinen, standaarden en productfamilies te vatten in één enkel model leidt onvermijdelijk tot een model met zeer veel elementen en relaties dat moeilijk te maken is, moeilijk te verwerken is en moeilijk te onderhouden is.

Kenmerk-gebaseerde Configuratie gebruikt *Kenmerk Modellen* (Engels: Feature Models) om alle configuratie opties en beperkingen te representeren. Van deze modellen wordt vooropgesteld dat ze gemakkelijk te begrijpen zijn door niet-technische gebruikers en dat ze daarnaast ook een goede basis vormen voor geautomatiseerde analyse en verwerking. Een complex product kan gemodelleerd worden door één enkel model dat alle deeldomeinen afdekt of door een aantal verschillende modellen voor elk deeldomein of bekommernis. Wanneer verschillende modellen worden gebruikt zijn er een aantal operatoren nodig om deze modellen met elkaar te integreren. De benaderingen en operatoren die op dit moment worden voorgesteld in het gerelateerd onderzoek veronderstellen dat de verschillende modellen quasi orthogonale deeldomeinen representeren, zelfs indien er complexe interacties tussen de deelmodellen aanwezig zijn. Zij schieten tekort wanneer elkaar doorkruisende modellen moeten gecombineerd worden zoals bijvoorbeeld de modellen die standaarden voorstellen en die typ-

isch beperkingen opleggen aan de elementen die al voorkomen in de domein modellen.

Ons onderzoek heeft als doel om limitaties van bestaande benaderingen om, in de aanwezigheid van meerdere domeinen en meerdere standaarden, configuratiesystemen te modelleren en te creëren weg te werken.

Modelleren: We stellen een benadering voor waar de diverse domeinen en standaarden gemodelleerd worden door meerdere artefacten: domein-specifieke kenmerk modellen voor elk technisch deeldomein, constraint sets om interacties tussen verschillende technische domeinen te vatten, en standaard-specifieke kenmerk modellen. Een standaard kan verplichtingen of beperkingen opleggen aan welke kenmerken uit technische domeinen wel of niet kunnen geselecteerd worden, hoe ze mogen gecombineerd worden, of welke concrete waarden sommige kenmerken moeten aannemen. Anders dan een domein-specifiek model zal een standaard-specifiek model daarom typisch model elementen uit de andere domein modellen hergebruiken. Ook voor standaarden stellen we voor om per standaard een apart model te gebruiken dat beheerd wordt door een expert ter zake. Het model neemt ook de vorm aan van een weliswaar specifiek type van kenmerk model en is daardoor ook makkelijk te begrijpen voor die expert. We hebben een systematisch proces gedefinieerd om de modellen te maken, inclusief stappen om ze te valideren met meerdere experts. De modellen kunnen dan hergebruikt en gecombineerd worden om meerdere productfamilies voor te stellen.

Modellen verwerken: We stellen een aantal nieuwe operatoren voor op kenmerk modellen, in het bijzonder om domein-specifieke en standaard-specifieke modellen te combineren. Gezien standaarden doorsnijdende bekommernissen zijn die wel of niet moeten afgedwongen worden afhankelijk van de gebruikerseisen of de regelgeving van toepassing in het land waar het product zal in gebruik genomen worden zijn speciale operatoren nodig. We definiëren *Conditional Intersection Merge*, om een standaard af te dwingen als hij geselecteerd wordt en *Partial Conditional Intersection Merge*, om alleen dat deel van een standaard af te dwingen dat van toepassing is op een gekozen deeldomein. We definiëren daarnaast een algemene *Combination* operatie die onze nieuwe operatoren combineert met gekende operatoren om de diverse domein-specifieke modellen en standard-specifieke modellen voor een productfamilie samen te stellen.

Configuratie systemen afleiden: We ontwikkelen een model-gedreven benadering om van een set van domein- en standard-specifieke kenmerk modellen automatisch een configuratie systeem af te leiden. Deze benadering behelst, aan de ene kant, model-transformaties die alle modellen voor een productfamilie en een specificatie van gebruikersinterface componenten vertalen naar een corresponderend configuratie systeem. Aan de andere kant worden een aantal run-time componenten ingezet om keuzes van de gebruiker te verwerken en de gebruikersinterface overeenkomstig aan te passen. In tegenstelling tot gerelateerd onderzoek kunnen onze run-

time componenten taken zoals het ongedaan maken of veranderen van een keuze meer efficiënt aanpakken.

Experimentele implementatie: We bouwden complete implementaties voor tools die (1) gebruikers helpen om domein- en standaard-specifieke kenmerk modellen te combineren en te n=analyseren, (2) automatisch configuratie systemen af te leiden van deze modellen, en (3) gedurende een configuratieproces beslissingen en keuzes te verwerken,

We hebben ons onderzoek toegepast in een industriële gevalstudie. In een gemeenschappelijk onderzoeksproject tussen Siemens Columbia en de Universidad de los Andes werden met onze benadering en onze tools meerdere configuratie systemen voor elektrische transformatoren gerealiseerd.

De belangrijkste bijdragen van ons onderzoek zijn: (1) Een benadering om kenmerken van complexe producten waarin meerdere domeinen en standaarden een rol spelen te modelleren met kenmerk modellen (2) Een aantal operatoren om die diverse kenmerk modellen te combineren (3) Een model-gedreven benadering om configuratie systemen automatisch uit deze modellen af te leiden (4) Een volledige implementatie van alle voorgestelde technieken (5) Een reële wereld, industriële gevalstudie rond de configuratie van elektrische transformatoren

Sleutelwoorden: Configuratie Systemen, Variabiliteit Management, Kenmerk Modellen, Model-gedreven Engineering

Resumen

Un *Sistema de Configuración* es una herramienta de software que guía a los usuarios en la especificación de productos, permitiéndoles seleccionar opciones y previniendo conflictos entre las escogencias. Los Configuradores para productos complejos como carros o transformadores eléctricos son usualmente muy difíciles de modelar y crear. Primero, estos productos comprenden múltiples dominios de interés para distintos expertos de dominio. En un carro, por ejemplo, hay subsistemas para el motor, la transmisión, los sensores periféricos del carro, y las luces. Segundo, estos dominios muy frecuentemente tienen interacciones entre ellos. Por ejemplo, en un carro, modernos sistemas de luces como los sistemas de nivelación de luz automática o las luces que giran a medida que el carro da vuelta, requieren sensores especializados. Tercero, estándares y regulaciones que varían de país a otro pueden definir cuáles de estas características deben estar o no. Por ejemplo, los sistemas para nivelar luces automáticamente son requeridos en todos los carros nuevos en Europa, pero requeridos solo en los carros equipados con lámparas de alta intensidad en Estados Unidos. Finalmente, estos dominios y estándares pueden ser compartidos a través de varias familias de productos. Tratar de utilizar un único modelo que represente todos los dominios, estándares y familias resulta en un gran modelo con una gran cantidad de elementos, consecuentemente, este es difícil de crear, procesar y mantener.

La *Configuración basada en características* usa *modelos de características* para representar las opciones y las restricciones de configuración. Los modelos de características son fáciles de entender por usuarios no técnicos y fáciles de analizar usando herramientas automáticas y *solvers*. Un producto complejo puede ser modelado utilizando un único modelo de características para todos los dominios o un modelo diferente para cada dominio. Existen operaciones para procesar y fusionar estos modelos. Infortunadamente, estos enfoques y operaciones presuponen que los dominios son ortogonales. Estos enfoques no soportan preocupaciones transversales como los modelos de características que representan los estándares y regulaciones, que incluyen principalmente restricciones que involucran características presentes en los otros dominios.

Esta disertación pretende superar estas limitaciones de los enfoques existentes para modelar y crear sistemas de configuración en la presencia de múltiples dominios y estándares:

Modelamiento: En primer lugar, proponemos un enfoque en el que los diversos dominios y estándares se modelan usando múltiples artefactos: modelos de característica para representar cada dominio, modelos representando cada estándar y conjuntos de restricciones que representan las interacciones entre los dominios. Un estándar puede restringir qué características pueden ser seleccionadas o no en otro dominio, puede definir combinaciones válidas de características y especificar valores específicos de las características. Utilizando un modelo de características diferente, cada estándar puede ser modelado por expertos en la norma tomando ventaja de la estructura del modelo, que es de fácil comprensión, en vez de modelar solo restricciones. Hemos definido un proceso sistemático para crear los modelos, incluyendo pasos para revisarlos con diferentes expertos. Estos modelos pueden ser reutilizados y combinados para crear diversas familias de productos.

Procesamiento de los modelos: En segundo lugar, proponemos un nuevo conjunto de operaciones sobre los modelos de características principalmente destinadas para combinar modelos de dominio y estándares. Considerando que las normas son preocupaciones transversales que pueden ser impuestas o no dependiendo de los requisitos del cliente y de las regulaciones nacionales en los que se van a desplegar los productos, definimos operaciones que permiten fusionar los modelos de manera condicional (Conditional Intersection Merge) para obligar a que se cumpla el estándar cuando éste es seleccionado o la operación de fusión condicional parcial (Partial Conditional Intersection Merge) para obligar el estándar en algún dominio particular. Definimos una operación de Combinación que realiza múltiples operaciones como la reducción de la agregación de productos y la combinación de intersecciones condicionales parciales para combinar adecuadamente los diversos modelos específicos de dominio y de estándar de una familia de productos.

Derivación de un Sistema de Configuración: En tercer lugar, también proponemos una estrategia para derivar Sistemas de Configuración. Nuestro enfoque abarca, por un lado, un conjunto de transformaciones de modelos automatizadas que toma modelos de características de una familia de productos y unas especificaciones de la interfaz de usuario deseada, para producir el sistema de configuración correspondiente. Por otra parte, comprende un conjunto de componentes en tiempo de ejecución que procesa las decisiones del usuario y actualiza la interfaz de manera consecuente. En contraste con otras propuestas, nuestro procesador de decisiones de usuario utiliza las decisiones anteriores del usuario para realizar tareas tales como deshacer y cambiar decisiones de manera más eficiente.

Implementación Experimental: Además, construimos una implementación completa de herramientas que: (1) ayudan a los usuarios a combinar los modelos y analizar los modelos de características tanto de dominio como de estándar, (2) derivan de manera automática Sistemas de configuración

de esos modelos, y (3) procesan las decisiones del usuario durante el proceso de configuración.

Finalmente, hemos aplicado nuestro enfoque en un estudio de caso industrial. Trabajamos en un proyecto conjunto de investigación entre Siemens Colombia y la Universidad de los Andes para modelar y crear múltiples sistemas de configuración para Transformadores Eléctricos utilizando nuestro enfoque y herramientas.

Las principales contribuciones de esta tesis son: (1) un enfoque para modelar características de productos que involucran múltiples dominios técnicos y estándares, (2) un conjunto de operaciones automatizadas para combinar múltiples modelos que representan características en diversos dominios y estándares, (3) un enfoque basado en modelos para derivar sistemas de configuración a partir de estos modelos, (4) una implementación completa de todas las técnicas que proponemos, y (5) un caso de estudio sobre el modelamiento de las características de Transformadores Eléctricos.

Palabras clave: Sistemas de Configuración, Gestión de la Variabilidad, Modelos de Características, Ingeniería Basada en Modelos.

Acknowledgements

This dissertation would not have been possible without the tremendous support from my colleagues, friends and family. With these acknowledgements I would like to show my gratitude to all these people that helped me throughout the process.

First and foremost, I would like to thank my promoters Rubby Casallas and Viviane Jonckers for giving me this opportunity and promoting this thesis. I am very thankful to Rubby for believing in me and giving me the opportunity to start a Phd under her supervision. I am also greatly indebted to Viviane for her support, her input in my work and her useful advice during the time I was in the VUB. Without Rubby and Viviane, all I have done would not have been possible.

I would like to thank my advisor Carlos Noguera. There is no doubt that his discussions, questions and suggestions greatly shaped the work of this thesis. I cannot express my gratitude for all the time and effort he put into supervising my work.

I would like to thank to Carlos Rangel and all the people at Siemens Colombia. They gave me the opportunity to test and improve our ideas in a real life setting. In addition, I truly appreciate the many fruitful discussions we have. They were the main motivation of some of the contributions I am presenting here.

I sincerely thank the members of my jury for the time and effort they put into the evaluation of this work and in giving insightful comments: Mathieu Acher, David Benavides, Philippe Cara, Dario Correal, Wolfgang De Meuter and Kelly Garces. All their questions and comments help me to clarify part of my work and improve the document. Thanks.

I want to thank all the members of the Software Languages Lab for all their collaboration. They have created in the Lab a very stimulating environment, not only for researching, but also for learning about friendship and collaboration. I particularly want to thank Dennis Wagelaar and Ragnhild Van Der Straeten for their support during my first weeks in Brussels, and Coen de Roover, Jens Nicolay, Engineer Bainomugisha, Nicolás Cardozo, Angela Lozano, Dirk Van Deun and all the others for being willing to help me when I needed. I thank you all.

I would also like to thank the members of the TiCSw Research Group and the Departamento de Sistemas y Computación en Uniandes who supported me during my PhD. Special thank goes out to Ferney Maldonado, Carlos Lozano, Carlos Velasquez, Carlos Rozo, Dorys Bolivar and Yomara Rincón. They have helped me out with academic duties and administrative issues on countless occasions. In addition, I would thank Hugo Arboleda for showing me the path and motivating me to start the PhD.

This dissertation have been partially funded by a PhD Scholarship of COLCIENCIAS. It have been also funded by a teaching assistance of Universidad de los Andes and a joint-project with Siemens Colombia. I am very grateful with all these institutions for the opportunity they gave me.

I would like to thank to all my family: Lubier, Betty, Carlos, Ricardo; and the family of my wife: Eduardo, Cecilia and Federico. I've been blessed with the happy and very supportive family they are. I admire them profoundly and I have tried hard to be someone they can feel proud. Gracias por todo su apoyo y todo su amor.

Last but not least, I would like to thank to my wife Sandra and my son Joaquin for their love, their patience and understanding during all these time. Ustedes son lo más importante en mi vida. Espero que algún día pueda recompensarlos por todo el amor, todo el esfuerzo y toda la dedicación que han tenido conmigo en estos últimos años. Gracias de todo corazón.

Contents

Abstract	i
Samenvatting	iii
Resumen	vii
Acknowledgements	xi
1 Introduction	1
1.1 Feature-based Configuration Systems	1
1.1.1 A Motivation Case: Feature-based Configuration for Cars	3
1.1.2 Challenges Modeling multiple Domains	4
1.1.3 Challenges Modeling Standards and Regulations	5
1.1.4 Challenges Creating the Configuration Systems	6
1.1.5 Our Case Study: Feature-based Configuration for Electrical Transformers	6
1.2 Regulations as a special case of external constraints	8
1.3 Problem Statement	9
1.4 Contributions	9
1.5 Organization of the Document	10
2 Background	13
2.1 Configuration Systems	13
2.2 Feature-based Configuration Systems	14
2.3 Feature Models	15
2.3.1 Concrete Syntax	15
2.3.2 Semantics	16
2.4 Automatic Processing of Feature Models	17
2.4.1 Model Analysis Operations	18
2.4.2 Approaches to implement Analysis Operations	18
2.4.3 Model Processing Operations	21
2.4.4 Approaches to implement Processing Operations	22
2.5 Automatic Processing of Interactive Configurations	23
2.5.1 Operations for Interactive Configuration	23
2.5.2 Approaches to implement Interactive Configuration	24

2.6	Summary	26
3	Related Work	27
3.1	Modeling multiple Domains and Standards	27
3.1.1	Proposals using a Monolithic Feature Model	28
3.1.2	Proposals using Views on a Feature Model	30
3.1.3	Proposals using multiple Feature Models	32
3.1.4	Proposals using multiple Types of Models	33
3.1.5	Discussion	35
3.2	Composing multiple Feature Models	36
3.2.1	Structure-based Merging of Feature Models	36
3.2.2	Semantic-based Merging of Feature Models	37
3.2.3	Discussion	40
3.3	Deriving Configuration Systems from Feature Models	40
3.3.1	Automated Derivation of Configuration Systems	41
3.3.2	Existing libraries to process Feature Configurations	41
3.3.3	Discussion	42
3.4	Summary	42
4	Modeling Features of multiple Domains and Standards	45
4.1	Modeling Domains and Standards	45
4.1.1	Overview	46
4.1.2	Running Example: Feature-Based Configuration for Cars	46
4.1.3	Modeling Multiple Domains	47
4.1.4	Modeling Domain Interactions	50
4.1.5	Modeling Standards	51
4.2	Reviewing and Analysing the Models	53
4.2.1	Motivation	53
4.2.2	Analysis of the Models	54
4.2.3	Testing of the Models	57
4.3	Discussion	57
4.4	Summary	59
5	Combining Feature Models representing Multiple Domains and Standards	61
5.1	A Glimpse to the Background	62
5.1.1	Feature Models	62
5.1.2	Operations to merge Feature Models	63
5.2	Challenges Merging Feature Models for Standards	65
5.2.1	Example	65
5.2.2	Unconditional enforcing of standards	66
5.2.3	Conditional enforcing of Standards	67
5.2.4	Conditional Partial enforcing of Standards	68
5.2.5	Combination (with conditional enforcing) of Standards	70
5.3	New Merge Operations	71
5.3.1	Conditional Intersection Merge	72

5.3.1.1	Definitions	72
5.3.1.2	Implementation	74
5.3.2	Conditional Partial Intersection Merge	80
5.3.2.1	Definitions	80
5.3.2.2	Implementation	82
5.3.3	Combination of Domains and Standards	83
5.3.3.1	Definitions	83
5.3.3.2	Implementation	83
5.4	Discussion	84
5.4.1	Classification of the Operations	84
5.4.2	Implementation of the Operations	85
5.5	Conclusions	86
6	Deriving Configuration Systems from Models representing multiple Domains and Standards	89
6.1	Deriving the User Interface	90
6.1.1	Example	90
6.1.2	Modeling Configuration Options	91
6.1.3	Mapping Configuration Options to User Interface	92
6.1.4	Deriving the Configurator User Interface	93
6.2	Processing User Decisions during Configuration	95
6.2.1	The Reasoner	95
6.2.2	The User Decision Processors	96
6.2.2.1	Select and Deselect Feature Operation	96
6.2.2.2	Undo Feature Operation	97
6.2.2.3	Revise Feature Operation	98
6.2.2.4	Force Change Feature Operation	99
6.2.3	Our Implementation of User Decision Processors	99
6.2.3.1	Representation of the Configuration State.	99
6.2.3.2	Select and Deselect Feature Operation	100
6.2.3.3	Undo Feature Operation	102
6.2.3.4	Revise Feature Operation	104
6.2.3.5	Force Change Feature Operation	106
6.3	Discussion	106
6.3.1	Addressing issues	106
6.3.2	Comparison to other proposals	107
6.3.3	Comparison to existing libraries	108
6.3.4	Evaluation	109
6.4	Conclusions	109
7	Experimental Implementation	111
7.1	Overview	112
7.2	Tools to create Feature Models and Constraints	113
7.3	Operations to merge Feature Models	115
7.4	Model-transformations to derive Configuration Systems	115
7.5	An Extensible Web-based Epsilon Runtime	116

7.6	Epsilon extensions to process Feature Models and Constraints .	119
7.7	Conclusions	124
8	Case Study: Electrical Transformers at Siemens	125
8.1	Intended Configuration Scenario	126
8.2	Modeling Process in Siemens	127
8.3	Models for Electrical Transformers	130
8.3.1	Modeling Multiple Domains	130
8.3.2	Modeling Domain Interactions	133
8.3.3	Modeling Standards	133
8.4	Reviewing and Analyzing the Models	134
8.4.1	Analysis by combining domains	134
8.4.2	Analysis by combining domains and standards	136
8.4.3	Analysis by combining all the domains and standards .	137
8.5	Reviewing and Testing the Models	139
8.6	Lessons Learned	140
8.6.1	Modeling	140
8.6.2	Validation of the models	141
8.6.3	Tooling	142
8.6.4	Impact on other engineering processes	142
8.7	Discussion	143
8.7.1	Research Method	143
8.7.2	Threats to Validity	144
8.8	Conclusions	145
9	Conclusion and Perspectives	147
9.1	Summary of Contributions	147
9.2	Research Perspectives	149
9.2.1	Regarding the Modeling of Domains and Regulations . .	149
9.2.2	Regarding the Combination of Feature Models.	149
9.2.3	Regarding the Derivation of Configuration Systems. . .	150
Appendix A	Approaches for Intersection Merge	153
A.1	Running example	154
A.2	Approaches to implement the intersection merge	155
A.2.1	Semantic-based implementation	155
A.2.2	Reference-based implementation	157
A.2.3	Hybrid implementations	159
A.3	Our approach: Adding constraints	160
A.4	Comparison	162
A.5	Conclusions	163
Appendix B	Alternative implementations of Conditional Intersection Merge	165
B.1	Conditional Intersection Merge	166
B.1.1	Running example	166

B.2	Alternative implementations	168
B.2.1	Semantic-based implementation: Composing operations	168
B.2.2	Semantic-based implementation: Formula Calculation .	171
B.2.3	Reference-based implementation	173
B.2.4	Hybrid implementations: Slice of a compound model . .	176
B.2.5	Implementation Adding Constraints	177
B.3	Comparison	181
B.4	Conclusions	181
	Bibliography	183

Chapter 1

Introduction

1.1 Feature-based Configuration Systems

Configuration Systems (or Configurators) are used in industries such as automotive, electronics and furniture to allow sellers and clients to customize the products according to their needs and expectations[47].

Companies such as Renault¹ and Audi² provide *Web-based Configurators* where customers can customize a vehicle before purchase. Figure 1.1 shows a screenshot of the Audi Configurator. The software captures user decisions and validates if the *Product Configuration* (i.e., the set of selected features) satisfies already defined constraints about which features can be included in the same product. When a user decides that a configuration complying with the constraints is complete, usually she can request for the corresponding product. Depending on the company, the product is then manufactured, assembled and delivered according to the specifications of the client.

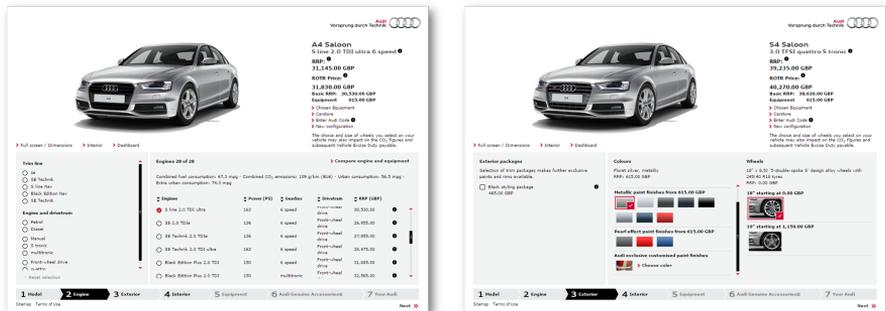


Figure 1.1: Screenshots of the Audi web configurator

¹<https://www.renault.co.uk/vehicles/configure.html>

²<http://configurator.audi.co.uk/>

In *Feature-based configuration*, configuration options and constraints are specified using *Feature Models* [19][67]. These models represent products as a hierarchy of features whose selection is ruled by constraints. They have been found easy to understand by non-technical users [80] and easy to analyse using automated tools and solvers [87][92].

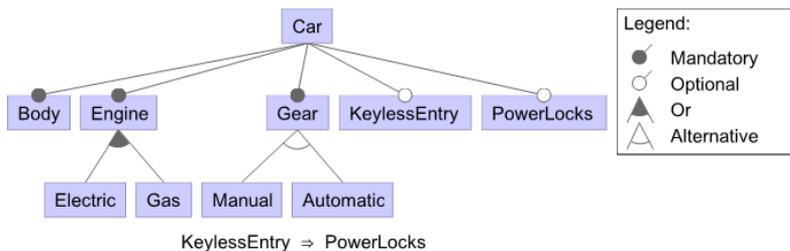


Figure 1.2: Example Feature Model of a Car [38]

For instance, Figure 1.2 shows a Feature Model representing a Car [38]. It comprises a tree where the root is the *concept*, i.e., the element to configure, and the branches and leaves are *features*, i.e., user-selectable options. The model includes *mandatory* features, such as the Body and the Engine, that are represented using a filled circle on top, and *optional* features such as the keyless-entry and the power-locks, represented using a hollow circle. The feature Engine is an *or-group* including Electric and Gas engines. That means that a user can select only one or both. In contrast, the feature Gear is an *alternative group*. Thus, a user can select either manual- or automatic-gear but not both. Finally, the feature model includes an *cross-tree constraint* at the bottom indicating that the feature keyless-entry requires the feature power-locks. If a user selects the first, the software must select automatically the other if the user has not selected it yet.

Although there are many proposals aimed to help, creating Feature-based Configuration Systems remains as a challenge. This is specially true for companies manufacturing multiple families of products in different countries that require configuration systems considering multiple domains and diverse standards and regulations. In such systems, the required Feature Models become large and hard to create and maintain. In addition, these companies may require multiple Configuration Systems, each one targeting a different market or product line. It is not easy (1) to create and manage the diverse models for the different domains and regulations, neither (2) to determine which subset of features must be used for each configuration system to develop the corresponding software.

Our work is focused on creating Feature-based Configuration Systems in presence of multiple domains and regulations. Next section presents first some concepts we use in all this document and then introduces the challenges we want to tackle.

1.1.1 A Motivation Case: Feature-based Configuration for Cars

It is well known that creating the Feature Models for Configuration Systems in the automotive industry is not easy because the complexity of the products [26][50][145]:

Multiple domains: First, cars are complex products where *multiple systems* co-exist. A typical car may include more than 300 different subsystems [50]. A correctly configured car implies a lot of correctly defined elements such as its engine, its power train and its set of electronic control components. In addition, a single component, such as one of the “Electronic Control Units (ECU)” of a car, may be instantiated in more than 10.000 different ways [106]. A typical car network, with more than 50 ECUs, may be instantiated in millions of different configurations. It is practically impossible for a single person to model the configuration options for all the car. The configuration options belongs to different technical *domains* that must be specified by different *domain experts*.

Multiple domain interactions: Second, although there are many subsystems, these subsystems are not fully independent one of the others. Some features require complex interactions among completely different subsystem. For instance, including a “park assistant” in a car requires the presence of a sensor to measure the relative position of the car to the parking space [50]. On some cars this sensor is a camera while on others it may be a sonar detector. Park assist also requires brakes that accept software control, and may require particular versions of steering controls. Modeling (and designing) these complex features requires the coordination of the groups that are responsible of the involved subsystems.

Multiple standards and regulations: Third, complex products usually must comply with diverse *standards and regulations* around the world. For instance, “Daytime Running Lights” are required equipment in Canada, Norway, and Sweden, prohibited in Japan and China and optional in the United States, Europe, Australia, and the rest of the world [50]. In the “Headlamps”, the “low-beams (or dipped-beams)” that are standard in USA do not comply the regulations of Europe [56]. There are so many differences in the regulations for elements such as high beams, brake lamps and fog lamps that impede to produce a single type of lighting for all the countries [56]. The configuration options for a car must be constrained by the specific regulations imposed by the country where the product will be sold. If the modelers create separate feature model for each country considering only the product options and the regulations that apply, the resulting models will include redundancies and may lead to inconsistencies [29][113].

Multiple product lines: Finally, these industries define *multiple product lines* sharing components among them. For instance, these companies usually define technological platforms to support multiple families of products. The *Volkswagen A series platform* is used in models such as the *Audi A3*, *Audi Q3*, *VW Golf*, *VW Jetta*, *SEAT León* and *SEAT Toledo* [73]. Although the platform itself has a definition of all its configuration alternatives, creating a configuration system for one of its car families require to define the specific subset of options that must be considered.

The intrinsic complexity of the cars introduces, at least, two sets of challenges: On one hand, challenges about how to create, review and analyse feature models that represent the diverse domains, domain interactions, regulations and product lines. On the other hand, challenges about how to specify and create configuration systems based on that models.

1.1.2 Challenges Modeling multiple Domains

Given the complexity of the cars, creating the corresponding models is not an easy task [50][105]. A typical process involves multiple technical domains and domain experts [105]. In addition, the proliferation of standards and regulations introduce new complexities. We will describe here challenges of modeling multiple domains using the existing approaches.

Many authors have proposed techniques and strategies to create and structure the corresponding feature models. According to a recent study from Oliynyk et al. [104][105], these proposals can be classified into two categories:

1. *Proposals using Monolithic Feature Models*, that use a single feature model to represent all the products, and
2. *Proposals using Multiple Feature Models*, that use multiple feature models to represent the product. These proposals are known as *Modular Feature Models*, when the models are used to represent just one family of products, or *Multiple Product Lines*, when some feature models represent parts and subsystems that can be reused in multiple product lines.

Monolithic Feature Models. Historically, the first proposals tried to create a single feature model representing the whole system [80][37]. The resulting feature model is typically organized in multiple abstraction levels. For instance, it may include a level including features representing customer features, another level with subsystem features and another level with component functions. The feature model included traceability links and constraints relating the features in the diverse levels.

Regretfully, creating a single feature model for complex products such as a car is not easy. Several studies have identified many problems related to use a single feature model [26][105][111]. First, the resulting feature model is hard to review. A feature model for a car may include thousands of features and constraints. Although there are means to automatically detect errors [19], it is necessary that the modelers review the models to determine if they represent

the features and constraints of the real products. This *semantic quality* of the models cannot be analysed without human review. Second, features cannot be reused. If a feature (or a set) is required multiple times or can be applied for other product lines, the feature should be duplicated [105]. This creates unnecessary complexity; the resulting model turns hard to maintain and understand. And third, there is not any mechanism for separating concerns, complicating the division of the work [67][105]. Any modification of any user may affect the features that are of the concern of other users. A user modifying some part of the model may introduce an error or inconsistencies on other parts of the model without notice it.

Modular Feature Models More recently, other approaches aim to create multiple feature models to represent complex products [58][67][111][120]. Each model represents a different subsystem or a different concern [67]. Additional models can be used to capture the variability in the context [58], e.g., to represent the multiple standards that may apply. In these approaches, relationships between features may be used to represent constraints and dependencies involving diverse concerns and standards. As a result, the models are smaller and easier to review, debug and reuse.

1.1.3 Challenges Modeling Standards and Regulations

Monolithic Feature Models. The challenges mentioned above using a single feature model are exacerbated by including standards and regulations in the monolithic model. For instance, despite recent efforts of many governments, car regulations may be quite different across the world. Almost all of the countries have different norms that must be considered. Modeling all the standards in the same model implies including a lot of features and constraints. Consequently, the models are even harder to review and debug. In addition, considering that many regulations may be similar among them, some constraints may result being repeated many times. Modellers creating or updating a standard may alter inadvertently other standards. Finally, using a single model impedes that experts on each standard may specify the corresponding constraints independently of the other domains and standards for the product.

Modular Feature Models Modelling the standards and regulations with multiple feature models may be challenging as well. The standards constitute *cross-cutting concerns* that affect features in more than one technical domain. A single regulation may restrict features in domains such as the lighting, i.e., by prohibiting a type of lamp, and in the Car Periphery System, i.e., by enforcing the installation of some type of parking or driving sensors or applications. Trying to model the standard using models for each domain may result in a large number of rules scattered in all the models.

Recently, Hartmann et al. [58] modeled standards in an independent feature model for *context variability*. This model represents parameters that are global

for the whole system and take their value according to the type of customer or the market to which the products are targeted. It may include information of which standards and regulations can be enforced in a product line. Regretfully, this approach has its drawbacks too. First, the resulting model includes all the related standards and difficults the division of the work among multiple experts. Second, the model may result with a lot of constraints relating standards with features in diverse feature models, making it hard to review and debug. And finally, during the modeling of the standard, it is possible that modelers introduce rules that contradict the constraints defined in the other feature models.

1.1.4 Challenges Creating the Configuration Systems

Car companies such as Audi and Renault build multiple configuration systems. For instance, they have a different web-based software for Europe than for USA. The car models and configuration options in each system are different.

To create multiple configuration systems, companies may opt for one of two alternatives: On one hand, they could create a different feature model for each system. However, this approach result in models including redundancies that may lead to inconsistencies [29][113]. On the other hand, they could define feature models representing all the products and specify a subset of these models for each configuration system [66]. Regretfully, this approach is not easy either. Once selected which features to include in a configuration system, it is necessary to review that the subset of the models represent correctly the products that must support. In addition, if the feature models changes, it is necessary to update the subset of features and the corresponding configuration accordingly. Existing software tools for managing feature models offer limited support for these activities.

1.1.5 Our Case Study: Feature-based Configuration for Electrical Transformers

Problems creating Feature Models and Configuration Systems are not exclusive to the Automotive Industry. Part of our research was performed in a joint research project between Siemens Colombia and Universidad de los Andes [32]. Our work was focused on modeling and creating configuration systems for Electrical Transformers, complex products that must comply with different standards and regulations across the world. As with the car example, we must dealt with multiple domains, multiple standards and multiple configuration systems.

Models for multiple domains are hard to build and review In our experience, creating a single model for complex products such as Electrical Transformers is tough [32]. Mainly, the domain experts were overwhelmed by the number of domains, features and constraints. For instance, defining a single structure of the features is not easy because each

expert may conceive the product using a different point of view and taking in account different concerns. In addition, during the development of the models, sometimes experts may not understand why some features are deactivated after a user select others. They spend time trying to understand how some features affect the others by looking not only for the constraints that they defined, but also the constraints defined by the other experts. Furthermore, if they find a constraint that causes a undesired behaviour, sometimes they do not recognize if the constraint is imposed by the technical domain, by the inter-domain interactions or by some country- or client-specific standard.

Standards impose a lot of constraints to all the domains Electrical

Transformers must comply with a large variety of standards. Each power transmission and distribution network owner may define its own set of standard specifications for transformers. Currently, almost all the countries have different standards (e.g., IEEE Std C57.12 [72] is the standard in USA, IEC 60076 [71] applies in Europe and ICONTEC NTC 819 [69] in Colombia). Additionally, private portions of the networks created by industries can also define their own standards (e.g., Ecopetrol, the oil company from Colombia, has its own standards). Each of these standards imposes constraints such as enforcing some values for voltage or prohibiting some components or accessories in all the other technical domains. In addition, a single standard may apply to different types of transformers (e.g., the NEMA TR-1 [98] applies to pole-mounted, pad-mounted and many other types of transformers). Trying to create and maintain all these constraints turn the models into artifacts hard to review and maintain.

Specifying multiple Configuration Systems is not easy Siemens Colombia

wants to create different systems for specific markets and product lines. For instance, they are trying to use different systems for each type of transformer in each country. The software to configure a medium-size transformer in Colombia may be different to the one used for transformers with the same size in North America. In addition, it is possible that the software that uses a customer in the company's website is different than the software used by engineers or specialized sellers. When the engineers in Siemens tried to create the configuration systems using a single model, they ended with systems where a lot of features in the model are not relevant. When they tried to create multiple models, they ended with a lot of models with redundant, and possible inconsistent, features and constraints. It was not easy to specify and maintain the diverse configuration systems they were interested in.

1.2 Regulations as a special case of external constraints

Above mentioned challenges to create configuration systems are not exclusive to products involving standards and regulations. In our experience, technical domains of a product are defined internally by the companies while standards and regulations are defined by external organizations. Basically, these regulations impose additional constraints to the products of the companies. We consider standards as a special case of externally defined constraints.

We consider that the challenges found working with standards and regulations are also present in externally defined constraints with three distinctive characteristics:

Introduce additional constraints Standards and regulations may enforce or prohibit features, components and configurations of products in ways not considered in the original design. Intuitively, these regulations enforce restrictions not imposed by the company or the designers of the product. It is very likely that a product may be built being not compliant to the standard.

Reuse the product vocabulary Instead of introducing new elements to the product, standards and regulations *reuse the vocabulary* (i.e., reuse the elements of the product design) to add new constraints.

Are Cross-cutting concerns These constraints affect many, if not all, the technical domains of the product. In addition, it is possible that affect many families of products as well. Considering that a company may have multiple configuration systems, an externally defined constraint may affect the options of more than one of these systems. In fact, it is possible that (1) only a subset of the options of a configuration system is affected, or (2) only a subset of the constraints affects a specific configuration system.

In addition to standards and regulations, there are other examples of externally defined constraints where above mentioned challenges must be tackled to create the corresponding feature models and configuration systems:

- **Customer-defined policies and regulations.** There are customers that impose restrictions to the products they acquire or integrate to their systems. For instance, telecommunication companies and data-centers usually define rules for the products they install in their infrastructure. Companies manufacturing these products must consider not only the constraints imposed internally by their design but also the constraints defined by these customers.
- **Local Factory and provider limitations.** Companies manufacturing the same products in multiple factories and countries, are usually affected by their local limitations. It is possible that some components are not available or are too expensive. Companies may represent these limitations as externally defined constraints not included in the product design nor its technical domains.

1.3 Problem Statement

To tackle the complexity of creating feature models and configuration systems, this thesis addresses three research questions that remain as a challenge despite the related work. Several of these questions are refined in the next chapters as we delve in each problem:

- RQ1** *How to model the features of multiple technical domains and externally defined constraints such as standards and regulations in a way that they can be later used to produce configuration systems for different product lines?*
- RQ2** *How to process automatically models representing multiple domains and externally defined standards and regulations to combine them, detect potential errors and simplify the resulting combined model?*
- RQ3** *How to process automatically these models representing multiple domains and externally defined standards and regulations and derive corresponding Configuration Systems that can process user decisions and update the user interface consequently?*

1.4 Contributions

The main contribution of this thesis is a comprehensive approach to model, build and evolve feature-based configuration systems that involve multiple stakeholders, multiple domains and multiple standards and regulations. Specifically, it consists of:

- C1** *An approach to model features of products involving multiple technical domains and standards.* We extend existing approaches by using multiple feature models and constraints among feature models. Instead of creating a single feature model, domain experts create a model for each domain standard and regulation independently. Interactions among the domains are represented using sets of constraints among the corresponding feature models. The experts may analyse and test these models by combining them. We propose an iterative process to specify all the options for the configuration system by creating, combining and testing models of the domains and standards.
- C2** *An automated process to combine multiple models representing features in diverse multiple domains and standards.* These models represent, on one hand, configuration options and constraints defined by the technical properties and designs of the products, and on the other hand, constraints imposed by organizations for standardization and governments. Engineers interested on creating configuration systems must be able to combine them in different arrangements according to the product family to configure and the market where the products will be sold. We propose a set of automated operations to analyze and merge such models.

- C3** *An automated approach to derive configuration systems from models representing multiple technical domains and standards.* Considering that companies are interested on creating Configuration Systems for specific markets and countries, it is important to produce the software based on the correct combination of models. We propose a model-driven approach where feature models are used to generate the software components that interact with the user during the configuration process. It comprises a set of runtime components that processes the user decisions and updates the user interface consequently.
- C4** *A case study on modeling Configuration Options of Electrical Transformers.* We have developed part of our approach in a two-years research project aimed to create configuration systems for Electrical Transformers to be sold in diverse countries of North, Central and South America. Our experience in that project was used to improve, test and validate our approach. We present here a report of our experience including some lesson learned during the process.
- C5** *A complete implementation of all the techniques we propose.* We have developed the *FaMoSA toolset*, a set of software tools to (1) model, test and analyze models representing multiple domains and standards (2) analyze and test these models (3) combine these models according to the intended configuration process, and (4) derive automated configuration systems that stakeholders can use to decide and select features for a product.

1.5 Organization of the Document

This thesis document is organized in four parts. There is an *Introduction*, a part describing the *State of The Art*, other part describing our *Proposal* and a final part with the *Conclusions*. The Figure 1.3 shows the structure and chapters.

Introduction	Ch 1	Introduction
State of the Art	Ch 2	Background
	Ch 3	Related Work
Our Approach	Ch 4	Modeling Features of multiple Domains and Standards
	Ch 5	Weaving Feature Models representing multiple Domains and Standards
	Ch 6	Deriving Configuration Systems from Models representing multiple Domains and Standards
	Ch 7	Experimental Implementation
	Ch 8	Case Study: Modeling Configuration Options of Electrical Transformers
Conclusions	Ch 9	Conclusions

Figure 1.3: Structure of the Document

In the *State of the Art*, Chapter 2 presents a background on Feature-based Configuration. It describes the Feature Models, the Automated Analysis and Merging of these types of models, and the use of these models as a foundation for the Configuration Systems. Chapter 3 describes related work on Feature Models and Feature-based Configuration Systems involving multiple domains and standards.

Our *Approach* is presented in five chapters. Chapter 4 discusses our approach to model multiple domains and standards, Chapter 5 discusses the automated combination of these models, Chapter 6 the automated derivation to produce configuration systems and the strategy to process user decisions. Chapter 7 the experimental implementation that supports the approach, and Chapter 8 presents a case study on modeling configuration options for Electrical Transformers.

The final part presents the *Conclusions* of this thesis. Chapter 9 includes a discussion of our work and considers future work.

Chapter 2

Background

Configuration Systems can be found in a large number of industries in different forms. For instance, in 2015, researchers of the *Configurator Database Project* [23] reviewed 1050 different web-based configuration systems in 16 different industries. Companies selling products such as musical instruments, footwear and food offer systems where users may select features and components, and a specialized software assist them to find and buy the products most suitable for their interests.

Our work focuses on using *Feature-based Configuration*, i.e., configuration systems where users decide on the features to be included in a product. In such systems, the features that can be selected by a user are usually represented using *Feature Models*, a compact representation of the configuration options and the constraints that rule which combinations are valid. This section presents an overview of the concepts related to Feature-based configuration, the Feature Models and the automated processing of Feature Models and Configurations.

2.1 Configuration Systems

A **Configuration System (or Configurator)** is a software application for specifying products matching the individual needs and expectations of the customers [23][47].

A Configuration System assists users in the selection of the features and components for a product [15]. The system presents the user with the set of options that she can decide, and infers necessary decisions based on her decisions [77]. It relies on a series of pre-defined rules or constraints to determine which features can be included in a product at the same time and which not. It uses these constraints to discourage the user from making inconsistent decisions, and to suggest decisions that are necessary to satisfy them.

According to Sabin et al. [117], there are three types of configuration systems according to the technique they use to represent and reason about the configuration options and constraints: (1) *rule-based configurators* that validate selections using *if-then* style rules, (2) *case-based configurators* that use

the user selections to find and adapt existing products based on their similarity, and (3) *model-based configurators* that use model-based representations to describe the configuration rules and reasoner libraries (e.g., CSP, BDD or SAT solvers) to validate user selections.

In contrast to other types of software, creating a *Configuration Knowledge* representation is an essential part of the development of Configuration Systems[47]. These models are not used not only as a specification of requirements but also as an internal structure that support the automated analysis [47][137]. Engineers and developers typically use libraries and solvers that may analyse automatically user decisions using that knowledge representation as an input.

2.2 Feature-based Configuration Systems

A **Feature-based Configuration System** is a model-based configuration system that uses (1) *Feature Models* to represent configuration rules, and (2) *Automated Analysis on Feature Models and Configurations* to reason on user decisions [64]. In such systems, the users do not specify product parts or components but product features. A user can specify expected properties and attributes such as the final color or size of the product. Usually, at the end of the process, the system take the features selected by the user to find the corresponding product, calculate a Bill of Materials or provide the input for an automated process that derives the product.

One of the first steps when developing a *Configuration System* is to collect and organize the required domain knowledge and build the corresponding models [47]. Engineers and developers must identify the different stakeholders and sources of information to acquire and model that knowledge. We say that a **Configuration involves multiple domains** when the information regarding the options and constraints belongs to multiple domains and diverse stakeholders must be consulted.

Once the engineers capture and model the configuration knowledge, they must check and test those models to ensure their quality [47]. Many researchers have proposed automated techniques to analyse and detect errors in Feature Models [19]. The **Automated Analysis of Feature Models** are the automated techniques aimed to detect errors and extract information from Feature Models before engineers create the configuration system and users configure products [19].

Feature-based Configuration systems are usually built using libraries and solvers such as FaMa [20], Familiar [7] and SPLOT [93] that recognize the feature models and the constraints therein. These tools can take the selections of a user (i.e., a configuration) and a feature model and determine if the former satisfies the constraints in the latter. The **Automated Analysis of Feature Models and Configurations** aims to assist users and detect conflicts among the features selected by a user and the options and constraints defined in a Feature Model.

2.3 Feature Models

Feature Models were proposed by Kang et al. [79] to specify the similarities and differences among the members of a family of products. They are widely used in Software Product Lines to analyse the variations in a family of products as well as the options that can be used to configure each product.

2.3.1 Concrete Syntax

A **Feature Model** represents the features of a product and their dependencies [118] using a hierarchical structure: a tree with a root feature known as the *concept* (i.e., the product to configure), a set of *child features* that form the branches and leafs, and a set of *feature groups* and *feature relationships* representing configuration constraints.

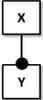
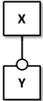
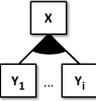
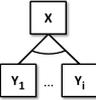
Element	Graphic	Description
root		Concept or Product to configure. It must be selected.
mandatory		if X is selected, Y must be selected
optional		if X is selected, Y can be selected (or not)
or group		if X is selected, one or more of Y_1, \dots, Y_n must be selected (at least one of them)
alternative		if X is selected, one and only one of Y_1, \dots, Y_n must be selected
requires		if X is selected, Y must be selected
excludes		if X is selected, Y must be not selected

Table 2.1: Feature Model elements

Table 2.1 shows the elements that a feature model may comprise. Basically, in a feature model:

- the *Root* feature represents the *concept* depicted in the model,

- *Mandatory features* represent commonalities, features that must be selected when the parent feature is already selected,
- *Optional features* capturing variations, features that may be selected or not when the parent is selected,
- *Or (inclusive-or) groups*, represent a set of features where one or more features can be selected, and
- *Alternative (exclusive-or) groups*, are groups where just one of the features can be selected.

Feature relationships are:

- *requires* relationships that indicates that one feature must be selected when the other is selected, and
- *excludes* representing that two features cannot be selected at the same time.

Figure 2.1 shows an example feature model for Cellular Phones. The feature model has the root feature of *Cellular Phone* which has three mandatory subfeatures: *LCD*, *Input Device* and *Battery* and one optional subfeature of *External Memory*. In turn, an *LCD* can be *Normal* or *Touch Screen* but not both. The *Input Device* can be a *Keypad*, a *Stylus*, or both. There is a *excludes* relationships indicating that when a cell phone includes a *Stylus*, cannot include a *Normal LCD* (it must include a *Touch Screen*). Finally, the *Battery* can be of *Small Size* or *Large Size*. There is also a *requires* relationship denoting that the use of a *Touch Screen* requires the inclusion of a *Large Size Battery*.

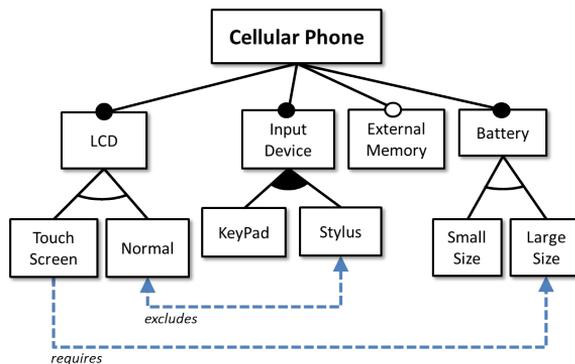


Figure 2.1: Example Feature Model, adapted from [83]

2.3.2 Semantics

The semantics of a Feature model is given by the corresponding set of *Feature Configurations*.

A **Feature Configuration (or Configuration)** is a set of the features of a FM. A configuration is *valid* if the set of features satisfies the constraints

defined in the model. For instance, considering the above feature model, a valid configuration for a *Cellular Phone* is $C = \{ CellularPhone, LCD, Normal, InputDevice, Keypad, Battery, SmallSize \}$

The **Feature Model Semantics** can be represented using a set of configurations or a Feature Matrix. For instance, considering the feature model fm depicted in the Figure 2.1, the semantics $\llbracket fm \rrbracket$ can be represented using the matrix in the Table 2.2. In that matrix, all the valid configurations are represented as a row where the selected features are marked with a check mark (\checkmark).

	Cellular Phone	LCD	Touch Screen	Normal	Input Device	Keypad	Stylus	External Memory	Battery	Small Size	Large Size
P1	\checkmark	\checkmark	\checkmark		\checkmark	\checkmark			\checkmark		\checkmark
P2	\checkmark	\checkmark	\checkmark		\checkmark		\checkmark		\checkmark		\checkmark
P3	\checkmark	\checkmark	\checkmark		\checkmark	\checkmark	\checkmark		\checkmark		\checkmark
P4	\checkmark	\checkmark	\checkmark		\checkmark	\checkmark		\checkmark	\checkmark		\checkmark
P5	\checkmark	\checkmark	\checkmark		\checkmark		\checkmark	\checkmark	\checkmark		\checkmark
P6	\checkmark	\checkmark	\checkmark		\checkmark	\checkmark	\checkmark	\checkmark	\checkmark		\checkmark
P7	\checkmark	\checkmark		\checkmark	\checkmark	\checkmark			\checkmark	\checkmark	
P8	\checkmark	\checkmark		\checkmark	\checkmark	\checkmark			\checkmark		\checkmark
P9	\checkmark	\checkmark		\checkmark	\checkmark	\checkmark		\checkmark	\checkmark	\checkmark	
P10	\checkmark	\checkmark		\checkmark	\checkmark	\checkmark		\checkmark	\checkmark		\checkmark

Table 2.2: Feature Matrix representing Semantics for the Example Feature Model

Note that not all the combinations of features are valid configurations. For instance, if a configuration includes *LCD Normal* and *LCD Touch Screen*, it is invalid because the model states that the user must select one of them but not both. In addition, if a configuration does not include neither *Keypad* or *Stylus*, the configuration is invalid because the user must select an *Input Device* with at least one of *Keypad* or *Stylus*.

2.4 Automatic Processing of Feature Models

In real families of products, feature models may grow to hundreds or thousands of features [93][87]. Manipulating and detecting problems in these models becomes a hard task for humans. In the last decades, an increasing set of operations have been proposed to analyse feature models to detect problems and extract relevant information [19].

There are two types of operations for processing automatically Feature Models. On one hand, some researchers have focused on *Model Analysis Operations* aimed to answer questions that may be of use to the people creating or debugging a Feature Model [19]. On the other hand, other researchers have focused on *Model Processing Operations* that allow modelers to decompose, combine or simplify models according to their intended applications [9][2].

2.4.1 Model Analysis Operations

A **Feature-Model Analysis Operation** copes with the extraction of information from feature models[19]. A great deal of analysis operations have been proposed to determine if a feature model is valid or if it contains errors such as contradictory constraints. For instance, there is an operation to detect *dead features*, i.e., features that cannot be included in any product, and *core features* (a.k.a., *full mandatory features*), i.e., the set of features included in all the products. Additional operations haven been proposed to *explain* and *fix* the errors found.

Benavides et al. [19] present a very comprehensive catalog of analysis operations in their systematic review. Among these operations, we can mention:

- **Validating a Feature Model** (or detecting FM satisfiability), i.e., detecting if there is at least one valid configuration regarding the model. An invalid model is synonym of an over-constrained model from which no valid configuration can be defined.
- **Obtaining Core Features** (or obtaining the full-mandatory features), i.e., finding the features that appear in all the valid configurations. For instance, consider the Feature Model semantics presented in Table 2.2. The *LCD* is a core feature that appears in all the valid configurations.
- **Obtaining Dead Features**, i.e., finding the features that never appear in any valid configuration. Note that a feature model may not include any dead feature. For instance, considering the above example, all the features appear at least in one valid configuration in the Table 2.2.
- **Obtaining Free Features**, i.e., finding the features that are neither core or dead features. In the above example, the *Touch Screen* is a free feature because it may appear (or not) in a valid configuration.
- **Obtaining Explanations of invalid models or dead features**, i.e. providing explanations about why a feature model is invalid or a feature is dead.
- **Proposing Fixes to invalid models or dead features**, i.e. providing suggestions on how to modify a feature model to make it valid or to make a feature not dead.

2.4.2 Approaches to implement Analysis Operations

To implement operations that analyse feature models, many authors rely on formal semantics that allow straightforward translations of these models into well-established formalisms like propositional logics [21][54][88][118][97]. These

approaches implement the analysis operations by (1) translating the feature models into a formal representation, (2) analysing the alternative representation using solvers, and (3) presenting the results translating the solver response to the elements in the feature model.

Benavides et al. [19] made a systematic review of the diverse approaches that implement operations to analyse feature models. They identified four categories according to the reasoning technology they use. There are approaches (1) that translate Feature Models to *Propositional-logic* and use *Satisfiability (SAT)* or *Binary Decision Diagram (BDD)* solvers, (2) that translate the models to *Constraint-Satisfaction Problems* and use *Constraint Satisfaction Programming (CSP)* solvers, (3) that use *Description Logics*, and (4) others that implement *ad-hoc algorithms* for analysis. More recently, other authors have proposed new approaches that translate the models into *Optimization Problems* [142] and approaches that use technologies such as *Satisfiability Modulo Theory (SMT)* [95], Quantified Boolean Formula Solvers (QBF Solvers or QSAT) [82] and *Answer Set Programming (ASP)* solvers [96].

Implementing Analysis Operations using SAT solvers. For example, in order to analyse a Feature Model, it is possible to translate the features and constraints in the model into propositional formulae and use a SAT solver to perform the analysis.

A *Propositional Formula* is a logic expression defined over boolean variables, using operators such as conjunctions (i.e., AND, \wedge), disjunctions (OR, \vee), and negation (NOT, \neg). A *Satisfiability Problem (SAT)* is the problem of determining for a given formula whether there is a set of logical values (i.e., TRUE or FALSE) for the variables where the complete expression evaluates to TRUE. A problem is said to be *satisfiable* if exists at least one solution. A *Satisfiability solver (SAT solver)* is a software able to take an propositional formula, usually expressed as a CNF formula, and check whether it is satisfiable.

A *Encoding for Feature Models* describes how to translate a given feature model into a propositional formula. There are many proposals on how to perform this encoding [92][87]. In general, all these encoding have the following general form:

- each feature is encoded as a boolean variable,
- and each feature, group and relationship are encoded as a propositional sub-formula.

Table 2.3 summarizes how to encode each type of feature and relationship according to Mendonca et al. [92].

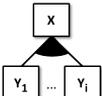
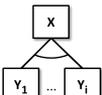
Element	Graphic	PL encoding
root		r (i.e., $r = true$)
mandatory		$x \Leftrightarrow y$
optional		$y \Rightarrow x$
or group		$x \Rightarrow (y_1 \vee y_2 \vee \dots \vee y_n)$ $(y_1 \Rightarrow x) \wedge (y_2 \Rightarrow x) \dots \wedge (y_n \Rightarrow x)$
alternative		$x \Rightarrow ((y_1 \wedge \neg y_2 \wedge \dots \wedge \neg y_n)$ $\wedge (\neg y_1 \wedge y_2 \wedge \dots \wedge \neg y_n)$ $\dots \wedge (\neg y_1 \wedge \neg y_2 \wedge \dots \wedge y_n))$ $(y_1 \Rightarrow x) \wedge (y_2 \Rightarrow x) \dots \wedge (y_n \Rightarrow x)$
requires		$x \Rightarrow y$
excludes		$(x \Rightarrow \neg y) \wedge (y \Rightarrow \neg x)$

Table 2.3: Feature Model encoding in Propositional Logic, based on Mendonca et al.[92] and Janota et al. [74][75]

For instance, consider the feature model for Cell Phones presented above in Figure 2.1. The corresponding propositional formula is the following:

```

 $\phi = CellularPhone$ 
 $\wedge CellularPhone \Leftrightarrow LCD$ 
 $\wedge LCD \Rightarrow ((TouchScreen \wedge \neg Normal) \vee (\neg TouchScreen \wedge Normal))$ 
 $\wedge (TouchScreen \Rightarrow LCD) \wedge (Normal \Rightarrow LCD)$ 
 $\wedge CellularPhone \Leftrightarrow InputDevice$ 
 $\wedge InputDevice \Rightarrow (KeyPad \vee Stylus)$ 
 $\wedge (KeyPad \Rightarrow InputDevice) \wedge (Stylus \Rightarrow InputDevice)$ 
 $\wedge ExternalMemory \Rightarrow CellularPhone$ 
 $\wedge CellularPhone \Leftrightarrow Battery$ 
 $\wedge Battery \Rightarrow ((Small \wedge \neg Large) \vee (\neg Small \wedge Large))$ 
 $\wedge (Small \Rightarrow Battery) \wedge (Large \Rightarrow Battery)$ 

```

Once we have translated a Feature Model fm into a corresponding formula ϕ , it is possible to determine if the model is valid by determining if the formula is satisfiable. We can use a SAT solver to find a solution. If a solution exists, the feature model has (at least) one valid configuration. We write $SAT(\phi)$ to denote a invocation to a SAT solver. That function will return TRUE if the formula ϕ is satisfiable or FALSE otherwise.

In addition to determine if a feature model is valid, we can use the SAT solver to perform other analysis operations. For instance, we can use a solver to determine if a feature is dead or if it is a core feature. Algorithm 2.1 describes how to obtain the set of core features. Considering that core features are those that are included in all the valid configuration, we know that a feature is core if a solver cannot find a configuration that do not include that feature.

Algorithm 2.1 Obtaining the set of core features

```

1: procedure CALCULATECOREFEATURES( $fm$ )
2:    $\mathcal{F}_{core} \leftarrow \{\}$ 
3:   for all  $f \in \mathcal{F}_{fm}$  do                                ▷ for each feature in the model
4:      $\phi = encode(fm) \wedge \neg f$                           ▷ force the feature  $f$  to be disabled
5:     if  $\neg SAT(\phi)$  then                                ▷ if the resulting  $\phi$  is not satisfiable
6:        $\mathcal{F}_{core} \leftarrow \mathcal{F}_{core} \cup \{f\}$           ▷ then  $f$  is a core feature
7:     end if
8:   end for
9:   return  $\mathcal{F}_{core}$ 
10: end procedure

```

There is a large number of analysis operations that can be performed in the same way [18]. A more complete description of the corresponding algorithms exists in the works of Benavides [18] and Henneberg [61].

2.4.3 Model Processing Operations

Besides the operations to analyse Feature Models, there are other operations aimed to process and alter the models.

A **Feature Model Processing Operation** is an operation that takes a feature model or other type of inputs, and yields to a new feature model which is more simple, represent a different set of products or exhibit a different value for some property. For instance, there are operations to reverse engineer [125], specialize [36][37], slice [6] and merge feature models [9][2].

The Processing Operations that modify a feature model (i.e., that edit a feature model) may affect the set of products that the model represents. For instance, it is possible that a configuration valid against a feature model results invalid against the modified model. Thum et al.[135] classifies these operations according to the resulting modification to the set of valid configurations of the feature model (i.e., modifications to the semantics of the model):

- **Refactoring operations** are those operations that do not modify the semantics. For instance, a modification in the structure of the feature model or its constraints where the resulting set of valid configurations is the same set of valid configurations of the original.
- **Generalization operations**, operations where the resulting set of valid configuration is a superset of the original.
- **Specialization operations**, operations where the resulting set of valid configuration is a subset of the original. And
- **Arbitrary edit operations**, when the resulting set of valid configuration is neither the same, a superset nor a subset of the valid configurations of the original.

Some of the Feature Model Processing Operations are relevant to create Configuration Systems. For instance, Czarnecki et al. [37] proposed operations to specialize a Feature Model to select or deselect a specific feature. These operations may be used during a staged configuration, after a user makes a decision, to obtain a feature model representing only the options she can select afterwards.

Among the operations to process Feature Models, we can mention:

- **Reverse Engineering a Feature Model**, i.e., given a set of valid configurations or a propositional formulae, calculating one or more feature models that represent that set.
- **Specializing a Feature Model**, i.e., given a feature model, calculating another feature model which set of valid configurations is a subset of the original. The subset of features may be determined by some user provided criterion or decision [37].
- **Slicing a Feature Model**, i.e., given a feature model and a *slicing criterion*, calculating a new feature model that includes a subset of the features and constraints of the original [6][86].
- **Merging two Feature Model**, i.e., given two feature models, calculating a new feature model that combines their features and constraints. How these features are combined is determined by a *merging semantics*. For instance, it is possible to merge the models in a way that the resulting that the set of valid configurations of the new feature model is the *union* or the *intersection* of the valid configuration of the constituent models [2].

2.4.4 Approaches to implement Processing Operations

Model Processing Operations can be implemented in different ways [9][11]. On one hand, There are *syntactic-based approaches* that take the inputs and produce a new the feature model by applying transformation rules or by adding or removing features and relationships to or from an existing model. On the other hand, there are *semantic-based approaches* that compute a logical representation of the model (e.g., in propositional logic), operates on this representation, and compute a new feature model from the results of the operations [6][2].

Operations based Syntactic Modifications exploit the graph-oriented structure of the model to introduce or remove features and constraints, change the type of features or feature groups, or altering the hierarchy and the feature relationships. [6]. For instance, a syntactic-based operator to slice a Feature Model may simply extract a sub-tree of the model taking an arbitrary feature as the root of the new model and removing all the features that are not a descendant of it. However, operators implemented in such way usually have preconditions in order to assure their semantics. For instance, the merge operators proposed by Segura et al. [122], Van den Broek et al. [143][144] and Aydin et al. [16] require that the feature models to combine are *parent-compatible*, i.e., all the features included in both models must have the same parent in both models. The algorithms they propose should be adapted to handle propositional constraints and deal with different hierarchies [2].

Operations based on processing the semantic representation modify the formal representation of the feature models and synthesize a resulting feature model from the modified representation. These approaches exploit existing techniques to create feature models from a set of valid configurations [124] or a propositional formulae [38][125]. For instance, Acher et al. [2] implement operations to merge and slice feature models by combining and processing the encoding of feature models in propositional logic and synthesizing a new model from the resulting propositional formula.

Acher et al. [2][11] point out that the processing based on the semantic representation may achieve, by construction, the semantics of the operations. That means that the operations may work even with feature models that cannot be processed with syntactic modifications. For instance, operations based on logic techniques can be used to merge not parent-compatible models while operations based on model modifications can not.

2.5 Automatic Processing of Interactive Configurations

Finally, there are other operations aimed to analyse and process feature configurations. These operations are relevant for creating Configuration Systems where a user may decide on features from a model. Those operations can be used by the software to validate the user decisions, explain the conflicts found, and propose fixes to the configuration. Additional operations may be used to auto-complete the configuration or to optimize the decisions based on some criterion.

2.5.1 Operations for Interactive Configuration

There are many operations that are key for the implementation of Configuration Systems. For instance, it is important to determine if a configuration is complete or is partial. A configuration is complete if there is a decision for all the features. It is partial if the user has not decided on some of the features. A system may use an operation to auto-complete a partial configuration by

deciding on the features not decided by the user. We say the the system optimize a partial configuration when the system decide on these features based on some criterion.

Among the operations for Interactive Configuration, we can mention:

- **Validating a Configuration**, i.e., detecting if a configuration or a state of an interactive configuration is valid regarding the features and constraints defined in a Feature Model.
- **Validating a Partial Configuration**, i.e., detecting if a set of features does not include any contradiction regarding the Feature Model.
- **Propagating User Decisions**, i.e., taking user decisions and selecting and removing automatically the features that depend on them.
- **Auto-completing a Partial Configuration**, i.e., selecting and removing automatically undecided features in order to turn a partial configuration into a complete valid configuration. This is equivalent to find a valid complete configuration that is a superset of a given partial configuration.
- **Optimizing a Partial Configuration**, i.e., auto-completing a partial configuration in a way that one or more properties or functions on the selected features are maximized or minimized.
- **Recommending a Configuration**, i.e., finding valid configurations that are similar to a given partial configuration. Usually, these recommendations try to find the closest configurations by using a predefined *distance function* that may work even with invalid partial configurations.
- **Obtaining Explanations of Configuration Conflicts**, i.e., providing explanations about why a configuration is invalid.
- **Proposing Fixes to Configuration Conflicts**, i.e., providing suggestions on how to modify the configuration, by selecting or removing features, to make it valid.

2.5.2 Approaches to implement Interactive Configuration

The Literature review from Benavides et al. [19] surveyed techniques to implement operations on feature models and configurations. They found that the operations for Interactive Configurations have been implemented using *Propositional Logic* and *Constraint Programming* mainly. Usually, the Configuration Systems translate the Feature Model and the Configuration to a propositional formula or a Constraint-Satisfaction Problem and use a specialized solver to perform the operations.

Processing user decisions Almost all the approaches for processing user decisions during Interactive Configurations are based on seminal works from Batory [17] and Czarnecki [36][36]. First, Batory [17], proposed the use of a *Logic Truth Maintenance System (LTMS)*, in order to validate and propagate the users decisions. Then, Mendonca [90] proposed a hybrid *Reasoning System for Feature Models (FMRS)* combining techniques for propagating user decisions with SAT and CSP solvers. Janota [75] proposed a complete reasoning system based on SAT solvers. And Finally,

more recently, other authors have proposed techniques based on *CSP* [21], *QSAT* [82], *SMT* [95] and *Answer Set Programming* [96].

Operations to determine why a configuration is invalid and, optionally, obtain suggestions on how to fix it may rely on *explanations* or *diagnosis*.

Explanations Approaches based on explanations look for the cause of the conflict. Considering that feature models and configurations can be translated to boolean formulae and constraints, it is possible to use many techniques that determine which operators or constraints are conflicting. *QuickXplain* [78] can be used to determine a minimal (irreducible) set of faulty constraints with a CSP solver. Options to find *Minimal Unsatisfiable Sets (MUS)* can be used in modern SAT solvers [77]. A configuration can be repaired by excluding all the faulty constraints [101]. The result is a partial valid configuration that a user can complete. Note that retracting these decisions may result in deselecting features that the user wants but have conflicts with other decisions.

Diagnoses and Fixes Approaches for diagnosis look for an alternative valid configuration that maximize the user preferences. These approaches use as input not only the configuration but also which subset of that configuration the user wants to keep. They usually look first for explanations and perform additional processing to determine an alternative configuration. While some approaches are based on *FastDiag* [49], an algorithm proposed by Felfernig et al. for CSP; others such as the proposed by White et al. [147] and Xion et al. [148] are based on the Reiter's Theory of Diagnosis [115][55] or on the *CURE* strategy proposed by White et al. [146] specially for Feature Models. Some of these approaches determine a set of fixes and let the user to decide which use to repair the configuration.

Configuration Systems can implement the operations in different ways depending on their intended use [100]. Usually, they are *backtrack-free* systems designed to *prevent conflicts*. When a user makes a decision, the decision is propagated to select and disable other features and prevent that the user may cause a conflict by selecting the wrong feature [77]. More recently, some Configuration Systems have been designed to *tolerate conflicts*. These systems allow users to pick features even if they cause conflict with other decisions. Additional processes are performed when the user decides to determine configurations without conflicts that maximize the user selections [100][101]. Other systems allow users to define *selections and preferences* as hard and soft constraints. In such systems, conflicts are not permitted among selections but are tolerated among preferences. An automated process may look for valid configurations that includes the selections and maximizes the preferences.

2.6 Summary

Recently, many proposals use *Feature Models* as a foundation for engineering configuration systems. The Feature Models are used to specify a hierarchy of the features in a product and define the set of legal combinations of these features. Engineers can take these Feature Models to define which options and constraints must be considered to configure a product. In addition, there are many *analysis operations* that can be used to extract information of the models and detect potential errors in the configuration systems before building them. Other *manipulating operations* can be used to combine or slice the models during the specification of the systems. Furthermore, there are *algorithms and operations for interactive configuration* that can be used at runtime to process the user decisions and determine which features must be automatically selected during a configuration process.

In this chapter we sketched the major concepts of feature-based configuration systems and the syntax and semantics of feature models. We also highlighted the existence of automated techniques to analyse and manipulate these models such as the operations to validate a model and to merge multiple models into a new one. Finally, we also described automated operations that can be used during the interactive configuration of a product such as propagating user decisions and auto-completing configurations based on feature models.

Chapter 3

Related Work

There is a large number of proposals related to Feature Models and Feature-based Configuration Systems. Several authors have systematically reviewed the literature on using Feature Models to represent variability in multiple domains [67], on analysing Feature Models [19], on composing and decomposing Feature Models [9], and on creating Feature-based Configuration Systems [1][117].

In this chapter, we introduce the state-of-the-art in approaches to create Configuration Systems for complex products in three areas: (1) the modeling of complex products considering multiple domains and standards, (2) the composition of feature models representing different concerns, and (3) the derivation of configuration systems from feature models, namely the activities that are the subject of our research.

3.1 Modeling multiple Domains and Standards

Modeling configuration options for complex products have been of special interest in industries such as the automotive [132][145] and consumer electronics [57]. In these industries, realistic models tend to include hundreds and even thousands of features and, therefore, tend to be hard to be created and debugged by humans. Many authors have propose different strategies to create and structure the feature models trying to facilitate the process [22][67][140][44][105].

Approaches aimed to create feature models for complex products can be classified into three categories: (1) Proposals that organize all the product features in a single monolithic feature model, (2) Proposals that use a single model but define views for the diverse stakeholders or concerns, and (3) Proposals that use multiple feature models. The following sections will give a short overview of representative proposals in each category describing how they support the modeling of multiple domains and standards.

3.1.1 Proposals using a Monolithic Feature Model

Initial proposals aimed to define a single model to represent all the features of the products. They organized the features using different strategies to facilitate the creation and maintenance of the models.

Feature Categories In the *Feature-Oriented Reuse Method (FORM)*, Kang et al. [80] organize the features in a model according to a set of categories. Each feature belongs to a unique category. For instance, the examples included in FORM classify features into four categories: capabilities, domain technologies, implementation techniques, and operating environments. To create a Configuration System, the products are modeled by specifying the features in each one of these categories. Although the authors do not provide a similar example, this approach can be used to structure the model considering each technical domain as a category in the model. To model standards, modelers must choose to create a category for all standards or model the standard scattered across all the categories.

Feature Levels Buhne et al. [29] and Svahnberg et al. [130] organize the features in levels. Features in the higher levels are further specified in features in the lower levels. In the Svahnberg et al.'s examples, the features at the top level are user requirements. These features are decomposed into features of the architecture, then into features of the software design, later into features of the code, and finally into features of the running system. Buhne et al.'s examples define a level for the Vehicle at the top, a level for the systems, another for the functions and a lower level for the software features. During a configuration process, if a user selects a high-level feature, she must decide on the corresponding low-level features. This approach does not consider subsystems nor technical domains as a way to structure the model. Standards are not considered either. The final models include the standards and the corresponding constraints scattered across all the levels.

Summary. There are many proposals to structure the features of complex products in a single model. They may use feature categories or levels to organize the models according to technical domains or a decomposing hierarchy. Table 3.1 presents a comparison. Although they aim to reduce the complexity of models involving multiple subsystems or sequence of decisions, they do not consider standards explicitly. Standards may be modeled as a category where each feature represents a standard and has a lot of cross-cutting concerns to the other categories, or as a set of features and constraints scattered across all the levels.

	Modeling of domains	Modeling of standards	Comments
Feature Categories. Kang et al. [80]	<ul style="list-style-type: none"> • all in a single feature model. • there are subtrees representing categories or subsystems. 	<ul style="list-style-type: none"> • as features and constraints in the same model. 	<ul style="list-style-type: none"> • standards end scattered along all the model. • Hard to create and maintain.
Feature Levels. Buhne et al.[29] Svahnberg et al. [130]	<ul style="list-style-type: none"> • all in a single feature model. • there are subtrees representing concerns or subjects to decide. 	<ul style="list-style-type: none"> • as features and constraints in the same model. 	<ul style="list-style-type: none"> • standards end scattered along all the model. • Hard to create and maintain.

Table 3.1: Related work on Modeling Multiple Domains and Standards using a Monolithic Feature model

3.1.2 Proposals using Views on a Feature Model

In Feature Model Views, each view includes a subset of the features of the model. Usually it includes the features that are of concern of a group of stakeholders. They are used primarily in a Configuration Process allowing each stakeholder to configure parts of the feature model in his own view until all variability is bound. Views can be used during modeling to reduce the complexity and present stakeholders only the features where they are experts.

Representative proposals using views on Feature Models are:

Multi-Criteria Product Lines: Buhne et al [29] model vehicle product lines in DaimlerChrysler using multiple levels on a single model. The feature model comprises the features for all the vehicles. To facilitate the understanding and maintenance, they create different views according to different criteria such as geographical location, cost, or the vehicle line. These views are used to facilitate the review of the models. For instance, they can be used to display only the features that are of the concern of a stakeholder. However, because standards define rules for features in all the concerns, a view representing a standard may be not very useful because it will include (1) features of multiple domains and (2) constraints related to these features that do not belong to the standard. Other drawbacks of this approach is the need of using the complete feature model for editing the model and the absence of supporting tools [105].

Multiple Perspectives on Feature Models: Schroeter et al. [120][119] introduced *feature cluster models* that comprise a feature model, a view and a mapping between both. In their approach, each view is a filtered feature model. One of these views is *FM-consistent* if each configuration that is valid in the view is also a valid partial configuration of the original feature model. They defined a *view composition* operator to aggregate multiple views of the same feature model into perspectives (FM-consistent view aggregations) for different stakeholders.

View-based Configuration Processes: Hubaux et al. [65][67] proposed algorithms to define and extract views (sub-models) from an existing feature model. They also define some operations to simplify the views and validate if combining the configurations for a set of views results into a complete configuration of the original model. This proposal is used to support configuration processes with multiple stakeholders. Basically, they use the views to present each stakeholder the features that are of her concern.

Summary. Several proposals use multiple views to model complex products. Each view usually comprises the subset of features that are of the concern of one of the stakeholders. There is a single feature model that represent all the products and their features, and the set of views are excerpts of that model used to display, review or configure the models. Table 3.2 presents a summary.

These approaches do not consider standards explicitly. A standard may be defined as an additional view that comprises features and constraints scattered across the model.

	Modeling of domains	Modeling of standards	Comments
Multi-Criteria Product Lines. Buhne et al. [29]	<ul style="list-style-type: none"> • in a single feature model. • a stakeholder can review the features of a single domain. 	<ul style="list-style-type: none"> • as features and constraints in the same model. • A stakeholder can review the features of a standard. 	<ul style="list-style-type: none"> • operations are performed on a model integrating all the views. • standards end scattered along all the model.
Multiple Perspectives on Feature Models Schroeter et al. [119][120]	<ul style="list-style-type: none"> • in a feature model and in a view. • a mapping among the models and their views must be defined. 	<ul style="list-style-type: none"> • as features and constraints in the models or the views. 	<ul style="list-style-type: none"> • domains and views are independent models and may be inconsistent. • standards end scattered along all the model.
View-based Configuration Processes Hubaux et al. [65][67]	<ul style="list-style-type: none"> • in a single feature model. • a stakeholder can review and configure the features of a single domain. 	<ul style="list-style-type: none"> • as features and constraints in the same model. 	<ul style="list-style-type: none"> • operations are performed on a model integrating all the views. • standards end scattered along all the model.

Table 3.2: Related work on Modeling Multiple Domains and Standards using Views on Feature models

3.1.3 Proposals using multiple Feature Models

Other proposals represent complex products using multiple feature models. Each feature model represent a technical domain or a concern. Relationships among the feature models are used to specify dependencies between features in different domains and concerns. Additional processing is required to combine the feature models to represent a specific product line.

Multi-level feature trees: Reiser et al. [113][112] model vehicles in Daimler-Chrysler using multiple feature models. Each model represent a technical domain or a decomposition level of the product. They use *Configuration Links* [111][110] between the feature models to describe how a *configuration decision* on a *source feature model* affect the configuration on a *target feature model*. Using these links, it is possible to specify how a set of decisions in one side must affect the decisions in the other.

Context Variability Modeling: Hartmann et al. [59] consider the modeling of multiple product lines using a set of feature models. Which models are part of a specific product line depends on the intended scenario or context. In their approach, there is a feature model for the context variability, a feature model for each Multi-product line, a model for each provider of product components, and a set of relationships to features in the other feature models that define which models and features in those models must be considered for a specific scenario. The authors describe multiple scenarios where the models are merged in different ways depending on the task to perform, e.g., scenarios where the models for many products are merged to a MPL-feature model or where the models of multiple suppliers are merged together. In addition, they describe the merging operators but do not include a formal specification of the algorithms.

Feature Models for System Composition: Friess et al. [52] aim to support modular embedded systems where each module (i.e., component) has its own variability. In their approach, a set of composition rules describing how the variability in the modules must be integrated to the system. They provide a Composition Checker that takes a configuration for the system and the related modules, creates a new composite feature model and validates the validity of the selected options. Their approach merges the feature models as an intermediary step in the validation process.

Summary. There are other proposals using multiple feature models to represent complex products. These models usually represent subsystems or components that have their own variability independently of the product variability. These proposals usually aim to model products that compose or assemble components and subsystems created by third parties. As shown in Table 3.3, these approaches do not consider standards and regulations explicitly. Considering that standards may impose constraints to different elements and subsystems

of the product, they must be modeled using features and constraints scattered across all the set of the models.

	Modeling of domains	Modeling of standards	Comments
Multi-level feature trees. Reiser et al. [113][112]	<ul style="list-style-type: none"> as independent feature models. 	<ul style="list-style-type: none"> as features and constraints in the feature models 	<ul style="list-style-type: none"> standards may end scattered along several models. modelers must review and update the models to incorporate a standard.
Context Variability Modeling. Hartmann et al. [59]	<ul style="list-style-type: none"> as independent feature models. 	<ul style="list-style-type: none"> the standards are modeled as features in a <i>Context Feature Model</i> and relationships to features in the other models. 	<ul style="list-style-type: none"> standards are represented as relationships to features in all the models. domains must be modeled first than standards to create the relationships. hard to create and maintain.
Feature Models for System Composition. Friess et al. [52]	<ul style="list-style-type: none"> as independent feature models. 	<ul style="list-style-type: none"> as features and constraints in the feature models 	<ul style="list-style-type: none"> standards may end scattered along several models. hard to create and maintain.

Table 3.3: Related work on Modeling Multiple Domains and Standards using multiple Feature models

3.1.4 Proposals using multiple Types of Models

Other approaches represent complex products by combining Feature Models with other types of Models. Feature Models may be used to represent a technical domain while other types of models may represent other domains. For instance, Feature Models may represent the user-selectable features of a product, while other types of models such as class diagrams or component models may represent the variability of the internal components or the constraints indicating which of these components can be combined or not. Similarly to the approaches using multiple feature models, additional processing is required to combine these models to represent or analyse a specific product line.

Heterogeneous Variability Modeling Dhungana et al. provide support for modeling complex software products using diverse types of variability models [42][43]. Galindo et al. [53] provide additional support for distributed configuration of products using these heterogeneous models. In addition to Feature Models, they support Orthogonal Variability Models [94][108] and Decision Models [40][39] to represent the variability of the different domains of multiple product lines. The authors describe how these models can be related and combined to analyze the variability or to configure products. They define a set of cross-type relationships (such as “extends model”, “extends feature”, and “provides features” relationships) to represent equivalences and relationships among the models and features. However, the authors do not consider the use of different models for standards.

Integration to Goal Models Many authors have proposed integrations of Feature Models to Goal Models. Asadi et al. [14], Silva et al. [127] and Clotet et al. [34] have presented approaches that relates features in Feature Models to goals and requirements in i^* models. They provide automated processes that may take goals and requirements defined by stakeholders and create an initial product configuration by selecting the corresponding options of the feature model. A standard may be included in an i^* model as a client requirement. Which features must be enforced or prohibited when a standard is required must be defined as relationships between the goal and feature models. It is possible that models representing product lines that must support multiple standards end with a large number of relationships scattered along these models.

Integration to Architectural Models Other authors have proposed integrations of Feature Models to Architectural Models. Janota et al. [76] and Acher et al. [4][5] presented formal approaches to integrate Feature Models and Component Models. They provide means to introduce the constraints defined in the component models into the corresponding feature models. In addition, they provide means to extract and evolve feature models from the source code of applications based on plug-ins. Although these approaches may introduce new constraints to the feature models, these constraints correspond to internally defined design constraints but not to externally defined constraints such as standards and regulations.

Summary. There are some other approaches that integrate feature models to other types of models. Table 3.4 presents a summary. They relate the variability represented in feature models to the variability and the constraints defined in other types of models. These approaches aim to support different representations of the domains and components of a product line. While some approaches consider different types of models that represent the variability of the products, other proposals relate the variability to other design artifacts to

include or compare the constraints defined in these models. However, these approaches do not consider standards explicitly. Considering that standards may impose constraints to multiple elements and subsystems of the product, standards must be considered in all the different type of models. It is possible that a single standard must be modeled using multiple elements scattered across all these models.

	Modeling of domains	Modeling of standards	Comments
Heterogeneous Variability Modeling. Dhungana et al. [42][43][53]	<ul style="list-style-type: none"> the domains can be modeled using Feature Models, Orthogonal Variability Models or Decisions Models. 	<ul style="list-style-type: none"> as features, decisions and constraints in the diverse models. 	<ul style="list-style-type: none"> standards may end scattered along several models. hard to create and maintain.
Integration to Goal Models Asadi et al. [14], Silva et al. [127] and Clotet et al. [34]	<ul style="list-style-type: none"> all the domains in a single feature model 	<ul style="list-style-type: none"> as features and constraints in the goal model or the feature model 	<ul style="list-style-type: none"> standards must be modeled as relationships scattered along the models. hard to create and maintain.
Integration to Architectural Models Janota et al. [76] and Acher et al. [4][5]	<ul style="list-style-type: none"> all the domains in a single feature model 	<ul style="list-style-type: none"> probably, as features and constraints in the feature model 	<ul style="list-style-type: none"> It is focused on including design constraints into the feature models. it is not well suited to externally defined constraints.

Table 3.4: Related work on Modeling Multiple Domains and Standards using multiple types of models

3.1.5 Discussion

Existing proposals do not support standards explicitly. The standards may be modeled as feature categories, feature levels, an independent view or an independent feature model depending on the approach. However, the constraints defined by the standards end up scattered across all the models, diffculting their creation and maintenance.

3.2 Composing multiple Feature Models

Approaches to merge feature models can be classified into: (1) Proposals that describe how to merge feature models using transformation rules on the structure of the model, and (2) proposals that use semantics instead of structure to merge models.

3.2.1 Structure-based Merging of Feature Models

There are many authors proposing rules to merge parent-compatible feature models:

Feature Model Refactorings: Alves et al. [12] defined a set of rules and operations, that they named *refactorings*, to modify feature models. These operations included operators such as one to insert a feature into a model, convert a feature group from one type to the other and merge feature groups.

Feature Model Merge by Transformation: Segura et al. [122], inspired on the operations defined by Alves et al.[12], defined a set of ACG transformation rules to merge feature models. They defined rules (30 in total) to merge different types of features, different types of feature groups and different types of *requires* and *excludes* relationships. The resulting model of merging two models represent, as a minimum, the union of the products of both models. Although it is not mentioned in their work, these transformations rules assume that the models to merge are parent-compatible and include only cross-tree constraints represented as relationships.

Viewpoint Oriented Variability Modelling: Mannion et al. [89] proposed a method to use different feature models to represent the diverse viewpoints of the stakeholders of a product. In contrast to Segura et al. [122], they do not define rules to merge the features but rules to resolve inconsistencies between the input feature models. They determined a set of *conflict resolution rules* (24 in total), each one describing the inconsistency that may exist and their resolution. Some of these rules (5) require user intervention and cannot be solved automatically. Their approach does not aim to implement a single coherent model as a result of merging, instead they want delay inconsistency resolutions until the rationales which cause distinctive choices are better understood.

ViewPoint Integration: Niu et al.[99] extend the work of Mannion et al. [89] to take advantage of a lattice ordering to support late binding of variability and stakeholder traceability.

Parent-Compatible Feature Model Merge and Intersection: Van den Broek et al. [144][143] proposed methods not based on rules to merge and intersect feature models. Their approach starts by taking two input

parent-compatible feature models with excludes and requires relationships, removing their dead features and transforming them into generalised feature trees. Then, it determines requires and excludes relationships among all the features in both models and, finally, uses those relationships to determine the types for each feature and the cross constraints that must be included.

Feature Model Merge by Local Rules: Aydin et al. [16] defined a *merging by conformance* process that is similar to the method proposed by Broek et al. [144]. Their proposal merges the models processing a single parent feature with its children features each time. It creates a *feature selection map* that describes, for each feature, the effect that it has on the others (e.g., if a feature forces, prohibits or do not affect the selection of other feature). This information is later used to determine the type of variability for each feature in the resulting model. Their work reproduced the rules defined by Mannion et al. [89] and proposed some automatic solutions for situations which that approach cannot solve. Their method supports basic feature models and feature models with cross-tree constraints relationships, is based on local rules and only considers one abstraction level at every step.

Summary. There are many proposals aimed to merge feature models exploiting their tree-based structure. They propose different merge semantics and techniques. However, these approaches apply only on feature models that comply with some pre-conditions: they must be parent-compatible models and must include only cross-tree constraints specified using *requires* and *excludes* relationships.

3.2.2 Semantic-based Merging of Feature Models

To overcome the limitations of the structure-based merging of feature models, the Acher et al. proposal aims to merge feature models considering the equivalent formulae in propositional logic. It is based on previous works from Czarnecki [38], She et al.[125][126] that support transforming feature models into propositional logic and transforming back from these representations to new feature models.

Feature Model Composition Operators: Acher et al. [9][2] defined a set of merge operations where the user may specify if the resulting feature model will represent an intersection, a strict union or a reduced product. In addition, they defined algorithms that rely on structural modifications [8], and can be applied to parent-compatible feature models, and others that rely on propositional logic and solvers and can be applied to a large number of models [10]. These operations are implemented as part of FAMILIAR, a DSL for processing feature models.

	Operations	Comments
Structure-based Merging of Feature Models		
Feature Model Refactorings Alves et al. [12]	<ul style="list-style-type: none"> • Unidirectional Refactorings • Bidirectional Refactorings 	<ul style="list-style-type: none"> • Structural modifications to the models • Only bidirectional refactorings maintain the set of valid configurations • It is focused on reflecting changes and refactorings on programs to changes on the FMs.
Feature Model Merge by Transformation Segura et al. [122]	<ul style="list-style-type: none"> • Merge 	<ul style="list-style-type: none"> • Graph Transformations (ACG rules) to combine the structure of the input models • The resulting semantics contains the union of the semantics of the inputs • It applies to parent-compatible feature models that include only cross-tree constraints represented as relationships.
Viewpoint Oriented Variability Modelling Mannion et al. [89]	<ul style="list-style-type: none"> • Merge 	<ul style="list-style-type: none"> • Set of Rules to resolve differences among the feature models. Some cases require user intervention to solve the conflict. • The intended resulting semantics is the union of the semantics of the inputs. It may depend of the user interventions. • It applies to parent-compatible feature models that include only cross-tree constraints represented as relationships.
ViewPoint Integration Niu et al.[99]	<ul style="list-style-type: none"> • Merge 	<ul style="list-style-type: none"> • A lattice ordering that determine how to resolve differences among the feature models. • The intended resulting semantics is the union of the semantics of the inputs. • It applies to parent-compatible feature models that include only cross-tree constraints represented as relationships.

Table 3.5: Related Work on Composing Feature Models

	Operations	Comments
Parent-Compatible Feature Model Merge and Intersection Van den Broek et al. [143][144]	<ul style="list-style-type: none"> • Merge • Intersection 	<ul style="list-style-type: none"> • An algorithm that transform the input models into Generalised Feature Trees, match the features that exists in both models and combine their children features. • The intended resulting semantics is the union or the intersection of the semantics of the inputs. • It applies to parent-compatible feature models that include only cross-tree constraints represented as relationships.
Feature Model Merge by Local Rules Aydin et al. [16]	<ul style="list-style-type: none"> • Merge 	<ul style="list-style-type: none"> • An algorithm that transform the input models into Generalised Feature Trees, match the features that exists in both models and combine their children features. • The intended resulting semantics is the union of the semantics of the inputs. • It applies to parent-compatible feature models that include only cross-tree constraints represented as relationships.
Semantic-based Merging of Feature Models		
Feature Model Composition Operators Acher et al. [2][9]	<ul style="list-style-type: none"> • Insert • Aggregate • Merge • Union Merge • Strict Union Merge • Intersection • Diff • Slice 	<ul style="list-style-type: none"> • An algorithm that (1) matches the features of the input models, (2) calculates a propositional formula for the semantics of the resulting model, and (3) creates a tree for the resulting feature model. The tree is created based on the implication graph of the calculated propositional formula. The parts of the formula not represented in the tree are added as additional constraints to the model. • The resulting semantics is a well defined operation (e.g., union or intersection) of the semantics of the inputs. • It applies to non-parent compatible feature models that may include constraints as formulae

Table 3.5: Related Work on Composing Feature Models

3.2.3 Discussion

As with the approaches to model complex products, existing approaches to merge feature models do not consider standards or additional constraints as a special case. There are many proposals of operations to merge feature models representing different domains or concerns of a complex products. To use these operations, standards must be specified as part of the constituent models. Regretfully, modeling the standards in this way ends including the corresponding constraints as relationships and rules in the feature models for each domain, but not in an independent model. The use of feature models mixing domains and standards makes difficult the analysis, maintenance and reuse of the standard.

3.3 Deriving Configuration Systems from Feature Models

The development of correct Product Configuration Systems remains as a challenge. On one hand, it is necessary to create a software that enforces completely the options and constraints in the feature model. And, on the other hand, it is necessary to process the user decisions during an interactive configuration to provide proper feedback and update correctly the user interface.

There are multiple studies focused on best practices and issues to develop Configuration Systems. Streichsbier et al. [129] evaluated 126 web configurators to determine *de-facto* standards in user interface design. Rogoll et al. [116] evaluated 12 configurators to report usability and visual representation problems. Trentin et al. [136] surveyed 63 users to determine UI characteristics that reduce the effort during product configuration. More recently, Abbasi et al. [1] evaluated 111 Product Configurators to determine how these systems represent the configuration options, handle their constraints and support the configuration process. This study revealed some issues in the real configurators used by people everyday in the web. We present here three issues detected by the Abbasi et al.

Correct handling of constraints : Problems in constraint handling lead to Product Configurators that allow erroneous configurations because they do not detect some conflicts. In their study, Abbasi et al. found that the 97% of the configurators propagate automatically the decisions, only the 3% identified conflictual decisions and the 11% provide explanations about the conflict to the user. The 26% of the configurators do not check constraints involving multiple features and the 4% do not even check if the mandatory options are indeed selected.

Correct updating of the user interface : Even if the Product Configurator is able to determine correctly which features are enabled or not, this information must be used to update the user interface consistently. Studies from Rabiser [109] and Hubaux et al. [68] show that many Product Configurators provide incomplete or incorrect information and ad-

vices. Abbasi et al. found that the 11% of the configurators with cross-constraints do not support hiding or disabling options.

Ability to revise the decisions : Including options to reset the configuration and undo actions is one of the recommended capabilities for user guidance in Product Configurators [109]. However, these options are hard to implement if the reasoning mechanism is not properly implemented. Abbasi et al. found that the 69% of the analysed configurators allow users to revise their decisions and only the 11% provide *undo last decision* functionality. Configurators without that options force users to start from scratch each time they want to alter their configuration.

3.3.1 Automated Derivation of Configuration Systems

Recently, several approaches aim to create automatically the software from a feature model and a specification of the intended configuration system. Two of the most representative are:

Model-driven generation of Web Configurators: Boucher et al. [25][24] proposed a model-driven approach to generate web configuration tools¹. They use multiple DSLs to specify the models: Feature models written in *Textual Variability Language (TVL)* to represent configuration options, models in *Textual View Definition Language (TVDL)* to specify the user interface elements, and models in *Featured Cascading Style Sheets (FCSS)* to represent the visual appearance of these elements. Once created the models, they are processed in *Acceleo*² to generate an HTML-based configuration tool. The proposal aims to create a single configuration system from a single feature model. They do not consider that multiple independent systems can be created from the models.

Model-driven generation of Multi-view Configurators: Sottet et al. [128] propose a model-driven approach where multiple views of the same model can be defined. In their approach, the resulting Configuration System allows a multi-step configuration process where each stakeholder decide on the features of her concern. As the above, this proposal do not consider the creation of multiple configuration systems based on the same models.

3.3.2 Existing libraries to process Feature Configurations

There are many tools and libraries that can be used as a foundation to create Configuration Systems. For instance, software systems such as *Familiar*, *FAMA* and *SPLOT* can be used to analyse feature configurations and determine their validity against a feature model and, therefore, to update the configuration user interface after a user makes a decision.

¹<https://staff.info.unamur.be/qbo/Tools/>

²<https://eclipse.org/acceleo/>

For instance, to validate configurations, *Familiar* provides two operations: `isComplete` and `isValid` to determine if a configuration is complete and valid. *FaMa* provides other two: `ValidProduct` and `ValidPartialConfiguration` to determine the validity of configurations and partial configurations. *SPLIT* provides an `isValidConfiguration` method to determine if a configuration is valid and a `ConfigurationEngine` class to interactively select and deselect options.

However, these libraries are not intended to provide complete support, through a user interface, to interactive configure a product. For instance, *FaMa* does not support directly configuration processes and does not provide options to undo selections. *Familiar* provides operations to `deselect` and `unselect` options that support other types of user decisions, but do not maintain information of which features are selected or disabled by the user or the software, difficulting the implementation of the undo operation and the update of the user interface. *SPLIT* provides a `ConfigurationEngine` class that support decision changes and undo last selection, but does not support undo of the other decisions.

3.3.3 Discussion

There are many proposals and works to support configuration systems using Feature Models [1][44][117]. A few of these proposals aim to derive automatically configuration systems. In these proposals, developers specify user interfaces and views on a feature model and a software system takes that information and produce source code for the corresponding configuration system. Regretfully, these proposals are based on a single model and do not support explicitly the existence of multiple domains and standards therein. For instance, if the feature model is created by merging other models for multiple standards and domains, each time one of the constituent models changes, it is necessary to determine if the feature models is affected and a new configuration systems must be generated.

Regarding the existing software that can be used to support the configuration tasks, tools and libraries such as *Familiar*, *FaMa* and *SPLIT* offer methods to analyze configurations against a feature model. A Configuration System may take a configuration provided by a user and perform some analyses on her decisions. However, *Familiar* and *FaMa* are not aimed to support interactive configuration processes and do not report information about the sequence of decisions made by the user and the features that have been selected or deselected automatically. *SPLIT* maintains more information about the configuration steps but does not offer options to undo decisions that are not the last one.

3.4 Summary

There is a lot of related works. The number of proposals aimed to use feature models to represent complex products and configuration systems has increased

in the last years [67][19][9][1][117]. However, a few of them support explicitly the modeling of standards and regulations that may introduce or affect the constraints. Almost all the proposals consider the standards and regulations as additional features that do not require special handling. In this chapter we surveyed proposals in the state-of-the-art highlighting the difficulties caused by the confluence of multiple domains and standards.

Modeling using a single Feature Model represents standards and regulations as features and constraints in the same model used for the technical domains of the product. The resulting models end including features and constraints for each standard. For the modelers, it is hard to determine if a constraint is defined by the domain or by the standard. When a standard changes, it is hard to determine where the model must be modified. At the end, these approaches difficult the maintenance of the models and division of the work.

Modeling using Views on a Feature Model uses a single feature model to represent all the product and concerns. Modelers can define multiple views to focus on a single concern during the creation and review of the models. For instance, it is possible to define a view including only the elements related to a single standard. However, a view belongs to a single models and cannot be reused for other products. This is an inconvenience for industries where the same standard may apply to multiple products. In addition, because there is a single model behind the scenes, using views may not reduce all the problems associated to the maintenance. A change on a standard may introduces additional changes, and possibly errors, in the other views. It is necessary to use specialized tools that support views on feature models.

Modeling using Multiple Feature Models uses different feature models to represent each technical domain and concern. Merge operations are used to compose the models and create the combinations that are of the interest of the stakeholders. This approach not only allows the separation of concerns but also the reuse of the models across multiple product lines. However, existing tools provides support to merge orthogonal (or almost-orthogonal) feature models where each model includes a different set of features or has very few in common with the others. The resulting model includes the union of these features. This is an inconvenience for modeling standards as independent feature models. Standards and regulations are mostly sets of constraints. A single standard may impose restrictions on features in multiple of the other feature models. In consequence, existing model merging operations may end introducing features that are not of the interest for a specific domain.

Creating Configuration Systems from Feature Models implies the creation of programs that takes user decisions and use the information in the feature models to reason on these decisions. Nowadays, these programs

are created by hand in almost all the related work. Few approaches take the models and derive source code for the corresponding configuration system. In addition, the existing approaches take a single feature model or a view on that model but do not consider the existence of multiple models. This may result in an inconvenience if the products are modeled using multiple feature models. For a specific product family, when a domain or a standard is modified, it is possible that the configuration system must be build again. It is desirable a comprehensive approach that consider the constituent models, the required merge operations and the derivation of the configuration systems.

The following chapters present our proposals to tackle these difficulties and offer a better support for creating Feature-based Configuration Systems considering multiple standards.

Chapter 4

Modeling Features of multiple Domains and Standards

Representing complex products using a single Feature Model results in large models hard to create and maintain by humans [67][81][114][140]. Instead of creating a single feature model, we propose a modeling approach where each technical domain and each standard and regulation of the product are modeled independently: *Feature Models for Domains* are used to model each domain, *Feature Constraint Sets* are used to represent interactions between the models, and *Feature Models for Standards* are used to represent standards and regulations. These models are later combined in different ways for reviewing by the stakeholders and generating the corresponding software.

This chapter describes our modeling approach. First, Section 4.1 introduces the diverse types of models using an example of the Automotive Industry. Section 4.2 presents how these models can be automatically combined and analysed according to the different needs of the stakeholders. Finally, Section 4.3 presents a discussion and Section 4.4 summarizes the chapter.

Contribution: Main contribution of this chapter is a novel modeling approach that (1) uses feature models for domains, inter-domain constraint sets, and feature models for standards to represent complex products, (2) relies on specialized operations to combine and analyse these models, and (3) derives the corresponding configuration system based on these combined models.

4.1 Modeling Domains and Standards

An important objective of our approach is to allow domain experts to create models of their own area of expertise without being distracted or worried by features and constraints of other domains. Relationships among different domains are discussed by multiple experts. If a feature in a domain constrains features in other domain, the experts in these domains must work collaboratively to precise the interactions. In addition, standards and regulations are

modeled independently by experts. As a benefit, each domain expert creates and debugs smaller models that represent only elements that she dominates.

4.1.1 Overview

The following list describes, for each concern, which type of models we use:

Domains: Each domain is modeled using an independent feature model. Each model includes only the features and constraints that correspond to that domain. These models for the domains are disjoint. Thus, the features included in a model cannot be included into another other model representing a different domain. In addition, the constraints included in a model comprise only references to features on that model and cannot refer to features in other models.

Domain interactions: Relationships and dependencies among feature models are specified using sets of constraints independently of the feature models. Each *constraint set* is created independently. It includes a definition of which feature models are being related. In addition, it includes propositional formulae where each term coincide with features in these models.

Standards and Regulations: Externally defined constraints are modeled independently. Each standard and regulation is specified in a different feature model. However, in contrast to the feature models for the domains, the models for the standards include features that are references to features in other models. These references can be used to define constraints affecting features in the other domains.

Configuration Systems: Features that are part of a configuration system are specified by a set of commands that slice, merge, and combine feature models. The result of executing these commands is a new feature model that includes only the features and constraints to include in the system. Additional files specifying valid and invalid configurations can be used to test if the resulting model is able to represent real products.

4.1.2 Running Example: Feature-Based Configuration for Cars

To illustrate our approach, we will use a running example based on the automotive industry. Another case study modeling Electrical Transformers is presented in Chapter 8. We prefer to introduce our approach using a case study on modeling cars because this domain has been well-studied in the literature and may result more amenable to understand by non-experts.

As mentioned by many authors [50][132], designing and manufacturing automobiles require the specification of multiple subsystems, of the interactions

of these subsystems, and the multiple standards that these products must comply with around the world. It is an excellent scenario to describe our approach for modeling multiple domains, standards and configuration systems.

4.1.3 Modeling Multiple Domains

Nowadays, cars are very complex systems: A typical car comprise more than 50 different subsystems [50] distributed in a network of more that 80 electronic control units (ECUs) [26]. Modeling all these subsystems, components and their interactions is key for engineering cars.

In our approach, a first decision is to determine which domains to model. There are many alternatives to structure the feature models [41][105]. For instance, it is possible to use a structure that mimics the organizational units by creating a feature model for the diverse department stakeholders in the company. Many examples in the literature model cars considering a *functional architecture*, i.e., creating a model for each key subsystem and component [105].

Once we have defined the domains to model, we create an independent feature model, we named *A Feature Model for Domains*, to represent each one. Here we describe models to represent the systems for car lighting [50] and periphery supervision [131], systems that have been subject of many studies.

Car Lighting System One of the car systems is the lighting system, a critical part for the safety of both the driver and the other road users to see and to be seen [141].

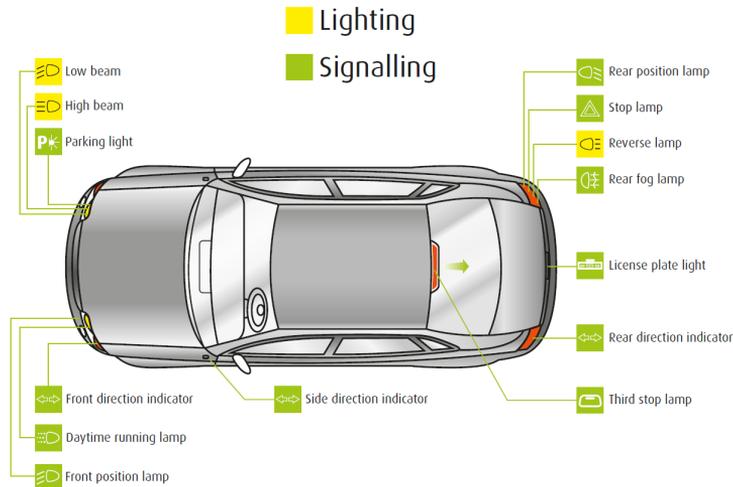


Figure 4.1: Elements in a Car Lighting System [141]

Figure 4.1 shows typical elements in the lighting system [141]. On one hand, there are **lighting elements** aimed to project light and illuminate the road: a

typical *Head-lamp* may include high-beams, low-beams (or dipped-beams), fog-beams, cornering-lamps and spot-lights; while a *Rear-Lamp* includes a reverse-beam. These beams can be projected using *Incandescent Lamps (Halogen)*, *Gas Discharge Lamps (Xenon)* or *LED Lamps*. On the other hand, there are other **signaling elements** that allow light to inform car activities to other drivers and pedestrians: *Head-lamps* may include parking-lights, front-direction indicators and day-time running lights; *Side-lights* may include side-direction indicators; and *Rear-lamps* may include rear-fog lamps, stop-lamps, rear-position lamps and rear-direction indicators. A car may also include other lamps such as a license plate light and a third stop lamp. In addition, emergency cars such as fire engines, ambulances and police cars may include special warning lights.

Modern Car Lighting does not include lamps only. They also include additional devices to improve the driver’s safety and comfort. For instance, there are “Automated Leveling Systems” that keep the headlights aimed down to the road. These systems use level sensors in the car to determine if the car is tilted forward or back and electric servomotors to move the lamp accordingly. Similarly, “Adaptive headlights” monitor the car’s speed and rotation, as well as the angle of the steering wheel, to turn the headlights projects by up to around 15 degrees to match the car’s intended direction. In addition, “Automated cornering lights” are activated when the car is parking or in a sharp turn at low speeds.

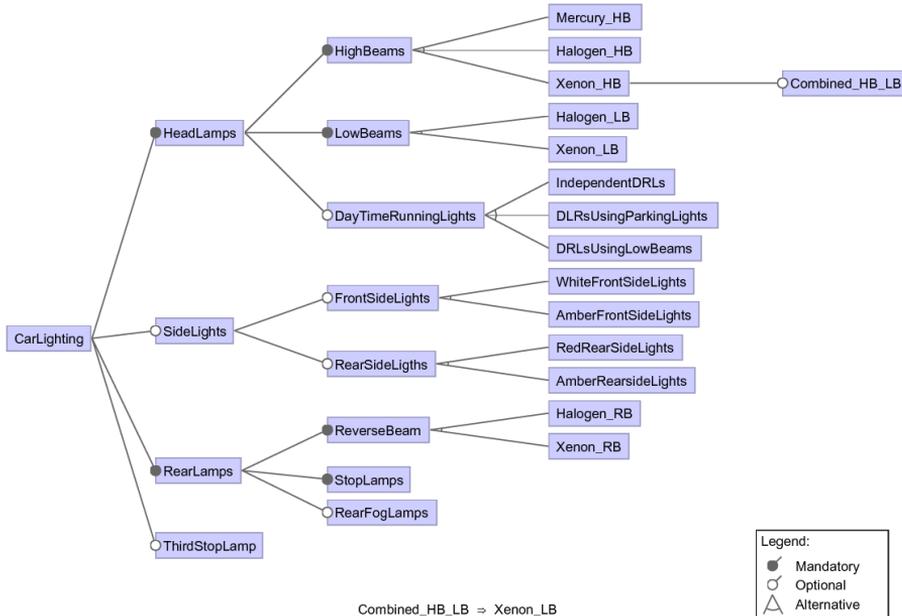


Figure 4.2: Feature Model of a Car Lighting System

In our approach, we create a Feature Model for Domains to specify the

elements that comprise the Car Lighting. Figure 4.2 shows an excerpt of the feature model for a Car Lighting System. It comprises features that represent the diverse alternatives for beams, types of lamp and assistance systems that can be included in a car. In addition, it describes constraints on which elements can be combined into the same car. For instance, it states that a car with “Xenon High-beams” and “Combined High-Low Beams” must be configured including “Xenon Low Beams”.

Car Periphery Supervision System A different domain is the Car Periphery Supervision, a complex subsystem that monitors the local environment of a car on the basis of sensors installed around a vehicle [131]. It comprises diverse type of sensors that may be installed in many locations of a car. Figure 4.3 depicts the diverse ranges of measurements of the sensors in a car may exhibit. Engineers select which location and type of sensors include in a car depending on the applications they want to support.

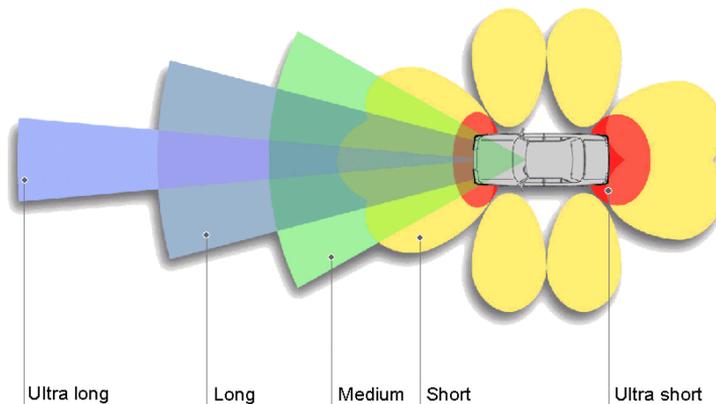


Figure 4.3: Sensors in a Car Periphery Supervision System [133]

Measurements and sensor data are used by Electronic Control Units (ECUs) in the car to enable different kinds of applications. These can be grouped into *safety-related* applications such as pre-crash detection, blind spot detection, and adaptive control of airbags and seat belt tensioners; and *comfort-related* applications such as parking assistance and adaptive cruise control [131]. There are applications that require a specific sets of sensors. For instance, an application for *parking assistance* require sensors at the rear.

A feature model different that the one used for the Lighting is created to represent the Car Periphery Supervision. Figure 4.4 shows a simplified Feature Model based on others that exist in the literature [60][131][132][133]. There are some features representing the applications to include in the system. For instance, there is a module for parking assistance that determine the distance of the car to a wall or another car behind. In addition, besides the modules, the model includes other features representing the type and the location of the

sensors in the car. There are Low-end and High-end sensors for the front and the rear of the car. In addition, there are light-sensors that can detect the conditions of visibility outside the car.

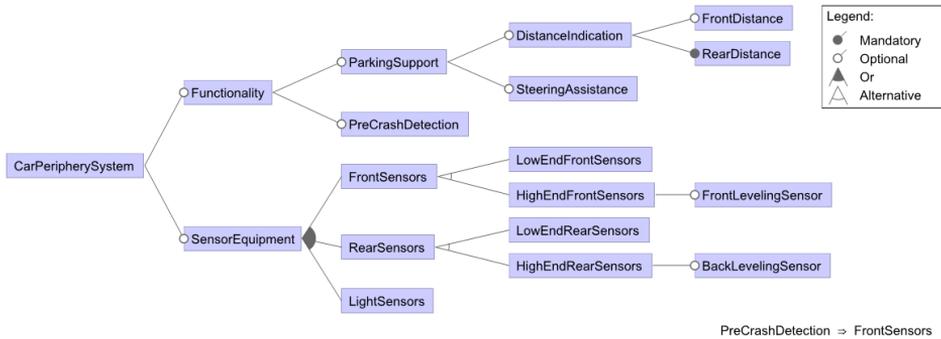


Figure 4.4: Feature Model of a Car Periphery Supervision System

4.1.4 Modeling Domain Interactions

Car subsystems do not exist in isolation. There are many interactions that impose cross-domain constraints. In our approach, these domain interactions are specified using *Constraint Sets*.

For instance, the *Automated Leveling Systems* imply relationships among the Lighting and the Car Periphery Supervision systems. Figure 4.5 shows key elements of an Automated Leveling System. There is a *Leveling actuator* that is part of the Lighting system and a set of *Leveling sensors* included in the Car Periphery Supervision.

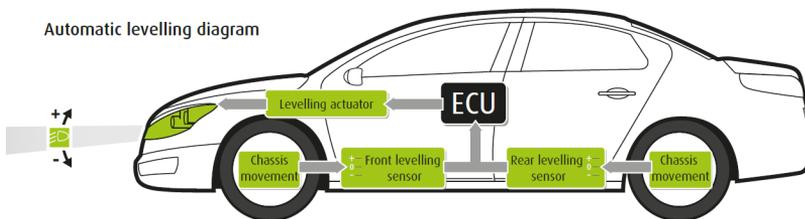


Figure 4.5: Key elements in an Automated Leveling System [141]

A *constraint set* is a propositional formulae that specifies the relationships between the features in one domain to the features in other domains. For instance, we can define a constraint set to represent the dependencies between the Car Lighting and the Car Periphery Supervision Subsystems. A rule may state that the Automated Leveling System in the Lighting system requires Front- and Back-leveling sensors in the Periphery Supervision system.

$AutomatedLeveling \Rightarrow FrontLevelingSensor \wedge BackLevelingSensor$
 $AutomatedLights_HB_LB \Rightarrow LightSensors$
 $AdaptiveCorneringLights \Rightarrow HighEndFrontSensors$

Figure 4.6: An example Constraint Set

Figure 4.6 shows a set of constraints relating these subsystems. Note that it comprises expressions where terms represent features in multiple models. For instance, in the first expression, the term *AutomatedLeveling* correspond to a feature in the Car Lighting and the terms *FrontLevelingSensor* and *BackLevelingSensor* belongs to the Periphery Supervision system.

4.1.5 Modeling Standards

Car specifications and functions are defined and designed according to stringent regulations around the world. Despite recent efforts of many governments, car regulations are quite different across the world. A large number of countries follow the regulations of the *United Nations Economic Commission for Europe (UNECE)*¹. Around 50 countries formally participate in the WP-29 agreement to adopt uniform technical regulations. In 2015, that agreement comprised 135 UN regulations, most of them covering a single vehicle component or technology. United States, on the other hand, has its own *Federal Motor Vehicle Safety Standards and Regulations (FMVSS)*² that does not recognise the UN regulations. Other countries such as Canada³ and Japan⁴ recognise or either mirror some of the UN and USA regulations in their own requirements. A recent study counted around 25 different country-specific regulations only for the Standard Emissions in cars [102].

To mention some differences among the regulations, *Automated Leveling Systems* are required in all the new cars in Europe, but required only in cars equipped with bi-xenon headlights in USA. *Daytime Running Lights* are mandatory in Canada but optional in Europe and USA. *Side-lights* have different sizes and colors in Europe than in USA. A Configuration System aimed to these markets must consider the additional constraints imposed by the corresponding regulations.

We create a separate feature model to represent each standard. Our Feature Models for Standards represent a single standard and may be specified by an expert on the standard without worrying about other regulations.

¹http://www.unece.org/trans/main/wp29/meeting_docs_wp29.html

²<http://www.nhtsa.gov/cars/rules/import/FMVSS/>

³<http://www.tc.gc.ca/eng/acts-regulations/regulations-crc-c1038.htm>

⁴http://www.bookpark.ne.jp/jsae/book_e.asp

For instance, Table 4.1 shows some differences in the regulations of Europe and USA regarding the side-lights. Instead of creating a single model, we define a Feature Model for the European Regulations and another for the American Standards related to the Car Lighting. Figure 4.7 and 4.8 shows excerpts of the corresponding Feature Models for Standards

EU regulations and US end-outline marker lamps [clearance lamps] at 2015
(R48: UN Regulation No. 48; F108: FMVSS Standard No. 108; R7: UN Regulation No. 7; SAE Standard No. J2042)

Property	EU (UN Regulations)	US (FMVSS/SAE Stds)	Comparison
Applicability	Optional, option of AM/RM1/RM2 category lamps	Optional	Identical for applicability The EU permits the use of variable intensity rear end outline marker lamps, while the US prohibits their use
Number	4-8	2x Front 2x Rear	Number of side marker lamps can range from 4-8 in the EU, but must be 4 (2x rear and 2x front) in the US
Color	Front: White Rear: Red	Front: Amber Rear: Red	Color must be white at the front and red at the rear in the EU, while the color must be amber at the front and red at the rear in the US
Position			
Height	Front: Upper edge not lower than upper edge of wind-screen Rear: At maximum height possible	As near the top as practicable	Minimum height at front is lower in the EU Identical for the rear
Width	Outer: ≤ 400 mm and as close as possible to the extreme outer edge of the vehicle	Indicate the overall width of the vehicle and symmetric about the vertical center line	Widths are more prescriptive in the EU, while the US is more subjective
Length		Front: On the front Rear: On the rear Other: Any other location to ensure that overall width of vehicle is indicated	Lengths are not defined in the EU, while the US provides subjective length definitions
Other	Distances must be ≥ 300 mm vertically from position lamps		Minimum vertical distance from position lamps are prescribed in the EU, while the US does not define these minimum distances

Table 4.1: Differences in regulations between EU and USA side turn-signal lamps, based on a study from Freund and Oliver [51]

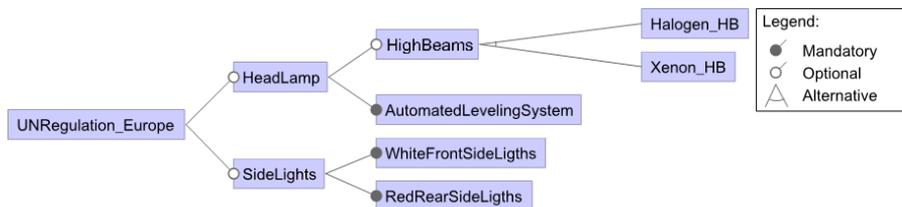


Figure 4.7: Feature Model for the Standard of UN regulations for Car Lighting

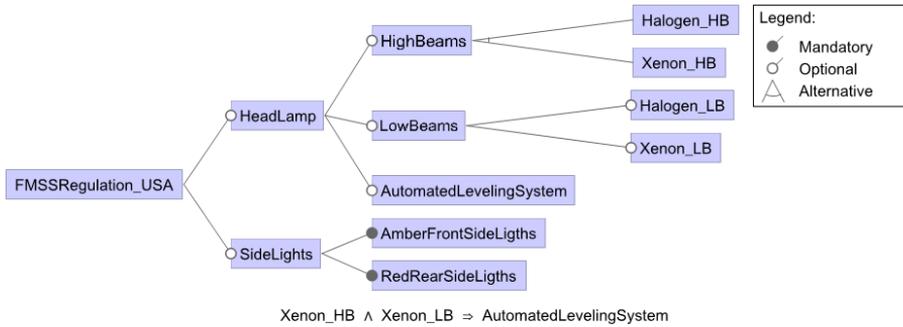


Figure 4.8: Feature Model for the Standard of FMVSS (USA) regulations for Car Lighting

Note that the *Feature Models for the Standards* reference features that exist in the other models. The root of each model is the Standard itself. The other features are references to features in the model for the Car Lighting. This model represent constraints on the selections of that features that only applies when the standard must be obeyed.

4.2 Reviewing and Analysing the Models

4.2.1 Motivation

In our approach, each model is created by experts in each domain: While the *Feature Models for Domains* are created by experts in the technical domains, the *Feature Models for Standards* are created by experts in each standard and regulation. However, there is a need to review the interactions of the diverse technical domains in light of the standards that must comply. For instance, it is desirable to review (1) how the constraints defined for a domain affect the features of other domain, (2) how a constraint set affects the features of multiple domains, and (3) how a standard affects the features of one or more domains.

Experts of different domains can work collaboratively to review the models and their interactions. For instance, a typical car can be modelled considering additional domains such as the engine, the power train and the alarm/security system. An expert in the alarm system may be interested on working with the experts in the Car Lighting and the Periphery Supervision systems. An alarm may require sensors and use the lighting as part of its operation. We allow modelers to combine and analyse the models in arbitrary arrangements.

An important element of our proposal is the ability to combine and analyse the diverse feature models we define. We have extended the *Epsilon family of languages*⁵ to accomplish these tasks. Epsilon is a family of domain specific

⁵<http://www.eclipse.org/epsilon/>

languages to manage different types of models, including options to transform them into other models or into source code. We created a set of drivers to load the diverse feature models, constraint sets and configurations in our proposal; and modified the language runtime to support feature model analyses. The resulting DSL allow modelers to process the diverse models of our proposal, to combine the feature models in different ways, and to perform a set of analyses on them.

4.2.2 Analysis of the Models

Analysing multiple Domain-Specific Feature Models To analyse the interactions among the multiple domains, we can use Epsilon or a library such as *Familiar DSL* to combine the corresponding *Domain-specific Feature Models* and *Constraint Sets* and perform analyses on the resulting models.

For instance, consider domain experts trying to analyse the interactions between the Car Lighting and the Periphery Supervision systems. They might be interested on a model that combines the features and constraints of both systems. The resulting model can be used to perform multiple types of analysis. For instance, it is possible to determine if some features are *dead* and cannot be included into any product because the existing constraints impede to do it.

Figure 4.9 shows the result of combining the models representing the Car Lighting, the Periphery Supervision System and the related Constraint Set (see Figures 4.2, 4.4 and 4.6 respectively). Note that the resulting model has a new root feature named *Car* and includes the features and constraints defined in the source models and constraint sets.

Analysing Feature Models for Standards To analyse the interactions among domains and standards, we combine the corresponding models. We first combine the domain-specific models and the constraint sets. Later, we use an additional operation to combine the resulting model with a feature model for standard. The resulting models include the features of all the domain-specific models plus the constraints defined in the standard.

Figure 4.10 shows a model combining the example Car Lighting and Car Periphery Supervision systems with the related USA standards. Note that the model is similar to the model that combines the domains depicted in Figure 4.9. It includes an additional optional feature *FMSSRegulation_USA* that represents the standard. In addition, it includes a set of constraints that coincide with the restrictions defined in the feature model for a standard.

Note that the resulting model includes new constraints imposed by the standard. If the USA standard is selected and the car has xenon lamps for the high- and the low beams, the car must include a Automated Levelling System.

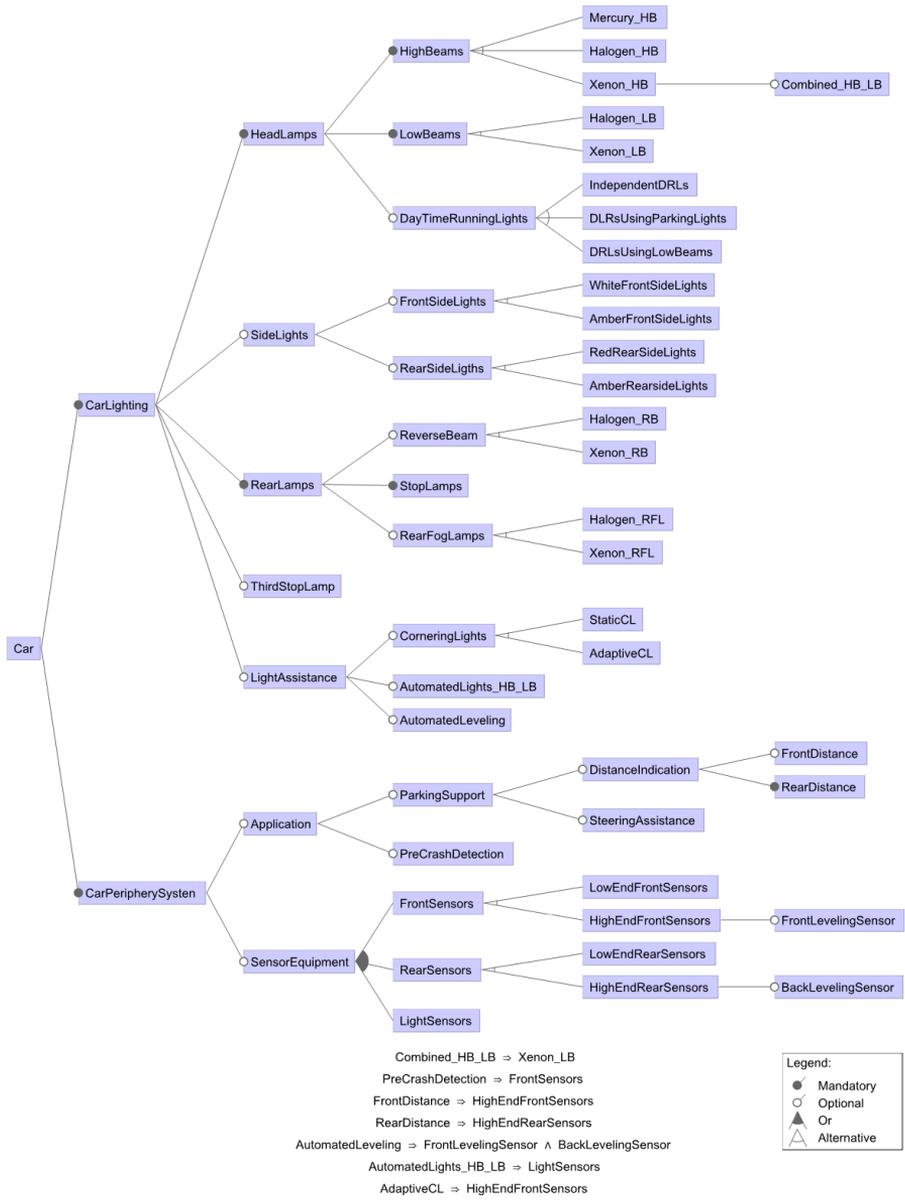


Figure 4.9: Model combining the example Car Lighting and Car Periphery Supervision systems.

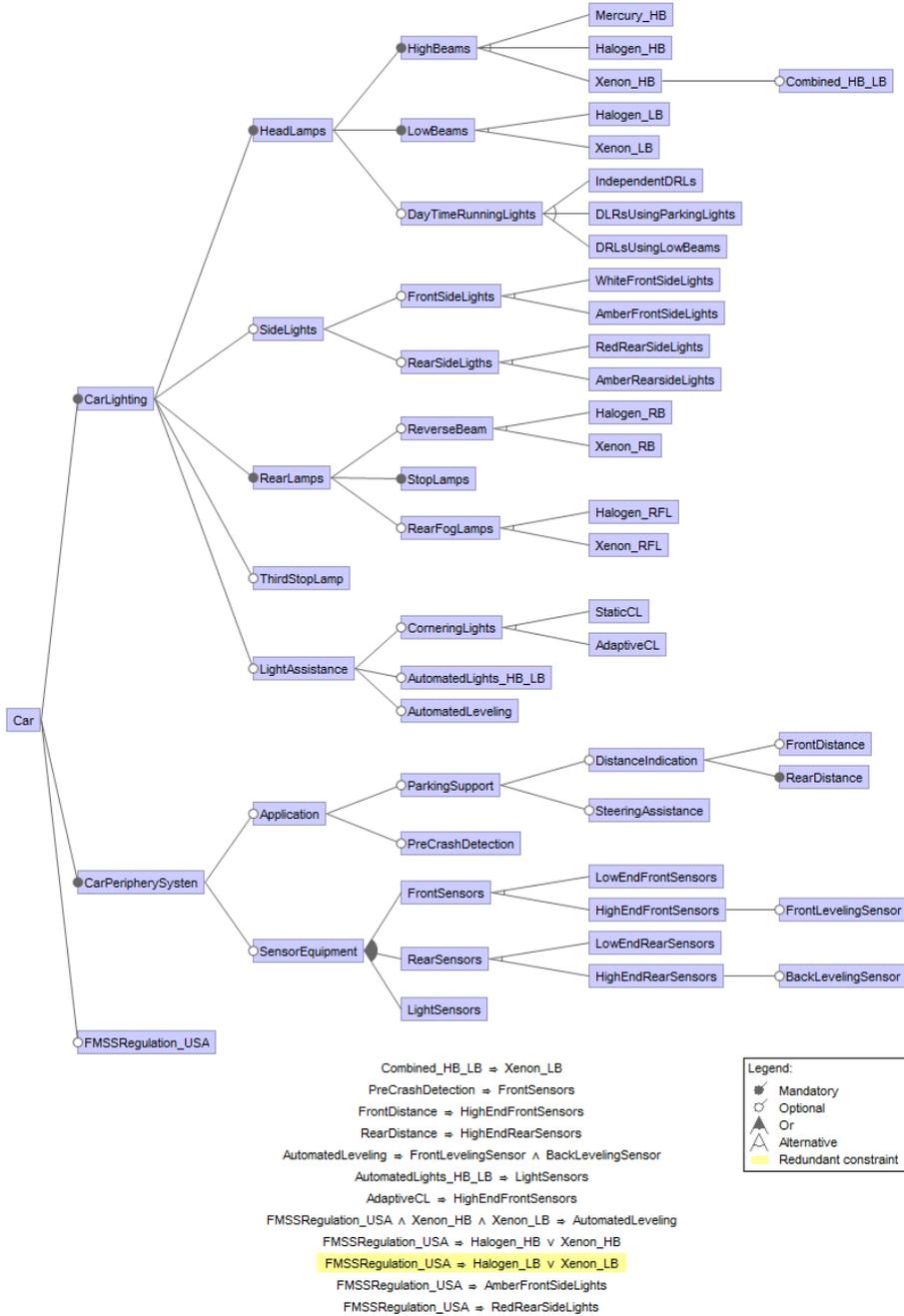


Figure 4.10: Model combining the example Car Lighting and Car Periphery Supervision systems with the related USA standards.

4.2.3 Testing of the Models

Review and Testing of Feature Models We must mention that the automated analysis of the Feature Models and their combination is not enough to assure their quality. It is necessary to perform reviews and tests to validate if the models represents correctly the products that a company produces and the constraints imposed by a standard [134].

There are many techniques that can be used to review and test the Models. For instance, domain and standard experts can meet together to perform a *walkthrough*. In addition, they can perform *manual tests* using a configuration system to check if the model constraints allow the specifications of real products from the factory. Other tests may include faulty products to determine if the problems can be detected. Finally, engineers can use techniques for *automated testing* to check if the models can validate correctly the configuration of the products in the catalog and historic databases of the company. They can use *generated tests* based on distinguished configurations and mutations on valid configurations [13]. As part of a modeling process, experts and engineers can define which combination of these techniques must be performed to produce models with a high level of quality.

This chapter presents a set of models as an example of our approach. These models were built based on other models from the literature and cannot be checked against a real product catalog of a company. Instead, these models were reviewed by peers. The Chapter 8 presents a case study where we participated in the modeling of real products for a company, and describes the techniques for review we used.

4.3 Discussion

There are many approaches to model complex products using Feature Models. Some of them focus on creating a single feature model and others on creating multiple feature models [67][105]. A few use an additional *context feature model* that maintains parameters and standards that the products must obey [58]. Our approach is different in three concrete aspects: we (1) Use a model for each standard, (2) Use constraint sets in addition to relationships and graphs to represent dependencies between domains, and (3) Combine the models using different arrangements depending of the intended application or the type of analysis to perform. This section discusses these differences in more detail by comparing them to the related work.

A Model for each Standard We use a model for each domain and for each standard. We agree with many authors that consider using a single feature model brings more problems than benefits [50][111][105]. However, in contrast to existing approaches we use a feature model for each standard. Hubaux et al. [66] structure the feature models according to the stakeholder's concerns. Reiser et al. [111] use a model for each subsystem. They do not make a special consideration for the standards and, therefore, their rules may result scattered

on the diverse models. Hartmann et al. [58] defines an independent *context variability* model. This model includes all the product's parameters and standards that may constrain the features in the other models. In this approach, all the standards are specified in a single model and cannot be modeled or reviewed independently. Using a model for each standards, our approach allows experts on each standard to create the corresponding model. This improves the separation of concerns and division of the work.

We use *Feature Models for Standards* to represent the standards. These models do not specify new features of the product. Instead, these models define constraints on the features in the other domains. A Feature Model for Standard has the standard as the root. The other features in the model are references to features in the models for domains. We defined a new operation that takes a feature model for a domain and a feature model for standard and produces a feature model with the structure and the constraints of the first and including additional constraints representing the information conveyed in the second. This operation, described in the Chapter 5, is different from the existing operations to merge feature models [2].

Constraint Sets In contrast to other approaches, we use constraint sets instead of relationships or graphs to specify dependencies amongst features in different domains. Bruin et al. [27] use Feature-Solution graphs that relate features in the problem space to features in the solution space. Hartmann et al. [58] use Feature-relationships to relate features in the context to features of the product. Reiser et al. [111] use Configuration Links to relate features in different domains. Instead of, we use sets of constraints expressed as propositional formulae. These constraints may involve more than two features and are more suitable to represent elaborated constraints such as those included in complex products such as a car.

Multiple Arrangements for the Models We may combine and analyse the feature models for domains and for standards in different ways. Depending on the intention of the modellers, it is possible to combine and analyse only some feature models for domains, many feature models for domains and standards, or all the models. Other approaches define a unique way to integrate the models. Hubaux et al. [66] use multiple models, but they are views of the same model and cannot be combined in a different way. Hartmann et al. [58] combines the context variability model with the other models varying only a set of pre-selected features. Our approach is similar to the proposals of Reiser et al. [111] and Acher et al. [2]. We use a domain specific language like the *Familiar DSL* proposed by Acher et al. [2]. However, we have introduced new operations to combine and analyse constraint sets and feature models for standards.

4.4 Summary

In this chapter, we presented the diverse elements of our modeling proposal:

Feature Models for Domains A set of disjoint feature models that represent the technical domains of the product. Each model represents a single domain and is created and reviewed by experts on that model.

Constraint Sets Propositional formulae that relate features in different domain-specific feature models. These formulae are used to express relationships and dependencies amongst the domains. For instance, they are used to specify, after selecting a feature in a domain, which features in other domains must be selected or deselected automatically during a configuration process.

Feature Models for Standards Feature Models that represent standards and Regulations. Each model represents a single standard. In a model, the root represents the standard and the other features are references to features in the diverse domains of the product. The model is used to specify the constraints over the product features that the corresponding standard defines.

Operations to Combine Models A domain-specific language to combine, merge and weave feature models for domains, feature models for standards and constraint sets. These operations can be used in different ways according to the intentions of the modelers and engineers. For instance, there are operations that can be used to combine multiple domain-specific feature models to allow domain experts to analyse the interactions of multiple domains. In addition, there are operations that can be used to combine the models and create a feature model that represent the products for a specific configuration system.

This chapter illustrates our approach using an example of the automotive industry. It includes models for a Car Lighting and a Car Periphery Supervision Systems, a sample of UN and USA standards, and their relationships. The example also presents how these models can be combined to analyse the domains. Finally, the chapter includes a discussion of the differences with other proposals in the related work.

Chapter 5

Combining Feature Models representing Multiple Domains and Standards

In our approach, complex products are modeled using multiple feature models. On one hand, the options and features of each technical domain are specified using *feature models for domains*. On the other hand, the constraints and restrictions defined by the standards and regulations are modeled using *feature models for standards*. These models are first created independently by diverse experts on each domain and standard. Later, they are combined to (1) analyse and review the models, and (2) create the models used for specific applications such as family-specific configuration systems and product recommenders.

This chapter presents the techniques we use to combine *feature models for domains* with *feature models for standards*. First, Section 5.1 presents a background, introducing some concepts for feature models and describing existing operations to merge feature models. Then, Section 5.2 introduces an example to illustrate why the existing techniques cannot be used to combine *feature models for domains* and *for standards*. Section 5.3 describes our approach and implementation, Section 5.4 presents a discussion, and Section 5.5 concludes the chapter.

Contribution: Main contribution of this chapter is a set of new operators to combine *feature models for domains* and *for standards*: (1) *Conditional intersection merge*, an operator for conditional enforcing of standards, (2) *Conditional partial intersection merge*, for enforcing only a subset of a standard, and (3) *Sequences of union merge and conditional intersection merge*, to combine multiple technical domains and standards.

5.1 A Glimpse to the Background

There are many proposals for *merge operations* that allow modelers to combine feature models using different semantics [9][16][144]. They are used, for instance, to combine models representing diverse views of the same product created for different people. This section presents an overview of the feature models, the operations to merge these models and their possible application for supporting standards.

5.1.1 Feature Models

Mentioned in Chapter 2, a *Feature Model* is a compact representation of a set of products [35][79]. It describes the common and variable features of these products along with constraints that determines which of these features can be and/or must be present in a product at the same time.

For instance, consider the feature model depicted in Figure 5.1. The root element r represent the product for the model and, therefore, is present in all the products. The feature A is an *optional* feature, i.e., a product may not include it. The feature B is a *mandatory* feature and must be included in all the products. The features C and D are in an *alternative-group*. A product must include one of them but not both.

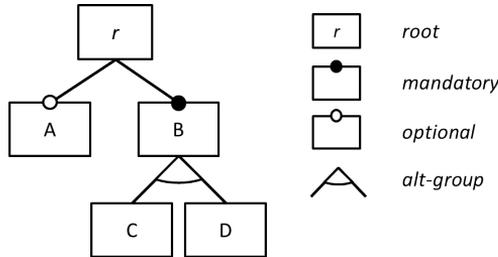


Figure 5.1: Example Feature Model

We use \mathcal{F}_{fm} to denote the set of features in a model. Considering the feature model fm_1 in Figure 5.1, the set \mathcal{F}_{fm} is:

$$\mathcal{F}_{fm_1} = \{r, A, B, C, D\}$$

A *Feature Configuration* is a set of the features of a Feature Model. It is *Valid* if the set satisfies the constraints defined in the Model. For instance, considering the model in the Figure 5.1, the configuration $c = \{r, A, B, C\}$ is valid against the model because it satisfies all its constraints.

Semantics for a Feature Model is defined by the corresponding set of valid configurations. Considering the above model fm_1 , the semantics $\llbracket fm \rrbracket$ is:

$$\llbracket fm_1 \rrbracket = \{\{r, A, B, C\}, \{r, A, B, D\}, \{r, B, C\}, \{r, B, D\}\}$$

5.1.2 Operations to merge Feature Models

Intuitively, the set of features of the model resulting of combining multiple feature models is the super-set of the features of all the constituent models. However, considering that feature models represent the set of products (i.e., configurations) in a product line [118][63], the semantics of the resulting model may vary from one approach to the other.

In their work on Feature Model semantics, Schobbens et al. [118] defined three kinds of operations to merge feature models:

- **intersection** when the merge results into a feature model that represents only the products that exist on the constituent models,
- **(strict) union** when the merge results into a model that represents the union of the products of all the models, and
- **reduced product** when the resulting feature model includes products that combine the features of each valid product of a model with the features of the valid products of the other model.

Definition 5.1. (Union-merge operation) *The (strict) union merge operation takes two feature models fm_1 and fm_2 and produces a resulting feature model fm_r , such that the set of configurations that are valid against the resulting model is the union of all the valid configurations of the source models, i.e., $\llbracket fm_r \rrbracket = \llbracket fm_1 \rrbracket \cup \llbracket fm_2 \rrbracket$* ■

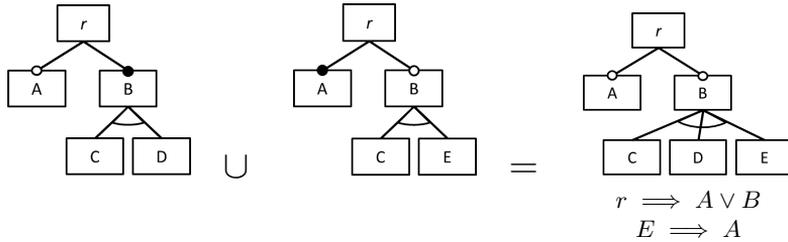


Figure 5.2: Example Union Merge

For example, consider the union merge of the feature model presented in Figure 5.1 with another one that includes the same mandatory root feature r . This model also includes the feature A as mandatory and the feature B as optional. In addition, B comprises the features C and E in an alternative group. Note that, for this fm_2 feature model, the semantics $\llbracket fm_2 \rrbracket$ is:

$$\llbracket fm_2 \rrbracket = \{\{r, A\}, \{r, A, B, C\}, \{r, A, B, E\}\}$$

The union merge of these models, depicted in Figure 5.2, is a new feature model which set of valid configurations is the union of the valid configurations of the others. Note that the merge operation combined the features in both models and introduced new constraints to invalidate the combination of features that are not valid in the source models.

$$\llbracket fm_r \rrbracket = \{\{r, A\}, \{r, A, B, C\}, \{r, A, B, D\}, \{r, A, B, E\}, \{r, B, C\}, \{r, B, D\}\}$$

Definition 5.2. (Intersection-merge operation) The intersection merge operation takes two feature models fm_1 and fm_2 and produces a resulting feature model fm_r such that the set of configurations that are valid against the resulting model is the intersection of the valid configurations of the source models, i.e., $\llbracket fm_r \rrbracket = \llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket$ ■

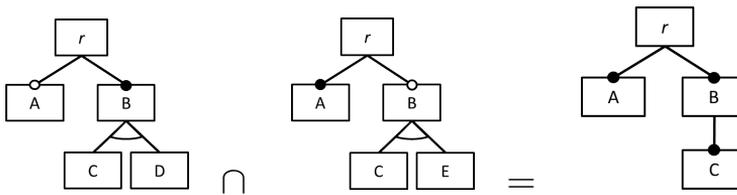


Figure 5.3: Example Intersection Merge

Figure 5.3 shows an example of the intersection merge. Note that the resulting model represent only the configurations that are valid to both source models:

$$\llbracket fm_r \rrbracket = \{\{r, A, B, C\}\}$$

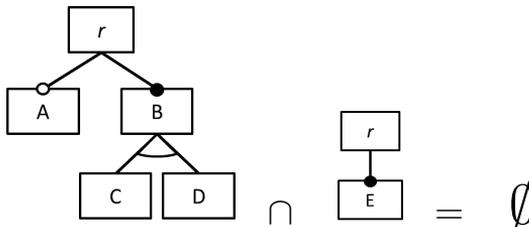


Figure 5.4: Example of Intersection Merge producing an error

Intersection merge may fail (or produce an empty feature model) when the intersection of the valid configurations of the models to merge is empty. For instance, consider the feature models depicted in Figure 5.4. The first is the same used in previous examples. The second includes the root feature r and a mandatory feature E , i.e., $\llbracket fm_2 \rrbracket = \{\{r, E\}\}$. Considering that the intersection of the configurations of these models is empty, i.e., $\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket = \emptyset$, the intersection merge fails.

Definition 5.3. (Reduced Product Aggregation) The reduced product aggregation takes two feature models fm_1 and fm_2 and produces a resulting feature model fm_r such that the configurations that are valid against the resulting model is the reduced product^a of the valid configurations of the source models, i.e., $\llbracket fm_r \rrbracket = \llbracket fm_1 \rrbracket \otimes \llbracket fm_2 \rrbracket = \{C_1 \cup C_2 \mid C_1 \in \llbracket fm_1 \rrbracket \wedge C_2 \in \llbracket fm_2 \rrbracket\}$. ■

^aWe reuse here the *reduced product* term introduced by Schobbens et al. [2][118]

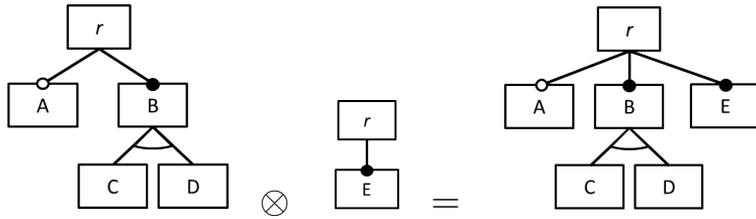


Figure 5.5: Example of Reduced Product Aggregation

For example, consider the same models presented before. The reduced product (or aggregation) will produce a feature model where the set of valid configurations is the combination of the valid configurations of both models. Figure 5.5 shows the result of the aggregation.

$$\llbracket fm_r \rrbracket = \{\{r, A, B, C, E\}, \{r, A, B, D, E\}, \{r, B, C, E\}, \{r, B, D, E\}\}$$

In contrast to the other merge operations, the reduced product aggregation can be used to combine orthogonal (or almost-orthogonal) feature models that represent different aspects or technical domains of the same products.

5.2 Challenges Merging Feature Models for Standards

Existing techniques for merging feature models, i.e., the *union-merge*, *intersection-merge* and *reduced product merge*, cannot be used to combine *feature models for domains* and *for standards* allowing the conditional enforcing of one or more of the standards.

5.2.1 Example

Consider a set of *feature models for domains* representing a family of electrical transformers: Some experts may model the electrical features of the transformer, other experts the mechanical properties and others its thermal and acoustic properties [32]. The feature model for the whole family may result from combining (e.g., by merging) the models for each domain. Existing compositional operations [9] aim to combine correctly these models.

Electrical Transformers may obey one or more standards depending of the country or the specific electric distribution network where it will be installed. The options and constraints of each standard may be specified using *feature models for standards*. However, these models must be handled differently than the above feature models representing technical domains, for three reasons:

- First, **a feature model representing a standard may affect more than one domain of the product**. For instance, a single standard for electrical transformers (e.g., the IEEE Std C57.12 [72]) imposes features and constraints to aspects such as their electrical features, their mechanical characteristics and their acoustic properties.
- Second, **a product may comply with more than one standard**. A medium-size electrical transformer may comply with multiple standards such as the ICONTEC NTC 3997 [70], NTC 819.4 [69] or the NEMA-TR1 [98].
- And Third, **when a product is being specified, the standards may be optional features**. For instance, a customer may request an electrical transformer that does not adhere to any standard. In consequence, the constraints defined in a standard such as the IEEE Std C57.12 must be enforced only when the product must adhere to it.

5.2.2 Unconditional enforcing of standards

Consider the example in the domain of electrical transformer depicted in Figure 5.6. On one hand, suppose that a feature model for domain states that the transformers can support a *Power* of X, Y or Z. On the other, suppose that a standard states that the transformers should support *Power* with one of W, X and Y. If we perform an *intersection merge* with these models, the resulting model will enforce the constraints defined in the standard. Note that the resulting model excludes the W feature that is not included in the domain and the Z that is not included in the standard.

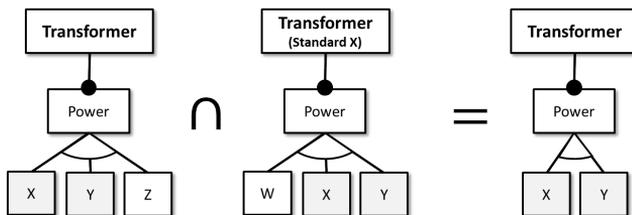


Figure 5.6: Unconditional Enforcing (Intersection) of standards

Note that the *intersection merge* enforces the standard. The resulting model includes only the features of the domain that obey the constraints of the standard. The standard is not an option.

5.2.3 Conditional enforcing of Standards

In companies selling products for different markets, standards cannot be not enforced in all the products. For instance, electrical transformers for United States must enforce standards that should not be enforced in transformers for Colombia, and vice-versa. Standards must be modeled as optional features that may be included (or not) in a product.

We say that optional standards introduce *conditional constraints*. Basically, if a product should not be compliant to a standard, the product does not require to satisfy the constraints defined in the standard. The product should satisfy the constraints in the standard only if the product must be compliant.

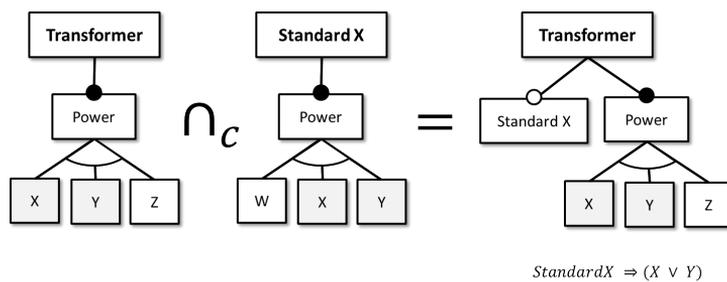


Figure 5.7: Conditional Enforcing of standards

Consider the example presented in Figure 5.7. If the standard is optional, (1) the constraints defined in the feature model for the domain must be enforced all the time, but (2) the constraints defined in the feature model for the standard must be enforced only when the standard is also included in the configuration. The resulting model includes a feature representing the standard. In addition, it includes a new constraint that enforce the rules for the standard, e.g., it states that, when the *standard* is selected, the *Power* should be one of *X* and *Y*. We say the the feature representing the standard is the *selector feature* because the constraints are enforced only of that feature is selected by a user and included in a configuration.

Note that the *conditional enforcing of standards* presented above cannot be implemented using the *intersection merge*. We need a new operation that combines the feature models for the domain and the standard but not invalidates the features in the domain not present in the standard. The standard must be enforced only when a *selector* feature is included in the configuration.

We named this new operation **Conditional Intersection Merge**. Intuitively, this operation must:

1. Introduce a selector feature for the standard into the domain (if it does not exist), and
2. Introduce new constraints to enforce the standard when the selector feature is selected.

5.2.4 Conditional Partial enforcing of Standards

Standards may affect only a subset of the features in the domains. For instance, in electrical transformers, a standard such as NEMA-TR1 only refers to the noise level of the devices and does not include constraints for accessories or mechanical properties. This represent a problem for using the union and the intersection merge operations where both feature models must represent all the features in the product.

For instance, consider the example shown in the Figure 5.8. While the feature model for the domain includes options for Power and Noise Level, the feature model for the standard defines only options for Power. Note that the Noise Level is a mandatory feature in the first model but is not included in the second one and, in consequence, there is not any configuration that is valid against both models. On one hand, if we perform an *intersection* of these models the result will be an empty set. On the other hand, if we perform a *conditional enforcing of the standard* as introduced above, the result will include a model where the standard cannot be selected. Note in the example the constraint: $StandardX \Rightarrow \neg NoiseLevel$. A user cannot select the selector feature, i.e., $StandardX$, because it conflicts to a core feature of the model, the $NoiseLevel$.

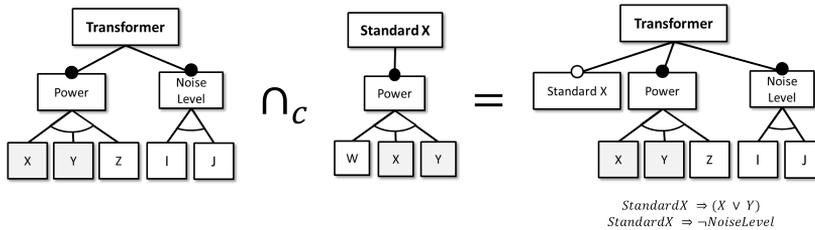


Figure 5.8: Conditional Enforcing of a technical domain and a standard

We need a new operation that determines which subset of the feature model for the domain must be considered during the enforcing of the standard. Features not related with the standard should not be removed by enforcing the standard. The Figure 5.9 shows the **conditional partial intersection** of the same models of the above example. Note that a user can select the $StandardX$ selector feature because the model does not include any constraint to the features not included explicitly in the standard.

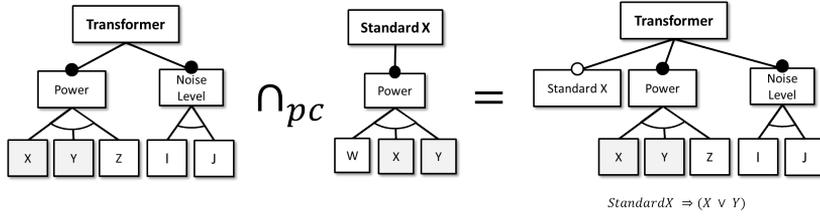


Figure 5.9: Conditional Partial Enforcing of a technical domain and a standard

Figure 5.10 shows a *conditional enforcing of standards* for a technical domain and two standards. Note that the operations introduce two new selector features to the first model, one for each standard. In addition, it introduces new constraints to enforce the standards when these selectors are selected. Note also that a modeler can combine multiple standards, one after the other, in any order, without affecting the final result.

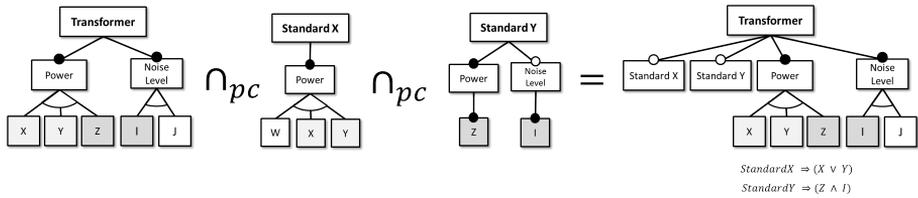


Figure 5.10: Conditional Partial Enforcing of a technical domain with multiple standards

We named this new operation: **Conditional Partial Intersection Merge**. Intuitively, this new operation:

1. Introduces a selector feature for the standard into the domain (if it does not exist),
2. Determines the set of common features in standard and domain feature models, and
3. Introduces new constraints to enforce the standard on the common features when the selector feature is selected.

5.2.5 Combination (with conditional enforcing) of Standards

Multiple technical domains may be affected by multiple standards. For instance, in a medium-size electrical transformer, the values for many properties in the electrical and the mechanical domains may be restricted by the IEEE Std C57.12, ICONTEC NTC 819.4 and NEMA-TR1 standards. If modelers define different models for the domains and the standards, it is necessary that the combination of models applies correctly the rules defined in the standards on the related features in all the domains.

Consider, for example, the feature models depicted in Figure 5.11. Combining the feature models for these domains and standards must produce a single feature model with selector features representing the standards and constraints that enforce the corresponding standards when one of these selector features is included in a configuration. To obtain the correct result, the models must be combined and conditionally intersected in the correct order.

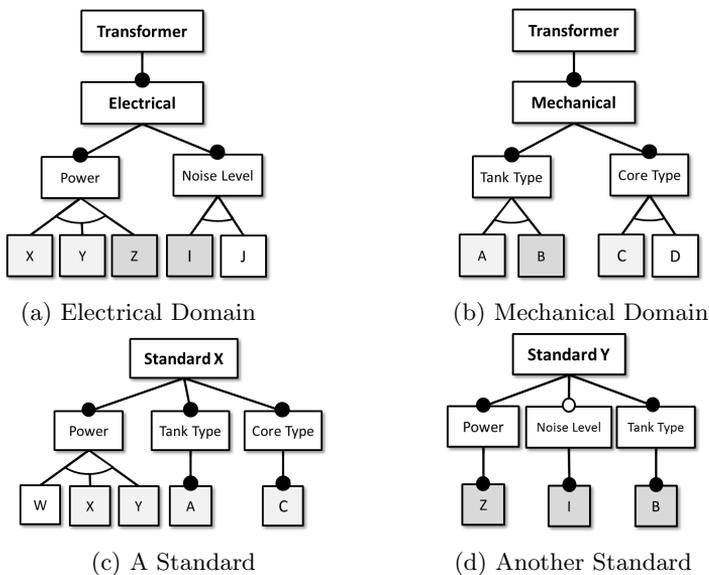


Figure 5.11: Example Domains and Standards

It is necessary to combine first the technical domains before the standards. The technical domains are combined using the *Reduced Product Aggregation* operation. This operation introduces the features for each domain into the result. Then, the feature models of standards are combined using the *Conditional Partial Intersection Merge* operation. This operation introduces the constraints of the standards that are related to features that already exists in the combined domains. The Figure 5.12a shows the aggregation of the domains and the Figure 5.12b shows the model after combining the standards.

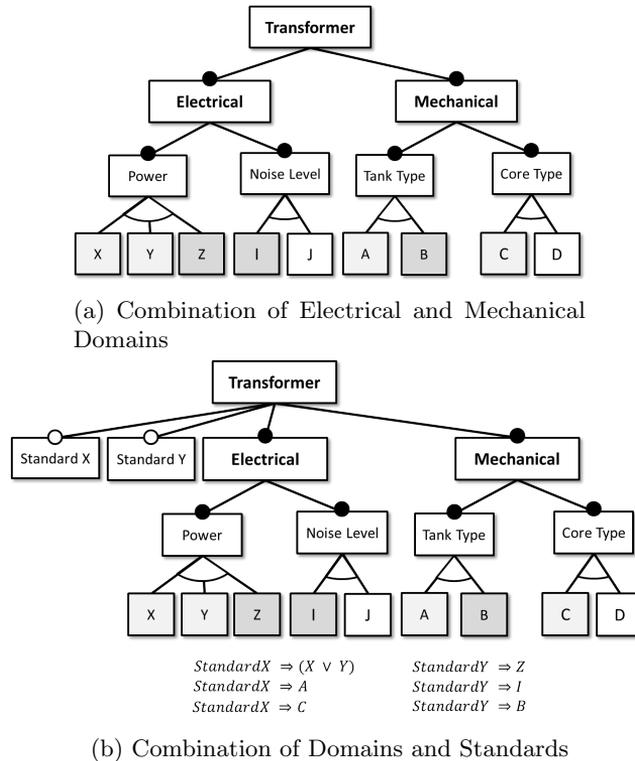


Figure 5.12: Combination of Domains and Standards

5.3 New Merge Operations

We have implemented new merge operations to support the combination of feature models for domains and for standards. We define three new operations that extend the existing operations to support the requirements presented above:

- **Conditional Intersection Merge** supports the Conditional enforcing of Standards.
- **Conditional Partial Intersection Merge** supports the Conditional Partial enforcing of Standards, and
- **Combination of feature models for domains and for standards** supports the combination of multiple domains and standards.

This section presents the operations and describes their implementation.

5.3.1 Conditional Intersection Merge

5.3.1.1 Definitions

We defined the *Conditional Intersection Merge* operation to combine (1) a *feature model of a domain* with (2) a *feature model of a standard*. This operation yields a feature model that represents all the valid configurations for the feature model for the domain that do not include the standard and the subset of configurations that include the standard and satisfy the constraints defined in the feature model for that standard.

Definition 5.4. (*Conditional Intersection Merge*) The *Conditional Intersection Merge* is an operation that takes two feature models fm_1 and fm_2 and produces a feature model fm_r that includes an optional Selector Feature f_s and its semantics $\llbracket fm_r \rrbracket$ represents the union of (1) the products of the first model $\llbracket fm_1 \rrbracket$ (where f_s is not included), and (2) the reduced product aggregation of the selector feature and the intersection of the products of both models $\{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket)$.

$$\llbracket fm_r \rrbracket = \llbracket fm_1 \rrbracket \cup (\{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket))$$

■

Conflicts in Conditional Intersection Merge. This operation produces a valid feature model if the first feature model is valid, The resulting model includes, at least, the same set of valid configurations of that model.

However, there are some circumstances where the resulting model is invalid or has inconsistencies:

- when the second model is invalid, the selector feature in the resulting model is a dead feature; and
- when the intersection of both models is an empty set, the selector feature in the resulting model is a dead feature.

Conditional Intersection Merge using an existing selector. We defined a variation of the above operation that can be applied to combine feature models for standards with legacy feature models combining domains and standards. These legacy models may include a feature representing the same standard that is represented in an independent feature model for a standard. In such situations, the Conditional Intersection Merge must not introduce a new feature but use the existing feature that represent the standard.

The *Conditional Intersection Merge using an existing selector* is used when the selector feature f_s already exists in the first feature model (i.e., $f_s \in F_{fm_1}$).

Definition 5.5. (Conditional Intersection Merge using an existing selector) The *Conditional Intersection Merge using an existing selector* is an operation that takes (1) two feature models fm_1 and fm_2 , and (2) a selector feature included in the first model $f_s \in \mathcal{F}_{fm_1}$, and produces a feature model fm_r which semantics $\llbracket fm_r \rrbracket$ represents the union of (1) all the products of the first model $\llbracket fm_1 \rrbracket$ where f_s is not included, and (2) the intersection of the products of both models $\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket$ that includes f_s .

$$\llbracket fm_r \rrbracket = \{C \in \llbracket fm_1 \rrbracket \mid f_s \notin C\} \cup \{C \in \llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket \mid f_s \in C\}$$

■

Conflicts in Conditional Intersection Merge using an existing selector. This operation produces valid models if both feature models are valid and the selector feature is truly optional (it is neither full mandatory nor dead) in the first feature model, The resulting model represents, at least, the valid configurations of the first model that do not include the selector feature.

It may produce models that are invalid or have inconsistencies:

- when the selector feature is truly optional in the first model and the second model is invalid, the selector feature in the resulting feature model is a dead feature.
- when the selector feature is truly optional in the first model and the intersection of the semantics of both models is empty, the selector feature in the resulting feature model is a dead feature.
- when the selector feature is full mandatory in the first model and the second model is invalid, the resulting feature model is invalid.

5.3.1.2 Implementation

Conditional Intersection Merge The *Conditional Intersection Merge* receives two feature models fm_1 and fm_2 and yields a new model fm_r . It introduces to the first model: 1. a feature, the selector feature f_s , and 2. a set of constraints to produce the result. In the resulting model, the selector feature is always an optional child feature of the root feature in fm_1 , and the constraints are determined automatically to enforce the constraints in fm_2 when the selector feature f_s is selected.

As mentioned before, the Conditional Intersection Merge aims to combine feature models for domains and for standards. It applies to standards defines restrictions to the features in the domain and does not consider features in other domains. In such scenarios, the feature model of the domain is fm_1 and the model for the standard is fm_2 . At the end, the resulting model fm_r will include a selector feature f_s representing the standard and a set of constraints that enforce the standard when f_s is included in a configuration.

FAMILIAR implementation The operation can be easily implemented using FAMILIAR [7], a domain specific language for managing feature models. This language provides sentences to perform operations on feature models such as the *intersection merge* and the *union merge*. The implementation of our operator can be a script that uses that operations implementing the semantics presented in the Definition 5.4.

```

1 // given fm1 and fm2
2
3 // define a feature model for the selector
4 fms = FM ( fs )
5
6 // intersect the input models
7 fmTemp = merge intersection { fm1 fm2 }
8
9 // insert the selector as mandatory in the resulting intersection
10 f2root = root fmTemp
11 insert fms into f2root with mand
12
13 // merge the fm1 with the temporary result
14 fmr = merge sunion { fm1 fmTemp }
```

Listing 5.1: Example FAMILIAR Implementation of the Conditional Intersection Merge

Note that FAMILIAR provides data types for feature models and features, and functions to find the root of model and perform the diverse merge and intersections required by our operator. Each operation is executed one after the other. Internally, the merge operations performs three tasks: the match of the feature models, the calculation of the propositional formula that represent the result, and the calculation of the tree that will be used in the resulting

model. After the propositional formula is calculated, the tree is generated by creating a implication graph of the formula and determining a spanning tree in the graph [7][11].

An alternative implementation We have considered an alternative implementation based on some properties of the *intersection* merge operation on feature models. Instead of creating a new structure for the resulting feature model, we are interested on a solution that may reuse the structure of the input models.

Intersection Merge. According to Definition 5.2, the result of intersecting two feature models is a new model which semantics is the intersection of the valid configurations of the input models. Each configuration that is valid against the result is valid against both input feature models. On one hand, this means that these valid configurations can include only the features that exists on both models. The set of features in the result models is the intersection, i.e., a subset, of the features in each one of the input models. On the other hand, that means that the set of valid configurations of the result is a subset of the valid configurations of the input models.

The intersection of feature models can be defined using propositional logic. Acher et al. [2] named *Configuration Semantics* of an operation to the intended semantics of the result. For each operation, there is a *Formula Calculation* that may take the semantics of the input models and determine the configuration semantics of the result. According to the them, the formula calculation for the intersection merge is the following:

Definition 5.6. (Formula Calculation for Intersection Merge)

Given a feature model fm_r resulting of the intersection of two feature models fm_1 and fm_2 , the corresponding propositional formula ϕ_r is:

$$\phi_r = (\phi_1 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})) \wedge (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}))$$

where,

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

■

The Formula Calculation for the intersection combines the formulas of the input models but negates all the features that exists in one feature model that do not exist in the other. Negating these features, the resulting formula prevents the inclusion of these features in the configurations of the resulting model.

Note that the resulting feature model of an intersection can be one of the input feature models after adding some new constraints. Considering that the model fm_1 is equivalent to ϕ_1 , we can conclude that the resulting model fm_r may be the same input model fm_1 after adding the constraints for: $\phi_2 \wedge \text{not}(F_{fm_2} \setminus F_{fm_1}) \wedge \text{not}(F_{fm_1} \setminus F_{fm_2})$.

To intersect the feature models fm_1 and fm_2 , we can take the first model fm_1 and add to it a set of constraints: Algorithm 5.1 describes how to obtain the set of constraints to add. On one hand, in line 3, it adds clauses to negate all the features in fm_1 that do not exist in fm_2 . On the other hand, starting at line 3, it adds the constraints in ϕ_2 but negating the features of fm_2 that do not exist in fm_1 . It translates the feature model fm_2 into a corresponding propositional formula ϕ_2 , using an operation $encode(fm)$. The resulting formula comprises a set of clauses c . It is possible to determine the features in the clause using the operation $vars(c)$. The algorithm add the clauses c after negating the features $vars(c)$ that do not exist in fm_1 .

Algorithm 5.1 Obtaining the set of constraints to add to fm_1 to produce the intersection of fm_1 and fm_2

```

1: procedure DETERMINEINTERSECTIONCONSTRAINTS( $fm_1, fm_2$ )
2:    $\phi_r \leftarrow \{\}$ 
3:   for all  $f \in (\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})$  do                                 $\triangleright$  adding:  $not(F_{fm_1} \setminus F_{fm_2})$ 
4:      $\phi_r \leftarrow \phi_r \cup \{\neg f\}$ 
5:   end for
6:    $\phi_2 \leftarrow encode(fm_2)$                                         $\triangleright$  adding:  $\phi_2 \wedge not(F_{fm_2} \setminus F_{fm_1})$ 
7:   for all  $c \in \phi_2$  do
8:     for all  $f \in (vars(c) \setminus \mathcal{F}_{fm_1})$  do
9:        $c \leftarrow negateVariable(c, f)$                                 $\triangleright$  equivalent to:  $c \leftarrow (c \wedge \neg f)$ 
10:    end for
11:     $\phi_r \leftarrow \phi_r \cup \{c_i\}$ 
12:  end for
13:  return  $\phi_r$ 
14: end procedure

```

In the algorithm we want to add to fm_1 only constraints on features that exists on that model. We do not want to add a constraint $c \wedge \neg f$ where f is not part of fm_1 . Algorithm 5.1, in line 9, uses a $negateVariable(c, f)$ operation that takes the clause c , set the variable f in false and yields a new clause that do not include that variable. There are many techniques to simplify CNFs that can be used to achieve this.

Note that there are cases where the process to obtain the constraints to add can be simplified. For instance, consider two feature models fm_1 and fm_2 with the same set of features or where the set of features in the second is a subset of the features of first, i.e., $\mathcal{F}_{fm_2} \subseteq \mathcal{F}_{fm_1}$. All the features in the second feature model exist in the first, i.e., $\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1} = \emptyset$. In such cases, it is possible to take the constraints in the second feature model fm_2 and add it to the first without the need of negating variables.

Partial Intersection Merge¹. We are proposing an operation that combines two models enforcing the constraints of the second feature model only when a feature is selected. For instance, to enforce the constraints of a standard when the feature represented the standard is selected. The resulting model must include an optional selector feature and a set of constraints that enforces the new rules.

Considering the semantics presented above, we can determine the formula to calculate the *Conditional Intersection Merge* using the well-known formulas for the *intersection* and the *union merge* [2].

Definition 5.7. (Formula Calculation for Union Merge) Given two feature models fm_1 and fm_2 with the corresponding formulas ϕ_1 and ϕ_2 , the model fm_{\cup} resulting of the union merge $\llbracket fm_{\cup} \rrbracket = \llbracket fm_1 \rrbracket \cup \llbracket fm_2 \rrbracket$ is the formula ϕ_{\cup} such that:

$$\phi_{\cup} = (\phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})) \vee (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}))$$

where

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

■

We defined the Conditional Intersection Merge (see Definition 5.4) as:

$$\llbracket fm_r \rrbracket = \llbracket fm_1 \rrbracket \cup (\{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket))$$

Letting $fm_{1\cap 2} = \llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket$, the corresponding $\phi_{1\cap 2}$ is determined by

$$\begin{aligned} \phi_{1\cap 2} &= (\phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})) \wedge (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})) \\ &= \phi_1 \wedge \phi_2 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}) \end{aligned}$$

Letting $fm_{f\otimes 1\cap 2} = \{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket)$, the corresponding formula $\phi_{f\otimes 1\cap 2}$ is

$$\phi_{f\otimes 1\cap 2} = f_s \wedge \phi_1 \wedge \phi_2 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})$$

and the set of features of $fm_{f\otimes 1\cap 2}$ is:

$$\mathcal{F}_{f\otimes 1\cap 2} = f_s \cup (\mathcal{F}_{fm_2} \cap \mathcal{F}_{fm_1})$$

Note that $\mathcal{F}_{f\otimes 1\cap 2}$ includes the f_s feature and the features that are common to both models. In consequence,

$$\begin{aligned} \mathcal{F}_{f\otimes 1\cap 2} \setminus \mathcal{F}_{fm_1} &= f_s \cup (\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \\ \mathcal{F}_{fm_1} \setminus \mathcal{F}_{f\otimes 1\cap 2} &= \mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2} \end{aligned}$$

¹Appendixes A and B discuss on other alternative implementations for the Intersection and Partial Intersection merge operations.

Now, considering $\llbracket fm_r \rrbracket = \llbracket fm_1 \rrbracket \cup (\{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket))$, the corresponding formula ϕ_r is:

$$\begin{aligned}
\phi_r &= (\phi_1 \wedge \text{not}(\mathcal{F}_{f_{\otimes 1 \cap 2}} \setminus \mathcal{F}_{fm_1})) \vee (\phi_{f_{\otimes 1 \cap 2}} \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{f_{\otimes 1 \cap 2}})) \\
&= (\phi_1 \wedge \neg f_s \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})) \vee (\phi_{f_{\otimes 1 \cap 2}} \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})) \\
&= (\phi_1 \wedge \neg f_s \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})) \\
&\quad \vee (f_s \wedge \phi_1 \wedge \phi_2 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})) \\
&= \phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge (\neg f_s \vee (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}))) \\
&= \phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge (f_s \Rightarrow (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})))
\end{aligned}$$

Definition 5.8. (Formula Calculation for Conditional Intersection Merge) Given two feature models fm_1 and fm_2 with the corresponding formulas ϕ_1 and ϕ_2 , the model fm_r resulting of the conditional intersection merge can be represented with a formula ϕ_r such that:

$$\phi_r = \phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge (f_s \Rightarrow (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})))$$

■

This formula denotes that:

- (1) The resulting set of configurations contains all the configurations valid against the first model, i.e., ϕ_1 .
- (2) None of the configurations include features in fm_2 not included in fm_1 , i.e., $\text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$.
- (3) And, when f_s is included in the configuration, only the configurations valid against the second model that not include features of fm_1 not included in fm_2 are valid, i.e., $(f_s \Rightarrow (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})))$.

To implement the Conditional Intersection Merge, we must add to fm_1 the constraints that correspond to: $\phi_{c1} = \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$, $\phi_{c2} = (f_s \Rightarrow \phi_2)$ and $\phi_{c3} = (f_s \Rightarrow \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}))$

- $\phi_{c1} = \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$: In the original formula, there is not any feature $f \in (\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$. We can implement ϕ_{c1} without adding constraints. We must maintain the features of fm_1 without adding new features from fm_2 . It is possible that some clauses in ϕ_{c2} include these features. For each clause, we can take all the features in $(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$, set them to false and remove them before add the corresponding constraint to fm_1 .
- $\phi_{c2} = (f_s \Rightarrow \phi_2)$: ϕ_{c2} can be determined easily from the formula ϕ_2 that represents fm_2 . As mentioned before, in our implementation, we set to false and remove the variables that correspond to $(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$.
- $\phi_{c3} = (f_s \Rightarrow \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}))$: Here we must determine the features in fm_1 that do not exist in fm_2 . ϕ_{c3} can be determined easily from that set.

An algorithm to implement the *Conditional Intersection Merge* may add (1) an optional feature for the standard, (2) a set of constraints denoting that the feature representing the standard implies the constraints defined in the model of the standard (the second model) after setting to false and removing the variables that do not exist in the model of the domain (the first model), and (3) constraints denoting that the standard implies the negation of the features in the domain that do not exist in the standard.

Note that the Algorithm 5.1 obtains the constraints that determines the intersection of both models. That function can be used to determine which constraints to add to the first model for the *Conditional Intersection*. We must take each constraint c in that set and add to the feature model that the selector feature implies that constraint, i.e., $f_s \Rightarrow c$.

Algorithm 5.2 describes the process. First, in the line 4, it introduces a new optional child feature of root representing the selector feature. And then, in the line 4, it introduces additional constraints stating that the selector feature implies the constraints that implement the intersection of both models.

Algorithm 5.2 Conditional Intersection Merge of fm_1 and fm_2

```

1: procedure CONDITIONALINTERSECTIONMERGE( $fm_1, fm_2$ )
2:    $fm_r \leftarrow fm_1$ 
3:    $f_s \leftarrow \text{nameOfRoot}(fm_2)$ 
4:   addOptionalFeature( $fm_r, f_s$ )
5:   for all  $c \in \text{determineIntersectionConstraints}(fm_1, fm_2)$  do
6:     addConstraint( $fm_r, (f_s \Rightarrow c_i)$ )
7:   end for
8:   return  $fm_r$ 
9: end procedure

```

Note that the Algorithm introduces a new feature with the name of the root of the second model. Here we are assuming that this is the name of the standard and does not exists any other feature with the same name in the first model, i.e., the domain model.. The algorithm is presented in this way for the sake of simplicity. A real implementation may define a different strategy to name this new feature. For instance, it may be asked to the user.

5.3.2 Conditional Partial Intersection Merge

5.3.2.1 Definitions

We defined the *Conditional Partial Intersection Merge* operation to combine (1) a *feature model of a domain* with (2) a *feature model of an standard* that includes constraints for features in other domains. Intuitively, this operation obtains first the subset of the standard that is directly related to the domain and performs a *Conditional Intersection Merge* of both models.

Our *Conditional Partial Intersection Merge* relies on the *slice operation*. This operation, defined and formalized by Acher et al. [2][6], aims to obtain a subset of a feature model which valid configurations are also partial valid configurations of the original.

Definition 5.9. (**Slice of a Feature Model, $\prod_F(fm)$)** The slice is an operation that take a feature model fm and a set of features $\mathcal{F}_{slice} \subseteq \mathcal{F}_{fm}$ and yields a new feature model $fm_r = \prod_{\mathcal{F}_{slice}}(fm)$ which semantics represents the projected set of configurations of fm over \mathcal{F}_{slice} ,

$$\llbracket fm_r \rrbracket = \llbracket fm \rrbracket|_{\mathcal{F}_{slice}} = \{C \cap \mathcal{F}_{slice} \mid C \in \llbracket fm \rrbracket\}$$

■

We define the *Conditional Partial Intersection Merge* operation as follows:

Definition 5.10. (**Conditional Partial Intersection Merge**) The *Conditional Partial Intersection Merge* as an operation that takes two feature models fm_1 and fm_2 and produces a feature model fm_r that includes an optional Selector Feature f_s and which semantics $\llbracket fm_r \rrbracket$ represents the union of (1) the products of the first model $\llbracket fm_1 \rrbracket$ (where f_s is not included), and (2) the reduced product aggregation of the selector feature and the intersection of the products of the first model and the slice of the second model with the features in the first model, $\{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \prod_{\mathcal{F}_{fm_1}} \llbracket fm_2 \rrbracket)$.

$$\llbracket fm_r \rrbracket = \llbracket fm_1 \rrbracket \cup (\{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \prod_{\mathcal{F}_{fm_1}} \llbracket fm_2 \rrbracket))$$

■

Note that a valid configuration of the feature model for a standard may be not valid against the subset of the standard directly related to the features in the domain because the former may have more features and constraints than the latter.

Conflicts in Conditional Partial Intersection Merge. It produces a valid feature model if the first model is valid. The resulting model includes, at least, the same set of valid configurations of that model.

However, there are some circumstances where the resulting model is invalid or has inconsistencies:

- when the second model is invalid, the selector feature in the resulting model is a dead feature; and
- when the intersection of both models is an empty set, the selector feature in the resulting model is a dead feature.

Conditional Partial Intersection Merge using an existing selector.

As with the *Conditional Intersection Merge operation*, we have defined a variation of this operator that allow us to use an existing feature in the first model as the selector feature. The *Conditional Partial Intersection Merge using an existing selector* is an operation used when the feature f_s exists in fm_1 (i.e., $f_s \in \mathcal{F}_{fm_1}$) and should not be added in the process.

Definition 5.11. (Conditional Partial Intersection Merge using an existing selector) *The Conditional Partial Intersection Merge using an existing selector is an operation that takes (1) two feature models fm_1 and fm_2 , and (2) a selector feature included in the first model $f_s \in \mathcal{F}_{fm_1}$, and produces a feature model fm_r which semantics $\llbracket fm_r \rrbracket$ represents the union of (1) all the products of the first model $\llbracket fm_1 \rrbracket$ where f_s is not included, and (2) the set of products in the intersection of the products in the first model with the slice of products of the second model with the features in the first model, $(\llbracket fm_1 \rrbracket \cap \prod_{\mathcal{F}_{fm_1}} \llbracket fm_2 \rrbracket)$ that includes the f_s .*

$$\llbracket fm_r \rrbracket = \{C \in \llbracket fm_1 \rrbracket \mid f_s \notin C\} \cup \{C \in (\llbracket fm_1 \rrbracket \cap \prod_{\mathcal{F}_{fm_1}} \llbracket fm_2 \rrbracket) \mid f_s \in C\}$$

■

Conflicts in Conditional Partial Intersection Merge using an existing selector. This operation produces valid models if both feature models are valid and the selector feature is truly optional (it is neither full mandatory nor dead) in the first feature model, The resulting model represents, at least, the valid configurations of the first model that does not include the selector feature.

It may produce models that are invalid or has inconsistencies:

- when the selector feature is truly optional in the first model and the second model is invalid, the selector feature in the resulting feature model is a dead feature.
- when the selector feature is truly optional in the first model and the intersection of the semantics of both models is empty, the selector feature in the resulting feature model is a dead feature.

- when the selector feature is full mandatory in the first model and the second model is invalid, the resulting feature model is invalid.

5.3.2.2 Implementation

Conditional Partial Intersection Merge The *Conditional Partial Intersection Merge* receives two feature models fm_1 and fm_2 and yields a new feature model fm_r . It introduces to the first model a selector feature f_s and a set of constraints. In contrast to the previous operation, this operation does not include all the constraints in fm_2 that do not exist in fm_1 . It only introduces the constraints in fm_2 that affect directly the features that exist in fm_1 .

To implement this operation, we use the *slice* operation defined by Acher et al. [2] to remove safely all the occurrences of the features that do not exist in fm_1 . The slice operation takes a feature model and a set of features and yields a new feature model. We use that to remove from the second model fm_2 , the set \mathcal{F}_{remove} with the features in fm_2 that do not exist in fm_1 , i.e., $\mathcal{F}_{remove} = \mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}$. Algorithm 5.3 describes our implementation.

Algorithm 5.3 Conditional Partial Intersection Merge of fm_1 and fm_2

```

1: procedure CONDITIONALPARTIALINTERSECTIONMERGE( $fm_1, fm_2$ )
2:    $fm_r \leftarrow fm_1$ 
3:    $f_s \leftarrow \text{nameOfRoot}(fm_2)$ 
4:    $f_{remove} \leftarrow \mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}$ 
5:    $fm'_2 \leftarrow \text{slice}(fm_2, f_{remove})$ 
6:   addOptionalFeature ( $fm_r, f_s$ )
7:   for all  $c_i \in \text{determineIntersectionConstraints}(fm_1, fm'_2)$  do
8:     addConstraint( $fm_r, (f_s \implies c_i)$ )
9:   end for
10:  return  $fm_r$ 
11: end procedure

```

5.3.3 Combination of Domains and Standards

5.3.3.1 Definitions

To combine multiple feature models for domains and for standards, we defined an operation for *Combination of Domains and Standards*.

Definition 5.12. (Combination of Domains and Standards) The *Combination of Domains and Standards* is an operation that takes (1) a set of feature models for domains $dfm_1 \dots dfm_n$, and (2) a set of feature models for standards $sfm_1 \dots sfm_m$, and produces a feature model fm_r which corresponds to the conditional partial intersection of (1) the reduced product of all feature models for the domains, and (2) all the models for standards.

$$fm_r = ((dfm_1 \otimes \dots \otimes dfm_n) \otimes_{\text{partial} \cap} sfm_1 \otimes_{\text{partial} \cap} \dots \otimes_{\text{partial} \cap} sfm_m)$$

■

5.3.3.2 Implementation

Combination of Domains and Standards The combination of multiple feature models for domains and for standards is performed by using the previously defined operations. The combine operation (1) takes a set of feature models for domains dfm and a set of feature models for standards sfm , (2) perform a reduced product aggregation of all the models for domains, and (3) perform a conditional intersection merge of all the models for standards. Algorithm 5.4 describes the implementation.

Algorithm 5.4 Combination of Domains and Standards

```

1: procedure COMBINE( $dfm, sfm$ )
2:    $fm_r \leftarrow dfm_1$ 
3:   for all  $dfm_i \in dfm$  do
4:      $fm_r \leftarrow fm_r \otimes dfm_i$ 
5:   end for
6:   for all  $sfm_i \in sfm$  do
7:      $fm_r \leftarrow fm_r \otimes_{\text{partial} \cap} sfm_i$ 
8:   end for
9: end procedure

```

5.4 Discussion

5.4.1 Classification of the Operations

Thum et al. classifies the modifications on feature models according to the effect on the semantics [135]. Acher et al. uses this classification to characterize the operations on feature models [2]. According to them, operations can be classified into: (1) *refactorings*, when the semantics of the model remains the same; (2) *specializations*, when the resulting set of configurations is a subset of the original; (3) *generalization*, when the result is a superset; and (4) *arbitrary edit*, otherwise.

As pointed out by other authors [2], an *intersection merge* is a specialization. Given two feature models, the intersection produces a feature model which semantics, i.e., the set of valid configurations, is a subset of the semantics of the originals.

In contrast, although our operations introduces constraints to the feature models and they are named as “intersections”, not all of the them can be classified as specializations:

Conditional Intersection Merge is a *generalization*. The resulting feature model of the operation includes all the configurations of the first feature model and a set of new configuration according to the existent in the second model. The operation generalizes the first feature model.

Conditional Intersection Merge with an existing selector is a *specialization*. Considering that the first feature model includes the selector feature, it includes configurations that include the selector. The operation will produce a new model that restrict these configurations with the selector feature with additional constraints defined in the second model. The operation specializes the first feature model.

Conditional Partial Intersection Merge is a *generalization*. As with the *Conditional Intersection Merge*, this operation maintains the configurations of the first feature model and introduces new configurations with the selector feature. This operation generalizes the first model.

Conditional Intersection Merge with an existing selector is a *specialization* like the *Conditional Intersection Merge with an existing selector*. This operation specializes the first feature model.

We must mention that our operators may produce a feature model with the same semantics of first of their inputs depending on properties of the operands. For instance, if we have two feature models fm_1 and fm_2 , perform a *conditional intersection merge* of these models $fm_a = fm_1 \otimes_{\text{partial} \cap} fm_2$ and perform another *conditional intersection merge with an existing selector* of the result with the second feature model $fm_b = fm_a \otimes_{\text{partial} \cap} fm_2$, the result of the first operation is the same of the second, i.e., $fm_a = fm_b$

As another difference among the operations, the intersection merge is *commutative* regarding the *configuration semantics* but our operators are not. Our operations produce a feature model that maintain all the configurations of the first model and introduces new configurations, e.g., representing those products that comply with a standard. The order of the models determine which set of configurations will be maintained in the result.

5.4.2 Implementation of the Operations

The operations to merge feature models are implemented in two phases: (1) a *matching phase* that identifies which elements (e.g., features) in the models to be composed represent the same concept, and (2) a *merging phase* where the matched elements are grouped together and a new feature model is generated according to the intended semantics.

There are many strategies for these matching and merging phases:

For matching: the most simple strategy is to match features that have the same name or the same id. This strategy is plausible for models created by the same company or the same modelers where the names of the features can be standardized [2]. More sophisticated strategies such as correspondence tables or inter-model relationships [53] are required when the models are created by different suppliers and it is unlikely that the features get the same name.

For merging: it is possible to merge the models manipulating the structure of the models or by processing the semantics of them². The semantic-based operations usually overcomes the limitations of the structure-based ones and can be applied to a larger number of feature models [2][9]. For a given operation, a semantic-based operation consider, on one hand, its *configuration semantics* i.e., the intended set of valid configurations of the resulting model, and on the other hand, its *ontological semantics* i.e., the intended tree-based structure of the result.

We have developed an implementation of our operations using Epsilon and Java. This implementation *matches* the features that have the same *name* in the input models. At the moment we have been working with models created by a single modeller or by a modellers of a single company. The use of the *name* is plausible because we can define rules and reviews to check that all the features are consistently identified in all the models. However, we know that this may be a problem if we try to use models created by different companies and suppliers.

Regarding the merge of the models, we define the operators considering the semantics of the models. Instead of the strategy proposed by Acher et al. [2] where the structure of the model is created from an implication graph, we were

²A more extensive discussion exists in the Section 3.2.2 in the Background chapter and in the works of Acher et al. [2][9]

interested on reusing the structure of one of the input models for two reason: On one hand, we believe that modelers feel more comfortable with models that are similar to those that they have created. On the other hand, we consider that adding constraints instead of generating the tree may result more efficient. We are exploiting the semantics of the intersection that allow obtain a result by adding constraints. The same strategy of just adding constraints may be not appropriated to implement other operators.

We must mention that we rely on the implementation of the *slice* included in FAMILIAR. FeatureIDE has another implementation. Recently, some work has been done to optimize these implementations [86]. These optimizations may be a source of inspiration for improving our work.

5.5 Conclusions

In this chapter we presented three operations we defined to combine *feature models for domains* and *for standards*. Instead of enforcing the constraints in the standard with the existing operations, these new operators allow us to define the standards as optional elements. The constraints defined in the standard are enforced only when the standard is selected by the user during a configuration.

The operations we defined are:

Conditional Intersection Merge An operation that takes a feature model for a domain, a model for a standard and a common selector feature that represents the standard. The operations yields a new model where all the valid configurations of the first model that do not include the selector, i.e., the standard, are valid; and where only the configurations that contain the standard and are in both the domain and the standard are valid. This means that the configurations that contain the selector feature but are not valid against the feature model for the standard are invalid in the resulting model.

This operation can be used to combine a feature model for a domain with a model for a standard that only includes features of that domain. It cannot be used in scenarios where the standard applies to many domains and, therefore, the model includes more features than the existing in the model for the domain.

Conditional Partial Intersection Merge An operation that takes a feature model for a domain, a model for a standard and a common selector feature representing the standard and yields a new feature model. In the new feature model, all the valid configurations of the model of the domain that do not include the selector feature are valid too. In addition, all the valid configurations of the first model that include the selector and the subset of features included in both models is valid against the second model, are valid in the resulting model.

This operation is used to combine a feature model for a domain with a feature for a standard that includes features of other domains.

Combination of models for domains and standards An operation that takes one or more feature models for domains, inter-domain constraint sets and feature models for standards and yields a new feature model that represent the subset of the reduced product of the models for the domains that satisfies all the rules defined in the constraint sets and that enforces the constraints of the standards when the corresponding selector features are included in the configuration.

This operation is used to produce a feature model for a product family according to a set of models created by multiple stakeholders.

This chapter includes examples to illustrate the need of the new operators, and presents a formal definitions of these operations

Chapter 6

Deriving Configuration Systems from Models representing multiple Domains and Standards

The development of *Product Configuration Systems* is not an easy task. A recent study [1] evidences the multiple issues that exist in contemporary web Product Configurators: Many of these tools *omit constraints* or *erroneously update the user interface*. In addition, some tools *do not provide advanced options* to undo or to reconsider the value of a decision previously made. Furthermore, only a few notify users when a change affects already made decisions and the user has to revise some of the choices.

This chapter presents our proposal to tackle these problems: On one hand, a model-based approach to derive automatically, from a given feature model, a Product Configuration system. On the other hand, a set of runtime components to process user decisions during the configuration process.

Rest of this chapter is organized as follows: Section 6.1 describes how we use feature models and a user-interface definition models (*UI-definitions*) to derive the user interface. Section 6.2 details the model-transformations we use to process user decisions and update the user interface at runtime. Section 6.3 discusses our approach and compares it with other proposals and libraries, and Section 6.4 concludes the chapter.

Contribution: As main contributions, this chapter (1) introduces a model-based approach to engineer user interfaces for Configuration Systems that support these operations, and (2) describes how the models can be used at runtime to process user decisions, determine the impact of these decisions, and update the user interface accordingly

6.1 Deriving the User Interface

This section presents our proposal for model-driven derivation of configuration systems. In our approach, each Configuration System is created from a set of models. On one hand, the options and constraints are defined in a feature model. For product families specified using multiple domain- and standard-specific models, it is necessary to combine first these models into a single feature model. On the other hand, the elements of the user interface are defined in a *UI-definition model*. To explain how we derive the user interface, we use a simple Car Configurator example.

6.1.1 Example

Here we present a Configurator inspired on the Car Configurator offered by Audi in its web site¹. The example configurator depicted in Figure 6.1 supports only a few of the configuration options to configure the Audi A4 and A6 models: the *A4Saloon*, *A4Avant*, *S4Saloon*, *S4Avant*, *A6Saloon* and *A6Avant* models.

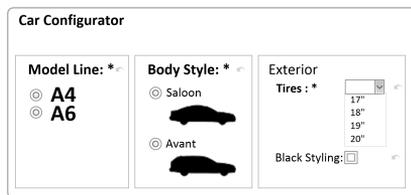


Figure 6.1: User interface of our Car Configurator

In our example, a user configuring an *Audi Car* must select a *ModelLine*, a *BodyStyle*, and some *Exterior* features. The *ModelLine* can be one of *AudiA4* or *AudiA6*. The *BodyStyle* can be either *Saloon* (a.k.a., Sedan) or *Avant* (a.k.a., Station-Wagon). In the *Exterior*, the user must select the type of *Tires* (one of *17inches*, *18inches*, *19inches* or *20inches*). A user can, optionally, select a *BlackStyling* package.

However, some features cannot be selected at the same time. Regarding the car models, the selection of the *AudiA4* model line implies the selection of one of the *A4Saloon*, *A4Avant*, *S4Saloon* or *S4Avant* models. Selecting the *AudiA6* model line implies one of the *A6Saloon* or *A6Avant* models. In turn, the *Saloon* body style implies the selection of one of the *A4Saloon*, *S4Saloon* or *A6Saloon*. And finally, selecting the *Avant* body style implies the selection of *A4Avant*, *S4Avant* or *A6Avant*.

Regarding the other features: selecting *A4Saloon* or *A4Avant* models implies the use of *17inches* tires. The *S4Saloon* and *S4Avant* models can use *18inches* or *19inches* tires, the *A6Saloon* and *A6Avant* only use *19inches* or *20inches* tires. Finally, only models in the *AudiA4* model line can include the optional *BlackStyling*.

¹<http://configurator.audi.co.uk/>

6.1.3 Mapping Configuration Options to User Interface

We use a *UI-definition model* to describe how the configuration options (i.e., the features) must be presented in the user interface.

Each feature in the feature model can be displayed using some type of widget [1]. The developers of the Configuration System must map each feature to a type of widget. The following list describes the best practices defined by Boucher et al. [24] on which widgets use according to the type of each feature.

Or-groups can be presented as a list-box or multiple check-boxes that allow multiple selections.

Alternative groups can be presented using radio buttons or combo-boxes that prevent multiple choices.

Mandatory features can be hidden (because they must be selected all the time) or presented as a notification text.

Optional features can be presented using a check-box where the user can select (or not) an option.

Besides a specific type of widget, each feature is presented using their name, a descriptive text or an image. Figure 6.3 summarizes the mentioned alternatives.

Or-Groups	Feature Group: <input checked="" type="checkbox"/> Feature Name <input checked="" type="checkbox"/> Feature Name
	Feature Group: Feature Name Feature Name Feature Name
Alternative Groups	Feature Group: <input type="checkbox"/> Feature Name <input type="checkbox"/> Feature Name
	Feature Group: <input checked="" type="radio"/> Feature Name <input type="radio"/> Feature Name
Optional Features	Feature Name: <input checked="" type="checkbox"/>

Figure 6.3: Summary of widgets to represent features

A *UI-definition model* specifies the presentation for each feature. For instance, consider the feature model in Figure 6.2 and the example product configurator in Figure 6.1. There are many alternative groups: the *BodyStyle* is represented with radio buttons with images, the *FuelType* as radio buttons with texts, and the *Tires* as a combo-box.

Table 6.1 presents the mappings for our example configurator. Note that the type of widget must be defined for *FeatureGroups* and *Features*. For instance, the *ModelLine* feature group uses a Radio Button widget. All the *GroupedFeatures* in that group are displayed using the same type of widget.

Also note that the representation for all the elements visible in the UI must be defined. While some are represented using a description (i.e., the *AudiA4* feature) and other are presented with an image (i.e., the *Saloon* feature).

Table 6.1: Feature model to widget mappings for our Car Configurator

	Element Type	Widget Type	Representation
ModelLine <i>AudiA4</i> <i>AudiA6</i>	FeatureGroup GroupedFeature GroupedFeature	RadioButton	desc="Model Line" desc="A4" desc="A6"
BodyStyle <i>Saloon</i> <i>Avant</i>	FeatureGroup GroupedFeature GroupedFeature	RadioButton	desc="Body Style" img="saloon.png" img="avant.png"
Tires <i>17inches</i> <i>18inches</i> <i>19inches</i> <i>20inches</i>	FeatureGroup GroupedFeature GroupedFeature GroupedFeature GroupedFeature	ComboBox	name desc="17" desc="18" desc="19" desc="20"
BlackStyling	Feature	CheckBox	name

6.1.4 Deriving the Configurator User Interface

We derive the user interface from the feature model and the UI-definition model. Basically, a model-to-text transformation generates the HTML code traversing the elements in the feature model and checking the widget to use in the UI-definition model.

The appearance of the widgets depends on the type of feature, the state of the feature and the type of the widget used to display. For instance, Figure 6.4 presents alternative presentations for the *Tires* and the *BlackStyling* in our example Configurator. According to the mappings defined above, the *Tires* must be presented using a combo-box with the *17inches*, *18inches*, *19inches* and *20inches* features. Because *Tires* is a mandatory feature, it is presented using an asterisk as a visual indicator. The *BlackStyling* is displayed using a check-box and, because it is not mandatory, without the mentioned visual indicator.

The state of the feature affects how it is presented. When the user has not selected any option (i.e., the feature has the *Undecided* state) the widgets are presented in blank and with all the options available. Figure 6.4b shows the presentation when the user selects the *17inches* feature, that feature changes its state and is presented as selected in the user interface. Because the user can undo their decisions in our configurators, the corresponding *undo* icon is enabled. In addition, all the other options for *Tires* remains enabled to allow users change their decision.

The widgets are presented differently when they have been selected or disabled by the Configurator. For instance, our configurators hide the options disabled automatically. Consider a situation where the Configurator Logic has selected automatically the *19inches* option (i.e., has the *Configurator Selected*

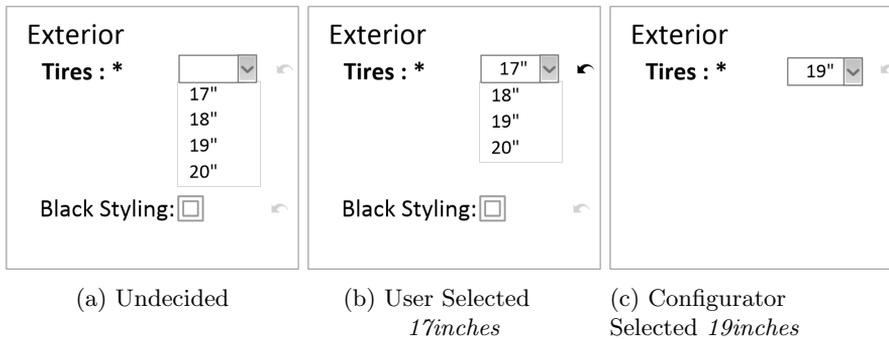


Figure 6.4: Example visualizations of the *Tires* Feature Group and the *Black-Styling* feature

state) and disabled the *BlackStyling* (i.e., has the *Configurator Disabled* state). As shown in Figure 6.4c, the *19inches* is presented selected and all the other options for *Tires* and the *BlackStyling* are hidden from the user interface.

Figure 6.5 presents alternative presentations for the *ModelLine*. This feature group is mapped to be presented as a radio-button with the *AudiA4* and *AudiA6* options. When the feature has the *Undecided* state, the widgets are presented with both options unselected. When the user selects *AudiA6* (i.e., has the *User Selected* state), that option is presented as selected and with an enabled *undo decision* icon. The *AudiA4* option remains enabled to allow users change their decision. If the Configurator selected the *AudiA4* option (i.e., has the *Configurator Selected* state) and disabled the *AudiA6* (i.e., has the *Configurator Disabled* state) the former is presented selected while the latter is hidden from the user interface.

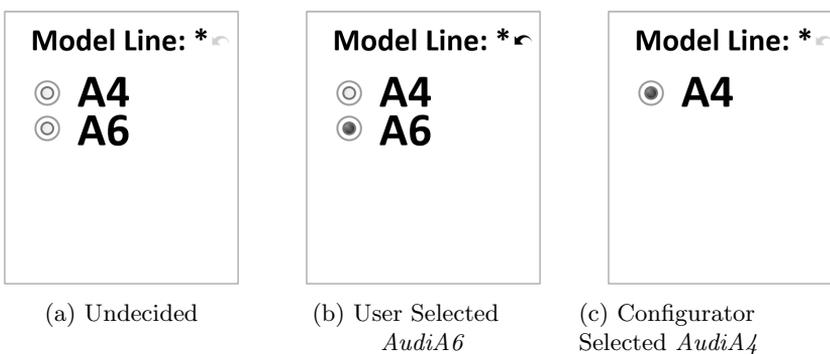


Figure 6.5: Example visualizations of the *ModelLine* Feature Group

In our approach, the HTML-code for the user interface includes a Javascript application to update the user interface according to the changes of the state of the features during the configuration process.

6.2 Processing User Decisions during Configuration

Our Configuration Systems are web applications where, each time the user performs an action, the *User Interface* invokes a *User Decisions Processor* to validate and propagate the user decision. Internally, each *User Decisions Processor* uses a *Reasoner library* (e.g., a CSP or a SAT solver) to determine which other features must be selected or disabled. After processing the decision, the User Interface in the browser is updated to reflect which features were selected or disabled automatically. In addition, our Configuration Systems maintain the state of the configuration process (i.e., the configuration steps the user has carried out) in order to implement advanced options such as undoing and revising decisions.

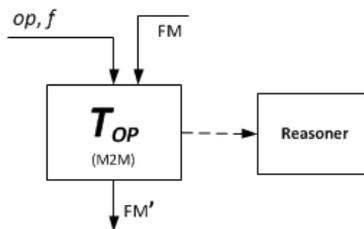


Figure 6.6: Schematic of the transformations to process user decisions

A *User Decisions Processor* is a model-transformation that processes user decisions and updates the feature models and the configuration steps consequently. Figure 6.6 shows schematically the processor for an operation op : It is a transformation T_{op} that is invoked when an operation op on a feature f is triggered from the UI; this T_{op} uses the current feature model FM , calls the *Reasoner* and produces a new feature model FM' .

We have defined processors for user decisions such as (1) *selecting* and *deselecting* a feature, (2) *undoing* a decision, (3) *revising* a decision, and (4) *forcing the change* a decision removing any other decision that may conflict.

6.2.1 The Reasoner

A *Reasoner* is an external library used by the User Decision Processors. It may be one of many well-known libraries for processing feature models. We have defined an API and a set of adapters to rely on a single interface independently of the library.

The Reasoner API comprises operations on a feature model to:

Validate a Configuration, i.e., to determine if a selection of features, in a feature model, do not violate any constraint defined in the model.

Propagate a Configuration, i.e., given a set of features and a selected feature, to determine which additional set of features must be automatically selected and disabled, and

Detect Conflicts, i.e., given an invalid partial configuration, to determine the subsets of features that have conflicts (i.e., the subsets of features that cannot be included at the same time in the configuration).

We defined adapters to use this API with SPLOT [93], FAMA [138], and FAMOSA [30] (our own implementation of these operations).

6.2.2 The User Decision Processors

Each time a user makes a decision, a processor is invoked to validate the decision, determine the set of selected and deselected features, and update the user interfaces consequently. Next sections describe the decision processors for the *Select*, *Undo* and *Force Change* operations.

6.2.2.1 Select and Deselect Feature Operation

Figure 6.7 shows a feature selection operation. In this case, the user selects the feature *BlackStyling*, and, as a consequence, a propagation occurs. In the figure, the *AudiA6* is disabled and the *AudiA4* is automatically selected because the *BlackStyling* only can be applied to *AudiA4* cars. In addition, the *20inches* are disabled because these cars do not use that tires.

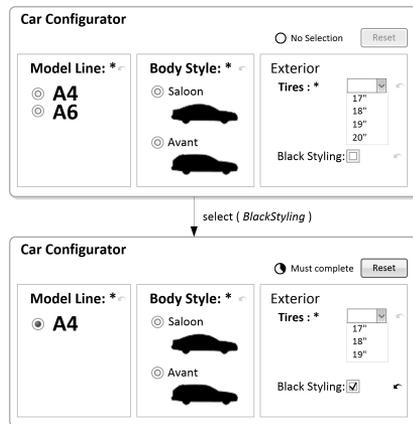


Figure 6.7: Example user interface sequence for a *Select* operation

In the example, the resulting state of *BlackStyling* is *User Selected*, while the state of *AudiA4* is *Configurator Selected* and the *AudiA6* is *Configurator Disabled*.

The transformation for the **Select** operation invokes the *Propagate* operation of the *Reasoner*. This transformation takes the selected feature, invokes the reasoner and obtains the other features that must be selected or disabled to abide the constraints in the feature model. The transformation changes the state of the affected features accordingly: The feature selected by the user

changes its state to *User Selected*, the features selected by the propagation change to *Configurator Selected* and those disabled to *Configurator Disabled*.

In our Product Configurators, this transformation stores in the feature model not only the new states but also information about the *Configuration Step*. For each selection, it keeps a history of which feature was selected by the user and which others were selected or disabled by the Decision Processor.

6.2.2.2 Undo Feature Operation

The Undo operation retracts a previous user decision in the Configuration Session. It occurs when the user wants to remove a feature from the Configuration and make that feature undecided. This operation does not generate any conflict because a subset of a valid partial configuration is also a valid partial configuration. However, this operation has to recalculate the propagation.

Consider the example depicted in Figure 6.8. The user has selects first *BlackStyling* (like the previous example) and then *19inches*. Later, he undoes the *BlackStyling*. Instead of returning the feature model to the state before *BlackStyling* (i.e., the initial screen), the configurator applies all the user selections except *BlackStyling*

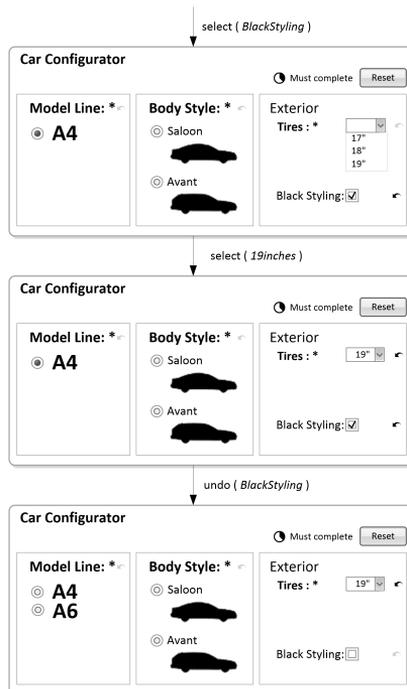


Figure 6.8: Example user interface sequence for a *Undo* operation

This transformation must consider all the configuration steps performed by the user except that to be undone. It may take initializes an empty configu-

ration, include the remaining user selections and invoke the *Propagate* of the Reasoner.

6.2.2.3 Revise Feature Operation

The Revise operation modifies a previous selection. It occurs when the user wants to deselect (i.e., to disable) an already selected feature. A new propagation has to be performed considering the change. This propagation may cause conflicts because the new selection may contradict other user decisions.

Figure 6.9 presents an example of a conflict caused by revising a decision. In this sequence, the user first selects the *AudiA4* and then the *17inches* tires. These selections are consistent because some *AudiA4* cars use these tires. However, later the user wants to change the model from *AudiA4* to *AudiA6*. There is a conflict because none of the *AudiA6* use *17inches* tires.

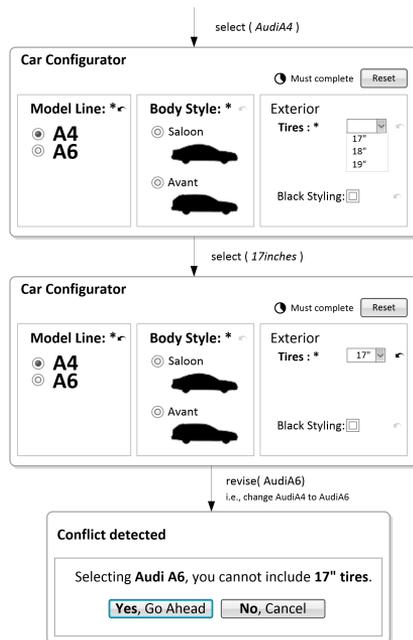


Figure 6.9: Example user interface sequence for a *Revise* operation

This transformation takes all the configuration steps performed by the user except that to be changed and invokes the *DetectConflicts* operation. If there is a conflict, a notification is issued to the user. If the change does not cause conflicts, the operation invokes the *Propagate* operation with all the user selection except that to be changed.

6.2.2.4 Force Change Feature Operation

When a conflict occurs after revising a decision, the user may confirm or cancel the change. The Force operation is invoked when the user confirms the change. It removes all the decisions that cause a conflict. It may invoke the *Detect Conflicts* operation to determine which features cause conflicts, remove these conflicting features from the configuration, and invoke the *Propagate* operation. At the end, the configuration includes all the features previously selected by the user except those in conflict with the change.

6.2.3 Our Implementation of User Decision Processors

We have implemented two sets of processors for user decisions. As other libraries, our processors use off-the-shelf solvers to perform tasks such as the validation and propagation of user decisions. However, in contrast to the others, our implementations maintain information of previous decisions to allow users retract decisions during the configuration process. While one implementation uses that information to determine which features consider at the moment of invoking the solvers, the other uses that information to process some types of decisions using with fewer calls to the solvers.

6.2.3.1 Representation of the Configuration State.

A configuration system keeps a track of the options selected and deselected by a user. This *state of the interactive configuration* is used to determine which other features select or deselect automatically. Before presenting our algorithms, we present here a formal definition for the state of an interactive configuration based on the existing literature [75][139].

Definition 6.1 (State of an Interactive Configuration). *The state for an interactive configuration ic of a feature model fm is a 2-tuple $ic = (S, R)$ such that*

- $S \subseteq \mathcal{F}_{fm}$ is the set of selected features
- $R \subseteq \mathcal{F}_{fm}$ is the set of removed features

In addition, each ic must satisfy

- $S \cap R = \emptyset$, there is not a feature that is selected and removed at the same time.

Undecided Features: *In a state of configuration ic , $(\mathcal{F}_{fm} \setminus (S \cup R))$ is the set of undecided features.*

Partial Configuration: *A state of configuration ic is partial (or incomplete) when it has undecided features, i.e., $(\mathcal{F}_{fm} \setminus (S \cup R)) \neq \emptyset$*

Complete Configuration: *A state of configuration ic is complete when all the features are decided, i.e., $(\mathcal{F}_{fm} \setminus (S \cup R)) = \emptyset$*

■

6.2.3.2 Select and Deselect Feature Operation

Each time a user makes a decision and *selects* or *deselects* a feature, the Configuration System performs, at least, two operations: First, it validates if the selection performed by the user is valid. This is a backtrack-free system, i.e., user selections are valid because invalid selections are removed or disabled in the user interface. Second, the configuration system propagates the decision to determine which other features must be selected and deselected automatically.

Algorithm 6.1 describes how to validate a user decision using off-the-shelf solvers [21][75]. The software encodes the feature model into a propositional formula (e.g., using an *encode* operation), adds constraints that sets the value of selected variables to true and deselected variables to false, and uses the solver to determine if the formula is valid. The configuration (partial or not) is valid if the corresponding formula is valid.

Algorithm 6.1 Validating a configuration

```

1: procedure VALIDATECONFIGURATION(ic, fm)
2:    $\phi = \text{encode}(\text{fm}) \wedge \bigwedge_{s \in S} s \wedge \bigwedge_{r \in R} \neg r$ 
3:   return SAT( $\phi$ )
4: end procedure

```

During an interactive configuration, a user may select or deselect some of the undecided features. Algorithm 6.2 propagates decisions by checking if the other undecided features can be set to true or false after adding the constraints that correspond to the state of the interactive configuration [75][77][31][30].

Algorithm 6.2 Propagating Decisions

```

1: procedure PROPAGATEDECISIONS(ic, fm)
2:   for all  $f \in (\mathcal{F}_{\text{fm}} \setminus (S \cup R))$  do ▷ for each undecided feature  $f$ 
3:      $\text{CanBeTrue} \leftarrow \text{TestLiteral}(\text{ic}, \text{fm}, f)$ 
4:      $\text{CanBeFalse} \leftarrow \text{TestLiteral}(\text{ic}, \text{fm}, \neg f)$ 
5:     if  $\neg \text{CanBeTrue} \wedge \neg \text{CanBeFalse}$  then
6:       error “Internal error: Invalid model or configuration state”
7:     end if
8:     if  $\neg \text{CanBeFalse}$  then ▷ if the feature  $f$  cannot be set to false
9:        $S \leftarrow S \cup \{f\}$  ▷ select the feature  $f$ 
10:    end if
11:    if  $\neg \text{CanBeTrue}$  then ▷ if the feature  $f$  cannot be set to true
12:       $R \leftarrow R \cup \{f\}$  ▷ remove the feature  $f$ 
13:    end if
14:    ▷ remain undecided  $f$  if it can be true or false
15:  end for
16:  return  $\text{ic}' \leftarrow (S, R)$ 
17: end procedure

```

The algorithm computes for each undecided feature whether there are satisfying valuations having the corresponding variable set to true and false. There are four possible outcomes:

- **if the variable can be neither true nor false**, the feature cannot be selected or removed. That means that the ϕ formulae representing the state of the configuration is unsatisfiable, i.e., the feature model or the sets of selected and removed features are invalid before the user selection. Note that this must occur only if the implementation has some error. This breaks the assumption that ϕ is satisfiable at all the times because our configuration system is backtrack-free [75][77].
- **if the variable cannot be false**, the feature must be selected during the propagation.
- **if the variable cannot be true**, the feature must be removed.
- **if the variable can be both true and false**, the corresponding feature must remain undecided.

Note that the above algorithm must check if a undecided feature can be true or false. This is done by checking if the conjunction of the encoding of the partial configuration and the possible value for the feature [77][75]. Algorithm 6.3 shows the helper function.

Algorithm 6.3 Propagating Decisions Helper

```

1: procedure TESTLITERAL( $ic, fm, l$ )
2:    $\phi = encode(fm) \wedge \bigwedge_{s \in (S-l)} s \wedge \bigwedge_{r \in (R-l)} \neg r \wedge l$ 
3:   return SAT( $\phi$ )
4: end procedure

```

Optimizations to the Algorithm. In order to propagate a decision, Algorithm 6.2 may be inefficient because it calls the SAT solver twice for each undecided feature. This may be improved by reducing the number of calls to the solver [75][77]. There are many strategies that can be used. For instance,

- **Exploiting the Atomic Sets** i.e., sets of features that must be selected or removed at the same time [121]. It is possible to reduce the number of variables to evaluate by consolidating all the features in an atomic set in a single variable. This requires a pre-processing of the feature model to determine its atomic sets before the configuration process.
- **Exploiting Structural Properties of the Feature Models.** There are some properties that can be exploited to reduce the number of invocations to the solver. For instance, when a feature is removed, all the descendant features in the model are also removed [36][31]. When a feature is selected, all the ascendant features in the model must be selected too [31]. It is possible to determine the values for other features using the output of the solver and the structure of the feature model [90]. And
- **Ignoring non-relevant features.** When a feature is selected or removed, this decision affects only to the features that have some direct or

indirect relationship with it by the model structure or by its constraints. It is possible to pre-process the feature model to determine “propagation spaces” [90] and ignore these features that are not relevant to the verification [149].

6.2.3.3 Undo Feature Operation

To *undo* the last user decision, it is possible to take all the previous user decisions, remove the last one and propagate the remaining. However, this approach implies invoking the solver multiple times: one to determine the validity of the new decision, and two for each not-decided feature to propagate the decision. Instead of it, one of our implementations caches the results of processing the previous decisions (i.e., use memoization) to improve the performance.

Our approach requires information about the sequence of configuration steps performed by the user. In the literature, a *Configuration Path* is the sequence of configuration step that transform an feature model with all its features undecided into a fully-configured feature model [33]. We named *Configuration Sequence* to a valid sequence of configuration steps at any point of time of the process. That means that the last step of a configuration sequence may result into a partial configuration instead of a complete one.

Definition 6.2 (Configuration Step). *In an interactive configuration, the i -th configuration step cs_i is a 4-tuple $cs = (f, op, S, R)$ such that*

- $f \in \mathcal{F}_{fm}$ is the feature selected or deselected by the user,
- $op \in \{SELECT, DESELECT\}$ is the type of decision made,
- $S \subseteq \mathcal{F}_{fm}$ is the set of selected features after propagating the decision.
- $R \subseteq \mathcal{F}_{fm}$ is the set of removed features after propagating the decision.

■

Definition 6.3 (Configuration Sequence). *In an interactive configuration, a configuration sequence CS is a finite sequence $CS = cs_1 \dots cs_n$ of length $n > 0$, such that*

- each cs_i is a configuration step, and
- $\forall i \in \{2 \dots n\}$, each cs_i is the result of propagate the decision f_{cs_i} over the previous configuration step cs_{i-1} .

■

In our approach, Undoing the last decision is straight forward. We use the information of the configuration sequence to restore the interactive configuration to the previous state of the configuration step.

Algorithm 6.4 Undoing Last Decision

```

1: procedure UNDOLASTDECISION( $CS, fm$ )
2:    $CS' \leftarrow CS' - \text{last}(CS)$  ▷ Remove last decision
3:   return  $CS'$ 
4: end procedure

```

Undoing another decision (i.e., a decision that is not the last one) requires more processing. In such case, it is necessary to recalculate the configuration steps starting from the decision to retract in order to maintain the information of the configuration sequence. Suppose that you want to *undo* the i -th decision. It is necessary to restore the configuration to the state before the i -th decision (i.e., return to the step $i - 1$) and recalculate the states from $i + 1$ to n , applying again the corresponding decisions.

Algorithm 6.5 applies a configuration step cs to an existing configuration sequence CS . Depending on the operation in the step, it includes the feature in the step f_{cs} to the selected or removed features and propagates the decision.

Algorithm 6.5 Apply a Decision

```

1: procedure APPLYDECISION( $CS, fm, cs$ )
2:    $cs_n \leftarrow \text{last}(CS)$  ▷ Get the last state
3:   if  $op_{cs} = \text{SELECT}$  then ▷ If operation is Select
4:      $ic \leftarrow (S_{cs_n} \cup \{f_{cs}\}, R_{cs_n})$  ▷ Include feature  $f_{cs}$  into  $S$ 
5:   else ▷ If operation is Deselect
6:      $ic \leftarrow (S_{cs_n}, R_{cs} \cup \{f_{cs_n}\})$  ▷ Include feature  $f_{cs}$  into  $R$ 
7:   end if
8:    $ic' \leftarrow \text{PropagateDecision}(ic, fm)$ 
9:    $cs' \leftarrow (f_{cs}, op_{cs}, S_{ic'}, R_{ic'})$ 
10:   $CS' \leftarrow CS + cs'$  ▷ Append the configuration step
11:  return  $CS'$  ▷ Return the updated sequence
12: end procedure

```

Algorithm 6.6 shows how to undo the i -th operation. It takes the state before that step and applies the other operations. The procedure returns the sequence without applying the i -th configuration step.

Algorithm 6.6 Undoing Another Decision

```

1: procedure UNDODECISION( $CS, fm, i$ )
2:    $CS' \leftarrow \text{subSequence}(CS, 0, i - 1)$  ▷ Previous decisions
3:   for all  $cs \in \text{subSequence}(CS, i + 1, \text{size}(CS))$  do ▷ Further decisions
4:      $CS' \leftarrow \text{ApplyDecision}(cs)$ 
5:   end for
6:   return  $CS'$ 
7: end procedure

```

Note that the *Undo* operation does not produce errors. Removing a decision at any point of the configuration sequence cannot produce a conflict with the following decisions. We can propagate the following configuration steps omitting their validation. Conflicts arise when a decision is changed for other.

6.2.3.4 Revise Feature Operation

We define a *Revise* operation to modify a previous decision, i.e., to deselect a previously selected feature or to select an already deselected feature. In contrast to the *Undo* operation, the revise operation can cause an error because the changed feature can conflict decisions made after.

Changing the decision in a configuration step without taking care of the other decisions is very simple. It only requires to change the operation included in the configuration step. Algorithm 6.7 shows the procedure.

Algorithm 6.7 Change a Configuration Step

```

1: procedure CHANGESTEP( $cs$ )
2:    $cs' \leftarrow cs$ 
3:   if  $op_{cs} = SELECT$  then                                ▷ Change the operation
4:      $op_{cs'} \leftarrow DESELECT$ 
5:   else
6:      $op_{cs'} \leftarrow SELECT$ 
7:   end if
8:   return  $cs'$                                             ▷ Return the modified step
9: end procedure

```

To check if the modified configuration step produces any conflict, it is necessary to validate if the new operation can be applied as part of a configuration sequence. Algorithm 6.8 describes how to validate a decision without invoking the solver. Considering that the propagation updates the sets of selected features and removed features, a decision is valid when it aims to select a non-removed feature or deselect a non-selected feature.

Algorithm 6.8 Validate a Decision

```

1: procedure VALIDATEDECISION( $CS, cs$ )
2:    $cs_n \leftarrow \text{last}(CS)$                                 ▷ Get the last state
3:   if  $op_{cs} = SELECT$  then                                ▷ If operation is Select
4:     return  $f_{cs} \notin R_{cs_n}$                             ▷ Check if  $f_{cs}$  has not been removed
5:   else                                                    ▷ If operation is Deselect
6:     return  $f_{cs} \notin S_{cs_n}$                             ▷ Check if  $f_{cs}$  has not been selected
7:   end if
8: end procedure

```

Note that redundant decisions can be detected using a similar approach. A

redundant decision is a decision that aims to select an already selected feature or deselect an already removed feature.

Algorithm 6.9 Check Redundancy of a Decision

```

1: procedure ISREDUNDANTDECISION( $CS, cs$ )
2:    $cs_n \leftarrow \text{last}(CS)$  ▷ Get the last state
3:   if  $op_{cs} = SELECT$  then ▷ If operation is Select
4:     return  $f_{cs} \in S_{cs_n}$  ▷ Check if  $f_{cs}$  has been selected
5:   else ▷ If operation is Deselect
6:     return  $f_{cs} \in R_{cs_n}$  ▷ Check if  $f_{cs}$  has been deselected
7:   end if
8: end procedure

```

Algorithm 6.10 implements the *Revise* operation: To revise the i -th decision, it restores the configuration to the state before (i.e., returning to the state $i - 1$), changes the operation for that configuration step (e.g., changing the decision i from *SELECT* to *DESELECT* and vice versa), and recalculates the remaining states applying again the corresponding decisions (i.e., applying again decisions from $i + 1$ to n). Considering that before the revise operation all the decisions were valid, if there are conflicts, they exist between the new decision i and the further decisions between $i + 1$ and n . If some of these decisions cannot be applied, the procedure returns the original configuration sequence (i.e., without adding a step) to indicate that the decision cannot be changed without producing a conflict.

Algorithm 6.10 Revising a Decision

```

1: procedure REVISEDDECISION( $CS, fm, i$ )
2:    $CS' \leftarrow \text{subSequence}(CS, 0, i - 1)$  ▷ Takes previous decisions
3:    $cs' \leftarrow \text{ChangeStep}(CS_i)$  ▷ Change the i-th decision
4:    $CS' \leftarrow CS' + \text{ApplyDecision}(cs')$ 
5:   for all  $cs \in \text{subSequence}(CS, i + 1, \text{size}(CS))$  do ▷ Other decisions
6:     if  $\neg \text{IsRedundantDecision}(CS, cs)$  then
7:       if  $\text{ValidateDecision}(CS, cs)$  then ▷ If it can be applied
8:          $CS' \leftarrow \text{ApplyDecision}(CS', fm, cs)$  ▷ Add it to the sequence
9:       else ▷ If it cannot be applied
10:        return  $CS$  ▷ Error !! – return the unmodified sequence
11:      end if
12:    end if
13:  end for
14:  return  $CS'$ 
15: end procedure

```

The *Revise* operation a decision requires more processing than the *Undo* operation. It requires to validate the decisions made after the configuration

step to be changed. However, an advantage of keeping a track of the configuration steps is that, when a conflict occurs, we can reduce the set of decisions where the conflict must be searched.

6.2.3.5 Force Change Feature Operation

When a conflict exists, a *Force Change* operation can be used to modify a previous decision removing any further decision that may conflict.

Algorithm 6.11 implements the *ForceChange* operation: To change the i -th decision, it restores the configuration to the state before (i.e., returning to the state $i - 1$), changes the operation for that configuration step (e.g., changing the decision i from *SELECT* to *DESELECT* and vice versa), and recalculate the remaining states applying again the non-redundant and non-conflicting decisions (i.e., applying again a subset of the decisions from $i + 1$ to n).

Algorithm 6.11 Forcing the change of a Decision

```

1: procedure FORCECHANGEOFDECISION( $CS, fm, i$ )
2:    $CS' \leftarrow \text{subSequence}(CS, 0, i - 1)$            ▷ Takes previous decisions
3:    $cs' \leftarrow \text{ChangeStep}(CS_i)$                  ▷ Change the  $i$ -th decision
4:    $CS' \leftarrow CS' + \text{ApplyDecision}(cs')$ 
5:   for all  $cs \in \text{subSequence}(CS, i + 1, \text{size}(CS))$  do   ▷ Other decisions
6:     if  $\neg \text{IsRedundantDecision}(CS, cs)$  then
7:       if  $\text{ValidateDecision}(CS, cs)$  then           ▷ If it can applied
8:          $CS' \leftarrow \text{ApplyDecision}(CS', fm, cs)$    ▷ Add it to the sequence
9:       end if
10:    end if           ▷ Ignore redundant and non-applicable decisions
11:  end for
12:  return  $CS'$ 
13: end procedure

```

The *Force Change* operation is usually performed after a *Revise*. Its implementation may exploit the information obtained revising the decision to determine redundant and invalid steps without processing them again.

6.3 Discussion

6.3.1 Addressing issues

Our approach aims at tackling the issues identified in the Related Work, in the Section 3.3:

Correct handling of constraints we integrate in the Product Configurator a *Reasoner library* that implements proven algorithms and techniques to validate user selections (i.e., partial configurations) and propagate the decisions. Instead of reinventing the wheel, this reasoning is entirely

based on existing libraries to analyse feature models and configurations. In addition, because these libraries use directly the feature models representing the configuration options, none constraint included there will be omitted during the processing.

Correct updating of the user interface is achieved by the *User Decision Processor* that maintains the state of the configuration process and the models that map the features to user interface elements. Each time the user selects some features or revise some decisions, the User Decision Processor determines the new state for the features in the model and notifies the user interface the changes. In addition, when some erroneous behaviour is detected, the User Decision Processor notifies the interface to refresh all the options and prevent displaying erroneous information.

Ability to revise the decisions is achieved by the *User Decision Processor* that maintains not only the state of the configuration, but also a list of the configuration steps performed by the user. The User Decision Processor can use the information of these steps to undo and revise already made decisions. Basically, these logic can identify which features were selected by the user and which were propagated, and then calculate the propagation of the user decisions except those to undo or revise.

6.3.2 Comparison to other proposals

There are many other proposals to support user decisions during interactive configurations. Our proposal is based on the works from Janota et al. [75][74] and Mendoca et al. [90] on using SAT solvers to validate and propagate these decisions. However, it differs from the others in the implementation of the operations to *undo* and *revise* decisions. This section discusses the coincidences and differences with the related work.

Select and Deselect Operations Our implementation is based on the works from Janota et al. [75][74] and Mendoca et al. [90]. The main difference is that our implementation uses the state of the interactive configuration to keep the selected and removed features, and to determine easily the set of undecided features. The use of *stateful feature models* during configuration was explored before by Trinidad et al. [137][139] but their approach do not use the same algorithms.

Undo Operation Although this operation is not formally described in their work [92], Mendonca et al. have an implementation in the SPLOT library [93]. As our implementation, they keep a history of the user decisions and allow users to undo some of them. Our implementation improves their work by exploiting that history to avoid unnecessary validations of the decisions after the one that is undone.

Revise Operation This operation is also implemented in the SPLOT library [93].

Our implementation exploits the history of decisions to reduce the number of user decisions that must be analyzed to determine a conflict. It checks in the history which decisions are contrary to the change proposed by the user. If a conflict occurs, it is only among the subsequent decisions. It is possible to use algorithms for *explaining conflicts* [75][101] that use specialized SAT solvers to determine the corresponding *Minimal Unsatisfiable Sets*, i.e., the sets of constraints (and hence of decisions) that cause a conflict. In contrast to other proposals, we use the history of decisions to reduce the number of constraints where these algorithms will look for the conflict.

Force Change Our implementation uses the set of conflicting decisions detected on the *Revise* operation to force a change. Basically, it takes the state of the configuration at the moment previous to the decision to change and applies the new decision and all the subsequent decisions that do not conflict with it. This operation does not produce a new complete valid configuration. It is different than other operations that provide *diagnosis* [48] [147][146] or *fixes* [148] that look for an alternative configuration that maximizes the user preferences. Our approach forces a change by removing all the other decisions that conflict with it, and the user may require to make more decisions later. Other operations to auto-complete [75][77] or optimize [142] can be used to assist in these other decisions.

Our implementation is focused on “backtrack-free” configuration systems. These systems, instead of tolerating conflicts, try to prevent them. There are many works discussing the diverse strategies that exist to tolerate and fix conflicts during an interactive configuration [100][48]. A possible extension to our work is the implementation of some of these strategies that implement conflicts and the incorporation of operations to propose diagnoses and fixes.

6.3.3 Comparison to existing libraries

As mentioned in Chapter 3, there are tools and libraries such as *Familiar*, *FaMa* or *SPLOT* that can be used to analyze user provided configurations. A Configuration System can use them to validate user supplied configurations against a feature model and provide feedback. However, these libraries are not intended to provide complete support, through a user interface, to interactive configure a product.

In contrast to other libraries, our implementation of processors for user decisions supports operations to undo and revise decisions: *FAMA* does not support directly configuration processes and does not provide options to undo selections. *Familiar* provides operations to **deselect** and **unselect** options but does not maintain information of the user decisions during a configuration and cannot undo user decisions.

Similar to our implementation, *SPLIT* provides support for configuration processes. It includes a `ConfigurationEngine` class that can be used to maintain information of the user decisions, propagate decisions and undo the last decision. We provide a similar support using our User Decision Processors. However, in contrast to *SPLIT*, our implementation combines model-transformations using the *Epsilon* languages [85] with existing solvers and libraries, giving use more flexibility to create or modify the algorithms used to manipulate the models and configuration.

Familiar [7] defines a DSL that can be used to process feature models and configurations. For instance, A developer can use *Familiar* to create scripts that process feature models using diverse types of operations to merge and slice these models. Our Epsilon extension is similar. We have defined built-in objects for Epsilon that allow programmers create scripts and operations that load feature models, validate configurations and support interactive configuration processes. Contrary to *Familiar*, our Epsilon extensions allows to use multiple libraries and redefine existing operations. In addition, because we are extending Epsilon, we can use feature models and configurations directly in model transformations and code generation.

6.3.4 Evaluation

In order to test and evaluate our approach, we have implemented three case studies on Product Configurators²:

A Car Configurator based on the Audi Car Configurator. This is a superset of the example presented in this paper that includes 38 configuration options for all the cars in the AudiA4 and AudiA6 model lines.

The TREK's Bike Catalog implemented in the *SPLIT* website that includes 549 features and represents 376 products.

The DELL's Laptop Catalog implemented in the *SPLIT* website too, that includes 47 features, has 105 constraints and represents 2,319 products.

In addition, we have applied our approach to create configuration systems for Electrical Transformers in an industrial case study. The solution is used actually by tens of sellers and engineers to specify products and produce configurations that are the foundation to create designs, budgets and proposals for clients.

6.4 Conclusions

We have presented our approach to (1) specify product configurators using feature models for the configuration options and other additional models for the

²Web configurators and source code available at: <http://goo.gl/9YG8gN>, and <http://github.com/FaMoSA> respectively

user interface, (2) derive the user interface taking these models and transforming them into the corresponding HTML and Javascript code, and (3) support the behaviour of the configurator using a model-based layer that obtains the request from the user, use existing frameworks to validate the selected options and update the user interface using model-transformation at runtime.

Our approach facilitates the development of Product Configurators that support all the features and constraints defined in a feature model. It support many options to display the configuration options consistently. For instance, it supports combo-boxes, radio-buttons and check-boxes to represent the options. However, it does not support advanced product visualizations such as pictures obtained from a database or images modified according to the selected options.

Some complex products such as information systems, cars and industrial machines tend to be modeled using views or multiple models for each technical domain. These scenarios introduce new challenges to product configurators that must support collaborative configuration on the same domain or a simultaneous configuration of multiple domains. Future work is planned in supporting configuration processes with multiple stakeholders and multiple feature models.

Chapter 7

Experimental Implementation

Using Feature Models to represent products and create configuration systems requires a set of supporting tools. Different tools and libraries have been proposed in the last 20 years. There are well established tools like SPLOT¹, FAMA², Familiar³ and Feature-IDE⁴. However, these tools do not offer appropriate support for managing feature models for domains and for standards. While tools such as SPLOT allow engineers to create the models using a web-based graphical user interface, they do not support merging and analysing multiple models. Other tools such as Familiar, support processing multiple feature models but the software does not support our operations to merge feature models (i.e., those described in Chapter 5). We create and extend tools to support our approach.

This chapter describes: On one hand, our tools where multiple stakeholders can create and merge feature models for domains, feature models for standards and constraints. And, on the other hand, the tools we created to derive Configuration Systems and to process user decisions.

Contribution: As main contributions, this chapter presents (1) an extensible Epsilon runtime to process and transform models in web applications, (2) a set of Epsilon extension to load and process feature models using custom algorithms and using the existing SPLOT, FAMA and Familiar libraries, and (3) a set of Epsilon-based libraries and programs to process and analyse feature models for domains and for standards.

¹<http://www.splot-research.org/>

²<http://www.isa.us.es/fama/>

³<http://familiar-project.github.io/>

⁴http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

7.1 Overview

We created a set of tools and libraries to support our approach. In turn, these tools are based on our own extensions to Epsilon [84], a family of languages for model transformations. Figure 7.1 shows an overview.

The main components of our solution are the following:

- FM Studio**, a web-based application used to create feature models for domains and standards along inter-model constraints.
- FM Operators**, an Epsilon-based library that processes and merges these domain- and standard specific feature models. The above mentioned *FM Studio* depends on this library to process the diverse models of our approach.
- FM Configuration System Generator**, a set of model transformations that takes a feature model and a definition of the user interface and produces the corresponding Configuration System.
- FM Configuration Runtime**, a set of model transformations that processes the decisions made by a user during an interactive configuration. These model transformations implement the User Decision Processors described in Chapter 5.
- Web-based Extensible Epsilon Runtime**, a customized Epsilon runtime aimed to process Feature models and configurations. And
- FM abstraction layer**, a set of additional objects that can be included in our Epsilon runtime to load feature models and invoke off-the-shelf solvers.

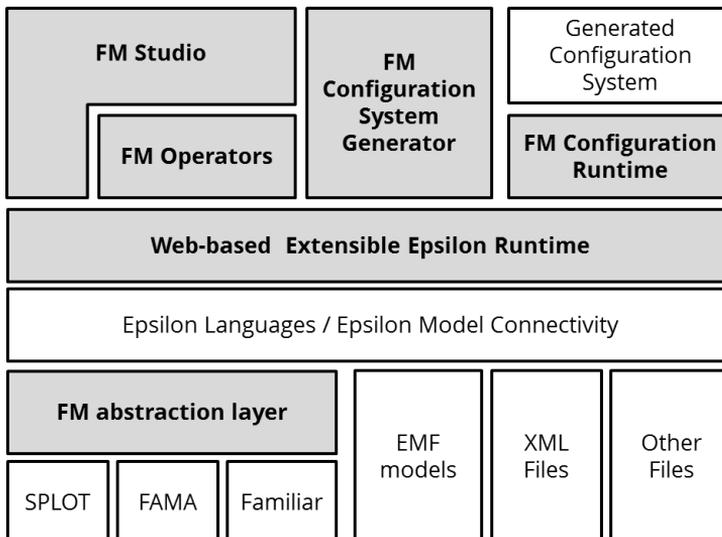


Figure 7.1: Overview of the developed Tools and Components

The next sections describe these components.

7.2 Tools to create Feature Models and Constraints

We built an integrated environment to create, edit and analyse the feature models and constraints. It was developed using Javascript⁵ and Angular⁶ for the front-end and a modified Epsilon runtime for the back-end. All the processing of the models is performed at the backend using our Epsilon extensions to process feature models.

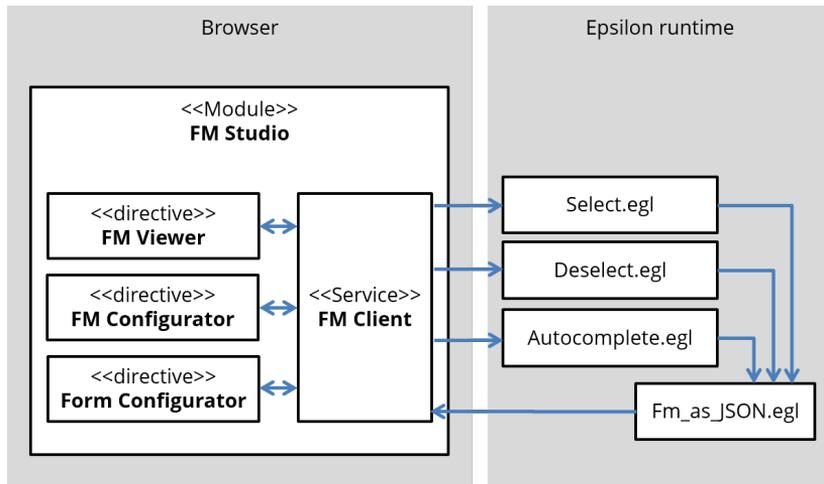


Figure 7.2: Excerpt of the architecture of our web-based editor for Feature Models and Constraints

Figure 7.2 shows the architecture of the application. On one hand, there are Angular modules and directives. They send requests to the Epsilon runtime in order to load, process and transform the models. A set of scripts in our runtime are the User Decision Processors mentioned in Chapter 6. These processors perform the tasks and return the updated models to the browser.

Figure 7.3 shows some screenshots. Once a user has entered into the application, the application may display a list with the projects and models she has access to. The user can edit the models and configure them in the application. She can also use other options to create constraint sets and to combine the models.

⁵<http://www.ecmascript.org/>

⁶<https://angularjs.org/>

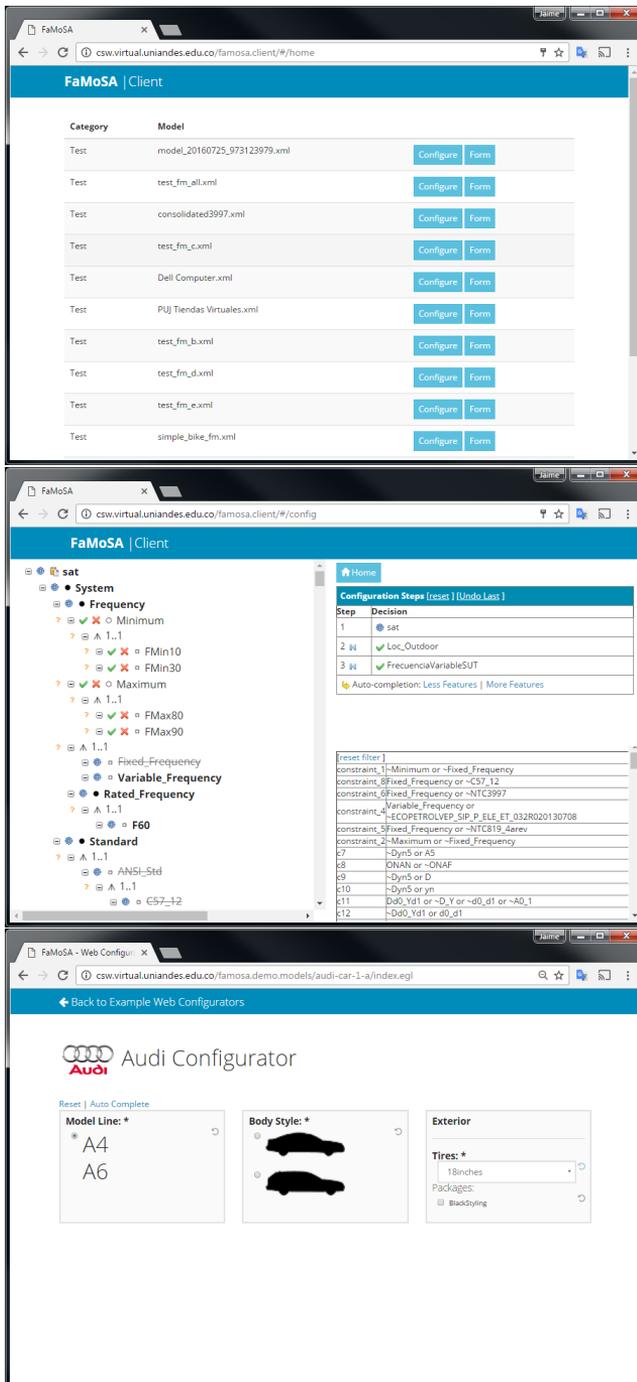


Figure 7.3: Screenshots of the web-based editor for Feature Models and Constraints

7.3 Operations to merge Feature Models

The operations to merge the Features Models and Constraint sets have been implemented as Epsilon libraries. An Epsilon script for model transformation, such as the one used in the above mentioned *FM Studio*, can use these libraries to merge domain- and standard specific feature models. A developer can use these libraries in other projects.

An example is the implementation of the merge operations described in Chapter 5. We defined a `famosa.eol` library with them. There are operations to perform traditional merge operations such as `Insert` and `IntersectionMerge`, and to perform our new operations such as `conditionalIntersectionMerge` and `partialConditionalIntersectionMerge`. Listing 7.1 shows an example Epsilon script that use the `conditionalIntersectionMerge` operation.

```
1 import "famosa.eol";
2
3 ModelManager.load('fm1', 'domain-model.xml', 'Splot');
4 ModelManager.load('fm2', 'standard-model.xml', 'Splot');
5
6 var fm_result = conditionalIntersectionMerge(fm1, fm2, 'standard');
7
8 ModelManager.save(fm_result, 'output.xml', 'Splot');
```

Listing 7.1: Example Epsilon Script using the `conditionalIntersectionMerge` operation

7.4 Model-transformations to derive Configuration Systems

We implement model-driven tools to derive configuration systems. On one hand, we created Angular components (known as directives) that can be used to display and configure feature models. Considering that all the processing is performed at the back-end, these components must interact with our Epsilon runtime and extensions. On the other hand, we created a model-driven approach to derive configuration systems from these models. Our tools may take a feature model and derive the corresponding configuration system.

7.5 An Extensible Web-based Epsilon Runtime

In order to implement the above tools, we created an extensible Epsilon runtime that allow us (1) to invoke model transformations from Javascript in a browser, and (2) to define additional extensions that invoke external libraries such as CSP and SAT solvers.

Basically, our *Epsilon Server runtime* processes requests from a browser. It extends an existing *EGL Servlet* that, depending on the request, executes an EGL script that may load a model and produce an output that is sent back to the browser. In the original *EGL Servlet*, the script may use a set of built-in objects to access the information of the request and to load EMF models, but not to load feature models and invoke external libraries. In addition, this *EGL Servlet* has a predefined processing cycle that loads these objects and executes the scripts but cannot load user-defined extensions and does not consider the security information in the request. We modify the processing cycle to introduce our extensions.

A new EGL-processing cycle Figure 7.4 shows the processing cycle of the existing EGL Servlet and the our modified runtime. The original Servlet uses the URL in the web request to determine the EGL Script to run. It creates a EGL runtime and introduces built-in objects for the web request, the web session and response. Then, it uses the runtime to execute the script. The resulting content is included in the response and sent back to the browser.

In contrast, our runtime uses the URL not only to determine the EGL script to run but also the configuration data for its execution. The configuration includes, for instance, which web applications can invoke the scripts. The runtime checks the request and the security configuration to abort or permit the execution. It also uses the configuration to determine which built-in objects and model loader drivers inject into the runtime. Finally, the runtime sets security headers in the response, executes the script, and appends the result of the script to the response.

Built-in Objects for Additional Functionality Our Epsilon runtime allows the definition of built-in objects that Epsilon scripts can use. The original *EGL servlet* includes some built-in objects to access information managed by the web server, such as the data in the user requests and sessions. We define a Java abstract class, `EpsilonBuiltinObject` that the classes for the built-in objects must implement.

Figure 7.5 shows a diagram presenting the `EpsilonBuiltinObject` class and an `Example` built-in object. The abstract `getName()` method of the super-class must be implemented in the `Example`. There are other methods such as `setContext`, `setSessionProvider` and `setFileNameConverter`, aimed to set the environment where the built-in object, that are final. The `Example` built-in object inherits those methods.

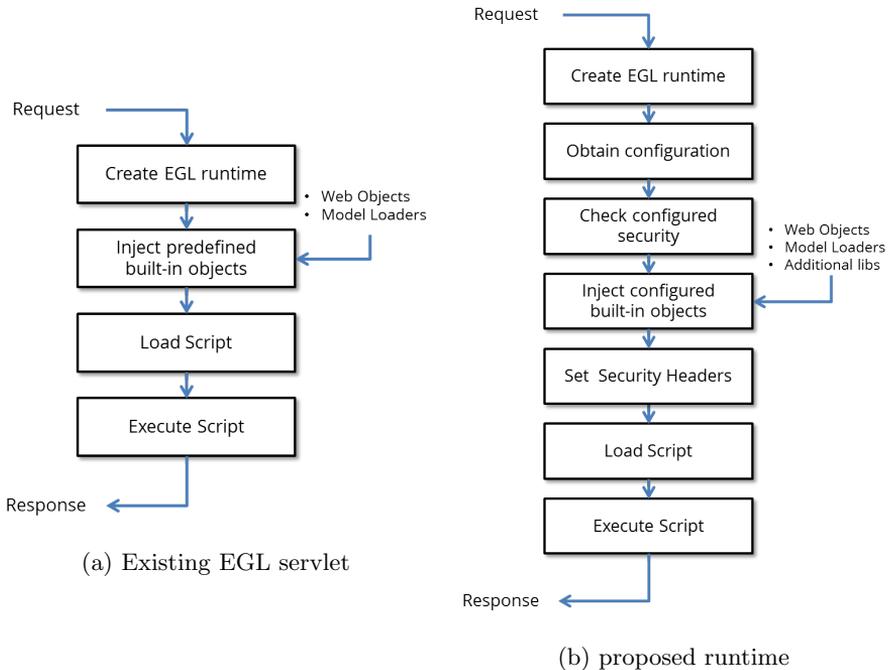


Figure 7.4: Processing Cycle of the Epsilon runtime

Listing 7.2 shows an example class for a built-in object. That `Example` class extends the `EpsilonBuiltinObject`. It includes a method named `sayHello` that receives a `String` and produces a greeting using that input.

```

1 public class Example extends EpsilonBuiltinObject {
2
3     public String getName() {
4         return "example";
5     }
6
7     public String sayHello(String name) {
8         return "Hello " + name + " !!";
9     }
10
11 }

```

Listing 7.2: Example implementation of a Epsilon Built-in Object

If these classes for built-in object are included in a proper java library file, i.e., a `.jar` file including a `META-INF` directory with the extension metadata, our runtime detects these classes and creates the corresponding instances for each user request. For instance, given the above mentioned `Example` class, our

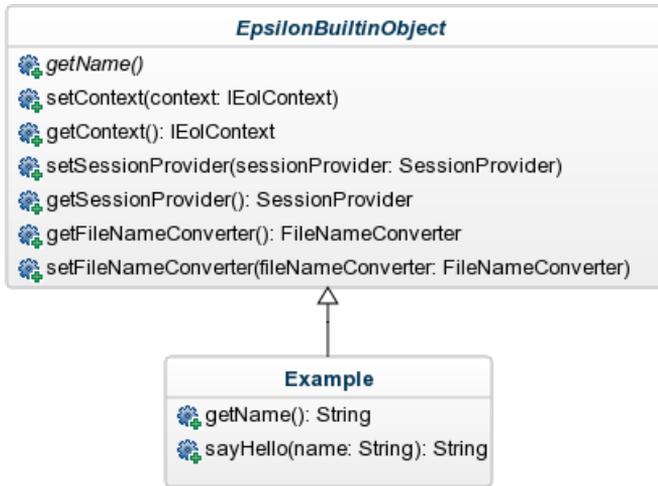


Figure 7.5: Class Diagram for the EpsilonBuiltinObject

runtime creates an `example` instance for each request. The name of the instance is determined by invoking the `getName()` method of the class. Listing 7.3 shows a script using that object. It uses the automatically instanced `example` object, by invoking the `sayHello` method.

```

1 // uses the "example" built-in object by invoking the sayHello method
2 var x = example.sayHello("Alice");
3 x.println();
4
5 example.sayHello("Bob").println();
  
```

Listing 7.3: Use of the `example` Epsilon Built-in Object

We use these built-in objects to allow scripts access to external libraries. For instance, they are used to invoke functions on libraries such as SPLOT and FAMA. In addition, we use them to implement our improved model loader.

An improved Model Loader The original *EGL servlet* includes a `Model-Manager` built-in object that allow scripts load Eclipse EMF Models. It cannot be used to load other types of files, such as XML files or spreadsheets. Considering that third-party libraries store feature models using file formats different from EMF, we decide to create a more flexible way to load the models.

Our `ModelManager` defines a consistent way to load models. It hides the complexity of the existing Epsilon Modelling Connector that uses different classes and methods depending of the type of model or file to load. In consequence, a developer can use the same method to load different types of models. Listing 7.4 shows an example using the same `load` function to load different types of models. For instance, it loads an EMF mode in the line 2 and an XML file in the line 5. The line 8 loads an XML file with an optional “hint” used internally to determine which driver must be used.

```

1 // load an EMF Model
2 ModelManager.load("Sample", "Graph.ecore");
3
4 // load an XML file (as XML)
5 ModelManager.load("Fm1", "dell-computer.xml");
6
7 // load an XML file (as a SPLIT file)
8 ModelManager.load("Fm2", "dell-computer.xml", "Split");
9
10 // load an Excel file
11 ModelManager.load("Tests", "dell-data.xls");

```

Listing 7.4: Use of the `ModelManager` Built-in Object

We defined a *ModelLoader* interface to integrate our *ModelManager* to existing or new Epsilon Model Connectivity drivers. Figure 7.6 shows a class diagram. A *ModelLoader* class is required for each type of model or file. It has a `canLoad` method to determine if it can load a specific file, `load` to load a model and `isHandlingModel` to determine if it has loaded a specific model. The `ModelManager` has a collection of loaders. Each time a script tries to load a non-EMF model, it calls to each loader to determine which one is able to load the model. The script can provide, as a hint, the name of the preferred loader.

7.6 Epsilon extensions to process Feature Models and Constraints

We created extensions to integrate our web-based Epsilon runtime to existing libraries and tools that process feature models. In concrete, we (1) create a set of model loaders for Feature Models, (2) define a unified API for reasoning and configuring these models, and (3) create a solution that allow us to execute additional Epsilon scripts when custom algorithms are provided.

Model loaders for Feature Models We created model loaders for feature models in SPLIT, FaMa and FeatureIDE. Our loaders use the Java libraries provided in the first two projects. An additional loader, based on an XML parser, is used to load the FeatureIDE models.

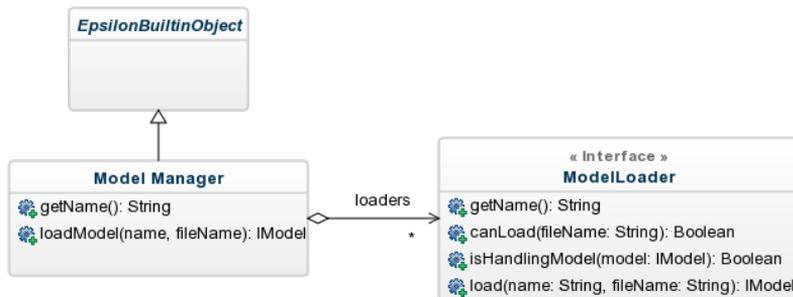


Figure 7.6: Class diagram including the `ModelManager` and `ModelLoader` classes

Common Metamodel Our loaders use a common metamodel for all the models of our approach. There are meta-classes for Feature Models, Feature Configurations, Inter-model constraints.

Figure 7.7 shows an excerpt of the meta-classes Feature Model. There is a *FeatureModel* that represent the whole model. It comprises *Features*. One of the is the *RootFeature*. The others are *GroupedFeatures* or *SolitaryFeatures*. There are *FeatureGroups* such as *OrGroups* and *AlternativeGroups*. The model also includes a *ConstraintSet*, a collection of *CNFConstraint*, where each constraint is an *ORClause* of *Literals*.

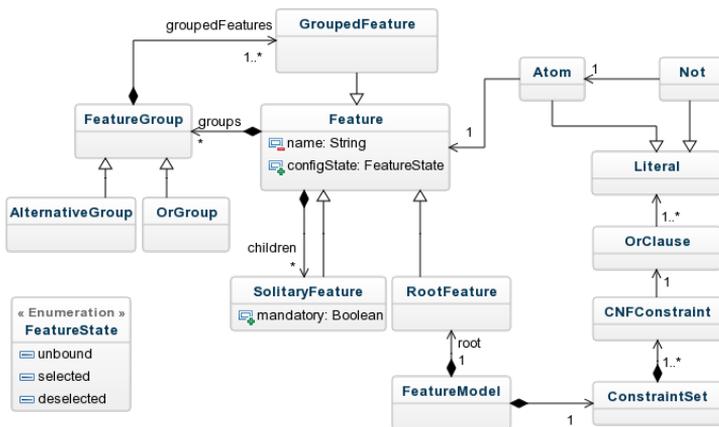


Figure 7.7: Excerpt of Feature Model metamodel

Note that the Features have information about their state. During an in-

teractive configuration, the state of the features are updated to reflect if they are selected or deselected by a user or by the machine after propagating the user decisions.

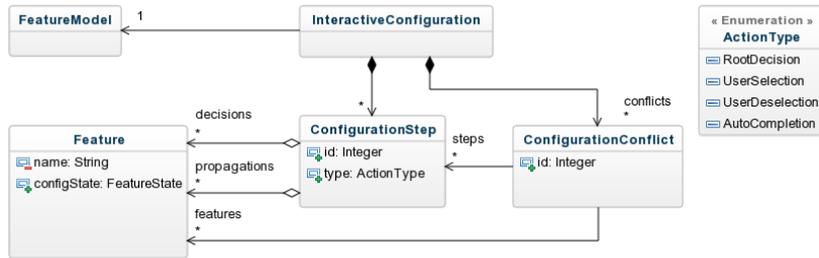


Figure 7.8: Excerpt of Configuration metamodel

Figure 7.8 shows additional classes used during an interactive configuration. A *Configuration* references an existing *FeatureModel* and comprises an ordered list of *ConfigurationSteps*. Each step describes the action performed by the user and the features selected or deselected in consequence. Additionally, the configuration may include a set of *Conflicts*. Each conflict describes a subset of the *ConfigurationSteps* and *Features* that cannot be applied at the same time because they violates the constraints defined in the model.

Unified API In addition to the common metamodel, to define a unified API, we created two built-in objects to process the feature models and configurations:

The *Reasoner* object provides operations to *validate*, *propagate* decisions and *detect conflicts*. The object can use SPLOT as underlying library to process feature models. In addition, it can use custom algorithms defined in Epsilon.

The *Configurator* object provides operations to support an interactive configuration. It provides operations to obtain and update the state of the features and the configurations steps. It provides operations such as *getConfiguration* and *updateConfiguration*. In addition, we have implemented operations to *select* features, *undo* decisions, *revise* decisions and *forceChanges*. These operations can use SPLOT as a underlying library or custom algorithms in Epsilon.

In simple terms, the **Reasoner** and **Configurator** objects define functions that implement operations for automated analysis of feature models and configurations [19]. Table 7.1 shows an excerpt of these functions. On one hand, the

Reasoner includes methods for tasks such as detecting the validity of feature models, checking if a feature is dead and obtaining all the valid configurations for a model. On the other hand, the **Configurator** includes operations such as selecting and deselecting features, detecting conflicts and completing a configuration.

Object	Operation	Description
Reasoner	isValid(fm): Boolean	determines if the Feature Model fm is valid
	isDeadFeature(fm, f): Boolean	Determines if a Feature f is dead, i.e., if it is not included in any valid configuration.
	isCoreFeature(fm, f): Boolean	Determines if a Feature f is a core feature, i.e., if it is included in all the valid configurations.
	isFreeFeature(fm, f): Boolean	Determines if the Feature f can be selected, i.e., if the feature is neither dead nor core.
Configurator	startConfiguration(fm): InteractiveConfiguration	Starts an interactive configuration based on the Feature Model fm. All the subsequent configuration steps will consider the features and constraints defined in that model.
	select(cfg, f): ConfigurationStep	Selects a Feature f trying to change the state of that feature. If everything is ok, it returns the configuration step including information of which other features are selected or deselected in consequence.
	deselect(cfg, f) : ConfigurationStep	Deselects a Feature f. If everything is ok, it returns the configuration step including which other features are selected and deselected in consequence.
	autoComplete(cfg): ConfigurationStep	Complete the configuration by taking a decision for all the undecided features in the model fm.

Table 7.1: Excerpt of methods of the **Reasoner** and **Configurator** built-in objects

Support for custom algorithms In addition to using underlying libraries, we were interested on supporting custom algorithms. Basically, we want to use Epsilon to test our ideas on how to process the models.

Listing 7.5 shows a simplified version of the *select* operation implemented using Epsilon. It uses the SPLIT reasoner. Basically, it receives an Interactive Configuration *cfg* and a Feature *f*. If the feature is a dead feature it returns an error for *InvalidConfigurationStep*. In addition, if the propagation of the selection results in a error, also returns an *InvalidConfigurationStep*. If everything is Ok, it updates the state of the features in the feature model, appends the configuration step to the interactive configuration and returns the step.

```
1 operation ConfiguratorDelegate select(cfg: InteractiveConfiguration, f:
    Feature) : ConfigurationStep {
2
3     // obtain the SPLIT reasoner
4     var reasoner = Reasoner.get('SPLIT');
5
6     // obtain the feature model being configured
7     var fm = cfg.featureModel;
8
9     // use the reasoner to determine if the feature is dead
10    if ( reasoner.isDeadFeature(fm, f) ) {
11        return InvalidConfigurationStep;
12    }
13
14    // use the reasoner to determine the effect of selecting
15    // the feature given the current configuration
16    var cfgStep = reasoner.propagate(cfg, fm, f, FeatureState#selected);
17    if ( not cfgStep.isValid() ) {
18        return InvalidConfigurationStep;
19    }
20
21    // if everything is ok...
22    // update the state of the features in the model
23    fm.update(cfgStep);
24    // append the step to the configuration
25    cfg.steps.add( cfgStep );
26    // return the step
27    return cfgStep;
28
29 }
```

Listing 7.5: Example implementation of a custom `Select` operation for the `Configurator` built-in object

7.7 Conclusions

In this chapter we presented diverse tools we created to support our approach:

Tools to create and edit Feature Models A web-based application where users can create and edit multiple domain-specific feature models, standard specific feature models and constraints.

Tools to merge Feature Models for Domains and Standards Tools that allows users to merge the diverse types of feature models in our approach,

Tools to derive configuration systems A set of model transformations that takes a feature model and produces the corresponding Configuration System, and

Runtime Components to process user decisions A set of model transformations that processes user decisions and updates the state of a configuration and the corresponding user interface.

In addition, we presented in this chapter the corresponding software designs and the platform extensions we created to develop them:

An extensible Epsilon runtime A modified *EGL servlet* that supports security configurations, custom built-in objects that can be invoked by the scripts, and an improved model loader able to load non-EMF models.

An improved Epsilon Model Manager A redesigned Epsilon Model Manager that is able to load non-EMF models uniformly. Instead of requiring different functions for different file types, it defines a single function that tries to load the models using a set of preloaded model connectivity drivers.

An extension to process Feature Models and Configurations A new built-in object that can be used by the Epsilon scripts to analyse and configure feature models. It is able to integrate both Java external libraries and Epsilon custom algorithms. Its design includes a unified API and an additional child-runtime used to execute the algorithms if it is required.

These tools, and their source code, are available at Github in the organization at <http://github.com/FaMoSA>.

Chapter 8

Case Study: Electrical Transformers at Siemens

During the work on this thesis, we started a joint research project between Siemens Colombia and Universidad de los Andes. On one hand, Siemens was interested on creating families of configuration systems for the Electrical Transformers they produce. On the other hand, we were interested in an industrial case study where we can explore the use and combination of multiple feature models. During two years we worked with a group of engineers modeling the configuration options for the electrical transformers and creating multiple configuration systems using our approach for structuring and analysing the related models.

This chapter describes the industrial case study we conducted in cooperation with Siemens. During the case study, our approach to model domains and standards independently was applied to a realistic scenario. First, Section 8.1 describes the intended configuration scenario at Siemens. Then, Section 8.2 introduces the process we use to model the diverse technical domains and standards, and Section 8.3 describes the resulting feature models. Finally, Section 8.6 discusses some lessons learned, Section 8.7 presents a discussion, and Section 8.8 concludes the paper.

Contribution: Main contribution of this chapter is a real-life case study on modeling electrical transformers that (1) describes real problems modeling complex products, (2) introduces an iterative process aimed to overcome these problems, and (3) presents lessons learned during our work.

Note that all models of the case study are property of Siemens Colombia and cannot be depicted or described in detail in this thesis. This chapter presents some example models and excerpts to explain our experience. However, the complete set of real models and constraints is not included.

8.1 Intended Configuration Scenario

During the first meetings with the Siemens engineers in the project, they commented on their vision.

Siemens Colombia produces Electrical Transformers for a large variety of applications for both North and South America markets¹. For instance, they produce transformers used in transmission and distribution networks, industrial applications and private households to deliver power to the points where the electricity is required. Instead of creating a single configuration systems for all the transformers, they are interested on multiple families of systems where each one is targeted to a specific type of stakeholder, a concrete market or a single line of transformers that Siemens produces.

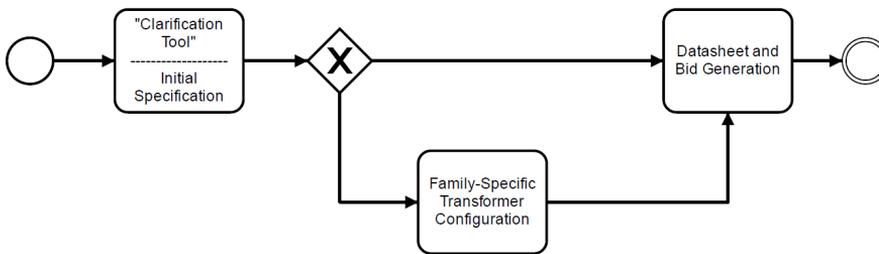


Figure 8.1: Workflow with the intended configurators for Siemens

Figure 8.1 shows an excerpt of the intended workflow with the diverse configuration systems. First, Siemens was willing to offer final customers a “clarification tool”: a system where customers can select from a general set of features and introduce some parameters, and the software (1) determines if the provided specifications are consistent, (2) searches in the standard product catalog and the historical database of designs which transformer models are the most suitable, and (3) if there is not a suitable model, presents users information on which families and models of transformers that Siemens offers can be customized to satisfy the requirements. A customer may take that information to order a specific model or specify her own transformer.

In addition, Siemens was interested on “family-specific configurators” that allow external and internal engineers to specify a customized electrical transformer. In these configuration systems, the users may introduce more detailed specifications for a specific type of transformer and the software must determine if the specifications comply with a set of pre-defined family-specific constraints. The resulting specification is later used as a foundation to design and manufacture the transformer according to the customer requirements.

¹<http://www.transformadores.siemens.com>

8.2 Modeling Process in Siemens

Considering the vision of using multiple configuration systems, we explore multiple approaches to create the corresponding feature models: Trying to create a single feature model for all the configuration systems may result into very large models hard to debug and maintain by all the different experts at the same time. Trying to create a different feature model for each configuration system may result into a lot of models with redundancies and, very likely, inconsistencies. Instead, we decide to create feature models for each domain and to process these models to create one for each configuration system based on the type of transformer and the market where it will be used.

Overview

We defined a process to elicit knowledge from the diverse domain experts in order to create a feature model for electrical transformers. This process aims to separate the domains in such a way that: (1) domain experts review, debug and introduce new options into the feature models that represent only the configuration options for electrical transformers that are part of their concerns, (2) domain experts model different standards using different feature models, and (3) domain experts join together to discuss and model how decisions in one concern affect decisions in other concerns. Figure 8.2 shows an overview of the proposed process to create the feature model.

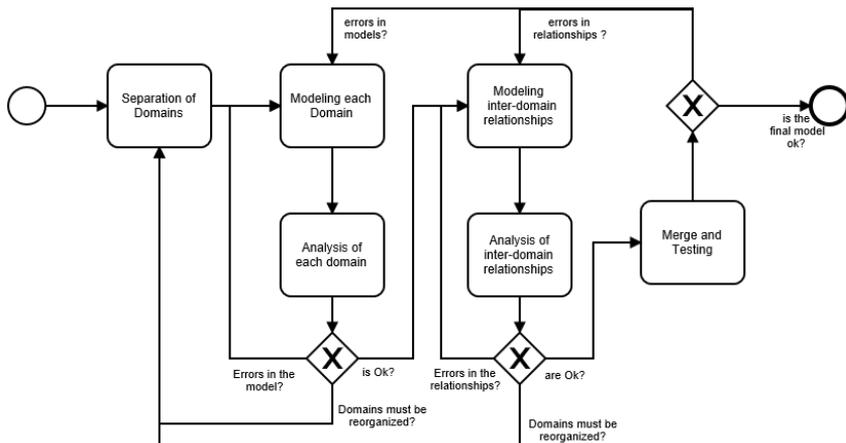


Figure 8.2: Process to Elicit Knowledge and create the Feature Model

Separation of Domains The process started defining the domains and standards to model. This was an iterative process where the different features and constraints for the product were grouped using the involved decisions as the main criteria. For instance, in Siemens we started with three different domains and ended with seven. The domains were organized to

maximize the *cohesion* by grouping the decisions performed by the same kind of experts, and to minimize the *coupling* by trying to keep together the decisions where one affects to the other. For instance, the *accessories* for transformers, initially built using a single feature model, was later divided into a model for *electrical design accessories* and *mechanical design accessories* because they represent decisions usually performed by different experts and are not highly related to each other.

Modeling of each domain and standard Once a separation of concerns was defined, we proceeded to model the options in each domain and standard using independent feature models. The models were created by a modeling leader (involved in the creation of all the models) and a group of experts for each domain. Each model was created including features that represent the options that each expert decided and including the constraints among these options that the domain expert identified.

Analysis per domain The models were later reviewed using three different activities: (1) a peer-review activity where domain experts discussed which options and constraints to include or not; (2) automated processes aimed to identify dead features (i.e., options that cannot be selected), and conflicting or redundant constraints; and (3) additional activities where domain experts used some of the automated tools we built to create configurations of existing products and detect problems in the models. During these configuration tests, experts entered options and then checked if the constraints in the feature model enabled or disabled some of the remaining options as planned. Based on these analyses, modifications and improvements were proposed and discussed with the modeling leader and domain experts.

Modeling relationships among domains Although each domain puts together decisions related to a single concern, these domains are not orthogonal. There are constraints and relationships among the options in these domains. In our approach, these relationships were represented as constraints external to the feature models. To define these inter-domain relationships, we discussed with experts in multiple domains the impact that decisions in one domain may have on decisions in the other domains. As a result of these decisions, sometimes improvements to each model were also proposed.

Analysis of the relationships among models As with the feature models, we reviewed the inter-domain relationships with peer-reviews and automated processes. Basically, we took each pair of feature models and a specification of the relationships between them to create a combined feature model. The resulting feature model was reviewed by experts and manually tested by trying to configure some products. In addition, those models were also analyzed using an automated process to detect conflicts and redundancies on the constraints.

Merge and testing Once the feature models and the inter-domain relationships were analyzed, we merged the models and performed further analyses and tests. On one hand, we automatically combined and analyzed the models for different combinations of the domains to detect dead features and conflicting constraints. On the other hand, we used existing catalogs to test the resulting models. Basically, we took the information of each product in the catalog and tried to configure that product using the resulting feature model. This process detects errors when an existing product cannot be configured using the resulting model.

We defined three roles for the project: A *project leader*, who was in charge of the administration of the project; a *modeling leader*, who supervised the modeling activities and led the activities to review the models with many experts; and the *domain and standard experts*, who created the models. A *software engineer* participated too. He developed software to obtain information from the Siemens databases to perform tests and analyses, integrated the configuration systems to the existing ERP and supported the domain experts when problems with the infrastructure happened.

In order to support the project activities, we used several tools: For instance, the Feature Models were created using a modified version of SPLOT. The combinations, tests and analyses of these models were performed using other tools we created. These tools were developed using Model-driven technologies that allow experts edit the feature models, define a combination of them, and derive the corresponding configuration system without the intervention of software developers. In consequence, the people in the project was able to analyse the models and test the corresponding configuration system during all the process.

8.3 Models for Electrical Transformers

Electrical Transformers are complex devices that exhibit an enormous variability depending on the intended power transformation, environmental conditions, standards imposed and customer particularities.

8.3.1 Modeling Multiple Domains

Intuitively, all the transformers perform the same task: They transport electricity from one circuit to another, possibly modifying properties such as the electric current or voltage in the process. All the transformers have specifications that include the electrical properties of the input and the output. However, transformer specifications may differ in many other aspects. For instance, the properties to include in the specifications may vary according to elements such as the type of transformer, the location where it will be installed and the electrical and mechanical properties required for its operation.

Separation of Domains As mentioned before, an important decision in our approach is to determine which domains to model. For Siemens, we defined seven domains after multiple iterations:

1. **System**, a model representing the options and restrictions defined by the power transmission and distribution network;
2. **Installation**, a model including properties such as altitude and temperature of the place where the transformer will be located;
3. **Transformer Type**, one including transformer classifications such as pole-mounted or pad-mounted;
4. **Electrical, Thermal and Acoustic (ETA)** representing specifications directly related to the electrical design;
5. **Mechanical** comprising specifications related to the mechanical design,
6. **Electrical Accessories**; and
7. **Mechanical Accessories**.

The next sections present a subset of these models and exemplify some of the operations and analysis we perform during the process. We describe there excerpts of the models for (1) the type of transformer; (2) the ETA properties; (3) the mechanical properties, (4) and the electrical accessories.

Transformer Type domain There are many types of transformers depending of their intended application and installation. For instance, for residential applications, there are: (1) *indoor distribution transformers* that are installed inside a building to deliver electricity to the apartments and offices in its interior, and (2) *pad-mounted transformers* and (3) *pole-mounted transformers* that are installed outdoor delivering power to the apartments in a building or to the houses in a block. Figure 8.3a shows a feature model representing the mentioned types of transformers.

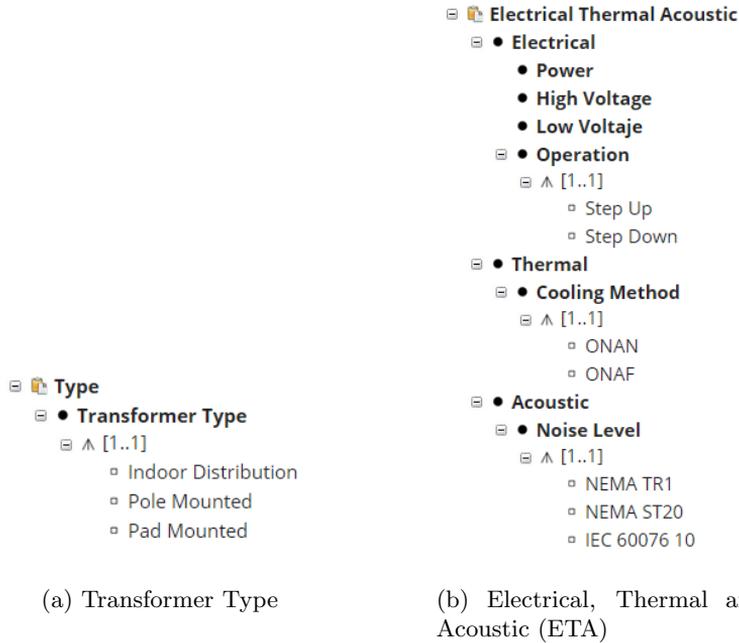


Figure 8.3: Excerpt of Feature Models for Transformer Domains (1)

Electrical, thermal and acoustic (ETA) domain In addition, independently of its type, transformer specifications include definitions for electrical, thermal and acoustic (ETA) properties. These definitions are used to design the electrical systems and components that are the core of the transformer. Figure 8.3b shows an excerpt of the corresponding feature model.

The electrical definitions include values for parameters such as power, high and low voltages, and the expected thermal and noise levels. Note that there are mandatory features for each one of these parameters. The values for these parameters must be specified for every transformer that Siemens produces.

Among the thermal characteristics of a transformer, there is the type of cooling system that must be used. For instance, the cooling systems may either ONAN or ONAF: ONAN, an acronym for "Oil Natural Air Natural", is a system that uses the natural flow of oil and air across tubes or radiators attached to the device to cool the transformer. In contrast, ONAF, for "Oil Natural Air Forced", is a system where fans blowing air on the cooling surface are employed to provide better cooling.

Similarly, the acoustic definitions include information about the intended levels of sound (or noise) for the transformers. For transformers, there are some definitions of the intended acoustic levels. For instance, a transformer may have either NEMA TR1 or NEMA ST 20 or IEC 60076-10 audible sound levels.

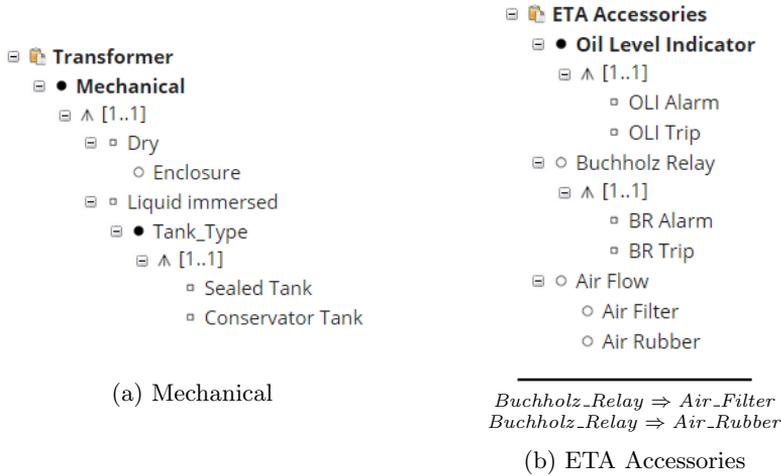


Figure 8.4: Excerpt of Feature Models for Transformer Domains (2)

Mechanical domain The mechanical specifications of a transformer includes definitions such as the type of refrigeration and the type of tank used by the transformer. Figure 8.4a presents an excerpt of the feature model. Basically, there are *dry* transformers, that are refrigerated by air, and *liquid-immersed* transformers, that are refrigerated by immersing the circuitry in mineral or vegetable oil. In addition, depending on the type of refrigeration, the transformer must include one of many alternatives for enclosing. For dry transformers, the device may include an *enclosure* to prevent users to accidentally access the circuits. For liquid-immersed transformers, they may use *sealed tanks* where the oil remains in a single container, or *conservator tanks* where the oil flows from one tank to facilitate its refrigeration.

ETA Accessories Another model describes the accessories to include in a transformer. These accessories include elements for electrical and thermal protection and control, such as diverse types of oil level indicators, protective valves and relays that a transformer uses. The feature model in Figure 8.4b shows some of these accessories. There are two features for specifying the *Oil level indicators*: these indicators may activate an *Alarm*, may *Trip* a circuit or may perform both tasks. In addition, there are other two similar types of Buchholz Relays: those that turn on an *Alarm*, those that *Trip* a breaker and those that do both. Finally, there are other features representing air flow accessories: *air filter* and *air rubber cell*.

Note that this model includes a cross-tree constraint. Including a *Buchholz relay* in a transformer implies to include also an *air filter* and an *air rubber cell*.

8.3.2 Modeling Domain Interactions

Domain interactions are modeled as cross-model constraints. For instance, as mentioned before, Buchholz relays can be used only in transformers with a conservator tank, i.e., the existence of a Buchholz relay in a transformer implies that the device includes a conservator tank. This is represented with a constraint relating a feature in the *mechanical domain* with a feature in the *ETA accessories domain*.

Figure 8.5 shows an example of cross-model constraints representing domain interactions. It includes other two constraints relating the features presented above: On one hand, *pole mounted* transformers are used to *step-down* the power and deliver electricity to houses, i.e., pole mounted transformers implies a step-down operation. And on the other hand, *pole mounted* transformers cannot comply with the ONAF specifications, i.e., pole mounted transformers excludes the ONAF.

Buchholz_Relay \Rightarrow *Conservator_Tank*
Pole_Mounted \Rightarrow *Step_Down*
Pole_Mounted \Rightarrow \neg *ONAF*

Figure 8.5: Inter-domain Constraint Set for Electrical Transformers

8.3.3 Modeling Standards

Standards are modeled as additional feature models. In contrast to models representing domains, the models representing standards may include features that already exists in other models. These models are used to represent the standards using a tree-based structure instead of using just a list of additional constraints.

Figure 8.6 shows an excerpt of the feature model for the NTC 3997 standard. Note that it is represented as a feature model and includes features that exists in the other domains. In the model, the root represents the standard: *NTC 3997*. It includes definitions of the allowed values for power, high and low voltage. It specifies that the transformer must be *pad mounted*. It defines only a single cooling method, *ONAN*, which is mandatory. In addition, it defines the *NEMA TR1* sound level as mandatory. All this information may be obtained from the text of the standard specification. Note that an initial version of the corresponding feature model can be created without the intervention of domain experts.

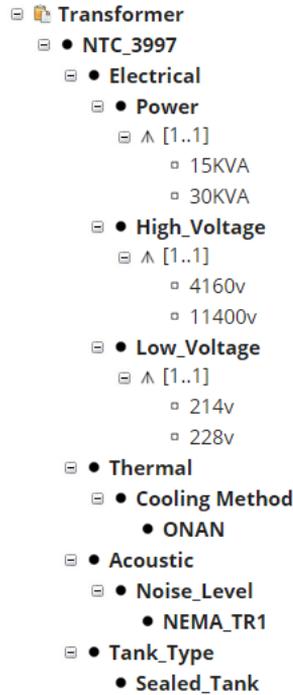


Figure 8.6: Excerpt of Feature Model for NTC 3997

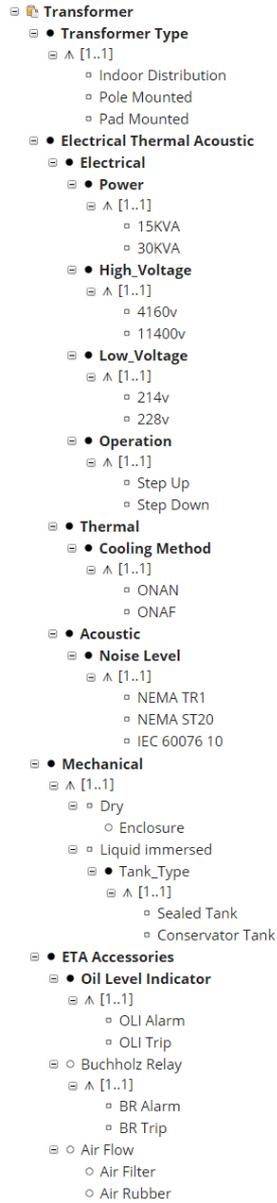
8.4 Reviewing and Analyzing the Models

Once each model was created, we performed additional processes to review and analyze them. These processes comprise the combining of these models in different arrangements:

- **Combining domain feature models and inter-domain constraints** to test and analyse the interactions among multiple domains.
- **Combining domain feature models with a standard**, to test and analyse the interactions among domains and standards, and
- **Combining all the domains and the standards**, to test and analyse the interactions among multiple domains and standards.

8.4.1 Analysis by combining domains

We combined the domain-specific feature models and inter-domain constraints to test and analyse the interactions among multiple domains. Basically, we combined first each pair of domains and reviewed the result with experts on both. Then, we combined all the models, and reviewed the combination trying to reproduce the configurations of real transformers.



Buchholz_Relay ⇒ *Air_Filter*
Buchholz_Relay ⇒ *Air_Rubber*
Buchholz_Relay ⇒ *Conservator_Tank*
Pole_Mounted ⇒ *Step_Down*
Pole_Mounted ⇒ ¬*ONAF*

Figure 8.7: Feature Model combining the domains

For instance, we combined the feature models to determine how the options in the ETA domain affect or are affected by the options in the ETA accessories domain. We used the result to determine if the constituent models included contradictions or problems that prevent to introduce valid configurations or lead to make options dead or full mandatory.

Once each pair of domains was combined and analysed, we combined all the domains. The figure 8.7 shows a model combining all the models presented above. Note that the root of the model, i.e., the concept to configure, is the *transformer*. Below, there are sub-trees based on the models for each domain. There are sub-trees for the *transformer type*, the *ETA*, the *Mechanical* and the *ETA accessories* domains. In addition, there are additional constraints representing the cross-tree constraints in each models and the inter-domain constraint sets.

8.4.2 Analysis by combining domains and standards

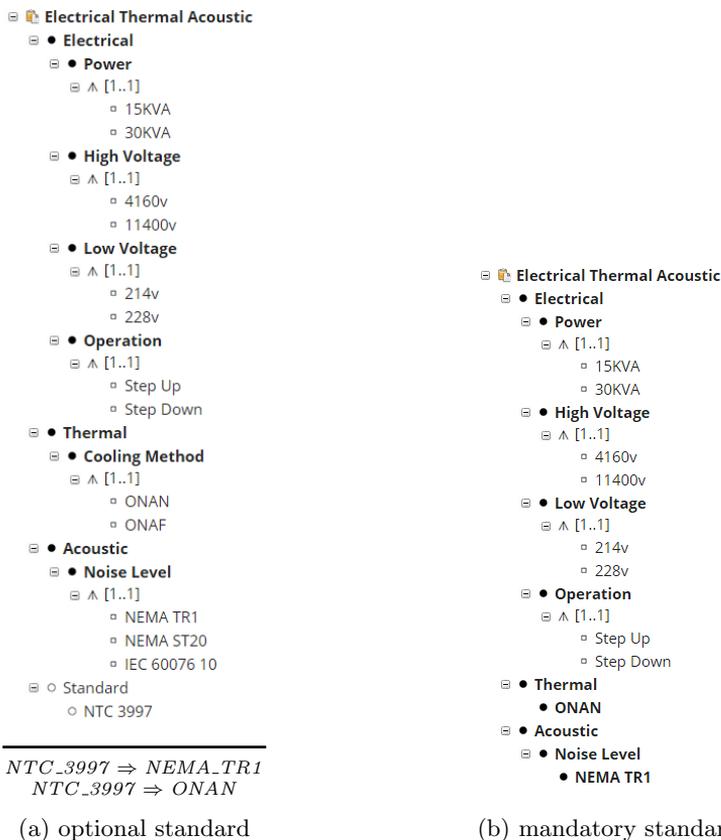


Figure 8.8: Feature Model combining ETA domain and the NTC-3997 standard

Simultaneously, we combined domain-specific feature models with standards. This allow us to determine interactions among the domains and the standards. For instance, we combined the ETA domain and the NTC-3997 standard. Figure 8.8 shows the resulting combined feature models.

We can combine domains and standards in two ways: On one hand, we can combine them setting the standard as an optional feature. In such case, a set of additional constraints is included in order to enforce the restrictions defined by the standard when it is selected. On the other hand, we can combine the models by enforcing the standard. The resulting model does not include any feature that is not allowed by the standard. Note the models in the figure 8.8. In the left model, the standard is optional and some constraints enforce the rules when it is selected. In the right model, the standard is not even a feature. The model includes only the features that are compatible with it.

We tested and analysed these models with experts on the corresponding domains and standards. Basically, we aimed to review if the constraints included in the model represent correctly the rules in the standard and do not contradict the restrictions defined by the domain itself.

8.4.3 Analysis by combining all the domains and standards

Once we analysed the different combinations of domains and standards, we created models combining all the models and the standards required for a specific configuration system. For instance, consider a system aimed to configure a family of pole-mounted transformers for the Colombian market. The required model must combine the feature models for the diverse transformer domains and the standards related to that type of transformer and that specific market.

Figure 8.9 shows feature models combining the NTC-3997 standard with all the domains presented above. As before, there is a model where the standard is defined as an optional feature, and another model where the rules of the standard are enforced. On one hand, note that the first model includes more constraints than the second one. For instance, there is a constraint stating that, if the *NTC-3997* is selected, the ONAN must be selected too. On the other hand, the second model includes less features than the first. For instance, the ONAN feature is mandatory because it is enforced by the standard and the ONAF feature is not included because it is prohibited by the norm.

8.5 Reviewing and Testing the Models

In conjunction with engineers from Siemens, we defined activities to assure the quality of the models. Additionally to the automated analyses of these models e.g., to determine dead features and conflicts among the constraints, we performed other activities to check if the models represented the real set of products that the company manufactures and sells.

Among these activities, we can mention:

Peer reviews and walkthroughs Several activities were aimed to review the models. On one hand, each domain and standard expert usually reviewed many models created by the others. In order to review the interactions among domains and among domains and standards, they usually took models from other users and checked that some required features were present in the models. Although some of these errors can be detected by automated analyses, many errors were detected in that way.

On the other hand, many meetings domain and standards experts was scheduled to review the models. Usually the author of the model presented the features in the model and an overview of the constraints included in it. The other participants reviewed the model at the same time to make questions and provide feedback. Some of these meetings were scheduled to define how to solve a conflict when differences among the models were detected.

Manual tests using Configuration Systems In addition to the reviews, domain and standard experts used configuration systems to test the model. Each time an expert modified or combined the feature models, she can use a configuration system to specify an electrical transformer and check if the constraints that she defined were defined correctly. Although the experts were free to configure any product, some activities were defined to request experts to configure products for specific markets and check if the model included all the relevant constraints.

Automated tests using Product Catalogs Siemens has databases with the actual catalog of products and with the history of all the products that the company has built. These databases include specifications, designs and a lot of additional information. We used these databases to obtain configurations of real products and test the feature models. The information in the databases was translated to configurations that included the features defined in the feature models. A program took that configurations and determined if they were valid against the feature model. The products that resulted invalid were checked to determine if the model had errors. Sometimes the products reported with errors were transformers that are not compliant with the current standards and regulations.

8.6 Lessons Learned

Our experience in modeling electrical transformers using feature models and specifying them as a Configuration Process can be summarized in the following lessons learned:

8.6.1 Modeling

Modeling complex products using a single feature model is tough

As a first attempt, we tried to create a single feature model representing the variability involved in a Transformer. The resulting feature model included features from multiple domains, constraints in each domain and constraints imposed by the diverse standards for electrical transformers. The final result had a large number of features and constraints and was very hard to understand, validate and maintain. The intermixed of features from diverse disciplines made the discussion among experts difficult. Furthermore, the amount of constraints imposed by the standards diffculted the analysis of the feature model. For instance, an initial model for a medium-sized transformers product line and only one standard included more than four hundred features and more than one hundred constraints. After many reviews and modifications, domain experts found it too hard to check and introduce new constraints in that model.

Using multiple feature models facilitates the modeling of multiple domains

Due to the issues found in the approach with a single model, we decided to create one feature model to capture the variability of each domain of expertise and a feature model per standard. The result was a set of feature models, each less complex than the former, single one. The expert validation was easier. As we presented in previous sections, the activity required an iterative process and a strong collaboration with the experts to identify relationships among feature models.

The separation of domains is an iterative process

Many times, after reviewing the model for a domain we found better ways to organize the domains or the features in the model. For instance, some domains were divided in two or three to focus on some type of decisions and simplify the modeling. Also some options were reorganized within a domain to simplify the constraints. And other options, initially defined in a domain, were moved to another domain. The iterative nature of the process and the participation of a modeling leader that understands how the domains have been divided facilitates the separation of domains.

While the initial feature model combining all the domains comprised more than 400 features, the resulting models for each domain contains between 10 and 123 features, and those for the standards between 69 and 71 features. Domain experts found these models easier to review and modify than the former. In addition, the final models include features from more products and represent

more standards than the initial. That means that these models are simpler and more complete.

Incremental modeling of products facilitates the work Initially, domain modelers tried to detect all the most exotic options included in any already developed product, introducing a lot of features and defining very complex constraints. Instead of trying to create a complete model from the beginning, we started focusing on the most common features (those included in the 80% of the products), and then on the more exotic ones. This incremental approach allowed us, (1) train the domain experts using simpler models, (2) practice with them how to introduce new features and constraints, and (3) define practices to review and debug the models continuously.

We consider that the models to represent transformer specifications will evolve with time to introduce new technologies and standards. Future activities for continuous modeling can be based on the presented process to elicit knowledge. In addition, more specialized tools to detect the impact of feature model modifications can be adapted or created to support these activities.

Each standard can be modeled independently An important source of complexity to model the transformers is the existence of multiple standards. Initially, we tried to model several standards using a single model. However this approach resulted in models with very complex constraints hard to review by the domain experts. Then we decided to use feature models for each standard and an automatic process to merge these models. Using multiple models we can focus on modeling each standard using simpler constraints, while our software tool combines these constraints into the complex constraints we detected before.

8.6.2 Validation of the models

Continuous testing facilitates the incremental modeling of products Each time we created a version of a model, that model was tested and reviewed with domain experts. This approach allowed us to detect errors using simpler models reducing the risks of trying to detect and correct errors on larger and more complex models. In addition, this approach allowed us to compare how different versions of the models validate some configurations and detect errors introduced when a model was modified.

Testing of models can be performed using product catalogs Because the models also represent products already manufactured, domain experts can use information from product catalogs and the production order history to test them. Sometimes they took information of standard catalogs and production orders to define custom configurations that they used to test the models manually. We consider that software tools can take that information to create configurations that must be valid against a correct set of feature models. Af-

ter any modification on the models, these tools may help us to detect errors introduced by accident.

8.6.3 Tooling

Existing tools provide limited support to create multiple inter-related models There are many tools to create feature models, define views on these models, merge sub-models and configure products based on them. However, they offer limited support to reorganize the domains we are modeling by moving features or automatically refactoring the constraints related to these features. We extended SPLOT [91] and create our own tools to edit multiple feature models, to perform the special merge of feature models for domains and for standards, and to process and test continuously the models. There is an opportunity to create more specialized tools to model complex products with multiple concerns.

Partial Configurations can be used to guide the process Existing tools can use the information in feature models to validate and auto-complete a partial configuration. This allows customers to provide incomplete specifications and use the software to validate them. In addition, if these incomplete specifications include all the features that are relevant to the customer, any configuration resulting of an auto-complete will include those features and will be appropriated to the user. Furthermore, those partial configurations can be used to find similar products and recommend customers either standard or already designed transformers that are faster to build. We consider that, at least for electrical transformers, the configuration software must include options to support user-provided partial configurations.

8.6.4 Impact on other engineering processes

Modeling the products allows engineers to define and enforce standards in the company During the process, we continuously discussed which features are part of a standard and which not. This discussion allows engineers to detect potential problems and conflicts in some combinations of standards. In addition, because we also discussed which features are the most commonly requested, this helped engineers in their permanent reflection on which standard products, assemblies and sub-assemblies can be defined in the company. We consider that the modeling of variability is not only useful to create configuration software but also to help companies to analyze and streamline their portfolio of products.

8.7 Discussion

Considering that we learned some lessons from the Siemens case-study, it is important to determine if these lessons are relevant to other companies and situations. This section describes the research method we used in our work and the threats for the validity of our conclusions.

8.7.1 Research Method

Our work with Siemens allowed us. not only to develop formal operations and software artifacts for using feature models to create configuration systems, but also to apply and evaluate these outcomes in a real-life environment. There are several *Design Science Research Processes* [62][107][103] proposed for scenarios like this where the expected outcomes of the research include the design, implementation or the improvement of artifacts for a specific problem domain. We followed the process outlined by Offermann et al. [103] where research, design and implementation activities are defined to (1) identify the problem, (2) design and (3) evaluate the solution.

Problem Identification We started working with Siemens to create configuration systems using Feature Models. Initial activities were centered on a literature review of the applicable approaches, the transfer of knowledge to Siemens engineers and the creation of a single feature model to represent the products. Additional activities were focused on using multiple feature models and determining advantages and problems of both alternatives. The challenges and problems addressed in our work were defined based on evaluations and interviews of the participants of these activities.

Solution Design After precisizing the problem, we started to design a solution. We worked on three main areas: the modeling of complex domains using multiple feature models, the automated processing of these models, and the derivation of configuration systems from them. For each area we performed literature reviews and technical evaluations with the participants during all the process. As a result we refined our proposal for the modeling process, the automated processing of the models and the design of the supporting software in multiple iterations.

Solution Evaluation Finally, we evaluated and improved our solution based on its application in real-life product lines. In conjunction with Siemens we created models to configure electrical transformers. The activities to analyse and test the models helped us, not only to improve the feature models but also to improve our solution to create and process these models.

As a result of this work we refined and evaluated our proposal to develop configuration systems using different feature models for the technical domains and the regulations of a product family.

8.7.2 Threats to Validity

In general, the *validity* of a research is concerned with the question of how the conclusions might be wrong [46]. A *validity threat* is a specific way in which it might be wrong [46]. This section discusses some validity threats of our work in the case study.

Internal Validity focus on how legitimate are the findings and solutions in our research. One possible threat to internal validity is a *researcher bias*. We have been working with feature models and operations on these models and it is possible that we influenced the work to use them and present them as a viable solution. In order to mitigate this threat we tried first to represent the products using a single feature model, and later tried with multiple feature models. As part of the problem identification, we interviewed the domain experts participating in the modeling about the pros and cons of both alternatives. Another possible threat is a *modeler bias*. It is possible that the results were influenced by the ability or the knowledge of the modelers. We consider that the likelihood of this threat is fairly low. On one hand, several modelers with different skills and backgrounds were involved in the process. The problems we describe were not perceived and commented by only one participant but by many of them. On the other hand, none of the participants had previous experience on using feature models. They learned to create the models as part of the project. Finally, we discussed our findings with the project leaders from Siemens, and presented our experience to other experts in feature models to mitigate any possible bias.

External Validity is concerned with whether we can generalize the results outside the scope of our study. In this case study, many domain and standard experts created models and configuration systems for multiple product families, domains and standards. Although some of these domains and standards were very different one from the others, they were able to specify these concerns using our proposal. That give us some confidence that the approach can be used to create models for product families in other domains and companies. However, all our work was limited to Siemens and to Electrical Transformers. Although, this is a point of caution in the generalization of our findings, we believe that our study shows that the approach is very comprehensive and offers benefits to the companies creating configuration systems for complex products that involve multiple domains and regulations.

8.8 Conclusions

In this chapter, we reported our experience on modeling configuration options for electrical transformers. Instead of modeling a set of existing products, we aimed to model the different options that can be included in user-supplied specifications. Initially, Siemens tried to use a single feature model to represent the different options in all the involved domains. However, this approach led them to create larger models with complex constraints difficult to understand, validate and maintain. Instead, we opted to use multiple feature models and automatic tools to combine them. We defined a process to elicit the knowledge on the diverse domains and create these multiple models.

Our findings are summarized as a set of lessons learned. Among others, we identified that using multiple feature models facilitates the modeling process because each domain expert can focus on the variability that she understands and is part of her concerns. In addition, we found that an iterative approach to separate and model the domains facilitates the work. Furthermore, we detected that existing tools can be improved to support the creation and testing of variability models that comprise multiple feature models.

Chapter 9

Conclusion and Perspectives

In this chapter, we first synthesize and conclude all the contributions of this thesis, re-enumerating the challenges and how we addressed each of them. Next and finally, we discuss some perspectives for future research.

9.1 Summary of Contributions

This thesis is an effort to support the tasks of modeling and building configuration systems for complex products. These tasks need special assistance due to the inherent complexity of the products, the need to involve multiple domains and experts, the existence of many standards and regulations, the diversity of configuration systems required by the companies to build product families.

To assist engineers and stakeholders on modeling and building these feature-based configuration systems, we have presented the following contributions:

- C1** *Feature Models for Domains and for Standards* We extend existing approaches by using multiple feature models and constraints among feature models. Instead of creating a single feature model, domain experts create a model for each domain and for each standard independently. Interactions among the domains are represented using sets of constraints among the corresponding feature models. A standard may restrict which features can be selected or not in the other domains, can define valid combinations of features and specify concrete feature values. Each standard can be modeled by experts on the standard, using a different feature model using the easy-to-understand structure of the model instead of a set of constraints. We propose an iterative process to specify all the options for the configuration system by creating, combining and testing models of the domains and standards.

- C2** *Operators to combine feature models for domains and standards.* These models represent, on one hand, configuration options and constraints defined by the technical properties and designs of the products, and on the other hand, constraints imposed by standards and regulations. We propose a set of automated operations to analyze and merge on such models. Considering that standards are crosscutting concern that can be enforced or not depending on the customer requirements and country regulations where the products are sold, we define operations for *Conditional Intersection Merge*, that can enforce a standard only when a feature is selected, and for *Partial Conditional Intersection Merge*, that can enforce only the part of a standard related to a domain when a feature is selected. We define an operation for *Combination* that performs multiple operations such as *Reduced Product Aggregation* and *Partial Conditional Intersection Merge* to combine properly the diverse feature models for the domains and the standards of a product family.
- C3** *Automated Derivation of Configuration Systems.* Considering that companies are interested in creating Configuration Systems for specific markets and countries, it is important to produce the software based on the correct combination of models. We propose a model-driven approach where feature models are used to generate the software components that interact with the user during the configuration process. Furthermore, we built a set of runtime components that processes the user decisions and updates the user interface consequently. In contrast to other proposals, our user-decision processor exploits the previous decisions of the user to perform tasks such as undoing and changing decisions more efficiently.
- C4** *Case Study on modeling Electrical Transformers.* We have applied our approach in an industrial case study. We worked in a joint research project between Siemens Colombia and Universidad de los Andes to model and create multiple configuration systems for Electrical Transformers using our approach and tools. We present here a report of our experience including some lesson learned during the process.
- C5** *Experimental Implementation of our approach.* We have developed the *FaMoSA toolset*, a set of software tools to (1) model, test and analyze models representing multiple domains and standards (2) analyze and test these models (3) combine these models according to the intended configuration process, and (4) derive automated configuration systems that stakeholders can use to decide and select features for a product.

9.2 Research Perspectives

In this section, we present some long- and short-term ideas for research around the contributions of this thesis.

9.2.1 Regarding the Modeling of Domains and Regulations

Support for other types of Models and Relationships. While our work uses multiple feature models and constraints to relate these models, other approaches combine other types of variability models and relationships among models to represent complex products. For instance, there are approaches that combine Feature Models, Orthogonal Variability Models and Decision Models [42][43][53], that combine Feature Models with Goal Models [34] and use multiple types of relationships [27][28][31][111]. It is important to extend our approach to support, on one hand, domains and standards represented using other types of models and, on the other hand, types of relationships among these domains and standards in addition to the already supported propositional constraints.

Support for additional analysis operations. Our approach models configuration options for complex products using multiple models for domains and externally defined constraints, such as standards and regulations. Additional operations can be defined to help to analyse and debug that set of models. For instance, we are considering operations to analyze the standards: some operations can be defined to determine if at least one product of a product family, or all of them, are compliant with some standard. Other operations can analyze if two or more standards are compatible and if it is possible to configure at least a product that is compliant with them at the same time. Additional methods can be considered to determine which modifications to a set of feature models must be performed to support a specific standard.

Case Studies on other Product Families. Part of our research involved a case study on Configuration Systems for Electrical Transformers at Siemens Colombia. We are interested on implementing our modeling approach in other companies and product families. Further cases studies will give us the opportunity to test and improve our approach.

9.2.2 Regarding the Combination of Feature Models.

Support of other types of solvers and formalizations. In order to improve the implementation of our approach, we are considering the use of diverse types of solvers and frameworks for testing. On one hand, we plan to explore Satisfiability Module Theory and Linear programming solvers to analyze [97] and optimize [142] [95] feature configurations. On the other hand, we are considering extending frameworks such

as FLAME [45] and Betty [123] to formalize the merge operations and perform automated metamorphic testing of our implementations.

Integration to other frameworks and libraries. We have implemented our merge operations using Epsilon and SPLOT. There are other libraries that are being used by the Software Product Line community such as Familiar, FaMa and FeatureIDE. We are interested on implementing our operations on these libraries to motivate other users to explore and use our approach.

9.2.3 Regarding the Derivation of Configuration Systems.

Support for Concurrent Configuration. We focused on developing feature-based configuration systems where a single user makes the decisions or where multiple users decide one after the other. However, there are many scenarios where multiple stakeholders decide at the same time without considering the decisions made by the others. In such scenarios, the software must evaluate the decisions to determine conflicts among the decisions of the same user and among the decisions made by the others. The software must support strategies to solve conflicts different than those used for a single user. For instance, the software may use a ranking of users to impose decisions of some users over the others, may use algorithms trying to satisfy a larger number of users or may propose fixes based on decision rules. It is necessary to extend our approach to derive configuration systems in order to support these tasks.

Support for Configuration Workflows. Instead of Concurrent Configuration, other authors have proposed the use of Configuration Workflows [66]. In this approach, multiple users configure a product, each one selecting options from a view of the product following a predefined workflow. The software to support this kind of configuration processes requires to support the definition of feature model views and the integration of the configuration systems with an workflow system. It is necessary to extend our approach to consider, on one hand, the multiple feature models representing each modeling perspective, and on the other hand, the multiple views representing the configuration perspectives in a workflow.

Support for Product Optimization during Configuration. Our approach was tested on a case study on Electrical Transformers. We use our approach to provide tools that validates user specifications and propose existing products based on that. However, during our work, many times the software presented a set of transformers that satisfies the requirements but does not recommend any in particular. We consider that the software may integrate algorithms for product optimization in order to recommend users which transformer choose based on some criteria, e.g., recommending the transformer that is cheaper, is easier to build, or has a larger customer base.

Support for Attributes and Values during Configuration. Products such as the Electrical Transformers must be configured by specifying values for attributes such as voltage or power. Currently, we model these attributes as features that may be selected or not. For instance, if a transformer may have a value of 110v or 220v for the voltage, these options must be represented as two different features. These options can be represented using an attribute for the transformer instead of using a feature. Using attributes, and domains of values for these attributes, requires modifications on the algorithms we defined, but this will open up new possibilities to the ways that products can be modeled and configured.

Appendix A

Approaches for Intersection Merge

Operations on feature models such as the *intersection merge* can be implemented in different ways. Acher et al. [11] described four approaches that can be used. There are *semantic-based* approaches that rely on the propositional encoding of the input feature models, *reference-based* approaches that build composite structures involving the inputs, and *hybrid techniques* that use *local synthesis* or *slice operations* to produce the resulting model. In Chapter 5, we are using an approach that takes one of the input models and introduces to it new constraints according to the semantics of the other model. This chapter compares our approach with the other techniques described above to implement the *Intersection Merge* operation.

Rest of this chapter is organized as follows: Section A.1 presents a running example. Section A.2 explains how the intersection merge can be implemented using the techniques proposed by Acher et al. [11]. Section A.3 presents our approach. Finally, Section A.4 presents a comparison and Section A.5 concludes the chapter.

Note that this appendix presents alternative implementations of the (traditional) intersection merge operation. In our work we defined other operations: Conditional Intersection Merge and Partial Conditional Intersection Merge. Alternative implementations for the Conditional Intersection Merge are discussed in the Appendix B.

A.1 Running example

This chapter presents diverse implementations of the *intersection merge*: an operation that takes two feature models and produces a new feature model which semantics (i.e., the corresponding set of valid configurations) is the intersection of the semantics of the input models.

To illustrate the diverse techniques, we present first a running example. Figure A.1 shows two feature models that we want to intersect.

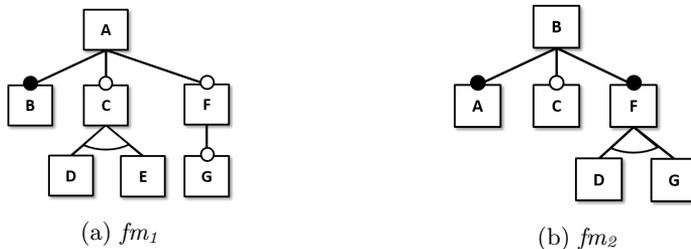


Figure A.1: Example Feature Models to intersect

The intersection merge operates on the semantics of these models. The Table A.1 shows the set of valid configurations of both models. Note that there are some configurations that are valid against both.

Configurations for fm_1	Configurations for fm_2
$\{A, B\}$	$\{A, B, D, F\}$
$\{A, B, C, D\}$	$\{A, B, F, G\}$
$\{A, B, C, E\}$	$\{A, B, C, D, F\}$
$\{A, B, F\}$	$\{A, B, C, F, G\}$
$\{A, B, F, G\}$	
$\{A, B, C, D, F\}$	
$\{A, B, C, E, F\}$	
$\{A, B, C, D, F, G\}$	
$\{A, B, C, E, F, G\}$	

Table A.1: Sets of valid configurations for the example Feature Models

The intended result of the intersection is a new feature model which semantics corresponds to the set of configurations valid on both models, i.e., $\llbracket fm_r \rrbracket = \llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket$. In our example, this is a feature model which set of valid configurations is $\{\{A, B, F, G\}, \{A, B, C, D, F\}\}$.

Note that the resulting feature model may include only the features that exist in both models, i.e., $\mathcal{F}_{fm_r} = \mathcal{F}_{fm_1} \cap \mathcal{F}_{fm_2}$. For instance, the feature E that exists in fm_1 but not in fm_2 is not included in any of the valid configurations of the result. That feature can be absent in the resulting model.

A.2 Approaches to implement the intersection merge

Acher et al. [11] described four techniques to implement operations on feature models: (1) a semantic-based approach, that uses the propositional encoding of the feature models, (2) a reference-based approach, that uses models that reference to the input feature models, (3) a local-synthesis hybrid approach, that takes a reference-based model to synthesize a new feature model, and (4) a slice-based hybrid approach, that uses the slice operation to process a reference-based model to produce the new feature model.

A.2.1 Semantic-based implementation

The *Semantic-based implementation* comprises three steps: (1) *Formula Calculation*, (2) *Implication Graph Derivation*, and (3) *Feature Tree Derivation* [2].

Formula Calculation The first step is to determine the propositional formula that represent the resulting feature model.

For the intersection merge [2]: Given two feature models fm_1 and fm_2 with the corresponding ϕ_1 and ϕ_2 propositional formulas, the resulting model fm_r can be encoded using the formula ϕ_r , defined as follows:

$$\phi_r = (\phi_1 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})) \wedge (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}))$$

where,

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

Implication Graph Derivation The next step is to build an *implication graph* from the resulting formula [9]. An implication graph of a formula is a directed graph $G \langle V, E \rangle$. Each a vertex v represents a variable of the formula or its negation, and each directed edge from vertex u to vertex v represents an implication $u \Rightarrow v$ (or the same, $\neg u \vee v$).

Feature Tree Derivation Finally, the last step is to determine a structure for the resulting feature model. The new structure is a spanning tree of the implication graph. Additional edges in the graph are included as cross-tree constraints to the model. The types for the features and relationships may be determined based on the formula. Additional structures may be used to help in the process, e.g., *Bi-implication graphs* can be used to determine atomic sets and *binary exclusion graphs* can be used to determine mutually exclusive features [3].

The result is a feature model which semantics represents correctly the intersection of the input feature models. Usually, it is one of the many possible models that represent that set of valid configurations. For instance, consider the example presented above. Figure A.2 shows some alternative solutions for the intersection of the example models.

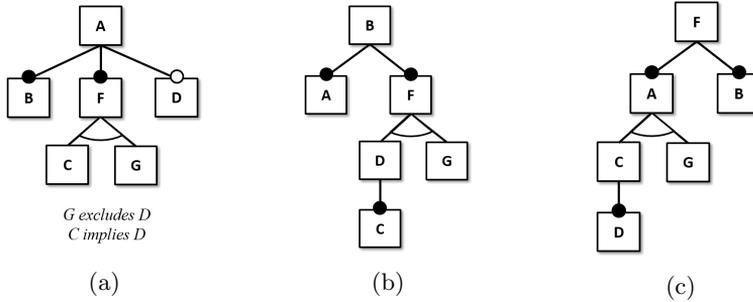


Figure A.2: Example Solutions for the intersection of the Feature Models

FAMILIAR implementation Executing a semantic-based intersection merge in FAMILIAR is straight forward. It provides a `merge intersection` command that executes all the steps. Listing A.1 shows an example intersecting the models in the example.

```

1 // define the fm1 and fm2
2 fm1 = FM ( A : B [C] [F]; C: (D | E); F: [G]; )
3 fm2 = FM ( B: A [C] F; F: (D | G); )
4
5 // intersect the input models
6 fmr = merge intersection { fm1 fm2 }

```

Listing A.1: Semantic-based Intersection Merge of the Example Feature Models

Figure A.3 shows the result of running the script. The resulting model is different to the examples shown above. It does not include any alternative group but additional constraints indicating that some features excludes to others.

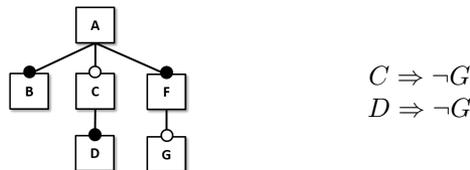


Figure A.3: Semantic-based intersection of the Example feature models

Pros and Cons Acher et al. discussed the pros and cons of this approach [11]. Among the pros, the semantic-based implementation provides a complete and sound solution for operations such as intersecting feature models. The resulting model represent the intersection of the valid configurations of the inputs. Among the cons, this resulting model may exhibit a hierarchy that is very different than the inputs, and therefore, hard to understand by the users.

A.2.2 Reference-based implementation

The *Reference-based implementation* does not use the encoding of the feature models into propositional logic. Instead of, it creates a new feature model that (1) aggregates the input models with a new feature model, named “the view”, that includes the features relevant to the user; and (2) includes a set of relationships and constraints that implement the operation.

For the *intersection merge*, the resulting model is a compound structure comprising:

- **a synthetic root**, the parent of three feature models: the view and the inputs.
- **the view model**, a feature model that includes the common features of the input models. All its features are optional to avoid additional constraints on these features.
- **the input models** that become mandatory children of the root, and
- **additional constraints** that establish bi-implications among the corresponding feature in all the models, and negate the features that are not common to all the input models.

Figure A.4 shows a reference-based solution to the example. Note that the synthetic root r is the parent of the view and the input models. In the view model, all the common features appear as optional children features of the A feature. And regarding the constraints, note that all the features in the view model are linked to the corresponding features in the input models. The features included in a model but not in the others are negated.

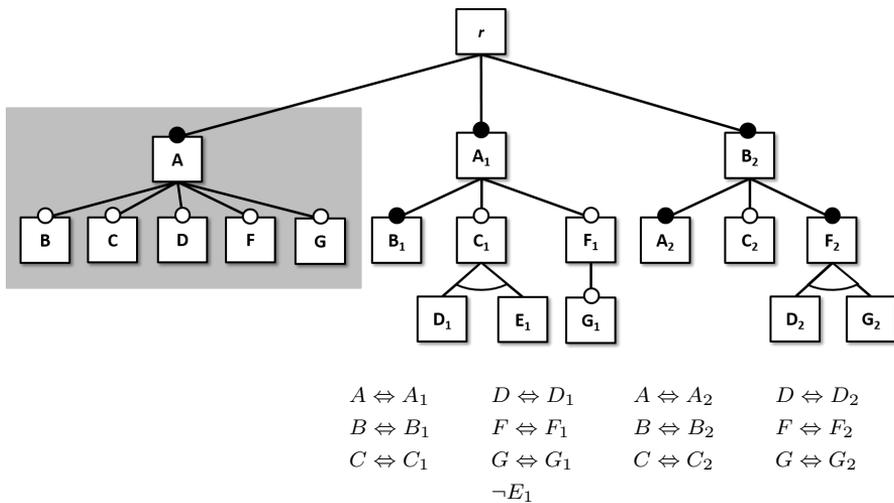


Figure A.4: Reference-based intersection of the Example feature models

FAMILIAR implementation To execute a reference-based intersection, it is possible to use the `aggregateMerge intersection` command.

```

1 // given the example fm1 and fm2
2
3 // intersect the input models
4 fmr = aggregateMerge intersection { fm1 fm2 }

```

Listing A.2: Reference-based Intersection Merge of the Example Feature Models

Figure A.5 shows the result using FAMILIAR. Note that its implementation vary from our previous example: The view model uses the same hierarchy of the first feature model instead of a flat hierarchy. The input feature models are modified to include all the features, e.g., the second model now includes the feature *E*. And finally, some constraints are included to negate these features. However, despite the differences, both solutions are equivalent.

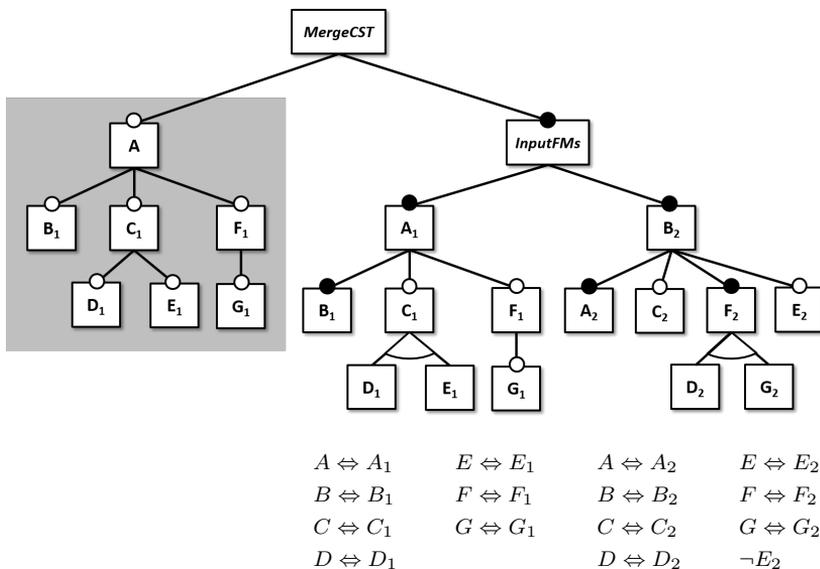


Figure A.5: Reference-based intersection of the Example feature models

Pros and Cons Acher et al. discussed this approach too [11]. Regarding its disadvantages, we can mention two of them. On one hand, the structure of the result is a composite model. A user may consider the resulting model awkward or hard to understand. On the other hand, the model may have a proliferation of constraints to maintain the same value on the all the corresponding features. The number of constraints may disturb the algorithms for processing and analysis and increase their computation time.

A.2.3 Hybrid implementations

The *hybrid-implementations* combines the above techniques. Basically, these approaches take the composite structure produced by the *reference-based* implementations, and use the propositional encoding of that structure to generate a simpler feature model using *semantic-based* techniques.

Two types of hybrid strategies have been defined [11]: (1) to perform a *Local Synthesis* of the result feature model, or (2) to *Slice* the composite structure to eliminate the non-relevant variables

Local Synthesis An alternative is to synthesize a new feature model from the features and constraints of the composite model. In general, this strategy performs similar steps to the used by the semantic based approach. It uses a propositional encoding of the composite model to generate a implication graph and determine a tree-structure for the result model.

Slice Other alternative is to *Slice* the composite structure. The Slice is an operation that produce a new feature model including only a set of user-provided relevant features. Internally, this operation processes the formula to remove the variables by *existential quantification*. The new formula represents the exact same valid configurations of the original. This formula is used to synthesize a new feature model using the techniques mentioned above.

FAMILIAR implementation There are commands for *synthesize* and *slice* feature models in FAMILIAR. An script may execute first an `aggregateMerge` and the execute on of these commands Listing A.2 shows an example.

```

1 // given the example fm1 and fm2
2
3 // intersect the input models
4 fmt = aggregateMerge intersection { fm1 fm2 }
5
6 // 0. obtain the features in the view
7 // in the result, the the root of the "view" is fmt.A
8 view_features = { fmt.A } ++ fmt.A.*
9
10 // 1. Local Synthesis
11 fmr1 = ksynthesis fmt over view_features
12
13 // 2. Slice
14 fmr2 = slice fmt including view_features

```

Listing A.3: Reference-based Intersection Merge of the Example Feature Models

Although both strategies rely on processing the propositional encoding of the composite models, they may produce different results. The *Local Synthesis* builds a new formula based on the features and constraints of the model. This

synthesis may result in a over-approximation of the set of valid configurations and, therefore, in a incomplete or erroneous intersection. The *Slice*, on the other hand, processes the actual propositional encoding of the input models, and produces correct responses all the time.

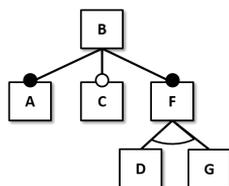
Pros and Cons Acher et al. discussed pros and cons of the hybrid approaches [11]. As an advantage, in contrast to the reference-based implementations, the hybrid strategies produce simpler feature models. Users trying to read or understand the model do not find repeated features as with these approaches. Regarding the different strategies that can be used, the *Slice* produces better results. The *Slice* strategy offers a complete and sound solution that can be customized and extended [11]. In contrast, the approaches using Local Synthesis may produce inexact results.

A.3 Our approach: Adding constraints

As mentioned in Chapter 5, we are considering a different approach. Considering the reference-based feature model, instead of creating a composite structure with the input models, we reuse the structure of one of the models and add additional constraints to the model to implement the operation.

Given two feature models fm_1 and fm_2 , the intersection can be implemented by (1) adding to the first model fm_1 the constraints that represent fm_2 negating the features that do not exist in fm_1 , (2) adding to fm_1 other constraints that negate the features in fm_1 that are not in fm_2 , and (3) slicing the resulting model in order to include only the features that are common.

For instance, Figure A.6 shows the constraints defined in the hierarchy of the feature model fm_2 .



(a) fm_2

$$\begin{aligned}
 A &\Leftrightarrow B \\
 C &\Rightarrow B \\
 F &\Leftrightarrow B \\
 F &\Rightarrow (D \vee G) \\
 D &\Rightarrow \neg G
 \end{aligned}$$

(b) Constraints represented in fm_2

Figure A.6: Constraints representing the second feature models of the example

FAMILIAR implementation FAMILIAR provides means to add constraints to a feature model. There is an `addConstraint` command that can be used. Listing A.4 shows code to take the first model of the example and add the constraints represented in the second.

```

1 // given fm1 and fm2
2
3 // copy the first model into a temporary model
4 fmt = copy fm1
5
6 // add the constraints of the second feature model
7 addConstraint constraint ( A biimplies B ) to fmt
8 addConstraint constraint ( C implies B ) to fmt
9 addConstraint constraint ( F biimplies B ) to fmt
10 addConstraint constraint ( F implies D or G ) to fmt
11 addConstraint constraint ( D implies ! G ) to fmt
12 // add the constraints negating the non-common features
13 addConstraint constraint ( ! E ) to fmt
14
15 // obtain the set of common features
16 fm1_features = features fm1
17 fm2_features = features fm2
18 fmr_features = setIntersection fm1_features fm2_features
19 // slice the result from the temporary model
20 fmr = slice fmt including fmr_features

```

Listing A.4: Example FAMILIAR Implementation of the Conditional Intersection Merge

Although the example uses literals to add the constraints, FAMILIAR provides a set of functions to obtain information of the constraints of a model. For instance, `getImpliesHierarchy` aims to obtain the constraints defined in the structure of a feature model. `getImpliesConstraint`, `getExcludesConstraint` and `getBiimpliesConstraint` obtain the cross-tree constraints. `computeBiimplies` and `computeExcludes` can be used to find others. The constraints obtained from these functions may be used as part of the process.

Discussion Our implementation is an hybrid approach. On one hand, like the reference-based implementations, it aims to reuse the structure of one of the input models. On the other hand, like the semantic-based implementation, it uses the encoding of the feature models to determine which additional constraints must be added, and use a synthesis algorithm to produce the resulting feature model.

Our approach exploits that the semantic formula calculation for the intersection is a conjunction of clauses. The operation can be implemented as a process that adds constraints to one of the models. Other operations, such as the union merge, that include disjunctions cannot be implemented just adding constraints. In order to implement these operations by modifying one of the input models, it is possible that some features must be added and copied. This strategy must be used with caution to achieve the expected results.

A.4 Comparison

Acher et al. proposed a framework to select the appropriated technique to implement an operation on feature models [11]. The framework comprises: (1) the set of *techniques* presented above, (2) a set of *dimensions for evaluation* of these techniques, and (3) a *comparison table*.

Dimensions of Evaluation The framework uses five dimensions to evaluate each approach.

Diagram Quality refers to how each approach produces feature models that are correct and easier to understand by users. As mentioned before, almost all the approaches produce complete and sound solutions. Only the hybrid implementation using local synthesis may produce an over-approximation. Regarding the structure of the result all the approaches, except the reference-based implementations produce simple models that do not repeat features. Our solution uses the slice operation to produce a simple feature model as result.

Reasoning refers to how amenable are the resulting feature model for other processing and analysis operations. Here, the reference-based implementation is weak because it produces models with features non-relevant to the user and additional constraints that maintains the same values for these features. Analysis operations may be disturbed by the large number of features and constraints. The hybrid approaches that use local synthesis are weak too. They may produce models with an over-approximation that affect the results of the analysis. Again, our approach uses the slice operation to produce a correct model with only the relevant features.

Traceability refers to how the approach can maintain relationships among the features in the resulting model and the elements in the source models. Here the semantic-based approaches are weak because they translate the models into a propositional formula and the references to the original models are lost. In contrast, Reference-based and hybrid approaches maintain references to the original features. Our approach maintains references to the features of the first feature model but not to the features of the second.

Customizability refers to how the approach can be customized or can be used to implement other operations. Here the hybrid approaches are the winners. A lot of new operations can be defined by combining multiple operations with a local synthesis or the slice operation. Our approach, however, it is based on modifying one of the feature models. Although this is very plausible to implement the intersection, may result hard to implement other operations.

Composability refers to the ability of compose (i.e., to combine) operations to produce different results. Operations implemented using the reference-based approaches are the hardest to compose. These approaches copies and renames the features in the process. It is very likely that two operations end referring to different features and producing non-expected results. Hybrid approaches using local synthesis are not the best option neither. Although they may maintain the names and references of the features in the input models, they may produce models with over-approximations. Composing these operations may produce unexpected results too. Our approach uses the slice operation. This help to produce correct models that can be used in other operations easily.

Comparison Table Table A.2 summarizes the discussions and results. Each approach is classified according to the same scale for each dimension. The best solutions are classified with *A* while the worst with *C*. Note that many solutions are equivalent for some criteria. For instance, there are many that produce high-quality models and produce correct results. This is the same table proposed by Acher et al. [11] but with an additional column presenting our approach.

	Semantic (Denotational)	Reference (Operational)	Hybrid		
			Local Synthesis	Slicing	Adding Constraints
Diagram Quality	A	C	B	A	A
Reasoning	A	C	C	A	A
Traceability	C	A	A	A	A
Customizability	C	B	A	A	B
Composability	A	C	B	A	A

Table A.2: Comparison of approaches to implement operations

Our evaluation is similar to the presented previously [11]. The slicing-based techniques are the most suited in the general case. They produce correct models that include only the relevant features. In fact, our own approach uses the slice operation for that reason.

A.5 Conclusions

In this chapter we presented different implementations of the *intersection merge* operation:

1. the *semantic-based implementation*, that operates on the equivalent propositional logic of the input feature models, produce a new propositional formula with the result, and derives a new feature model based on a spanning tree of the corresponding implication graph;
2. the *reference-based implementation*, that uses a new “view” feature model which includes the other models and relationships and constraints that implement the operation; and

3. *hybrid implementations* that combines techniques of the previous two approaches to produce simpler models.
4. and the *new type of hybrid implementation* we are proposing, an *approach that adds constraints* to one of the models and uses the slice operation to remove the non-relevant features.

The new approach we are proposing, described in Chapter 5, is a *hybrid implementation* that combines modifications to the feature models with the slice, a semantic-based operation. In contrast to the reference-based implementation, it adds constraints to one of the input models instead of creating a composite model with a synthetic view and all the input feature models. In addition, like the hybrid approaches, it uses the slice operation to produce a model that includes only the relevant features.

A comparison among the diverse proposals is included in this chapter. We found that adding constraints to a feature model and performing a slice allow us to implement the intersection merge. However, although the intersection can be implemented in that way, there are other operations such as the strict union merge that may result hard to implement using the same strategy. For these operations, a hybrid approach using slice may be a better alternative.

Our review confirms the observations of Acher et al [11]: Except for the hybrid approach with local synthesis, all the types of implementation produce complete and sound feature models. FAMILIAR can be used to implement operations using all these approaches. If the feature models can be matched by name, the semantic-based implementation of operations such as union and intersection in FAMILIAR is straightforward. If other strategies must be defined to match the features, an hybrid implementation can be used. The functions in the language can be used to implement the matching and produce a composed structure. Once matched the features, the resulting model can be produced by slicing that structure. Hybrid strategies using the slice operation are the most suitable in the general case.

Appendix B

Alternative implementations of Conditional Intersection Merge

In this thesis, we are proposing *Conditional Intersection Merge*, a new operation to combine feature models. We use this operation to combine a feature model for a domain with a feature model for a standard. The resulting model includes an optional feature representing the standard and a set of constraints that enforces the rules of the standard when that feature is selected.

This chapter presents some alternative implementations for that operation. Some of them are based on the approaches defined by Acher et al. to implement operations on feature models [11]. An additional implementation is based on our proposal presented in Chapter 5.

Rest of this chapter is organized as follows: Section B.1 presents an overview of the *conditional intersection merge* operation, Section B.2 presents the alternative implementations, Section B.3 compares the solutions, and Section B.4 presents our conclusions.

B.1 Conditional Intersection Merge

We model the configuration options of complex products using two types of feature models: (1) *feature models for domains*, that are created by domain experts to represent the variability of each technical concern, and (2) *feature models for regulations*, that are created by other experts to represent the rules defined in regulations and standards. In the process, each expert creates and maintains their own models. However, these models must be combined to perform inter-domain analyses and to build the corresponding configuration systems.

We have defined *Conditional Intersection Merge* as an operation to combine models representing domains with feature models representing regulations.

B.1.1 Running example

Figure B.1 shows an example of a *feature model for domain* and a *feature model for regulation*. They represent the variability for hypothetical electrical transformers. The first feature model fm_1 represents the devices that can be manufactured by a company while the second model fm_2 represents the options required by an specific standard.



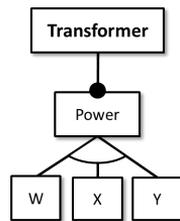
Figure B.1: Example Feature Models to apply Conditional Intersection

Regarding the product configurations: On one hand, there are options that can be manufactured but are not part of the standard. On the other hand, there are options in the standard that cannot be manufactured. Table B.1 shows the corresponding configurations.

Configurations for fm_1	Configurations for fm_2
$\{Transformer, Power, W\}$	$\{Transformer, Power, V\}$
$\{Transformer, Power, X\}$	$\{Transformer, Power, W\}$
$\{Transformer, Power, Y\}$	$\{Transformer, Power, X\}$
$\{Transformer, Power, Z\}$	$\{Transformer, Power, Y\}$

Table B.1: Sets of valid configurations for the example Feature Models

The *Intersection Merge* can be used to determine which products and options that are part of the standard can be manufactured by the company. Figure B.2 shows the intersection of the models in the example matching the Transformer and the Standard. Note that the corresponding set of configurations includes only the products of the standard that can be built.



(a) fm_{im}

Configurations for fm_{im}

- { Transformer, Power, W }
- { Transformer, Power, X }
- { Transformer, Power, Y }

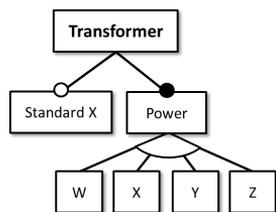
(b) Set of valid configurations

Figure B.2: Intersection of the Example Feature Models

However, the *Intersection Merge* may be too restrictive. The standard is not an option in the resulting model. We are interested on configuration systems where the user can select (or not) the standard.

Conditional Intersection Merge The *Conditional Intersection Merge* is an operation that takes a feature model for a domain and feature model for a regulation and yields a new model where the regulation is represented as an optional *Selector Feature*. In this model, the rules of the regulation are enforced only when the Selector Feature is included in the configuration.

Figure B.3 shows the *Conditional Intersection Merge* of the example models. Note that the standard is enforced when the corresponding feature is selected.



$$\text{StandardX} \Rightarrow (W \vee X \vee Y)$$

$$\text{StandardX} \Rightarrow \neg Z$$

(a) fm_r

Configurations for fm_r

- { Transformer, Power, W }
- { Transformer, Power, X }
- { Transformer, Power, Y }
- { Transformer, Power, Z }
- { Transformer, StandardX, Power, W }
- { Transformer, StandardX, Power, X }
- { Transformer, StandardX, Power, Y }

(b) Set of valid configurations

Figure B.3: Conditional Intersection of the Example Feature Models

B.2 Alternative implementations

As pointed out by Acher et al. [11], the operations on feature models can be implemented using different approaches: using *semantic-based* approaches that rely on the propositional encoding of the input feature models, *reference-based* approaches that build composite structures involving the inputs, and *hybrid techniques* that use *local synthesis* or *slice operations* to produce the resulting model. We are proposing a new technique that relies on *adding constraints* to one of the input models. This section presents some alternative implementations for the Conditional Intersection Merge.

B.2.1 Semantic-based implementation: Composing operations

The semantic-based implementation translates the feature models into a propositional formula, uses that formula to create an implication graph, and finds a spanning tree on that graph to create the result feature model. Tools and Libraries such as FAMILIAR implements operations for union, strict union and intersection of feature models using this approach.

An alternative to implement the Conditional Intersection Merge is to use these operations. Basically, we must understand the semantics of the operations that we want to build and use the corresponding operations to produce the expected result.

Semantics Operations on feature models can be defined in terms of the semantics of the participant models. Given a feature model fm , the semantics of the model $\llbracket fm \rrbracket$ is the corresponding set of valid configurations. In turn, each configuration c is a subset of the features in that model \mathcal{F}_{fm} .

The *Conditional Intersection Merge* is an operation that takes two feature models fm_1 and fm_2 and produces a feature model fm_r that includes an optional *Selector Feature* f_s and its semantics $\llbracket fm_r \rrbracket$ represents the union of (1) the products of the first model $\llbracket fm_1 \rrbracket$ (where f_s is not included), and (2) the reduced product aggregation of the selector feature and the intersection of the products of both models $\{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket)$.

$$\llbracket fm_r \rrbracket = \llbracket fm_1 \rrbracket \cup (\{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket))$$

FAMILIAR implementation FAMILIAR has support for the *union* and *intersection* merge. However, it does not support the *reduced product aggregation* with a single product. It must be implemented by modifying the feature model or by aggregating two feature models.

Configurations for $\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket$	Configurations for $\{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket)$
$\{Transformer, Power, W\}$	$\{StandardX, Transformer, Power, W\}$
$\{Transformer, Power, X\}$	$\{StandardX, Transformer, Power, X\}$
$\{Transformer, Power, Y\}$	$\{StandardX, Transformer, Power, Y\}$

Table B.2: Example product of $f_s = StandardX$ and $\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket$

In the Conditional Intersection Merge, the reduced product aggregation aims to include the feature representing the standard in all the configurations in the intersection. Note the effect in the Table B.2. After applying the intersection, the configurations in the intersection of the models in the examples are modified to include the standard. The same effect can be achieved by inserting a full-mandatory feature into the model that results of the intersection.

```

1 // example feature models
2 fm_domain = FM ( Transformer:Power; Power: ( W | X | Y | Z ); )
3 fm_std    = FM ( Transformer:Power; Power: ( V | W | X | Y ); )
4 std_name  = "Standard X"
5
6 // 1. intersect the input models
7 fmTemp = merge intersection { fm1 fm2 }
8
9 // 2. obtain the product of the selector and the intersection
10 // define a feature model for the selector
11 fms = FM ( dummy: fs; )
12 fms = extract fms.fs
13 renameFeature fms.fs as std_name
14 // insert the selector as mandatory in the resulting intersection
15 fmTemp_root = root fmTemp
16 insert fms into fmTemp_root with mand
17
18 // 3. merge the fm1 with the temporary result
19 fmr = merge union { fm1 fmTemp }

```

Listing B.1: Example FAMILIAR Implementation of the Conditional Intersection Merge

Listing B.1 shows an implementation of the conditional intersection of the models in the example using FAMILIAR. The code starts with defining the feature models and the name of the standard (see line 2). Then, the program intersects both models (line 7). Later, it determines the product of the selector and the intersection by adding a mandatory feature representing the standard (line 16). Note that a single feature cannot be added to the model. It is necessary to create a feature model to add the feature. Finally, the code obtains the union of the first feature model with the result of the previous steps (line 19).

The above mentioned reduced product aggregation can be implemented using the *aggregate* command. This command may aggregate two feature models that include at least a matching feature. Considering that the f_s in the example, i.e., *StandardX*, is not a feature in fm_1 nor fm_2 , it is necessary to perform an additional task. It is possible to create a feature model (1) which root is the root feature of the intersection and (2) the standard is a mandatory feature. Then, it is possible to aggregate both feature models matching the root features. Listing B.2 shows the code.

```

1 // example feature models
2 fm_domain = FM ( Transformer: Power; Power: ( W | X | Y | Z ); )
3 fm_std    = FM ( Transformer: Power; Power: ( V | W | X | Y ); )
4 std_name  = "Standard X"
5
6 // 1. intersect the input models
7 fmTemp = merge intersection { fm1 fm2 }
8
9 // 2. obtain the product of the selector and the intersection
10 // define a feature model for the selector
11 fms = FM ( dummy: fs; )
12 fms = extract fms.fs
13 renameFeature fms.fs as std_name
14 // rename the root with the name of the root of the intersection
15 fmTemp_name = name fmTemp
16 renameFeature fms.dummy as fmTemp_name
17 // aggregate both models
18 fmTemp = aggregate { fmTemp fms }
19
20 // 3. merge the fm1 with the temporary result
21 fmr = merge sunion { fm1 fmTemp }

```

Listing B.2: Example FAMILIAR Implementation of the Conditional Intersection Merge

Discussion The Conditional Intersection Merge can be defined by using the operations defined in FAMILIAR. There are some operations and step that are complicated than the others. For instance, to insert a feature into a model, it is necessary to create a feature model and insert that model into the other.

Regarding the configuration semantics, using the operations supplied by FAMILIAR ensures that the solution produces the expected results. Regarding the model hierarchy, this implementation converts the models to propositional logic and derives the corresponding model at least two times: one for the *merge intersection* and another for the *merge sunion* (the lines 7 and 19 in Listing B.1). It is possible that the resulting model end with a hierarchy very different than the exhibited by the input feature models.

B.2.2 Semantic-based implementation: Formula Calculation

Instead of using the existing operations, we can create a new one. In FAMILIAR each operation performs three steps: It (1) calculates a formula that represents the result, (2) derives an Implication Graph from that formula, and (3) derives a Feature Tree. One operation differs to the others only in the first step. Different operations calculate the formula for the result in a different way. To implement a new operation, we must define how to calculate the corresponding formula.

Formula Calculation Considering the semantics presented in Section B.2.1, we can determine the formula to calculate the *Conditional Intersection Merge* using the well-known formulas for the *intersection* and the *union* merge [2].

Union merge: Given two feature models fm_1 and fm_2 with the corresponding formulas ϕ_1 and ϕ_2 , the model fm_{\cup} resulting of the *union merge* $\llbracket fm_{\cup} \rrbracket = \llbracket fm_1 \rrbracket \cup \llbracket fm_2 \rrbracket$ is the formula ϕ_{\cup} such that:

$$\phi_{\cup} = (\phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})) \vee (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}))$$

where

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

Intersection Merge: The model fm_{\cap} resulting of the *intersection merge* $\llbracket fm_{\cap} \rrbracket = \llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket$ is the formula ϕ_{\cap} such that:

$$\phi_{\cap} = (\phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})) \wedge (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}))$$

Conditional Intersection Merge: The Conditional Intersection Merge (see Section B.2.1) has been defined as:

$$\llbracket fm_r \rrbracket = \llbracket fm_1 \rrbracket \cup (\{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket))$$

Letting $fm_{1\cap 2} = \llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket$, the corresponding $\phi_{1\cap 2}$ is determined by

$$\begin{aligned} \phi_{1\cap 2} &= (\phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})) \wedge (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})) \\ &= \phi_1 \wedge \phi_2 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}) \end{aligned}$$

Letting $fm_{f\otimes 1\cap 2} = \{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket)$, the corresponding formula $\phi_{f\otimes 1\cap 2}$ is

$$\phi_{f\otimes 1\cap 2} = f_s \wedge \phi_1 \wedge \phi_2 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})$$

and the set of features of $fm_{f\otimes 1\cap 2}$ is:

$$\mathcal{F}_{f\otimes 1\cap 2} = f_s \cup (\mathcal{F}_{fm_2} \cap \mathcal{F}_{fm_1})$$

Note that $\mathcal{F}_{f_{\otimes 1 \cap 2}}$ includes the f_s feature and the features that are common to both models. In consequence,

$$\begin{aligned}\mathcal{F}_{f_{\otimes 1 \cap 2}} \setminus \mathcal{F}_{fm_1} &= f_s \cup (\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \\ \mathcal{F}_{fm_1} \setminus \mathcal{F}_{f_{\otimes 1 \cap 2}} &= \mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}\end{aligned}$$

Now, considering $\llbracket fm_r \rrbracket = \llbracket fm_1 \rrbracket \cup (\{f_s\} \otimes (\llbracket fm_1 \rrbracket \cap \llbracket fm_2 \rrbracket))$, the corresponding formula ϕ_r is:

$$\begin{aligned}\phi_r &= (\phi_1 \wedge \text{not}(\mathcal{F}_{f_{\otimes 1 \cap 2}} \setminus \mathcal{F}_{fm_1})) \vee (\phi_{f_{\otimes 1 \cap 2}} \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{f_{\otimes 1 \cap 2}})) \\ &= (\phi_1 \wedge \neg f_s \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})) \vee (\phi_{f_{\otimes 1 \cap 2}} \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})) \\ &= (\phi_1 \wedge \neg f_s \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})) \\ &\quad \vee (f_s \wedge \phi_1 \wedge \phi_2 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})) \\ &= \phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge (\neg f_s \vee (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}))) \\ &= \phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge (f_s \Rightarrow (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})))\end{aligned}$$

Given two feature models fm_1 and fm_2 with the corresponding formulas ϕ_1 and ϕ_2 , the model fm_r resulting of the *conditional intersection merge* can be represented with a formula ϕ_r such that:

$$\phi_r = \phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge (f_s \Rightarrow (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})))$$

This formula denotes that: (1) The resulting set of configurations contains all the configurations valid against the first model, i.e., ϕ_1 . (2) None of the configurations include features in fm_2 not included in fm_1 , i.e., $\text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$. (3) And, when f_s is included in the configuration, only the configurations valid against the second model that not include features of fm_1 not included in fm_2 are valid, i.e., $(f_s \Rightarrow (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})))$.

FAMILIAR implementation FAMILIAR does not support the definition of new operations based on a formula calculation. Semantic-based operations are implemented as part of the platform in Java. In order to create a new operation using this approach, it is necessary to modify or extend the source code of the platform. The corresponding Java implementation is out of the scope of this chapter.

Discussion The formula of the Conditional Intersection Merge is a conjunction of the formula representing the first model and a set of additional constraints.

Regarding the configuration semantics, the approaches based on formula calculation produce feature models that represent the expected results. They use algorithms that produce a hierarchy and a set of constraints that represent that formula. However, they may produce models with hierarchies very different than the existing in the input models. This can be an obstacle to the users trying to understand and use the resulting models.

B.2.3 Reference-based implementation

The *Reference-based implementation* does not use the semantics or the encoding of the feature models into propositional logic. It yields a feature model that comprises a new feature model, named “the view”, the input feature models and a set of relationships and constraints that implement the operation. To implement the Conditional Intersection Merge, we must define how that structure can be derived from the input models.

Structure for the Conditional Intersection Merge The result feature model is a compound structure that comprises the view, both input models, and the additional constraints. Figure B.4 shows the resulting structure for the example.

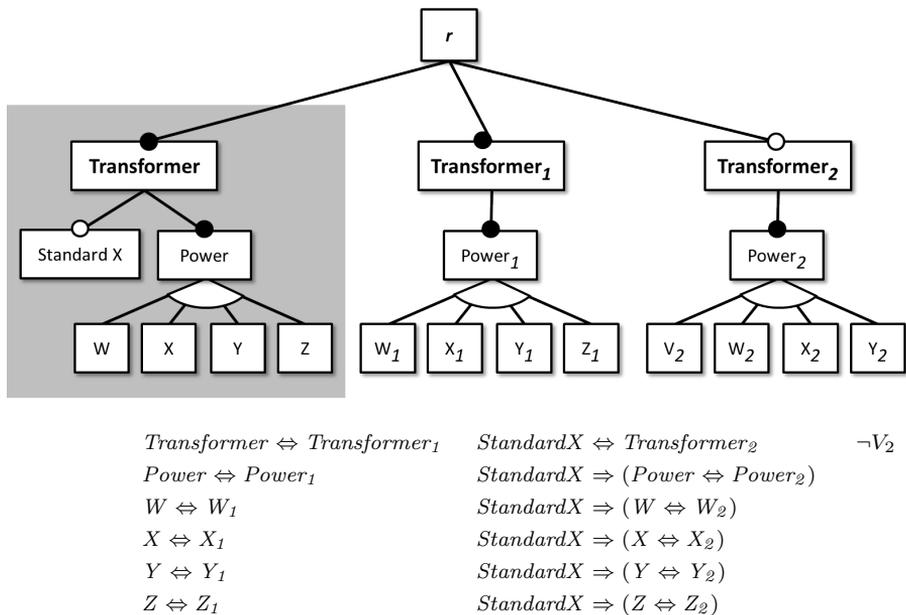


Figure B.4: Reference-based Conditional Intersection of the Example feature models

The View represents the features that are relevant to the user. It must have the same structure of the first feature model considering that all the products of that model can be configured in the result when the standard is not selected. In addition, it must include an optional feature representing the standard (e.g., *StandardX*).

The Model for the Domain, the first model, representing the products that can be selected when the feature is not selected, is mandatory. Basically, the constraints defined in that model must be enforced all the time.

The Model for the Regulation or Standard, the second model, representing the rules that must be enforced only when the standard is selected is optional.

and the **Additional Constraints** connect elements in all these models.

- Regarding **the view and the first model**, there are bi-implications that maintain the corresponding features in models with the same value.
- Regarding **the view and the second model**, there are constraints denoting that the corresponding values must have the same value only when the standard is selected.
- There are **constraints** negating all the features in the second model that do not exist in the first.

FAMILIAR implementation FAMILIAR includes several commands that can be used to copy and modify feature models. In addition, it is possible to iterate over a set of feature to, for instance, rename the features and create constraints. Listing B.3 shows an implementation of the Conditional Intersection of the models in the example.

```

1 // example feature models
2 fm_domain = FM ( Transformer: Power; Power: ( W | X | Y | Z ); )
3 fm_std    = FM ( Transformer: Power; Power: ( V | W | X | Y ); )
4 std_name  = "Standard X"
5
6 // 1. create the view
7 // copy the model for the domain
8 fm_view = copy fm_domain
9 // create a selector feature for the standard
10 fms = FM ( dummy: fs; )
11 fms = extract fms.fs
12 renameFeature fms.fs as std_name
13 // insert the selector as optional in the view
14 fm_view_root = root fm_view
15 insert fms into fm_view_root with opt
16
17 // 2. create the first model
18 // copy the model for the domain
19 fm_first = copy fm_domain
20 // renames the features adding 1 to the end
21 fs_first = features fm_first
22 foreach (f in fs_first) do
23   n = name f
24   n = strConcat n "1"
25   renameFeature f as n
26 end

```

Listing B.3: Example FAMILIAR Implementation of the Conditional Intersection Merge

```

16 // continuation of the previous listing
17
18 // 3. create the second model
19 // copy the model for the standard
20 fm_second = copy fm_std
21 // renames the features adding 2 to the end
22 fs_second = features fm_second
23 foreach (f in fs ) do
24     n = name f
25     n = strConcat n "2"
26     renameFeature f as n
27 end
28
29 // 4. aggregate the models
30 fmr = aggregate { fm_view fm_first fm_second }
31
32 // 5. add the constraints
33 // NOTE: for the sake of simplicity, we add the features using literals
34 // constraints between the view and the first model
35 addConstraint constraint ( Transformer <-> Transformer1 ) to fmr
36 addConstraint constraint ( Power <-> Power1 ) to fmr
37 // :
38 // constraints between the view and the second model
39 addConstraint constraint ( StandardX <-> Transformer2 ) to fmr
40 addConstraint constraint ( StandardX -> (Power <-> Power2) ) to fmr
41 // :
42 // additional constraints
43 addConstraint constraint ( ! V2 ) to fmr

```

Listing B.4: Example FAMILIAR Implementation of the Conditional Intersection Merge

Discussion The reference-based implementation of the Conditional Intersection produces models that are more complex than the produced in the other approaches. Although it produce correct results, the structure of the resulting model a major drawback. A user trying to understand and, for instance, debug the model may be distracted by the multiple structures and the additional constraints in the model. In addition, the repeated features increment the number of variables and constraints and may disturb the algorithms for processing and analysis.

B.2.4 Hybrid implementations: Slice of a compound model

The *hybrid-implementations* combines semantic-based and reference-based techniques. Basically, these approaches aim to improve the generation of the tree structure of the model which is the main weakness of the reference-based implementations.

Acher et al.[11] identified two hybrid approaches: one that use local synthesis of a feature model, and another that use the slice operation. The first can produce models with over-approximations and is not recommended. In contrast, the second is the most suitable for the general case. Here we describe the latter: the implementation using slice.

Slice of a compound model If you have a reference based implementation of the Conditional Intersection, implementing the hybrid approach using the slice operation is straightforward. It is only necessary to slice the model in order to maintain only the features of the view and remove the features of the other two models. The slice operation removes these features but defines new constraints that maintain the consistency of the result.

FAMILIAR implementation Slicing a feature model in FAMILIAR is straightforward. There is an *slice* command that can be used. Listing B.5 shows how to slice the model that results of the above reference-based implementation. Once we have defined the reference-based solution, we must define which features must be kept (see line 59) and use the `slice` (line 60).

```

55 // continuation of the previous listing
56
57 // 6. slice the model
58 // NOTE: for the sake of simplicity, we define the features using literals
59 to_include = { fmr.Transformer } ++ fmr.Transformer.*
60 fmr = slice fmr including to_include

```

Listing B.5: Example FAMILIAR Implementation of the Conditional Intersection Merge

Discussion The hybrid approach based on slice is very powerful. It allow us to define new operations using a “simple to define” reference-based approach, but that produce simpler models. This approach points to reduce the complexity of the resulting models in these approaches. Instead of a model with multiple sub-structures, the resulting model here is a simple hierarchy where the features are not repeated and there are not additional constraints to maintain synced their corresponding values.

B.2.5 Implementation Adding Constraints

As mentioned in Chapter 5, we are considering a different approach. Instead of generating a new structure, we use the structure of one of the models and add additional constraints to the model to implement the operation.

Effect of adding elements to the model Each modification to a feature model alters the corresponding formula. While some operations add constraints to the formula, others may modify the existing clauses. For instance, if we add an optional feature to a feature model, the modification adds a constraint. The corresponding formula is the conjunction of the formula of the original and the expression corresponding to the new element. Figure B.5 shows the propositional formula after adding an *StandardX* optional feature. Note that the result is the conjunction of the original formula and a new clause $StandardX \Rightarrow Transformer$.

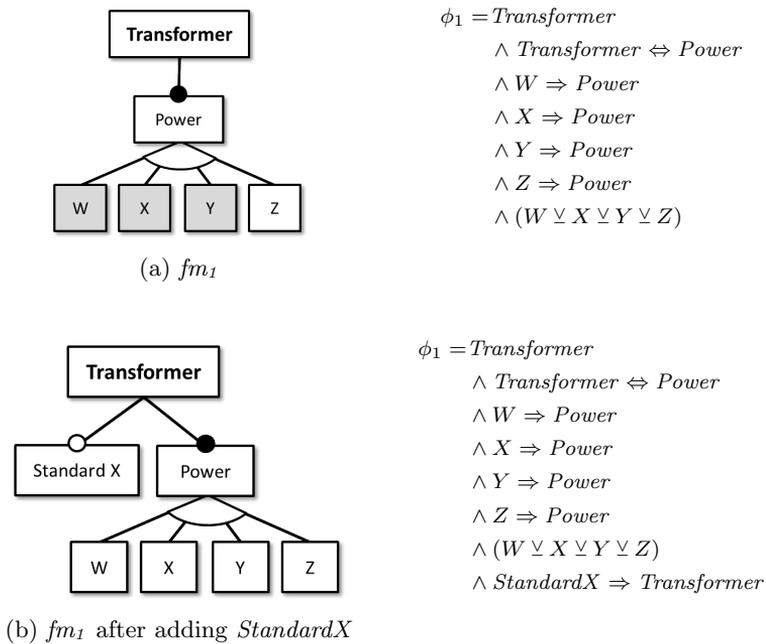


Figure B.5: Propositional Formula after adding an optional feature

Figure B.6 shows the effect of adding a constraint. The formula becomes the conjunction of the original formula and the expression corresponding to the new constraint.

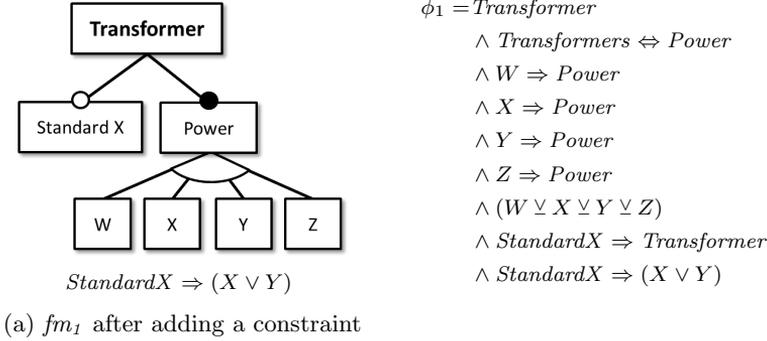


Figure B.6: Propositional Formula after adding a constraint

Implementation by Adding constraints Consider the formula calculation for the Conditional Intersection Merge.

$$\phi_r = \phi_1 \wedge \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1}) \wedge (f_s \Rightarrow (\phi_2 \wedge \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2})))$$

Note that ϕ_1 is the formula of the feature model fm_1 . To implement the Conditional Intersection Merge, we must add to fm_1 the constraints that correspond to: $\phi_{c1} = \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$, $\phi_{c2} = (f_s \Rightarrow \phi_2)$ and $\phi_{c3} = (f_s \Rightarrow \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}))$

$\phi_{c1} = \text{not}(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$: In the original formula, there is not any feature $f \in (\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$. We can implement ϕ_{c1} without adding constraints. We must maintain the features of fm_1 without adding new features from fm_2 . It is possible that some clauses in ϕ_{c2} include these features. For each clause, we can take all the features in $(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$, set them to false and remove them before add the corresponding constraint to fm_1 .

$\phi_{c2} = (f_s \Rightarrow \phi_2)$: ϕ_{c2} can be determined easily from the formula ϕ_2 that represents fm_2 . As mentioned before, in our implementation, we set to false and remove the variables that correspond to $(\mathcal{F}_{fm_2} \setminus \mathcal{F}_{fm_1})$.

$\phi_{c3} = (f_s \Rightarrow \text{not}(\mathcal{F}_{fm_1} \setminus \mathcal{F}_{fm_2}))$: Here we must determine the features in fm_1 that do not exist in fm_2 . ϕ_{c3} can be determined easily from that set.

The Conditional Intersection Merge can be implemented by adding (1) an optional feature for the standard, (2) a set of constraints denoting that the feature representing the standard implies the constraints defined in the model of the standard (the second model) after setting to false and removing the variables that do not exist in the model of the domain (the first model), and (3) constraints denoting that the standard implies the negation of the features in the domain that do not exist in the standard.

Example For instance, consider the feature models in the running example. The formula ϕ_1 for the model of the domain and the formula ϕ_2 for the model of the standard are the following.

$$\begin{aligned}\phi_1 &= \text{Transformer} \wedge (\text{Transformer} \Leftrightarrow \text{Power}) \\ &\quad \wedge (W \Rightarrow \text{Power}) \wedge (X \Rightarrow \text{Power}) \wedge (Y \Rightarrow \text{Power}) \wedge (Z \Rightarrow \text{Power}) \\ &\quad \wedge (W \vee X \vee Y \vee Z) \\ \phi_2 &= \text{Transformer} \wedge (\text{Transformer} \Leftrightarrow \text{Power}) \\ &\quad \wedge (V \Rightarrow \text{Power}) \wedge (W \Rightarrow \text{Power}) \wedge (X \Rightarrow \text{Power}) \wedge (Y \Rightarrow \text{Power}) \\ &\quad \wedge (V \vee W \vee X \vee Y)\end{aligned}$$

We can take the formula ϕ_2 and obtain a new formula ϕ'_2 after setting to false and removing the V variable that does not exist in the model of the domain.

$$\begin{aligned}\phi'_2 &= \text{Transformer} \wedge (\text{Transformer} \Leftrightarrow \text{Power}) \\ &\quad \wedge (\mathbf{false} \Rightarrow \text{Power}) \wedge (W \Rightarrow \text{Power}) \wedge (X \Rightarrow \text{Power}) \wedge (Y \Rightarrow \text{Power}) \\ &\quad \wedge (\mathbf{false} \vee W \vee X \vee Y) \\ &= \text{Transformer} \wedge (\text{Transformer} \Leftrightarrow \text{Power}) \\ &\quad \wedge (W \Rightarrow \text{Power}) \wedge (X \Rightarrow \text{Power}) \wedge (Y \Rightarrow \text{Power}) \\ &\quad \wedge (W \vee X \vee Y)\end{aligned}$$

To implement the Conditional Intersection, it is possible to add the constraint $\phi_{c2} = (\text{Standard}X \Rightarrow \phi'_2)$ to fm_1 .

$$\begin{aligned}\phi_{c2} &= \text{Standard}X \Rightarrow (\text{Transformer} \wedge (\text{Transformer} \Leftrightarrow \text{Power}) \\ &\quad \wedge (W \Rightarrow \text{Power}) \wedge (X \Rightarrow \text{Power}) \wedge (Y \Rightarrow \text{Power}) \\ &\quad \wedge (W \vee X \vee Y)) \\ &= (\text{Standard}X \Rightarrow \text{Transformer}) \\ &\quad \wedge (\text{Standard}X \Rightarrow (\text{Transformer} \Leftrightarrow \text{Power})) \\ &\quad \wedge (\text{Standard}X \Rightarrow (W \Rightarrow \text{Power})) \\ &\quad \wedge (\text{Standard}X \Rightarrow (X \Rightarrow \text{Power})) \\ &\quad \wedge (\text{Standard}X \Rightarrow (Y \Rightarrow \text{Power})) \\ &\quad \wedge (\text{Standard}X \Rightarrow (W \vee X \vee Y))\end{aligned}$$

Consider that, after adding the formula, the formula for the resulting feature model will be $\phi_r = \phi_1 \wedge \phi_{c2} \wedge \phi_{c3}$. If we have two clauses y and $x \Rightarrow y$, this is equivalent to y . We can obtain the same result by adding ϕ_{c2} after removing the redundant constraints. For instance, in the example we have two clauses $(X \Rightarrow \text{Power})$ and $\text{Standard}X \Rightarrow (X \Rightarrow \text{Power})$. We can remove the second.

The ϕ_{c3} can be calculated from the features of fm_1 and fm_2 . The Conditional Intersection can be implemented by adding the following constraints:

$$\begin{aligned}\phi_{c2} &= (\text{Standard}X \Rightarrow (W \vee X \vee Y)) \\ \phi_{c3} &= (\text{Standard}X \Rightarrow \neg V)\end{aligned}$$

FAMILIAR implementation FAMILIAR provides commands to modify the feature model by adding features and constraints. The Conditional Intersection can be implemented using these commands. Listing B.6 shows the code to implement the example. Note that it does not execute neither a *union* nor an *intersection*.

```

1 // example feature models
2 fm_domain = FM ( Transformer: Power; Power: ( W | X | Y | Z ); )
3 fm_std     = FM ( Transformer: Power; Power: ( V | W | X | Y ); )
4 std_name   = "Standard X"
5
6 // 1. create a copy of the model for the domain
7 fmr = copy fm_domain
8
9 // 2. add the feature for the standard
10 // create a selector feature for the standard
11 fms = FM ( dummy: fs; )
12 fms = extract fms.fs
13 renameFeature fms.fs as std_name
14 // insert the selector as optional in the view
15 fmr_root = root fmr
16 insert fms into fmr_root with opt
17
18 // 3. add the constraints
19 // NOTE: we add the constraints using literals
20 addConstraint constraint( StandardX -> ( W xor X xor Y ) ) to fmr
21 addConstraint constraint( ! V ) to fmr

```

Listing B.6: Example FAMILIAR Implementation of the Conditional Intersection Merge

The above example adds a set of predefined constraints. FAMILIAR provides some functions that can be used to determine which constraints must be added. For instance, `getImpliesHierarchy` and `getBiImpliesHierarchy` returns the constraints that represent the hierarchy of a feature models. Other functions, e.g., `getImpliesConstraints` and `computeImplies`, return information of the cross-tree constraints or the derived implications.

Discussion This approach implements the Conditional Intersection Merge by adding constraints to the first feature model. In contrast to the other approaches, this implementation do not require a re-calculation of the structure of the resulting feature model. Other approaches use union or intersection operations, or transform the propositional logic into a graph that is used to determine the structure of the resulting feature model. This is an advantage for two reasons: On one hand, this helps to produce a feature model that similar to the input models. The users using these models are no be disturbed by hierarchies of features that they do not know. On the other hand, this reduces the complexity of the solution.

B.3 Comparison

We have presented some alternative implementations of the *Conditional Intersection Merge*. In order to evaluate each one, we have considered two expected qualities for the resulting model: *Configuration Semantics* and *Diagram Hierarchy*.

Configuration Semantics An expected quality of any implementation is obtaining the expected results. On one hand, the conditional intersection merge must produce a feature model that: (1) allow users to configure all the products defined in the feature model for domain when the standard is not selected. (2) allow them to configure only products that are conforming to the standard when it is selected. All the implementations shown produce the expected results. None of the solutions has an advantage over the others.

Diagram Hierarchy Another expected quality is producing feature models with a hierarchy close to the feature model for the domain. Regarding this quality,

- the semantic-based implementations rely on calculating the structure for the resulting feature model from an implication graph. The implementation that composes operations calculates the structure twice and the implementation that compute the propositional formula calculates it once. This may produce hierarchies different than the existing in the original models.
- the reference based implementation uses a compound model that exhibits a hierarchy very different than the original
- the hybrid solution that uses the slice operation calculates the hierarchy of the resulting model from an implication graph. This implementation may produce models with a different hierarchy than the exhibited by the inputs.
- finally, the solution that adds constraints reuses the same structure of the feature model of the domain. It only adds to the hierarchy an optional feature model for the standard.

Here, the implementation based on adding features and constraints has an advantage over the others.

B.4 Conclusions

In this chapter we presented different implementations of the *Conditional Intersection Merge* operation: Two *semantic-based implementations*, one that compose existing operations and another that use the encoding of the input models to calculate the propositional formula of the result. A *reference-based implementation* that build the result by composing the input models with a synthetic view model. A *hybrid implementation based on the slice operation*

that takes the compound structures of the reference-based implementation and removes the non-relevant features. And, finally, an implementation that only *adds constraints* to one of the input models.

A comparison of these approaches is included in the chapter. We evaluated if the solutions can achieve the expected results and produce feature models with the hierarchy of one of the models. According to our evaluation, all the approaches are complete and sound solutions, i.e., all produce models that represent the expected set of configurations. However, only the solution adding the constraints enforce the hierarchy of one of the input models into the result. In addition, because this solution does not calculate the feature model hierarchy from an implication graph, it not only maintains the structure, but also may reduce the computation required.

Bibliography

- [1] E. Abbasi, A. Hubaux, M. Acher, Q. Boucher, and P. Heymans. The Anatomy of a Sales Configurator: An Empirical Study of 111 Cases. In *25th International Conference on Advanced Information Systems Engineering (CAiSE'13)*, pages 162–177, June 2013.
- [2] M. Acher. *Managing Multiple Feature Models: Foundations, Language, and Applications*. PhD thesis, Université de Nice–Sophia Antipolis, France, 2011.
- [3] M. Acher, B. Baudry, P. Heymans, A. Cleve, and J.-L. Hainaut. Support for reverse engineering and maintaining feature models. In *7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13)*, pages 20:1–20:8, 2013.
- [4] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Reverse engineering architectural feature models. In *5th European Conference on Software Architecture (ECSA 2011)*, pages 220–235, 2011.
- [5] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Extraction and evolution of architectural variability models in plugin-based systems. *Software and System Modeling*, 13(4):1367–1394, 2014.
- [6] M. Acher, P. Collet, P. Lahire, and R. France. Slicing feature models. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE'11), short paper*. IEEE/ACM, 2011.
- [7] M. Acher, P. Collet, P. Lahire, and R. France. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming (SCP)*, 78(6):657–681, 2012.
- [8] M. Acher, P. Collet, P. Lahire, and R. B. France. Composing feature models. In *2nd International Conference on Software Language Engineering, (SLE 2009)*, pages 62–81, 2009.
- [9] M. Acher, P. Collet, P. Lahire, and R. B. France. Comparing approaches to implement feature model composition. In *6th European Conference on Modelling Foundations and Applications (ECMFA 2010)*, pages 3–19, 2010.

-
- [10] M. Acher, P. Collet, P. Lahire, and R. B. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, 78(6):657–681, 2013.
- [11] M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire, and R. France. Composing your Compositions of Variability Models. In *16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13)*, pages 352–369, 2013.
- [12] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. J. P. de Lucena. Refactoring product lines. In *5th International Conference on Generative Programming and Component Engineering, (GPCE 2006)*, pages 201–210, 2006.
- [13] P. Arcaini, A. Gargantini, and P. Vavassori. Generating tests for detecting faults in feature models. In *8th International Conference on Software Testing, Verification and Validation (ICST 2015)*, pages 1–10, 2015.
- [14] M. Asadi, G. Gröner, B. Mohabbati, and D. Gasevic. Goal-oriented modeling and verification of feature-oriented product lines. *Software and System Modeling*, 15(1):257–279, 2016.
- [15] T. Axling and S. Haridi. A tool for developing interactive configuration applications. *The Journal of Logic Programming*, 26(2):147–168, 1996.
- [16] E. A. Aydin, H. Oguztüzün, A. H. Dogru, and A. S. Karatas. Merging multi-view feature models by local rules. In *9th International Conference on Software Engineering Research, Management and Applications, (SERA 2011)*, pages 140–147, 2011.
- [17] D. Batory. Feature models, grammars, and propositional formulas. In *9th International Conference on Software Product Lines (SPLC'05)*, pages 7–20, 2005.
- [18] D. Benavides, A. Felfernig, J. A. Galindo, and F. Reinfrank. Automated analysis in feature modelling and product configuration. In *13th International Conference on Software Reuse (ICSR 2013)*, pages 160–175, 2013.
- [19] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6):615 – 636, 2010.
- [20] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of Feature Models. In *First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2007)*, 2007.
- [21] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using Java CSP Solvers in the Automated Analyses of Feature Models. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, pages 399–408. Springer Berlin Heidelberg, 2006.

-
- [22] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. A survey of variability modeling in industrial practice. In *The 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13)*, pages 7:1–7:8, 2013.
- [23] P. Blazek, M. Partl, and C. Streichsbeir. *Configurator Database Report 2014*. Cylines Collection, 2014.
- [24] Q. Boucher. *Engineering Configuration Graphical User Interfaces from Variability Models*. PhD thesis, University of Namur, Sept 2014.
- [25] Q. Boucher, G. Perrouin, and P. Heymans. Deriving Configuration Interfaces from Feature Models: A Vision Paper. In *Sixth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2012)*, pages 37–44. ACM, Jan. 2012.
- [26] M. Broy. Challenges in automotive software engineering. In *28th International Conference on Software Engineering (ICSE '06)*, pages 33–42, 2006.
- [27] H. Bruin and H. Vliet. Feature and feature interaction modeling with feature-solution graphs. In *Proceedings of the GCSE'01 Feature Modeling Workshop*, pages 1–4, 2001.
- [28] H. Bruin and H. Vliet. Scenario-based generation and evaluation of software architectures. In *3rd International Conference Generative and Component-Based Software Engineering*, pages 128–139, 2001.
- [29] S. Bühne, G. Halmans, K. Pohl, M. Weber, H. Kleinwechter, and T. Wierczoch. Defining requirements at different levels of abstraction. In *12th IEEE International Conference on Requirements Engineering (RE 2004)*, pages 346–347, 2004.
- [30] J. Chavarriaga, C. Noguera, R. Casallas, and V. Jonckers. Supporting Multi-level Configuration with Feature-Solution Graphs: Formal Semantics and Alloy Implementation . Technical report, Vrije Universiteit Brussel, 2013.
- [31] J. Chavarriaga, C. Noguera, R. Casallas, and V. Jonckers. Propagating Decisions to Detect and Explain Conflicts in a Multi-step Configuration Process. In *17th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2014)*. Springer, 2014.
- [32] J. Chavarriaga, C. Rangel, C. Noguera, R. Casallas, and V. Jonckers. Using multiple Feature Models to specify configuration options for Electrical Transformers: an Experience Report. In *19th International Conference on Software Product Line, (SPLC 2015)*, pages 216–224, 2015.

-
- [33] A. Classen, A. Hubaux, and P. Heymans. A formal semantics for multi-level staged configuration. In *Proceedings of the Third International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2009)*, pages 51–60, 2009.
- [34] R. Clotet Martínez, D. Dhungana, J. Franch Gutiérrez, P. Grünbacher, L. López Cuesta, J. Marco Gómez, and N. Seyff. Dealing with changes in service-oriented computing through integrated goal and variability modelling. In *2nd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS' 2008)*, pages 43–52, 2008.
- [35] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
- [36] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):143–169, 2005.
- [37] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.
- [38] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *11th International Conference on Software Product Lines, (SPLC 2007)*, pages 23–34, 2007.
- [39] D. Dhungana and P. Grünbacher. Understanding decision-oriented variability modelling. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, pages 233–242, 2008.
- [40] D. Dhungana, P. Grünbacher, and R. Rabiser. The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering*, 18(1):77–114, 2011.
- [41] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer. Structuring the modeling space and supporting evolution in software product line engineering. *J. Systems and Software*, 83(7):1108–1122, 2010.
- [42] D. Dhungana, D. Seichter, G. Botterweck, R. Rabiser, P. Grünbacher, D. Benavides, and J. A. Galindo. Configuration of multi product lines by bridging heterogeneous variability modeling approaches. In *15th International Conference on Software Product Lines (SPLC 2011)*, pages 120–129, 2011.
- [43] D. Dhungana, D. Seichter, G. Botterweck, R. Rabiser, P. Grünbacher, D. Benavides, and J. A. Galindo. Integrating heterogeneous variability modeling approaches with invar. In *7th International Workshop on*

- Variability Modelling of Software-intensive Systems, (VaMoS '13)*, pages 8:1–8:5, 2013.
- [44] O. Djebbi, C. Salinesi, and G. Fanmuy. Industry survey of product lines management tools: Requirements, qualities and open issues. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 301–306, 2007.
- [45] A. Durán, D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Flame: a formal framework for the automated analysis of software product lines validated by automated specification testing. *Software & Systems Modeling*, pages 1–34, 2015.
- [46] R. Feldt and A. Magazinius. Validity threats in empirical software engineering research - an initial survey. In *Software Engineering and Knowledge Engineering Conference (SEKE 2010)*, pages 374–379, 2010.
- [47] A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen. *Knowledge-based Configuration: From Research to Business Cases*. Morgan Kaufmann, 2014.
- [48] A. Felfernig, S. Reiterer, F. Reinfrank, G. Ninausa, and M. Jerana. Conflict detection and diagnosis in configuration. In A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, editors, *Knowledge-based Configuration: From Research to Business Cases*, pages 73–87. Morgan Kaufmann, 2014.
- [49] A. Felfernig, M. Schubert, and C. Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 26:53–62, 2 2012.
- [50] R. Flores, C. Krueger, and P. Clements. Mega-scale Product Line Engineering at General Motors. In *16th International Software Product Line Conference (SPLC '12)*, pages 259–268, 2012.
- [51] C. Freund and S. Oliver. Gains from Harmonizing US and EU Auto Regulations under the Transatlantic Trade and Investment Partnership. Policy Brief PB15-10, Peterson Institute for International Economics, 2005.
- [52] W. Friess, J. Sincero, and W. Schroeder-Preikschat. Modelling compositions of modular embedded software product lines. In *25th Conference on IASTED International Multi-Conference: Software Engineering (SE'07)*, pages 224–228, 2007.
- [53] J. A. Galindo, D. Dhungana, R. Rabiser, D. Benavides, G. Botterweck, and P. Grünbacher. Supporting distributed product configuration by integrating heterogeneous variability modeling approaches. *Information & Software Technology*, 62:78 – 100, 2015.
- [54] R. Gheyi, T. Massoni, and P. Borba. A Theory for Feature Models in Alloy. In *First Alloy Workshop*, pages 71–80, 2006.

-
- [55] R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 11 1989.
- [56] W. Grueninger. The differences between American and European Car Lighting. *Motive Magazine*, 2007.
- [57] H. Hartmann. *Software product line engineering for consumer electronics: Keeping up with the speed of innovation*. PhD thesis, University of Groningen, 2015.
- [58] H. Hartmann and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *12th International Conference on Software Product Lines (SPLC 2008)*, pages 12–21, 2008.
- [59] H. Hartmann, T. Trew, and A. Matsinger. Supplier independent feature modelling. In *13th International Conference on Software Product Lines (SPLC 2009)*, pages 191–200, 2009.
- [60] A. Hein, M. Schlick, and R. Vinga-Martins. Applying feature models in industrial settings. In *Software Product Lines: Experience and Research Directions*, pages 47–70. Springer, 2000.
- [61] S. Henneberg. Next-generation feature models with pseudo-boolean sat solvers. Master's thesis, Universitat Passau, 2011.
- [62] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.
- [63] P. Heymans, P. Schobbens, J. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. Evaluating formal properties of feature diagram languages. *IET Software*, 2(3):281–302, 2008.
- [64] A. Hubaux. *Feature-based Configuration: Collaborative, Dependable, and Controlled*. PhD thesis, University of Namur, Belgium, 2012.
- [65] A. Hubaux, M. Acher, T. T. Tun, P. Heymans, P. Collet, and P. Lahire. Separating concerns in feature models: Retrospective and support for multi-views. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*, pages 3–28. Springer, 2013.
- [66] A. Hubaux, P. Heymans, P.-Y. Schobbens, D. Deridder, and E. Abbasi. Supporting multiple perspectives in feature-based configuration. *Software and Systems Modeling (SoSyM)*, 12(3):1–23, 2011.
- [67] A. Hubaux, T. T. Tun, and P. Heymans. Separation of concerns in feature diagram languages: A systematic survey. *ACM Computing Surveys*, 45(4):51:1–51:23, 2013.

- [68] A. Hubaux, Y. Xiong, and K. Czarnecki. A User Survey of Configuration Challenges in Linux and eCos. In *Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2012)*. ACM Press, 2012.
- [69] ICONTEC. Transformadores trifásicos autorrefrigerados y sumergidos en líquido. Standard NTC 819, Instituto Colombiano de Normas Técnicas y Certificación, 1995.
- [70] ICONTEC. Transformadores de distribución trifásicos tipo pedestal autorrefrigerados con compartimientos. Standard NTC 3997, Instituto Colombiano de Normas Técnicas y Certificación, 1996.
- [71] IEC. Power transformers – all parts. Standard 60076-SER ed1.0, International Electrotechnical Commission, 2013.
- [72] IEEE. Standard for general requirements for liquid-immersed distribution, power, and regulating transformers. Standard C57.12.00–2010, Institute of Electrical and Electronics Engineers, 2010.
- [73] Im fokus: Volkswagen - kernkompetenz: Sparen, March 2006.
- [74] M. Janota. Do SAT solvers make good configurators? In *12th International Conference on Software Product Lines (SPLC 2008), (Workshops)*, pages 191–195, 2008.
- [75] M. Janota. *SAT Solving in Interactive Configuration*. PhD thesis, Department of Computer Science, University College Dublin, 2010.
- [76] M. Janota and G. Botterweck. Formal approach to integrating feature and architecture models. In *11th International Conference Fundamental Approaches to Software Engineering (FASE 2008)*, volume 4961 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2008.
- [77] M. Janota, G. Botterweck, R. Grigore, and J. Marques-Silva. How to complete an interactive configuration process? *CoRR*, abs/0910.3913, 2009.
- [78] U. Junker. QuickXplain: Preferred explanations and relaxations for over-constrained problems. In *19th National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172, 2004.
- [79] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. (CMU/SEI-90-TR-021). Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.
- [80] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Software Eng.*, 5:143–168, 1998.

-
- [81] K. C. Kang and H. Lee. Variability modeling. In *Systems and Software Variability Management: Concepts, Tools and Experiences*, pages 25–42. Springer Berlin Heidelberg, 2013.
- [82] G. Khandu-Narwane, J. A. Galindo, S. Narayanan-Krishna, D. Benavides, J.-V. Millo, and S. Ramesh. Traceability analyses between features and assets in software product lines. *Entropy*, 18(8):269, 2016.
- [83] S. Kim, D.-K. Kim, L. Lu, and S. Park. Quality-driven architecture development using architectural tactics. *Journal of Systems and Software*, 82(8):1211 – 1231, 2009.
- [84] D. Kolovos, L. Rose, A. Garcia-Dominguez, and R. Paige. The Epsilon Book, 2014.
- [85] D. Kolovos, L. M. Rose, and J. R. Williams. Using Model-to-Text Transformation for Dynamic Web-based Model Navigation. In *6th International Workshop on Models at Runtime (Models@run.time 2011)*, pages 49–60, 2011.
- [86] S. Krieter, R. Schroter, T. Thum, W. Fenske, and G. Saake. Comparing algorithms for efficient feature-model slicing. In *20th International Systems and Software Product Line Conference (SPLC 2016)*, 2016.
- [87] J. H. J. Liang, V. Ganesh, K. Czarnecki, and V. Raman. Sat-based analysis of large real-world feature models is easy. In *19th International Conference on Software Product Line, (SPLC 2015)*, pages 91–100, 2015.
- [88] M. Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the Second Software Product Line Conference (SPLC2)*, pages 176–187. Springer, 2002.
- [89] M. Mannion, J. Savolainen, and T. Asikainen. Viewpoint-oriented variability modeling. In *33rd Annual IEEE International Computer Software and Applications Conference, (COMPSAC 2009)*, pages 67–72, 2009.
- [90] M. Mendonca. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, Canada, 2009.
- [91] M. Mendonça, D. D. Cowan, W. Malyk, and T. C. de Oliveira. Collaborative product configuration: Formalization and efficient algorithms for dependency analysis. *JSW*, 3(2):69–82, 2008.
- [92] M. Mendonca, A. Wasowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *13th International Software Product Line Conference (SPLC '09)*, pages 231–240, 2009.
- [93] M. Mendonça, M. Branco, and D. Cowan. S.P.L.O.T.: software product lines online tools. In *OOPSLA Companion*, pages 761–762. ACM, 2009.

- [94] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. 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 2007)*, pages 243–253, 2007.
- [95] R. Michel, A. Hubaux, V. Ganesh, and P. Heymans. An SMT-based Approach to Automated Configuration. In *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories (SMT'12)*, pages 107–117, 2012.
- [96] V. Myllärniemi, J. Tiihonen, M. Raatikainen, and A. Felfernig. Using answer set programming for feature model representation and configuration. In *Proceedings of the 16th International Configuration Workshop (ConfWS 2014)*, pages 1–8, 2014.
- [97] J. C. Navarro and J. Chavarriaga. Using Microsoft Solver Foundation to analyse Feature Models and Configurations. In *2016 8th Euro American Conference on Telematics and Information Systems (EATIS)*, pages 1–8, 2016.
- [98] NEMA. Transformers, regulators and reactors. Standard NEMA TR1, National Electrical Manufacturers Association, 2013.
- [99] N. Niu, J. Savolainen, and Y. Yu. Variability modeling for product line viewpoints integration. In *34th Annual IEEE International Computer Software and Applications Conference, (COMPSAC 2010)*, pages 337–346, 2010.
- [100] A. Nöhrrer, A. Biere, and A. Egyed. A comparison of strategies for tolerating inconsistencies during decision-making. In *16th International Software Product Line Conference (SPLC '12)*, pages 11–20. ACM, 2012.
- [101] A. Nöhrrer, A. Biere, and A. Egyed. Managing SAT inconsistencies with HUMUS. In *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*, pages 83–91. ACM, 2012.
- [102] S. of Automotive Engineers of Japan. The automobile and technical regulations. *Journal of the Society of Automotive Engineers of Japan (JSAE)*, 69, 2015.
- [103] P. Offermann, O. Levina, M. Schönherr, and U. Bub. Outline of a design science research process. In *4th International Conference on Design Science Research in Information Systems and Technology (DESRIST '09)*, pages 7:1–7:11, 2009.
- [104] O. Oliinyk. Applying hierarchical feature modeling in automotive industry. Master's thesis, Blekinge Institute of Technology, Sweden, 2012.

-
- [105] O. Oliinyk, K. Petersen, M. Schoelzke, M. Becker, and S. Schneickert. Structuring automotive product lines and feature models: an exploratory study at opel. *Requirements Engineering*, pages 1–31, 2015.
- [106] S. Oster. *Feature Model-based Software Product Line Testing*. PhD thesis, Technische Universität Darmstadt, 2011.
- [107] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007.
- [108] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering : Foundations, Principles, and Techniques*. Springer, 2005.
- [109] R. Rabiser, P. Grünbacher, and M. Lehofer. A qualitative study on user guidance capabilities in product configuration tools. In *27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 110–119. ACM, 2012.
- [110] A. Rein-Jury. *Feature Constraint Propagation along Configuration Links for Advanced Feature Models*. PhD thesis, Technischen Universität Berlin, Germany, 2013.
- [111] M. Reiser. *Managing Complex Variability in Automotive Software Product Lines with Subscoping and Configuration Links*. PhD thesis, Technischen Universität Berlin, Germany, 2008.
- [112] M. Reiser, R. T. Kolagari, and M. Weber. Compositional variability: Concepts and patterns. In *42st Hawaii International Conference on Systems Science (HICSS-42 2009)*, pages 1–10, 2009.
- [113] M. Reiser and M. Weber. Managing highly complex product families with multi-level feature trees. In *14th International Conference on Requirements Engineering (RE'06)*, pages 149–158, 2006.
- [114] M. Reiser and M. Weber. Multi-level feature trees: A pragmatic approach to managing highly complex product families. *Requirements Engineering*, 12(2):57–75, 2007.
- [115] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 4 1987.
- [116] T. Rogoll and F. Piller. Product configuration from the customer’s perspective: A comparison of configuration systems in the apparel industry. In *Technical and Organisational Aspects of Product Configuration Systems (PETO 2004)*, pages 179–199, 2004.
- [117] D. Sabin and R. Weigel. Product configuration frameworks-a survey. *IEEE Intelligent Systems and their Applications*, 13(4):42–49, Jul 1998.

- [118] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.
- [119] J. Schroeter, M. Lochau, and T. Winkelmann. Conper: Consistent perspectives on feature models. In *8th European Conference on Modelling Foundations and Applications (ECMFA'12), ACME Workshop*, 2012.
- [120] J. Schroeter, M. Lochau, and T. Winkelmann. Multi-perspectives on feature models. In *15th International Conference on Model Driven Engineering Languages and Systems, (MODELS 2012)*, pages 252–268, 2012.
- [121] S. Segura. Automated analysis of feature models using atomic sets. In *12th International Conference on Software Product Lines (SPLC 2008), (Workshops)*, pages 201–207, 2008.
- [122] S. Segura, D. Benavides, A. R. Cortés, and P. Trinidad. Automated merging of feature models using graph transformations. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, (GTTSE 2007)*, pages 489–505, 2007.
- [123] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, and A. Ruiz-Cortés. Betty: Benchmarking and testing on the automated analysis of feature models. In *Sixth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12)*, pages 63 – 71, 2012.
- [124] S. She. *Feature Model Synthesis*. PhD thesis, University of Waterloo, Canada, 2013.
- [125] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *33rd International Conference on Software Engineering, (ICSE 2011)*, pages 461–470, 2011.
- [126] S. She, U. Ryssel, N. Andersen, A. Wasowski, and K. Czarnecki. Efficient synthesis of feature models. *Information & Software Technology*, 56(9):1122–1143, 2014.
- [127] C. T. L. L. Silva, C. Borba, and J. Castro. A goal oriented approach to identify and configure feature models for software product lines. In *Workshop em Engenharia de Requisitos (WER11)*, 2011.
- [128] J.-S. Sottet, A. Vagner, and A. G. Frey. Model transformation configuration and variability management for user interface design. In *Third International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015)*, pages 390–404, 2015.
- [129] C. Streichsbier, P. Blazek, F. Faltin, and W. Fruhwirt. Are de-facto standards a useful guide for designing human-computer interaction processes? the case of user interface design for web based b2c product configurators. In *42nd Hawaii International Conference on System Sciences (HICSS '09)*, pages 1–7, Jan 2009.

-
- [130] M. Svahnberg, J. Bosch, and J. van Gorp. On the notion of variability in software product lines. In *Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, pages 54–54, 2001.
- [131] S. Thiel, S. Ferber, T. Fischer, A. Hein, and M. Schlick. A case study in applying a product line approach for car periphery supervision systems. In *In-Vehicle Software at SAW World Congress 2001*, pages 43–55, 2001.
- [132] S. Thiel and A. Hein. Modeling and using product line variability in automotive systems. *IEEE Software*, 19(4):66–72, 2002.
- [133] S. Thiel and A. Hein. Systematic integration of variability into product line architecture design. In *Second International Conference on Software Product Lines (SPLC 2002)*, pages 130–153, 2002.
- [134] C. Thörn. *On the quality of feature models*. PhD thesis, Linköping University, 2010.
- [135] T. Thüm, D. Batory, and C. Kastner. Reasoning about edits to feature models. In *31st International Conference on Software Engineering (ICSE '09)*, pages 254–264, 2009.
- [136] A. Trentin, E. Perin, and C. Forza. Sales configurator capabilities to prevent product variety from backfiring. In *Workshop on Configuration at ECAI 2012 (ConfWS-2012)*, pages 47–54, Aug 2012.
- [137] P. Trinidad. *Automating the Analysis of Stateful Feature Models*. PhD thesis, Universidad de Sevilla, Spain, 2012.
- [138] P. Trinidad, D. Benavides, A. Durán, A. R. Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896, 2008.
- [139] P. Trinidad, A. Ruiz-Cortés, and D. Benavides. Automated analysis of stateful feature models. In *Seminal Contributions to Information Systems Engineering: 25 Years of CAiSE*, pages 375–380. Springer, 2013.
- [140] T. T. Tun and P. Heymans. Concerns and their separation in feature diagram languages: An informal survey. In *Workshop on Scalable Modelling Techniques for Software Product Lines*, 2009.
- [141] Valeo. Lighting systems: From light to advanced vision technologies. Technical Handbook 998542, Valeo Service, 2015.
- [142] P. van den Broek. Optimization of product instantiation using integer programming. In *14th International Software Product Line Conference (SPLC 2010)*, pages 107–111, 2010.
- [143] P. van den Broek. Intersection of feature models. In *16th International Software Product Line Conference (SPLC 2012), Workshop Proceedings*, pages 61–65, 2012.

-
- [144] P. van den Broek, I. Galvão, and J. Noppen. Merging feature models. In *14th International Conference on Software Product Lines (SPLC 2010), Workshop Proceedings*, pages 83–90, 2010.
- [145] M. Weber and J. Weisbrod. Requirements engineering in automotive development: Experiences and challenges. In *IEEE Joint International Conference on Requirements Engineering (RE 2002)*, pages 331–340, 2002.
- [146] J. White, D. Benavides, D. C. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortés. Automated diagnosis of feature model configurations. *Journal of Systems and Software*, 83(7):1094 – 1107, 2010.
- [147] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *12th International Software Product Line Conference (SPLC '08)*, pages 225–234, 2008.
- [148] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *34th International Conference on Software Engineering (ICSE '12)*, pages 58–68, 2012.
- [149] H. Yan, W. Zhang, H. Zhao, and H. Mei. An optimization strategy to feature models' verification by eliminating verification-irrelevant features and constraints. In *11th International Conference on Software Reuse (ICSR 2009)*, pages 65–75, 2009.