ANALYSIS OF EXTENDED FEATURE MODELS WITH CONSTRAINT PROGRAMMING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

AHMET SERKAN KARATAŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

JUNE 2010

Approval of the thesis:

ANALYSIS OF EXTENDED FEATURE MODELS WITH CONSTRAINT PROGRAMMING

submitted by **AHMET SERKAN KARATAŞ** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Halit Oğuztüzün _____
Supervisor, **Computer Engineering Dept., METU**

Assoc. Prof. Dr. Ali H. Doğru _____
Co-Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. Hakkı Toroslu _____
Computer Engineering Dept., METU

Assoc. Prof. Dr. Halit Oğuztüzün _____
Computer Engineering Dept., METU

Prof. Dr. Mehmet Tolun _____
Computer Engineering Dept., Çankaya University

Assoc. Prof. Dr. Ferda Nur Alpaslan _____
Computer Engineering Dept., METU

Asst. Prof. Dr. Pınar Şenkul _____
Computer Engineering Dept., METU

Date: 29.06.2010

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Ahmet Serkan Karataş

Signature :

ABSTRACT

ANALYSIS OF EXTENDED FEATURE MODELS WITH CONSTRAINT PROGRAMMING

Karataş, Ahmet Serkan
Ph.D., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Halit Oğuztüzün
Co-Supervisor: Assoc. Prof. Dr. Ali H. Doğru

June 2010, 81 pages

In this dissertation we lay the groundwork of automated analysis of extended feature models with constraint programming. Among different proposals, feature modeling has proven to be very effective for modeling and managing variability in Software Product Lines. However, industrial experiences showed that feature models often grow too large with hundreds of features and complex cross-tree relationships, which necessitates automated analysis support. To address this issue we present a mapping from extended feature models, which may include complex feature-feature, feature-attribute and attribute-attribute cross-tree relationships as well as global constraints, to constraint logic programming over finite domains. Then, we discuss the effects of including complex feature attribute relationships on various analysis operations defined on the feature models. As new types of variability emerge due to the inclusion of feature attributes in cross-tree relationships, we discuss the necessity of reformulation of some of the analysis operations and suggest a revised understanding for some other. We also propose new analysis operations arising due to the nature of the new variability introduced. Then we propose a transformation from extended feature models to basic/cardinality-based feature models that may be applied under certain circumstances and enables using SAT or BDD solvers in automated analysis of extended feature models. Finally, we discuss the role of the context information in feature modeling, and propose to use context information in staged configuration of feature-models.

Keywords: Extended Feature Models, Constraint Logic Programming over Finite Domains, Global Constraints, Automated Analysis of Feature Models, Context Variability

ÖZ

GENİŞLETİLMİŞ ÖZELLİK MODELLERİNİN KISIT PROGRAMLAMA İLE ANALİZİ

Karataş, Ahmet Serkan
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Halit Oğuztüzün
Ortak Tez Yöneticisi: Doç. Dr. Ali Doğru

Haziran 2010, 81 sayfa

Bu tezde, genişletilmiş özellik modellerinin kısıt programlama ile analizi için bir zemin önerilmektedir. Özellik modelleri, yazılım üretim hatlarındaki değişkenliğin modellenmesi için yapılmış öneriler arasından etkinliği ile öne çıkmıştır. Ancak endüstriyel deneyimler özellik modellerinin yüzlerce özellik ve dallar arası karmaşık ilişkiler nedeniyle çok hızlı büyüdüğünü, bu nedenle de otomatik analiz desteğinin gerekli olduğunu göstermiştir. Bahsedilen ihtiyaca binaen karmaşık özellik-özellik, özellik-öznitelik, öznitelik-öznitelik ilişkileri ve bunların yanı sıra genel kısıtlar içerebilen genişletilmiş özellik modellerinden sonlu alanlar kullanarak kısıt programlamaya bir eşleme sunulmaktadır. Daha sonra özniteliklerin karmaşık ilişkilerin içine dâhil edilmesinin özellik modelleri üzerinde tanımlanmış çeşitli çözümleme işlemleri üzerindeki etkileri tartışılmaktadır. Özniteliklerin dallar arası karmaşık ilişkilerin içine dâhil edilmesi ile ortaya çıkan yeni tür değişkenlikler nedeniyle kimi çözümleme işlemleri için düzeltilmiş tanımlamalar, kimi işlemler içinse yeniden formüle etme önerilmektedir. Ayrıca ortaya çıkan değişkenliklerin doğası nedeniyle gündeme gelen yeni çözümleme işlemleri de önerilmektedir. Daha sonra, şartlar uygun olduğunda uygulanabilecek ve genişletilmiş özellik modellerinin otomatik çözümlenmesinde SAT ya da BDD çözücülerin kullanımını mümkün kılan, genişletilmiş özellik modellerinden temel/nicelik-tabanlı özellik modellerine bir dönüştürme önerilmektedir. Son olarak, bağlam bilgisinin özellik modellemedeki rolü tartışılmakta ve özellik modellerinin aşamalarla biçimlendirilmesinde bağlam bilgisinin kullanılması önerilmektedir.

Anahtar Kelimeler: Genişletilmiş Özellik Modelleri, Sonlu Alanlar Kullanarak Kısıt Programlama, Genel Kısıtlar, Özellik Modellerinin Otomatik Olarak Çözümlenmesi, Bağlam Değişkenliği

To My Parents; Gülser & Ahmet Karataş

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to my supervisor Assoc. Prof. Dr. Halit Oğuztüzün, who has always been there to answer my unending questions. I have learnt a lot from him and our invaluable discussions.

I would like to express my gratitude to my co-supervisor Assoc. Prof. Dr. Ali H. Doğru for his invaluable support. It has always been a joy for me to study with him.

My gratitude to my family is inexpressible in any words. As for many other things, I owe a lot to them for their endless love, support and caring.

As the last, but definitely not the least, I would like to express my deepest gratitude to Derya, who has always - even in my darkest hours - been there for me, and never let me feel alone.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

BDD	– Binary Decision Diagram
C–list	– Free–Choice List
C–product	– Free–Choice Product
CLP(FD)	– Constraint Logic Programming over Finite Domains
CSP	– Constraint Satisfaction Problem
CVM	– Context Variability Model
ECR	– Extra Constraint Representativeness
ECR–A	– Extra Constraint Representativeness for Attributes
F–list	– Fully Specialized List
F–product	– Fully Specialized Product
G–list	– Generalized List
LCA	– Lowest Common Ancestor
P–list	– Partially Specialized List
P–product	– Partially Specialized Product
SAT	– Boolean Satisfiability Problem
SPL	– Software Product Line
UML	– Unified Modeling Language

CHAPTER 1

INTRODUCTION

The idea of mass customization and building platforms to develop product families has been realized successfully since a long time in different areas of engineering from automobile industry to electronics manufacturing. The benefits offered by mass customization such as reduced development costs, enhanced quality of the products, decreased maintenance costs, simplified project management resulted in a growing tendency towards the use of production lines. With the increasing number of success stories originating from different disciplines, many organizations focused their attention on implementing the product line strategy.

Systematic reuse of existing code is one of the ultimate goals of software engineering. The reuse adventure of software engineering started in sixties when software engineers introduced subroutines that are created once and used many times. In seventies engineers started to group related subroutines in modules. When Object Oriented Programming came into scene in eighties, the idea of encapsulation leveraged reuse by packing data and methods to process the data, as it was possible to hide the details from the user. Nineties were the years of components that packed objects together, which offered more independent solutions. With the new millennium use of services became more and more popular.

As the markets became more competitive software engineers began seeking ways to reuse not just the code but also other project management artifacts such as requirements, architectures, and test objects to be able to focus on families of products rather than a product alone. Within the recent decades, as software products and software-intensive systems became larger and more complex, software researchers and practitioners inspired by the success stories initiated the adoption of production lines to the realm of software engineering. Since then Software Product Lines (SPLs) attracted more attention in both academic and industrial communities.

SPLs offer effective strategies to develop families of products. Members of a product family share some common concepts that relate them to the family, and include varying properties that make them a distinguished product. Thus, capturing these *commonality* and *variability* in SPLs is a key issue. Feature modeling has proven to be an effective activity for managing and modeling the commonality and variability in product families. Since their introduction important extensions have been devised to feature models to respond to growing expectations, and feature models have become more powerful to enable complex relationships among features to be expressed.

However, industrial experience showed that feature models tend to grow too large to manage manually. As a result automated support for reasoning on feature models has become a necessity. With the increasing need to automated reasoning on feature models the formal semantics of feature models gained more importance. Software researchers have been trying to establish a solid foundation for giving a formal semantics to feature models and reasoning on these models since their introduction. Theory and implementation behind these foundations have evolved and matured in parallel to the evolving feature model paradigm. But there are still some gaps between the theory and practice that has to be filled.

In this dissertation we address some of the challenges that arise due to the efforts to fill these gaps. We adopt constraint programming as a formalism to provide a formal semantics to feature models and as a base for automated reasoning. We propose suggestions for the evolution of automated reasoning literature in the light of the discussed concepts. We also propose transformations that will enable the use of previously proposed automated reasoning approaches on more powerful models, namely extended feature models.

As discussed in the previous paragraphs automated analysis of feature models is desirable. There are a number of proposals for automated reasoning on feature models reported in the literature [7]. However, none of the approaches reported so far can handle complex constraints that involve feature attributes effectively. As the first step for the construction of the groundwork we present a mapping from extended feature models, which may include complex feature-feature, feature-attribute, and attribute-attribute cross-tree constraints, to constraint logic programming over finite domains, designated CLP(FD). The mapping presented is capable of translating every extended feature model to a CLP(FD) program as long as the domains of the attributes in the model are finite. Note that, this mapping can also be used to translate basic and cardinality-based feature models into CLP(FD) programs as well.

It is often the case that modelers and developers face requirements that impose constraints on all or a set of features rather than a feature alone. Global constraints provided by the CLP(FD) systems offer eligible representations for such requirements. As the mapping discussed translates feature models into CLP(FD) programs, it enables using global constraints on feature models. We use this facility and as the second step for the construction of the groundwork, discuss using the global constraints in the automated analysis of feature models. We also present some performance test results and discuss the benefits of using global constraints on feature models.

As the next step we focus on the effects of the inclusion of feature attributes in the complex cross-tree constraints. We introduce a new variability type that emerges due to the discussed effects and classify features, products, and lists of products into categories in the light of the discussed variability type. Then we revisit the analysis operations defined on feature models; we suggest revised definitions for some of them and reformulations for some other. We also describe some new analysis operations that were not reported before.

Then we present a transformation from extended feature models to basic feature models. Such a transformation enables using existing automated reasoning approaches on extended feature models. However, this transformation can be used under certain circumstances, and we discuss the pros and

cons of the transformation. We also present a motivating example to illustrate the application of the transformation.

Finally we present some guidelines for using the context information in staged configurations of feature models. We discuss the properties of context entities, and classify context information into four categories. We discuss the relations between these categories and present an organization structure for the context variability model [24]. Then we present an ordering for the stages for the staged configuration of feature models using the classification given for the context entities.

The organization of this dissertation is as follows. Chapter 2 presents a literature survey on the feature models and automated reasoning efforts on feature models to provide background and related work information. In Chapter 3 we discuss a mapping from extended feature models to constraint logic programming over finite domains, and the groundwork for the inclusion of global constraints in the automated reasoning on feature models. In Chapter 4 we discuss the analysis operations defined on feature models, suggest revised definitions for some of them and reformulate some other in the light of the ideas originating from the mapping discussed in the previous chapter. Chapter 5 presents a simplification over extended feature models, which enables the use of automated reasoning approaches that were defined for less powerful models such as basic and cardinality-based feature models on the extended feature models as well under certain restrictions. In Chapter 6 we discuss the relations between the context and feature models and propose an approach to use context information in staged configuration of feature models. Chapter 7 presents discussions and conclusions. In the Appendices we present the sample feature models used in various discussions in the dissertation.

CHAPTER 2

FEATURE MODELS AND THEIR AUTOMATED ANALYSIS

An important consideration in software product lines is capturing the *commonality* and the *variability* in the family of the products to be produced. Thus, modeling and managing variability in software product families is a key concept. Among different proposals, feature modeling has been very popular since their introduction by Kang *et al.* [28]. However, industrial experiences showed that feature models often grow large with hundreds, or even thousands of features and complex cross-tree relationships among features and attributes of features. Thus, automated reasoning has become an important issue for both the researchers and practitioners working on feature models.

This chapter aims to provide background and related work information to the reader. We first present a brief discussion on feature models and the extensions devised to the feature models. For a detailed survey on feature models we refer the reader to [39]. Then, we present the well-known analysis operations defined on feature models. For a detailed literature review on the automated analysis of feature models we refer the reader to [7].

2.1 Feature Models

There are two widely used definitions for the concept *feature* in the domain engineering literature. The first one, which is described by Kang *et al.* [28], defines a feature as *"an attribute of a system that directly affects end–users"*. The second definition, which is used in the context of Organization Domain Modeling [42], describes a feature as *"a distinguishable characteristic of a concept (e.g. system, component, etc.) that is relevant to some stakeholder of the concept"*. As the latter definition is more general we will prefer that one in the scope of this study.

A Feature Model is a hierarchically arranged set of features, the relationships defining the composition rules among these features, the relationships defining the cross-tree constraints, and some additional information such as issues/decisions that record various trade-offs, rationale, and justifications for feature selection. Feature models are often represented using Feature Diagrams. For instance, the feature diagram in Figure 2.1 represents a sample feature model [7].

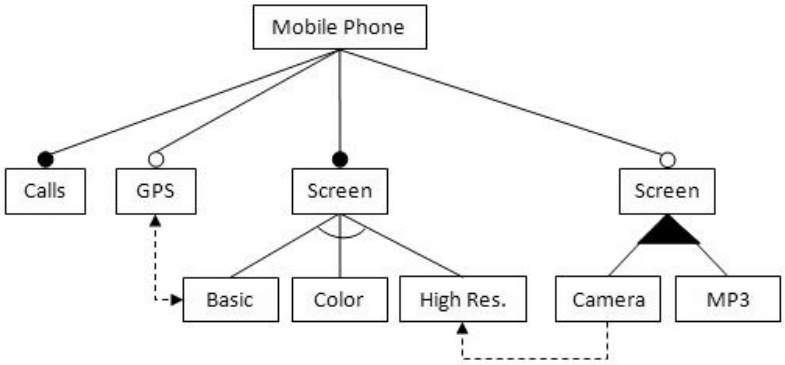

Figure 2.1 A sample feature model

The root of the feature diagram is called the *concept node*, and it appears in every product derived from the feature model. Note that the feature diagram in Figure 2.1 is in the form of a tree. However, in some of the extensions to the basic feature models, such as FORM [29] and FeatuRSEB [23], feature diagrams are allowed to be directed acyclic graphs.

There are two types of relationships in feature models; decomposition relationships, which define the relation between a parent feature and its child features, and cross-tree relationships, which are used to specify the cross-tree constraints among features.

2.1.1 Decomposition Relationships

There are four types of decomposition relationships:

- ***Mandatory Relation:*** Let P and C be features in a *mandatory relation*, where P is the parent of C. Then the feature C is called a *mandatory feature*. A mandatory feature is included in a product if and only if its parent is included.
- ***Optional Relation:*** Let P and C be features in an *optional relation*, where P is the parent of C. Then the feature C is called an *optional feature*. If the parent of an optional feature is included in a product, then the optional feature may be included or not; if the parent is not included, the optional feature cannot be included.
- ***Alternative Relation:*** Let P, C_1, C_2, …, and C_n be features in an *alternative relation*, where P is the parent of C_1, C_2, …, and C_n. Then the features C_1, C_2, …, and C_n are called *alternative features*. If the parent of some alternative features is included in a product, then exactly one alternative feature is included; if the parent is not included, then no alternative feature can be included.
- ***Or Relation:*** Let P, C_1, C_2, …, and C_n be features in an *or relation*, where P is the parent of C_1, C_2, …, and C_n. Then the features C_1, C_2, …, and C_n are called *or features*. If the parent of some or-features is included in a product, then a nonempty subset of the or-features is included; if the parent is not included, then no or-feature can be included.

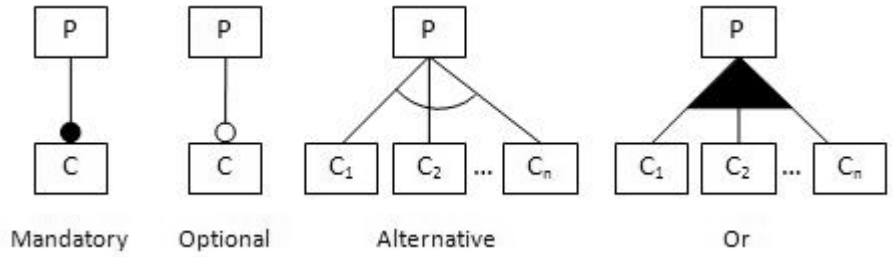

Figure 2.2 Symbols used to represent decomposition relationships

Consider the sample feature model given in Figure 2.1. The feature *Calls* is a mandatory feature, as it has a mandatory relation with its parent *Mobile Phone*, which means *Calls* will be included in every product that includes the feature *Mobile Phone*. However, the feature *GPS* is an optional feature, thus it may be included or not in a product that includes its parent *Mobile Phone*. The features *Basic*, *Color*, and *High Resolution* are alternative features, thus exactly one of them will be included in a product that includes their parent feature *Screen*. On the other hand, the features *Camera* and *MP3* are or-features, which means a nonempty subset of the set of features {*Camera*, *MP3*} must be included in every product that includes their parent feature *Media*. However, as the feature *Media* is an optional feature, if *Media* is not included in a product then neither *Camera* nor *MP3* can be included in the product.

2.1.2 Cross-tree Relationships

There are two types of cross-tree relationships:

- ***Requires:*** Let X and Y be features in a *requires* relation, such that *X requires Y*. Then the inclusion of the feature *X* in a product implies the inclusion of the feature *Y*.
- ***Excludes:*** Let X and Y be features in an *excludes* relation, such that *X excludes Y*. Then the inclusion of the feature *X* in a product implies the exclusion of the feature *Y* and vice versa (i.e. no product can include both of the features). Note that it would be possible to derive a product that includes only the feature *X*, only the feature *Y*, or none of them. However, it is not possible to derive a product that includes both *X* and *Y*.

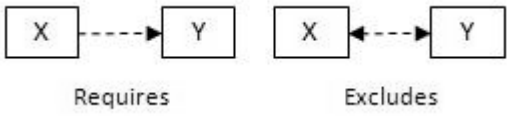

Figure 2.3 Symbols used to represent cross-tree relationships

Consider the sample feature model given in Figure 2.1. The feature *Camera* requires the feature *High Resolution*, thus every product that includes the feature *Camera* must also include the feature *High Resolution*. On the other hand, the feature *GPS* excludes the feature *Basic*, which means no product that includes the feature *GPS* can include the feature *Basic*, or vice versa (i.e. no product that includes the feature *Basic* can include the feature *GPS*).

2.2 Cardinality-Based Feature Models

Although less obvious than UML, it is possible to express some multiplicities in feature models. For instance consider the decomposition relationships given in Figure 2.4.

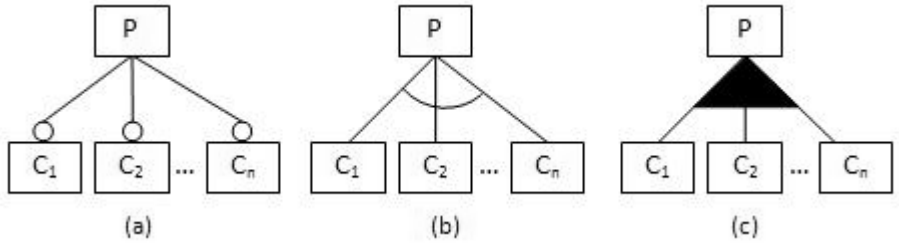

Figure 2.4 Multiplicities in basic feature models

The optional relations in Figure 2.4 (a) may be used to represent the multiplicity **0..n**, the alternative relation in Figure 2.4 (b) may be used to represent the multiplicity **1**, and the or-relation in Figure 2.4 (c) may be used to represent the multiplicity **1..n**. This list covers most of the cases, however in practice sometimes situations arise in which a set of features has the multiplicities such as 0..3, 1..3, 3..6, or simply 6. An important extension to feature models has been the introduction of UML-like multiplicities to represent such multiplicities, and these feature models are referred as *Cardinality-Based Feature Models*.

In their work [38] Riebisch *et al*. introduced *group cardinalities* (see Figure 2.5). Group cardinalities enabled to represent cardinalities in the form m..n, which means at least *m* and at most *n* child features must be selected when the parent feature is selected.

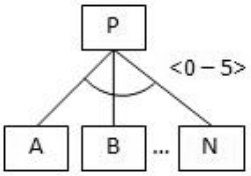

Figure 2.5 A decomposition with group cardinalities

Later Czarnecki *et al.* [14, 17] introduced feature cardinalities such as * (i.e. a feature may be selected an indefinite number of times), and m..n (i.e. a feature may be selected at least *m* and at most *n* times). The authors also cover group cardinalities in their works with the restriction that a feature in a decomposition relationship with group cardinality cannot include feature cardinality.

2.3 Extended Feature Models

Sometimes it may be desirable to provide more information about features in feature models. For instance, a designer would like to provide information on the clock speed of a CPU, which is a feature. Extended feature models provide further information about features using feature attributes. For instance, the extended feature model given in Figure 2.6 provides extra information about the features *CPU* and *RAM* by the attributes *speed* and *size* respectively.

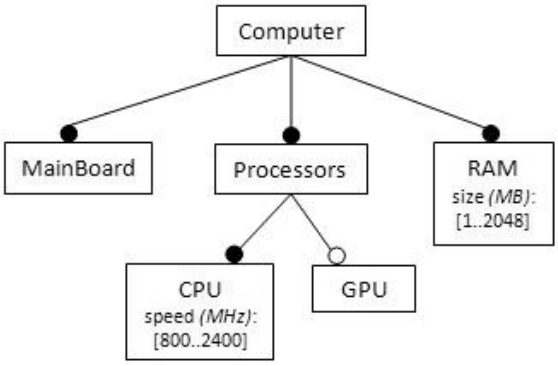

Figure 2.6 A sample extended feature model

An *attribute* of a feature is any characteristic of the feature that can be measured. Every feature attribute belongs to a domain, the space of possible values where the attribute takes its values [4]. For instance, the domain of the attribute *speed* in the sample extended feature model is [800..2400]. Domain of an attribute can be discrete (finite or infinite) or continuous.

2.4 Analysis Operations on Feature Models

A number of analysis operations on feature models have been reported in the literature to reveal certain characteristics of the feature models, and to answer questions such as *"how many products can be derived from a feature model?"*, *"is it a valid feature model?"*, *"what is the list of all products?"*. Typical inputs to the analysis operations, besides a feature model, are configurations and products. A configuration and a product are defined as follows [7]:

Configuration: Given a feature model with a set of features F, a *configuration* is a 2-tuple of the form (S, R) such that S and R are two disjoint subsets of F, where S is the set of features to be selected and R is the set of features to be removed. A configuration is called a *full configuration* If $S \cup R = F$, and *partial configuration* otherwise.

Product: A *product* is a full configuration where only selected features are specified and the removed features are implicitly omitted.

In this section we will briefly discuss some of the well-known operations. Definitions of these operations and a more detailed review can be found in [7].

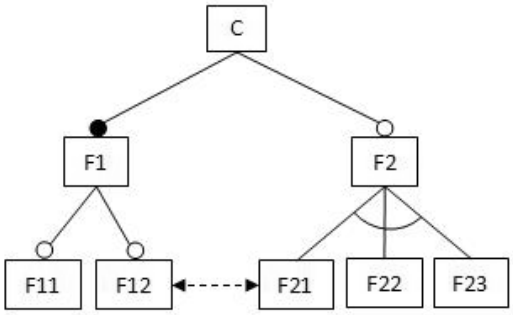

Figure 2.7 A sample feature model, $\mathcal{FM}$

2.4.1 Void Feature Model

This operation is used to answer the question *"does this feature model represent any products?"*. The input of the operation is a feature model, and the output is a value that indicates whether the feature model is *void* or not. A feature model is referred as *void* if no valid product can be derived from it (i.e. represents no products). For instance, the feature model $\mathcal{FM}$ is not void as at least one product (e.g. {C, F1}) can be derived from it. However, the sample feature model given in Figure 2.8 is void as it represents no products.

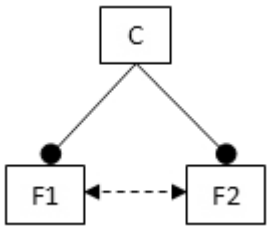

Figure 2.8 A void feature model

2.4.2 Valid Product

This operation is used to answer the question *"can this product be derived from this feature model?"*. The input of the operation is a feature model and a product (i.e. set of features), and the output is a value that indicates whether the product belongs to the set of products represented by the feature model or not. For instance the product {C, F1, F11, F12} is valid with respect to $\mathcal{FM}$, whereas the product {C, F1, F11, F12, F21} is not as it violates the cross-tree constraint *"F12 excludes F21"*.

2.4.3 Valid Partial Configuration

This operation is used to answer the question *"is it possible to derive a valid product using this configuration from this feature model?"*. The input of the operation is a feature model and a partial configuration, and the output is a value that indicates whether the configuration is valid with respect to the feature model or not.

A configuration is in the form (S, R) where *S* is the set of features to be selected and *R* is the set of features to be removed. A partial configuration is valid if it includes no contradiction (i.e. at least one valid product can be derived from it). For instance, the configuration (S, R) = ({F1, F11}, {F22}) is a valid partial configuration with respect to $\mathcal{FM}$ as it leads to no contradiction (e.g. the product {C, F1, F11} can be derived from this configuration). However, (S, R) = ({F11}, {F1}) is not a valid partial configuration as it is not possible to include a feature when its parent is not included.

2.4.4 All Products

This operation is used to answer the question *"what is the list of all products that can be derived from this feature model?"*. The input of the operation is a feature model, and the output is the list of all

products represented by the model. For instance, the list of all products that can be derived from the feature model $\mathcal{FM}$ is as follows:

- P1 = {C, F1}
- P2 = {C, F1, F11}
- P3 = {C, F1, F12}
- P4 = {C, F1, F11, F12}
- P5 = {C, F1, F2, F21}
- P6 = {C, F1, F2, F22}
- P7 = {C, F1, F2, F23}
- P8 = {C, F1, F11, F2, F21}
- P9 = {C, F1, F11, F2, F22}
- P10 = {C, F1, F11, F2, F23}
- P11 = {C, F1, F12, F2, F22}
- P12 = {C, F1, F12, F2, F23}
- P13 = {C, F1, F11, F12, F2, F22}
- P14 = {C, F1, F11, F12, F2, F23}

2.4.5 Number of Products

This operation is used to answer the question *"how many products can be derived from this feature model?"*. The input of the operation is a feature model, and the output is the number of products represented by the model. For instance, as we have shown in 2.4.4, fourteen products can be derived from the feature model $\mathcal{FM}$.

2.4.6 Filter

This operation is used to answer the question *"what is the list of all products that include this configuration and can be derived from this feature model?"*. The input of the operation is a feature model and a configuration that may be partial, and the output is the list of all products that include the configuration and represented by the model. For instance, if we feed this operation with the feature model $\mathcal{FM}$ and the configuration (S, R) = ({F1, F11}, {F22}), then the outcome of the operation would be as follows:

- P2 = {C, F1, F11}
- P4 = {C, F1, F11, F12}
- P8 = {C, F1, F11, F2, F21}
- P10 = {C, F1, F11, F2, F23}
- P14 = {C, F1, F11, F12, F2, F23}

2.4.7 Anomalies Detection

As the complexity of a feature model increases, chances to make errors for the modeler increase as well. Sometimes these errors may lead to anomalies in the feature model. These operations are used to answer the question *"what are the anomalies in this feature model?"*. The input of these operations is a feature model, and the output is the anomalies detected. Main types of anomalies are as follows:

- ***Dead Features:*** A feature is *dead* if it cannot appear in any of the products of the software product line.
- ***Conditionally Dead Features:*** A feature is *conditionally dead* if it becomes *dead* under certain circumstances.
- ***False Optional Features:*** A feature is *false optional* if it is included in all of the products that can be derived from the model, despite being not modeled as mandatory.
- ***Wrong Cardinalities:*** A cardinality is wrong if it cannot be instantiated.
- ***Redundancies:*** A feature model contains redundancies when some semantic information is modeled in multiple ways.

2.4.8 Explanations

This operation is used to answer the question *"why / why not this analysis operation did not respond as I have expected on this feature model?"*. The input of the operation is a feature model and an analysis operation, and the output is explanations that provide information on the reasons for the response of the operation. These explanations are often described in terms of the features and/or relationships involved in the operation, where explanations are mainly related to anomalies.

2.4.9 Corrective Explanations

This operation is used to answer the question *"how can I fix the problems detected about this feature model and this analysis operation?"*. The input of the operation is a feature model and an analysis operation, and the output is explanations that suggest some changes to be made.

2.4.10 Feature Model Relations

This operation is used to answer the question *"is there a relation between these two models?"*. The input of the operation is two feature models, and the output is the type of relation between the models. There are four types of relations reported in the literature:

- ***Refactoring:*** A feature model is a refactoring of another one if the two models represent the same set of products while having different structures.
- ***Generalization:*** A feature model, F, is a generalization of another one, G, if every product represented by G is also represented by F, whereas some products represented by F are not represented by G.
- ***Specialization:*** A feature model, F, is a specialization of another one, G, if the set of products represented by F is a subset of the products represented by G.
- ***Arbitrary Edit:*** This relation means that there is no explicit refactoring, generalization, or specialization relation between the two feature models.

2.4.11 Optimization

This operation is used to answer the question *"which is the product that can be derived from this feature model and fulfills the criteria established by this optimization function?"*. The input of the

operation is a feature model and an optimization function, and the output is the product that satisfies the optimization function.

2.4.12 Core Features

This operation is used to answer the question *"which are the features that are included in all of the products that can be derived from this feature model?"*. The input of the operation is a feature model, and the output is the set of features that are included in every product represented by the feature model. For instance, the set of core features for the sample feature model $\mathcal{FM}$ is the set {C, F1} as they are included in every product that can be derived from $\mathcal{FM}$.

2.4.13 Variant Features

This operation is used to answer the question *"which are the features that are not included in all of the products that can be derived from this feature model?"*. The input of the operation is a feature model, and the output is the set of features that are not included in every product represented by the feature model. For instance, the set of variant features for the sample feature model $\mathcal{FM}$ is the set {F11, F12, F2, F21, F22, F23}.

2.4.14 Atomic Sets

This operation is used to answer the question *"which are the sets of features that can be treated as a single unit when performing certain analysis operations?"*. The input of the operation is a feature model, and the output is the set of atomic sets (i.e. groups of features that can be treated as a unit). For instance, {C, F1} is an atomic set with respect to $\mathcal{FM}$.

2.4.15 Dependency Analysis

This operation is used to answer the question *"which configuration can be produced from this configuration as a result of the propagation of the constraints in this feature model?"*. The input of the operation is a feature model and a partial configuration, and the output is a new configuration with the features that should be selected and/or removed as a result of the propagation of constraints in the model. For instance, if we feed this operation with the feature model $\mathcal{FM}$ and the configuration ({F12, F2}, {F22}) then the output would be ({C, F1, F12, F2, F23}, {F21, F22}). The feature F1 is selected as it is the parent of the feature F12, and C is selected as it is the parent of the feature F1 (or F2). The feature F21 is removed due to the cross-tree constraint *F12 excludes F21*. The feature F23 is selected as the features F22 and F21 are removed, and exactly one feature among the alternative features F21, F22, and F23 must be selected as the feature F2 is selected.

2.4.16 Other Operations

These are the operations that perform some computation using the values generated as output by the previous operations. Some of these operations are as follows:

Homogeneity: This operation is used to answer the question *"what is the degree of homogeneity of this feature model?"*. The input of the operation is a feature model, and the output is the homogeneity degree of the model that is computed with the following formula:

$$Homogeneity = 1 - \frac{\#uf}{\#\ products} \tag{2.1}$$

Unique features, represented with *uf* in the formula, are the features that appear in only one product. For instance, degree of homogeneity of the feature model $\mathcal{FM}$ is 1 as there are no unique features in the model.

Commonality: This operation is used to answer the question *"what is the percentage of products including this configuration and represented with this feature model?"*. The input of the operation is a feature model and a configuration, and the output is the commonality degree of the model that is computed with the following formula:

$$Commonality(C) = \frac{|filter(C)|}{Number\ of\ Products} \tag{2.2}$$

For instance, for the feature model $\mathcal{FM}$, commonality of the configuration ({F1, F11}, {F22}) is 5/14 = 0.357.

Variability Factor: This operation is used to answer the question *"what is the ratio of the number of products represented with this feature model over 2^n, where n is the number of the features in the feature model?"*. The input of the operation is a feature model, and the output is the variability factor of the model that is computed with the following formula:

$$Variability\ Factor = \frac{Number\ of\ Products}{2^n} \tag{2.3}$$

In particular, 2^n is the number of potential products that can be derived from this feature model assuming that any combination of the features is a valid product. Note that, root and non-leaf features are often not considered while calculating the *n*. For instance, for the feature model $\mathcal{FM}$ the variability factor is $14/2^5 = 0.438$.

Degree of Orthogonality: This operation is used to answer the question *"what is the ratio between the total number of products represented by this feature model and the total number of products represented by this subtree represented with its root?"*. The input of the operation is a feature model and a feature that represents a subtree, and the output is the degree of orthogonality (i.e. the ratio between the total number of products represented by the feature model and the total number of products represented by the subtree).

Note that, only the constraints that involve the features in the subtree are considered when computing the total number of products represented by the subtree. For instance, for the feature model $\mathcal{FM}$ and the subtree represented by the feature F2 is 14/10 = 1.4.

Extra Constraint Representativeness (ECR): This operation is used to answer the question *"what is the ratio of the number of features involved in cross-tree constraints over the number of all features in this feature model?"*. The input of the operation is a feature model, and the output is the value indicating the ECR.

Note that, repeated features in the constraints are counted only once. For instance, the value indicating the ECR of the feature model $\mathcal{FM}$ is 2/8 = 0.25.

Lowest Common Ancestor (LCA): This operation is used to answer the question *"what is the lowest common ancestor of this set of features with respect to this feature model?"*. The input of the operation is a feature model and a set of features, and the output is the LCA (i.e. the shared ancestor that is farthest from the root) of the features in the set. For instance, LCA of the features {F11, F12, F21} with respect to $\mathcal{FM}$ is C.

Root Features: This operation is used to answer the question *"which are the features that are roots of the subtrees that include this set of features with respect to this feature model?"*. The input of the operation is a feature model and a set of features, and the output is the set of features that are the roots of the subtrees in the feature diagram that include all of the input features. For instance, the set of root features of the {F11, F12} with respect to $\mathcal{FM}$ is {F1, C}.

2.5 Automated Analysis of Feature Models

As feature models for realistic product families may be quite complicated, automated analysis of feature models is desirable. Proposals on automated analysis of feature models can be divided into four groups: propositional logic based analyses, description logic based analyses, constraint programming based analyses, and other proposals. In this section we will briefly discuss these approaches. We refer the reader to [7] for a detailed review on the automated analysis approaches reported in the literature.

2.5.1 Propositional Logic Based Analyses

Mannion *et al.* [35, 36] were the first to propose using propositional logic for reasoning on feature models. Batory [2] was the first author to propose using a SAT solver for the automated analysis of feature models in his work that he had also described a design for a *Logic Truth Maintenance System* as the reasoning tool. Sun *et al.* [44] proposed using Z [43], which is a formal specification language, to provide semantics to feature models. Benavides *et al.* [8, 9, 40] proposed to use a multi-solver approach and suggested that some solvers would perform better depending on the type of the analysis operation. Some of the tools proposed for use in propositional logic based analyses are SAT Solver [10], Alloy [45], SMV [46], and BDD Solver [48].

2.5.2 Constraint Programming Based Analyses

Benavides *et al.* [4, 5, 6] were the first to propose using constraint programming for automated analysis of feature models. The authors provided a set of mapping rules to translate feature models

into a CSP and described how constraint programming based automated reasoning can answer key questions such as *"how many products can be derived from a feature model?"*, *"is it a valid feature model?"*, *"what is the list of all products?"*. They have also provided tool support [9]. Some of the tools proposed to be used for constraint programming based analyses are Choco [13], JaCoP [27], OPL Studio [26], and GNU Prolog [22].

The work to be presented in this dissertation also falls into this category of approaches. However, there are major differences and contributions with respect to the previous proposals reported in the literature. For instance, in the mapping proposed by Benavides *et al.* [4, 5, 6] features make up the set of variables in the target program whereas attributes make up the set of variables in our mapping. Moreover, to the best of our knowledge, none of the approaches listed in this category can handle complex cross-tree constraints involving attributes effectively, whereas the solution to be presented in this dissertation is capable of handling basic, cardinality-based, and extended feature models, which may include complex feature-feature, feature-attribute, and attribute-attribute constraints.

2.5.3 Description Logic Based Analyses

Proposals in this category basically describe a feature model by a set of concepts (i.e. classes), a set of roles (e.g. properties or relationships) and set of individuals (i.e. instances). Then a description logic reasoner, which is a software package, is used for reasoning operations, checking for consistency and correctness.

Wang *et al.* [47] were the first to propose using description logic for automated analysis of feature models. Authors presented a mapping to translate feature models into OWL-DL ontologies [18] and proposed to use logic reasoning engines such as RACER [37]. Later, Fan *et al.* [20] also proposed translating feature models into description logic to use reasoning engines such as RACER. Abo *et al.* [1] proposed to use semantic web technologies for the automated analysis of feature models.

Wang *et al.* [47] provided support for analysis operations such as *void feature model*, *valid product*, *explanations*, and *refactoring*. Abo *et al.* [1] provided support for the analysis operations *dead features* and *explanations*. Fan *et al.* [20] provided support for the analysis operation *void feature model*. These works use basic feature model notation and Abo *et al.* [1] also provide support for extended feature models as well. However, to the best of our knowledge, none of the studies in this category can handle complex cross-tree constraints involving feature attributes.

2.5.4 Other Proposals

Studies in this category are described by Benavides *et al.* [7] as *"studies in which the conceptual logic used is not clearly exposed or ad–hoc algorithms, paradigms or tools for analysis are used"*. Kang *et al.* were the first to mention automated analysis of feature models in [28], however this idea has not been elaborated by the authors. Following the FODA report Deursen *et al.* [19] were the first to propose automated reasoning support using the ASF+SDF Meta-Environment [34].

CHAPTER 3

EXTENDED FEATURE MODELS AND CONSTRAINT PROGRAMMING

In this chapter we discuss some of the works of the author of this dissertation that establish the foundations for the analysis of extended feature models with Constraint Logic Programming over Finite Domains, designated CLP(FD). We first present a mapping from extended feature models, which may include complex feature-feature, feature-attribute and attribute-attribute relationships, to CLP(FD) [32]. Then we present a proposal that lays the groundwork for the inclusion of global constraints in automated reasoning on feature models [31].

3.1 Mapping Extended Feature Models to CLP(FD)

Basic and cardinality-based feature models use the traditional cross-tree relationships to define cross-tree constraints among the features in a feature model. However, extended feature models enable defining complex cross-tree constraints involving feature attributes, such as *"feature 3D Car Race requires Memory.size ≥ 1024"*. Although several approaches reported in the literature addressed automated analysis of feature models, feature-attribute and attribute-attribute relationships in extended feature models, such as the example presented above, were not handled effectively.

Karataş *et al.* [32] were the first to propose a mapping that can handle extended feature models including complex feature-feature, feature-attribute and attribute-attribute relationships effectively. This mapping translates an extended feature model to a CLP program and has the following properties:

- The attributes make up the set of variables.
- The domains of all attributes are finite.
- Every relationship (decomposition, cross-tree) becomes a constraint.

As the reader would have noticed features do not figure in the resulting CLP program. The reason for this is that every feature is assumed to have an *implicit attribute*, named *selected*, that ranges over the domain {true, false}. These implicit attributes represent the features in the CLP program such that a feature is considered to be included in a product if and only if the implicit attribute of the feature takes the value *true*. With the introduction of implicit variables the mapping can treat the features and attributes uniformly, which are at two separate levels of abstraction.

The translation maps the decomposition relationships using an approach similar to the proposals devised before, and it is capable of mapping all four decomposition relationships (i.e. mandatory relation, optional relation, alternative relation, and or-relation). Thus, basic and cardinality-based feature models are treated uniformly under this scheme. For instance, consider the decomposition relationships given in Figure 3.1.

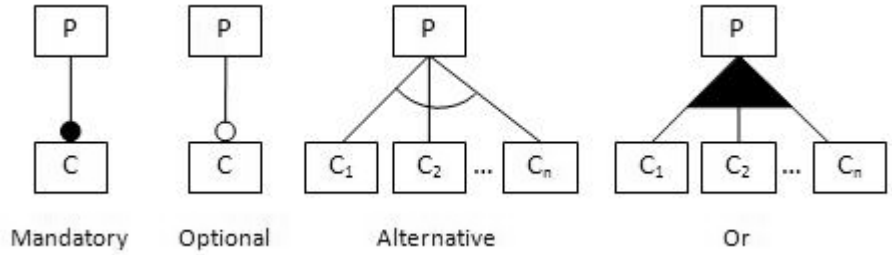

Figure 3.1 Decomposition relationships in feature models

The mapping produces the following constraints for the decomposition relationships:

- $P.selected \Leftrightarrow C.selected$ *(Mandatory Relation)*
- $C.selected \Rightarrow P.selected$ *(Optional Relation)*
- $(C_1.selected \Leftrightarrow (\neg C_2.selected \wedge \ldots \wedge \neg C_n.selected \wedge P.selected)) \wedge \ldots \wedge (C_n.selected \Leftrightarrow (\neg C_1.selected \wedge \ldots \wedge \neg C_{n-1}.selected \wedge P.selected))$ *(Alternative Relation)*
- $C_1.selected \vee C_2.selected \vee \ldots \vee C_n.selected \Leftrightarrow P.selected$ *(Or Relation)*

We have extended the definition of the *requires* relationship to allow complex expressions built by combining conditions with the logical operators. Conditions are constructed by applying relational operators (i.e. $=, \neq, <, \leq, >, \geq$) on numeric expressions built by combining integer constants and attribute values with the traditional integer operators (i.e. +, -, *, div, mod). Conditions of the form *Feature.attribute = truth-value* are also allowed.

For instance, the following are valid conditions that can figure in mapping:

- CPU.speed > 1800
- Hardware.cost = RAM.cost + CPU.cost + GPU.cost
- F.a = P.b + Q.c * 3 + R.d div 2
- Filter.on = false

Thus, the extended definition of the *requires* relationship enables to define cross-tree relationships such as:

- F requires X
- F requires X.a > 20 and Y.b = Z.c * 2
- F requires P.a + Q.b = M.c – N.d or P.a < Q.a + 2 and (M.c < 2 ⇒ N.d = 0)
- F requires (X.a < Y.b) or (Z and M.c = 100)

For instance, the mapping of the last *requires* example is as follows:

$$F.selected \Rightarrow (X.a < Y.b \wedge X.selected \wedge Y.selected) \vee (Z.selected \wedge M.c = 100 \wedge M.selected)$$

We also allow combining requires relationships, excludes relationships, and features with logical connectives to form complex constraints. For instance, the following are valid complex constraints:

- (F excludes X) or (F requires Y)
- X and Y implies not Z or (F requires X.a > 10)
- (X excludes Z) or (F requires Y.a = Z.b + 20)

Translation of the last example to CLP is as follows:

$$\neg (X.selected \wedge Z.selected) \vee (F.selected \Rightarrow (Y.a = Z.b + 20 \wedge Y.selected \wedge Z.selected))$$

We have introduced the concept of *guarded constraints* (i.e. constraints in the form of *if the guard evaluates to true then apply this constraint*) to express conditional constraints such as *"if F.a > 100 then X requires Y.a = Z.b"*. The translation of this guarded constraint to CLP is as follows:

$$F.a > 100 \wedge F.selected \Rightarrow (X.selected \Rightarrow (Y.a = Z.b \wedge Y.selected \wedge Z.selected))$$

This mapping enables automated reasoning support for the analysis of extended feature models, which may include complex feature-feature, feature-attribute and attribute-attribute relationships, which was not possible before. However, as we have pointed out, the inclusion of the attributes in cross-tree relationships introduces a new variability type.

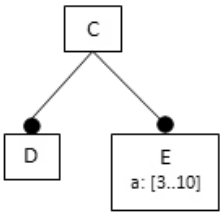

Figure 3.2A simple extended feature model

Assume that we have a cross-tree relationship *"D requires E.a > 6"*. When we seek the answer to the question *"which are the products of this model?"* there might be two possible ways to answer.

The first answer would include a single product:

P1 = {C, D, E where E.a ∈ {7, 8, 9, 10}}

Whereas the second answer would list 4 products:

P1 = {C, D, E where E.a = 7} through P4 = {C, D, E where E.a = 10}

This new type of variability, as we refer to *variability in terms of attributes*, has effects on some of the analysis operations discussed in 2.4. We will discuss these effects in the following chapter.

3.2 Global Constraints on Feature Models

In our work [31] we have laid the groundwork for the inclusion of global constraints in automated reasoning on feature models, which has been possible by the mapping discussed in the previous section. The aim of this work was to bring the benefits of global constraints to feature models.

A *global constraint* is basically a constraint that captures a relation between a non-fixed number of variables [25]. For instance, assume that we want to assign a value from a domain to *n* different variables such that no two distinct variables will have the same value. One can use the *alldifferent* global constraint to accomplish this task:

$$\text{alldifferent}(x_1, x_2, \ldots, x_n)$$

This constraint*, alldifferent*, assures that the values assigned to the *n* variables will be pair-wise distinct. Note that one can get the same effect without making use of the *alldifferent* global constraint by listing the following binary equations:

$$x_1 \neq x_2,\ x_1 \neq x_3,\ x_1 \neq x_4,\ \ldots,\ x_1 \neq x_n,$$
$$x_2 \neq x_3,\ x_2 \neq x_4,\ \ldots,\ x_1 \neq x_n,$$
$$\ldots$$
$$x_{n-1} \neq x_n$$

Typically, as it is the case for the example presented above, it may be possible to express a global constraint as the conjunction of several simple formulas. However, obviously, this will result in a more tedious coding process and decrease the readability and maintainability of the code. Moreover, one will miss the chance to make use of other advantages provided by the global constraints besides providing shorthand for recurring patterns.

One of the key issues of constraint programming is the propagation-search technique that is a traversal of the search space of the given constraint satisfaction problem (CSP). One of the aims of this technique is detecting dead-ends as early as possible. If one tries to perform an exhaustive search on the search space the cost will be exponential even in the best case, thus heuristics to prune the search space using the propagation techniques is desirable as the useless values that will lead to dead ends would be eliminated without enumeration.

For instance, assume that we want to solve the *n-queens* problem on a standard chess board with eight queens. When we place a queen on the board it will threaten some of the cells on the board, thus placing another queen in one of those cells will simply lead to a dead end and will require backtracking.

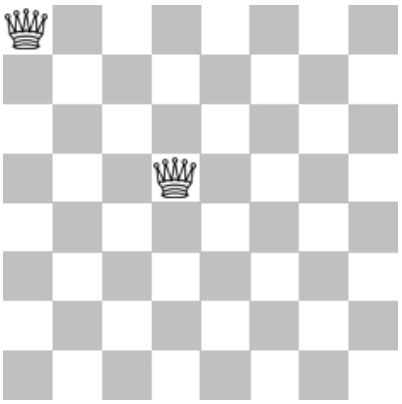

Figure 3.3 An invalid solution that will cause backtracking

However, if the domains of the rest of the queens are pruned as soon as a queen is placed on the board so that no other queen can be placed in a cell that is threatened by the queen placed, then the necessity to perform backtracking would be avoided to an extent.

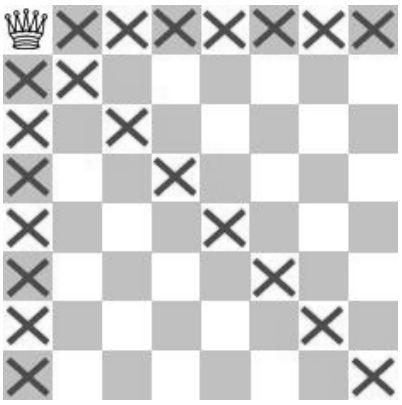

Figure 3.4 Domains of the rest of the queens are pruned as soon as a queen is replaced

Note that 22 possibilities have been pruned from the rest of the queens' domains when the first queen is placed on the board. Considering each queen's domain consists of 64 values, this would have a significant effect on the performance of the solver. This task is called *filtering* of the variable domains [25]. Basically filtering task is examining the domains of the unassigned variables and eliminating the useless values. As global constraints provide a better view on the structure of the problem to the solvers they facilitate the filtering process, which is another important advantage of using global constraints.

There are many global constraints reported in the literature. We refer the reader to [3] for an extensive listing of global constraints. Some systems such as clp(FD) [12] also provide facilities to the user to define their own user-defined global constraints.

The idea of using global constraints on feature models originates from the nature of project management. During the project management phases it is often the case that developers face requirements that impose restrictions on all or a set of features rather than a feature alone. Requirements such as *"total power consumption of the units in a mobile phone shall not exceed the*

power supplied by the battery", *"total cost of the product must reside in the prescribed interval to suit the targeted budget limits"*, or *"each PCI card to be installed on a computer board must be installed into a different PCI slot for a valid configuration of the computer"* are examples of such restrictions. Due to such restrictions the nature of the feature models invites, sometimes even requires, the use of global constraints. However, necessary background must be prepared, which we present next, before including the global constraints in feature models.

Attributes take meaningful values only if the features they belong to are included in a product. Thus, if the feature an attribute belongs to is not included in a product, the attribute should not figure in the global constraints. To eliminate such attributes we have proposed to use a filter operator, which is defined to work on a non-fixed number of *(truth-value, attribute)* pairs. For instance assume that a global constraint is defined on a number of variables $a_1, a_2, \ldots, a_n$ that belong to the features $F_1, F_2, \ldots, F_n$ respectively (note that F_i's would not be distinct). Then we apply the filter operator on the set of pairs $\mathcal{T} = \{(F_1.\text{selected}, a_1), \ldots, (F_n.\text{selected}, a_n)\}$ as the first step:

$$\text{filter}(\mathcal{T}) = \{a_i \mid (\text{true}, a_i) \in \mathcal{T}\}$$

As attributes that belong to the unselected features are eliminated by the filter process, the global constraint can be applied to the resulting set of variables obtained from the filter operation. Note that every global constraint can also be represented without making use of the filter operation by the mapping rules proposed in [32] by making use of the *guarded constraints* as follows:

$$\begin{gathered}((F_1.\text{selected} \wedge F_2.\text{selected} \wedge \ldots \wedge F_n.\text{selected}) \Rightarrow GC(F_1.a_1, \ldots, F_n.a_m)) \wedge \\ ((\neg F_1.\text{selected} \wedge F_2.\text{selected} \wedge \ldots \wedge F_n.\text{selected}) \Rightarrow GC(F_2.a_1, \ldots, F_n.a_m)) \wedge \\ \ldots \\ ((F_1.\text{selected} \wedge \neg F_2.\text{selected} \wedge \ldots \wedge \neg F_n.\text{selected}) \Rightarrow GC(F_1.a_1, \ldots, F_1.a_k))\end{gathered}$$

However, in such a case one cannot make use of the benefits of the global constraints.

For another solution, we have introduced the novel concept of *neutral value* (i.e. a value that neutralizes the effect of an attribute in a global constraint). For instance, we have discussed that 0 acts as the neutral value for the global constraints involving the summation of their arguments. We allow introduction of different neutral values for different attributes, but do not allow introduction of more than one neutral value for an attribute. We have proposed to extend the domain of each attribute with its neutral value during the mapping, and add two new constraints to ensure that: *(i)* the attribute takes a value from its original domain when the feature it belongs to is selected, *(ii)* the attribute takes its neutral value when the feature it belongs to is not selected.

For instance assume that we have an attribute *a* of feature *F*, such that the domain of the attribute is {5, …, 10}, and the neutral value of the attribute is 0. Then the mapping of the attribute to CLP would be as follows:

$$\begin{gathered}F.a \in \{5, \ldots, 10\} \cup \{0\} \wedge \\ (F.\text{selected} \Rightarrow F.a \in \{5, \ldots, 10\}) \wedge (\neg F.\text{selected} \Rightarrow F.a = 0)\end{gathered}$$

However, note that, it may not be possible to find a *common neutral value* (i.e. a value that will act as a neutral value for every global constraint the attribute figures in) in some cases. For instance, if an attribute figures in a summation in one global constraint, and in multiplication in another it is not

possible to find a value that will act as a neutral value for both of the global constraints. In such cases the general filter operator would be used. However, if it is possible to introduce common neutral values for the attributes that figure in global constraints, using common neutral values would provide more efficient implementations as the overhead caused by the filter operation is avoided.

We have also presented some examples for the usage of global constraints on feature models. For instance, consider the feature model part given in Figure 3.5 [31].

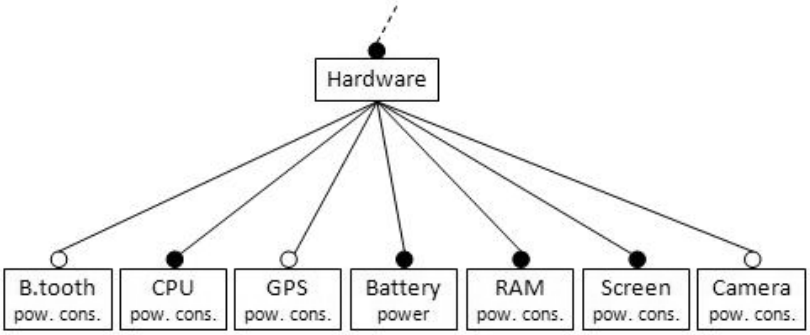

Figure 3.5 Part of a sample feature model for a mobile phone

Assume that a global requirement, *GR*, states that; *total power consumption of the electronic components in the mobile phone cannot exceed the power supplied by the battery*. This global requirement can be represented using the *sum* constraint as follows:

$$\text{FS=filter((Btooth.selected, Btooth.pc),...,(Camera.selected, Camera.pc))}$$
$$\wedge\ \text{sum(FS, total, 1)} \wedge (\text{total} \leq \text{Battery.power})$$

An alternative representation, if the attributes involved in *GR* has *0* as the *common neutral value*, would be:

$$\text{sum(Btooth.pc, CPU.pc, GPS.pc, RAM.pc, Screen.pc, Camera.pc, total, 1)}$$
$$\wedge\ (\text{total} \leq \text{Battery.power})$$

The attributes involved in the *sum* constraint are, then, mapped as:

$$(\text{Btooth.selected} \Rightarrow \text{Btooth.pc} \in \{\ldots\}) \wedge (\neg\text{Btooth.selected} \Rightarrow \text{Btooth.pc} = 0)$$
$$\wedge\ \text{Btooth.pc} \in \{\ldots\} \cup \{0\} \wedge \ldots \wedge \text{Camera.pc} \in \{\ldots\} \cup \{0\} \wedge$$
$$(\text{Camera.selected} \Rightarrow \text{Camera.pc} \in \{\ldots\}) \wedge (\neg\text{Camera.selected} \Rightarrow \text{Camera.pc} = 0)$$

This groundwork provided in our proposal enables using any global constraint on feature models. The results of the performance test we have performed on the *FM 2* (see Appendix B) indicate that using global constraints increased efficiency up to 25% in the most time consuming analysis operations such as *all products*, *commonality,* and *filter*. The increase in the efficiency was even above 50% in optimization operations (see Table 3.1).

Table 3.1 Performance results for some of the analysis operations

Analyses Operation	≈ Time (seconds)	
	With library GC predicates	Without library GC predicates
Void feature model	0.015	0.023
Valid product	0.005	0.005
Valid partial configuration	0.015	0.022
All products	16.305	20.360
Number of products	16.322	20.375
Commonality[1]	26.609	33.427
Filter[1]	10.422	12.974
Core features	0.217	0.362
Variant features	0.218	0.364
Dead features	0.218	0.363
False optional features	0.218	0.364
Optimization *(maximize cost)*	0.109	0.246
Optimization *(minimize cost)*	0.089	0.234

We have performed the tests on a computer with an Intel Core 2 Duo T5500 1.66 GHz CPU and 2 GB RAM, and running Microsoft Windows XP Professional. Note, however, that although the computer had 2 GB of physical memory, the SICStus Prolog version we have can utilize only 256 MB of memory on 32 bit systems [41].

The tool automatically derived 338,928 F-products (see 4.1) from the sample feature model. For the operations *number of products* and *commonality* we have generated the set of all products to find the answer. It would be possible to implement these operations without actually generating the set of all products; however this is not an issue we are intending to address here. For the optimization operation we have asked the tool to derive *the most expensive* and *the least expensive* products.

[1] Time for these operations heavily depend on the input configuration. For both of the operations we have used *"products including CPU 2"* as the input configuration.

CHAPTER 4

ANALYSIS OPERATIONS REVISITED

As we have briefly discussed in Chapter 3 inclusion of attributes in complex cross-tree relations caused a new type of variability, *variability in terms of attributes*, to emerge. This new variability type has effects on some of the analysis operations defined on feature models. In this chapter we discuss a proposal [33] that redefines and reformulates some of the analysis operations in the light of the new variability type introduced. We first categorize products considering the effects of the variability in terms of attributes. Then we suggest revised definitions for some of the analysis operations, and propose reformulation for some other. We also briefly discuss implementation issues on analysis operations, and present performance results we have obtained on two sample extended feature models.

4.1 A New Type of Variability

Feature models provide effective relationships to represent the variability in product families. Consider the sample feature model given in Figure 4.1.

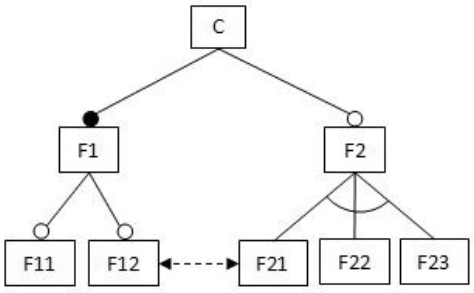

Figure 4.1 A sample feature model, $\mathcal{M}$

Decomposition relationships such as *optional relation*, *alternative relation*, and *or relation* are used to specify the variability points and the variants in feature models. For instance, the *optional relation* between the features F1 and F11 states that a product that includes the feature F1 would include the feature F11 whereas another one would not. Thus, {C, F1} and {C, F1, F11} are both valid products that can be derived from the given feature model. Or, as another example, the *alternative relation* between the parent feature F2 and its children F21, F22, and F23 states that exactly one of the children will be included in a product that includes the parent feature F2. Therefore, the products {C, F1, F2, F21} and {C, F1, F2, F22} belong to the product family represented by the feature model, whereas the

product {C, F1, F2, F21, F22} does not as it includes two children F21 and F22, which is a violation of the constraint specified by the alternative relation.

When all the variability in a feature model is removed we reach to a *fully specialized feature diagram*, in other words a full configuration [16]. The remaining features in the feature diagram make up a product belonging to the product family. For instance, the following list includes all the products that can be derived by removing the variability from the feature model $\mathcal{M}$.

- P1 = {C, F1}
- P2 = {C, F1, F11}
- P3 = {C, F1, F12}
- P4 = {C, F1, F11, F12}
- P5 = {C, F1, F2, F21}
- P6 = {C, F1, F2, F22}
- P7 = {C, F1, F2, F23}
- P8 = {C, F1, F11, F2, F21}
- P9 = {C, F1, F11, F2, F22}
- P10 = {C, F1, F11, F2, F23}
- P11 = {C, F1, F12, F2, F22}
- P12 = {C, F1, F12, F2, F23}
- P13 = {C, F1, F11, F12, F2, F22}
- P14 = {C, F1, F11, F12, F2, F23}

Products are represented by the set of features they include. A feature has just two possible states: it is either included in a product or not. For instance, the feature F2 is included in the products P5 through P14, whereas it is not included in the products P1 through P4. Note that, the only difference between the products P1 and P2 is that P2 includes the feature F11 whereas P1 does not. Thus, inclusion or exclusion of a feature would lead to two different products. However, the situation gets more complicated when the attributes are involved.

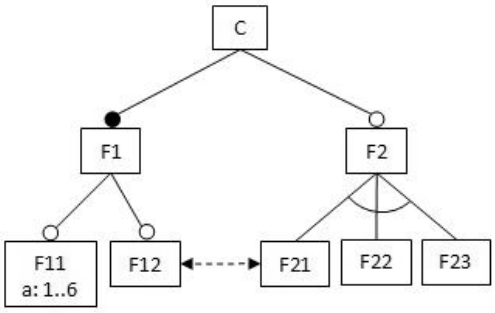

Figure 4.2 A sample extended feature model, $\mathcal{EM}$

The sample extended feature model, $\mathcal{EM}$, given in Figure 4.2 is a slightly modified version of the feature model $\mathcal{M}$. We have added the attribute *a*, which has the domain 1..6, to the feature F11.

Assume that we have a cross-tree relation, say *CTR*, such that *"F12 requires F11.a > 3"*. As, for a particular product, an attribute can have a meaningful value only if the feature it belongs to is included in the product, this constraint must be interpreted as *"F12 requires F11 and F12 requires F11.a > 3"*.

As attributes may take on a variety of values once the feature it belongs to is included in a configuration, the attribute *F11.a* would take different values when the feature F11 is included in a configuration. For instance consider the configuration $C_1 = (\{F11\}, R_1)$, as F11 is selected the attribute F11.a may take a value from 1..6. However, when we consider the configuration $C_2 = (\{F11, F12\}, R_2)$ the attribute F11.a may take a value from 4..6, which is a subset of its originally specified domain, due to the constraint *CTR*. Thus, the domain of an attribute may be restricted in a configuration due to the cross-tree constraints in the model.

Admissible Domain: *Admissible Domain (A-domain)* of an attribute *a* with respect to a configuration *C* is the space of possible values that the attribute *a* may take its value in configuration *C*.

For instance, A-domain of the attribute F11.a is 1..6 in C_1, and 4..6 in C_2.

Note that the A-domain of an attribute may be equal to its originally specified domain in some configurations, and may be restricted due to the cross-tree constraints in the model in some other. Thus, besides specifying the set of features to be selected and removed, it is also necessary to specify the A-domains of the attributes that belong to the selected features in a configuration to represent the configuration correctly. Therefore, we redefine a configuration as follows:

Configuration: Given a feature model with a set of features *F*, a *configuration* is a 3-tuple of the form *(S, R, A)* such that *S* and *R* are two disjoint subsets of *F*, where *S* is the set of features to be selected, *R* is the set of features to be removed, and *A* is the set of A-domains of the attributes that belong to the features in *S*. A configuration is called a *full configuration* If $S \cup R = F$, and *partial configuration* otherwise.

Similarly we redefine a product as follows:

Product: A *product* is a full configuration where only selected features and the A-domains of the attributes that belong to the selected features are specified, and the removed features are implicitly omitted.

For instance the product P4 must be represented as:

- P4 = ({C, F1, F11, F12}, {F11.a ∈ 4..6})

Note that this representation has no variability in terms of features as it is a full configuration. However, there is still some unresolved variability in the value the attribute *F11.a* may take. If we continue to remove this variability, it is possible to derive three different products from the product P4 as follows:

- P4_1 = ({C, F1, F11, F12}, {F11.a = 4})
- P4_2 = ({C, F1, F11, F12}, {F11.a = 5})
- P4_3 = ({C, F1, F11, F12}, {F11.a = 6})

Although these three products have exactly the same set of features they differ at the value assigned to the attribute *F11.a*, thus all these three products would be considered as different products. As stated above, the situation gets complicated because the attribute *F11.a* may get three different values in this particular product P4. Thus, removing the variability in terms of features would not be enough to reach to a fully specialized product, as features themselves would contain a new type of variability; *variability in terms of attributes*.

As this new variability type is defined on the level of attributes, some features would contain such variability whereas some other would not. For instance, the feature *F11* in the product P4 contains variability in terms of attributes, whereas the feature *F11* in the product P4_1 does not. Also note that the features *C*, *F1*, and *F12* do not contain such variability in any of the products as they do not have any attributes. Hence it is possible to classify the features in a product as *fully specialized features* and *partially specialized features*.

- A p***artially specialized feature*** is a feature that still has unresolved variability in terms of attributes (i.e. at least one attribute is allowed to take different values from its –possibly restricted– domain).
- A ***fully specialized feature*** is a feature that contains no variability in terms of attributes (i.e. either has no attributes or exact values have been assigned to all of its attributes).

Following this classification it is also possible to classify the products with respect to the features they include. For instance, the product P4 includes a partially specialized feature, namely *F11*, whereas the products P4_1, P4_2, and P4_3 include only fully specialized features. Hence it is possible to classify the products as follows:

- A ***partially specialized product*** *(P-product)* is a product that includes at least one partially specialized feature.
- A ***fully specialized product*** *(F-product)* is a product that includes only fully specialized features.

With this categorization the answer that would be given to the question *"which are the products that can be derived from the feature model FM?"* depends on the type of products that will be listed. For instance, if we want to list partially specialized products then the following list, which includes 11 products, would be produced:

- P1 = ({C, F1}, {})
- P2 = ({C, F1, F11}, {F11.a ∈ 1..6})
- P4 = ({C, F1, F11, F12}, {F11.a ∈ 4..6})
- P5 = ({C, F1, F2, F21}, {})
- P6 = ({C, F1, F2, F22}, {})
- P7 = ({C, F1, F2, F23}, {})
- P8 = ({C, F1, F11, F2, F21}, {F11.a ∈ 1..6})
- P9 = ({C, F1, F11, F2, F22}, {F11.a ∈ 1..6})

- P10 = ({C, F1, F11, F2, F23}, {F11.a ∈ 1..6})
- P13 = ({C, F1, F11, F12, F2, F22}, {F11.a ∈ 4..6})
- P14 = ({C, F1, F11, F12, F2, F23, {F11.a ∈ 4..6})

However, if we want to list fully specialized products then the following list, which includes 57 products, would be produced:

- P1 = ({C, F1}, {})
- P2_1 = ({C, F1, F11}, {F11.a = 1})
- …
- P2_6 = ({C, F1, F11}, {F11.a = 6})
- …
- P14_1 = ({C, F1, F11, F12, F2, F23}, {F11.a = 4})
- P14_2 = ({C, F1, F11, F12, F2, F23}, {F11.a = 5})
- P14_3 = ({C, F1, F11, F12, F2, F23}, {F11.a = 6})

With the same question asked it was possible to derive two different lists as the answer. This is due to the effects of the new variability type introduced. As we discuss in the following sections, this new variability type necessitates reformulation of some of the analysis operations defined in the literature.

4.2 Products and Lists of Products

4.2.1 More on Partially Specialized Products

Consider the sample extended feature model given in Figure 4.3.

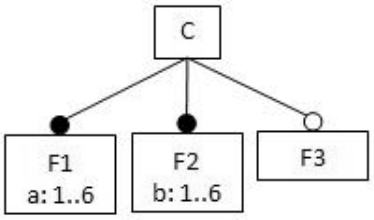

Figure 4.3 A sample extended feature model

Assume that we have the following cross-tree relationships defined on this extended feature model:

- If *F1.a* ≥ 3 then *F1* requires *F2.b* ≥ 3
- If *F1.a* < 3 then *F1* requires *F2.b* < 3
- If *F1.a* ≥ 5 then *F1* requires *F3*
- If *F1.a* < 5 then *F1* excludes *F3*

Consider the following two P-products that can be derived from this feature model:

- P1 = ({C, F1, F2}, {F1.a ∈ 1..2, F2.b ∈ 1..2})
- P2 = ({C, F1, F2}, {F1.a ∈ 1..4, F2.b ∈ 1..6})

Although both of the products are classified as P-products, there is a major difference between them. For the former product, P1, no assignment from an attribute's A-domain to the attribute has an effect on the A-domains of the other attributes, whereas this is not the case for the latter product, P2. For instance, the assignment *F1.a = 1* removes the values 3, 4, 5, and 6 from the A-domain of *F2.b* due to the constraint *"If F1.a < 3 then F1 requires F2.b < 3"* for the product P2. This characteristic leads to a new product type; *free choice product*.

Free-Choice Product: A *free choice product (C-Product)* is a P-product such that no assignment from one of its attribute's A-domain to the attribute has an effect on the A-domains of its other attributes (if there is any). Note that, as every C-product is also a P-product, C-products also contain unresolved variability in terms of attributes.

Some of the C-products that can be derived from the sample extended feature model given in Figure 4.3 are as follows:

- P1 = ({C, F1, F2}, {F1.a ∈ 1..2, F2.b ∈ 1..2})
- P2 = ({C, F1, F2}, {F1.a ∈ 3..4, F2.b ∈ 3..6})
- P3 = ({C, F1, F2, F3}, {F1.a ∈ 5..6, F2.b ∈ 3..6})

4.2.2 Lists of Products

Following the classification given for products in the previous sections we will classify the lists of products as follows:

- A list of products is called an *F-list (Fully Specialized List)* if all the products in the list are F-products.
- A list of products is called a *P-list (Partially Specialized List)* if all the products in the list are P-products.
- A list of products is called a *C-list (Free Choice List)* if all the products in the list are C-products.

Recall that every C-product is also a P-product, and C-products exhibit more interesting properties than P-products. Therefore we will focus our attention on the C-products and C-lists rather than P-products and P-lists in the remainder of this study.

Consider the following list of products:

- P1 = ({C, F1, F2}, {F1.a = 1, F2.b = 2})
- P2 = ({C, F1, F2, F3}, {F1.a ∈ 5..6, F2.b ∈ 3..6})

This list is not an F-list as it contains a P-product, P2, and not a P-list nor C-list as it contains an F-product, P1. However, there is an important characteristic this list possesses: No two products in this list share the same set of features. Thus, we add a new category for the classification of lists:

Generalized List: A list of products is called a *G-list (Generalized List)* if no two products in the list have the same set of features and for any product in the list no assignment from the A-domain of an attribute to the attribute has an effect on the set of features that make up the product (i.e. necessitates the inclusion or exclusion of a feature).

4.2.3 Relations between Lists of Products

Corresponding F-list: Let $\mathcal{M}$ be a feature model, and $\mathcal{L}$ a list of products derived from $\mathcal{M}$, then *the corresponding F-list* for $\mathcal{L}$ is the list of all F-products that can be computed by removing the variability (if there is any) from the products listed in $\mathcal{L}$ in accordance with the constraints specified in $\mathcal{M}$.

Example: Assume that a list contains a single product ({C, F1, F2}, {F1.a ∈ 1..3, F2.b ∈ 1..3}) that is derived from the sample extended feature model given in Figure 4.3 then the corresponding F-list contains the following five products:

- P1 = ({C, F1, F2}, {F1.a = 1, F2.b = 1})
- P2 = ({C, F1, F2}, {F1.a = 1, F2.b = 2})
- P3 = ({C, F1, F2}, {F1.a = 2, F2.b = 1})
- P4 = ({C, F1, F2}, {F1.a = 2, F2.b = 2})
- P5 = ({C, F1, F2}, {F1.a = 3, F2.b = 3})

Corresponding G-list: Let $\mathcal{M}$ be a feature model and $\mathcal{L}$ a list of products derived from $\mathcal{M}$. Then *the corresponding G-list* for $\mathcal{L}$, $\mathcal{G}$, is a G-list and the corresponding F-lists for $\mathcal{G}$ and $\mathcal{L}$ are exactly same.

Note that a G-list or a corresponding G-list would include some F-products, some C-products, and some P-products as there is no restriction on the type of products to be included.

Example: Assume that a list contains the following four products derived from the sample extended feature model given in Figure 4.3:

- P1 = ({C, F1, F2}, {F1.a ∈ 1..2, F2.b ∈ 1..2})
- P2 = ({C, F1, F2}, {F1.a ∈ 3..4, F2.b ∈ 3..4})
- P3 = ({C, F1, F2, F3}, {F1.a = 5, F2.b = 3})
- P4 = ({C, F1, F2, F3}, {F1.a ∈ 5..6, F2.b ∈ 5..6})

Then, the corresponding G-list would contain the following two products:

- P1 = ({C, F1, F2}, {F1.a ∈ 1..4, F2.b ∈ 1..4})
- P2 = ({C, F1, F2, F3}, {F1.a ∈ 5..6, F2.b ∈ {3, 5, 6}})

Corresponding C-list: Let $\mathcal{M}$ be a feature model and $\mathcal{L}$ a list of products derived from $\mathcal{M}$. Then *a corresponding C-list* for $\mathcal{L}$, $\mathcal{C}$, is a C-list and the corresponding F-lists for $\mathcal{C}$ and $\mathcal{L}$ are exactly same.

Example: Assume that we want to compute a corresponding C-list for the list of all F-products that can be derived from the sample extended feature model given in Figure 4.3, which contains 20 F-products. A corresponding C-list consists of three C-products:

- P1 = ({C, F1, F2}, {F1.a ∈ 1..2, F2.b ∈ 1..2})
- P2 = ({C, F1, F2}, {F1.a ∈ 3..4, F2.b ∈ 3..6})
- P3 = ({C, F1, F2, F3}, {F1.a ∈ 5..6, F2.b ∈ 3..6})

Another corresponding C-list consists of four C-products:

- P1 = ({C, F1, F2}, {F1.a ∈ 1..2, F2.b ∈ 1..2})
- P2 = ({C, F1, F2}, {F1.a ∈ 3..4, F2.b ∈ 3..6})
- P3 = ({C, F1, F2, F3}, {F1.a ∈ 5..6, F2.b ∈ 3..4})
- P4 = ({C, F1, F2, F3}, {F1.a ∈ 5..6, F2.b ∈ 5..6})

Thus, it may be possible to find more than one corresponding C-list for a given list. Moreover, it might be the case that no corresponding C-list for a list exists (e.g. when no feature in the products has an attribute). The same arguments apply for P-lists as well, thus we will mainly focus on F-lists and G-lists in the following sections.

4.3 Analysis Operations Revisited

In this section we discuss some of the well known analysis operations on feature models, propose a reformulation to some and suggest a revised understanding for some other considering the effects of the new type of variability introduced.

4.3.1 All Products

Recall that the input of this operation is a feature model, and the output is the list of all products represented by the model. For instance, we have seen that 14 products, P1 through P14, can be derived from the sample feature model given in Figure 4.1.

As discussed in the previous sections, the new type of variability introduced due to the inclusion of complex feature-attribute and attribute-attribute relationships in feature models leads to the classification of lists of products as F-list, C-list, and G-list. Hence the answer would differ depending on the type of products to be derived. For instance, the clp(FD) solver automatically derived 137,280 F-products from *FM 1*, and 338,928 F-products from *FM 2*. However, when we have asked to derive the corresponding G-list the solver produced lists consisting of 256 and 1,158 products respectively (see Table 4.1).

Table 4.1 Number of products derived by the clp(FD) solver

Feature Model	# of Products in the Solution Set	
	F-list	Corresponding G-list
FM 1	137,280	256
FM 2	338,928	1,158

It becomes necessary to specify the type of products to be derived, as different lists of products would be derived depending on the type of the products. Thus, we suggest revising the definition of the *all products* analysis operation as follows:

***All Products*:** This operation takes a feature model and the type of list (i.e. F-list, C-list, or G-list) to be derived, and returns the desired list of all products (i.e. the list of all F-products, a corresponding C-list for the list of all F-products, or the corresponding G-list for the list of all F-products) that can be derived from the model.

Note that, if we ask to compute a corresponding C-list it may not be possible to answer with a list of products as discussed in 4.2.3.

4.3.2 Number of Products

Recall that the input of this operation is a feature model, and the output is the number of products represented by the model. For instance, we have seen that 14 products can be derived from the sample feature model given in Figure 4.1.

As the arguments we have proposed for the analysis operation *all products* apply to this operation as well (see Table 4.1), we suggest revising the definition of the *number of products* analysis operation as follows:

***Number of Products*:** This operation takes a feature model and a list type (i.e. F-list, C-list, or G-list), and returns the number of products that the desired list of all products (i.e. the list of all F-products, a corresponding C-list for the list of all F-products, or the corresponding G-list for the list of all F-products) would include.

Note that, if we feed the operation with C-list as the list type it may not be possible to answer with a unique value as discussed in 4.2.3.

4.3.3 Valid Partial Configuration

Recall that the input of this operation is a feature model and a configuration that may be partial, and the output is the list of all products that include the configuration and represented by the model. For instance, the configuration *(S, R) = ({F12}, {F22})*, being *S* the set of features to be selected and *R* the set of features to be removed, is valid as it is possible to derive a product (e.g. P3 = {C, F1, F12}) from the feature model given in Figure 4.1 that contains this configuration. On the other hand, the configuration *(S, R) = ({F12}, {F1})* is invalid, as no product that includes a child (F12) without including its parent (F1) can be derived.

Recall that we have given a new definition for *configuration* in Section 4.1. With this new definition we can deterministically answer if a configuration (possibly partial) is valid or not. For instance the configuration ({F11, F12}, {}, {F11.a ∈ 5..6}) is valid, whereas the configuration ({F11, F12}, {}, {F11.a ∈ 1..2}) is not, with respect to $\mathcal{EM}$.

Note that with this redefinition another discussion arises: which answer must be given to the question *"is the configuration ({F11, F12}, {}, {F11.a ∈ 3..4}) valid with respect to ℰℳ?"*. One may adopt one of the two possible points of view to answer this question, as the assignment *F11.a = 3* invalidates this configuration whereas the assignment *F11.a = 4* would validate it:

- A configuration is valid if at least one assignment from the given domains to the attributes that does not lead to a contradiction can be found.
- A configuration is valid if no possible assignment from the given domains to the attributes leads to a contradiction.

In the scope of this work we will adopt the former point of view, thus we will answer the question mentioned above with "Yes".

Note that the old configuration definition (i.e. where A-domains are omitted) can be treated as a special case, which means no extra restriction on the domains of the attributes exists, of the general definition we suggest.

4.3.4 Valid Product

Recall that the input of this operation is a feature model and a product (i.e. set of features), and the output is a value that indicates whether the product belongs to the set of products represented by the feature model or not. For instance, the product {C, F1, F12} is valid with respect to the feature model given in Figure 4.1, whereas the product {C, F1, F12, F21} is not as F12 excludes F21.

However, as discussed in 4.1, the definition *set of features* is not enough to specify a product due to the new variability type introduced. Thus, the revised product definition given in 4.1 must be used in this analysis operation.

Also recall that we have classified the products as F-products, C-products, and P-products, and provided definitions for these product types. For instance a product would be a valid P-product but an invalid C-product. Thus, we suggest revising the definition of the *valid product* analysis operation as follows:

***Valid Product*:** This operation takes a feature model, a product and the type of the product (i.e. F-product, C-product, P-product), and returns a value informing whether the product belongs to the set of products of the specified type that are represented by the model or not.

Note that a P-product is considered to be valid if the following restriction holds: an assignment to an attribute from its A-domain would restrict other attributes' A-domains; however no assignment would have an effect on the set of features that make up the P-product (i.e. does not require the inclusion/exclusion of a new feature to/from the set of features). For instance, consider the feature model *ℰℳ*. The product ({C, F1, F11, F12}, {F11.a ∈ 1..6}) is not considered as a valid P-product as the assignment *F11.a = 1* requires the exclusion of the feature *F12* due to the constraint *"F12 requires F11.a > 3"*.

4.3.5 Filter

Recall that the input of the operation is a feature model and a configuration that may be partial, and the output is the list of all products that include the configuration and represented by the model. For instance, assume that we have the input configuration *(S, R) = ({F12}, {F22})*, being *S* the set of features to be selected and *R* the set of features to be removed, and we apply filter using this configuration to the feature model given in Figure 4.1. Then, the outcome is:

- P3 = {C, F1, F12}
- P4 = {C, F1, F11, F12}
- P12 = {C, F1, F12, F2, F23}
- P14 = {C, F1, F11, F12, F2, F23}

As discussed before, with the inclusion of complex feature-attribute and attribute-attribute relationships a particular assignment to an attribute may require or exclude a set of features. Hence, we shall use the general definition given for *configuration* for the *filter* operation.

An issue to be discussed about the *filter* operation is the type of products to be derived. For instance, assume that we have the following configurations C1 and C2 for the sample feature models *FM* 1 and *FM 2* respectively:

- C1 = (S1, R1, A1) = ({CPU 2}, {GPS}, {CPU 1.speed = 520})
- C2 = (S2, R2, A2) = ({CPU 2}, {Organizer}, {RAM.size ∈ {512, 1024}})

Table 4.2 Number of products derived by the clp(FD) solver

Feature Model	# of Products in the Solution Set	
	F-List	Corresponding G-List
FM 1 (C1)	11,968	64
FM 2 (C2)	79,840	484

Table 4.2 summarizes the results of applying the *filter* operation using the configurations C1 and C2 on the feature models *FM* 1 and *FM 2* respectively. As in the case of the analysis operation *all products* the answer set depends on the type of products to be derived. For instance, as it can be seen in Table 4.2, the clp(FD) solver automatically derived 11,968 F-products, however only 64 products exist in the corresponding G-list when we have applied the *filter* operation using C1 on *FM* 1.Thus, we suggest revising the definition of the *filter* analysis operation as follows:

***Filter*:** This operation takes a feature model, a configuration (potentially partial), and the type of list (i.e. F-list, C-list, or G-list) to be derived, and returns the list of all products (i.e. the list of all F-products, a corresponding C-list for the list of all F-products, or the corresponding G-list for the list of all F-products) that include the input configuration and can be derived from the model.

Note that, if we ask to compute a corresponding C-list it may not be possible to answer with a list as discussed in 4.2.3.

4.3.6 Commonality

Recall that the input of the operation is a feature model and a configuration, and the output is the commonality degree of the model that is computed with the following formula:

$$Commonality(C) = \frac{|filter(C)|}{Number\ of\ Products} \quad (4.1)$$

For instance, assume that we have the input configuration *(S, R) = ({F12}, {F22})*, and we want to compute the *commonality* of this configuration with respect to the feature model given in Figure 4.1. Then, the outcome is 4 / 14 = 0.286.

As it is discussed in 4.3.2 and 4.3.5 the outcome of the analysis operations *number of products* and *filter* depend on the type of list specified. For instance, consider the configurations C1 and C2 presented in 4.3.5, the outcome of the *commonality* analysis operation differs when we consider F-list and corresponding G-list (see Table 4.3).

Table 4.3 Commonality of the configurations C1, and C2

Feature Model	Commonality	
	F-list	Corresponding G-list
FM 1 (C1)	0.087	0.250
FM 2 (C2)	0.236	0.418

As the results imply it is necessary to specify the type of the product list that will be taken into consideration. Thus, we suggest revising the definition of the *commonality* analysis operation as follows:

***Commonality*:** This operation takes a feature model, a configuration (potentially partial), and the type of product list (i.e. F-list, C-list, or G-list) that will be taken into consideration and returns the percentage of products represented by the model including the input configuration.

Note that, if we ask to compute the commonality of a configuration with respect to a C-list it may not be possible to answer with a unique value as the outcome of the *number of products* operation for the C-list would not produce a unique value as discussed in 4.3.2.

4.3.7 Variability Factor

Recall that the input of the operation is a feature model, and the output is the variability factor of the model that is computed with the following formula.

$$Variability\ Factor = \frac{Number\ of\ Products}{2^n} \quad (4.2)$$

For instance, the variability factor of the feature model given in Figure 4.1 is 14 / 2^5 = 0.438, where the root and non-leaf features are not considered.

However, as discussed in 4.3.2 the outcome of the *number of products* operation depends on the type of product list to be considered. Also the formula to be used for the calculation of the potential number of products represented by the model, which is 2^n, would have to be revised. Hence, we will first revise the definition of the operation and then discuss how the potential number of products can be calculated for different types of products.

***Variability Factor*:** This operation takes a feature model and the product list type (i.e. F-list, or G-list) that will be taken into consideration and returns the ratio between the numbers of products in the given list type that are actually represented by the model and the potential number of products of the given list type represented by the model.

When we consider the potential number of products in the F-list we see that the domains of the attributes affect the formulation as we count two F-products with at least one different attribute assignment distinct, even if they include the same set of features. Thus, assuming we have an extended feature model with n features $\{F_1, \ldots, F_n\}$, where k features, $0 \leq k \leq n$, have attributes the formula to calculate the variability factor is:

$$Variability\ Factor = \frac{Number\ of\ F-products}{2^{n-k} * \left(|D_{1.1}| * \ldots * |D_{1.p}| + 1\right) * \ldots * \left(|D_{k.1}| * \ldots * |D_{k.q}| + 1\right)} \quad (4.3)$$

where

- $D_{i.j}$ is the domain of the j^{th} attribute of the i^{th} feature.
- $|D|$ is the cardinality of domain D.

Note that we calculate the number of all possible assignments to the attributes of a feature for the case that the feature is selected to be included in the product, and then add 1 to this value for the case that the feature is not selected. For instance, the variability factor computed using the F-list type for the feature models $\mathcal{FM}$ 1and $\mathcal{FM}$ *2* are respectively 0.268 and less than 0.001 (see Table 4.4).

Due to their definition, recall that no two products in a G-list can have the same set of features, the potential number of products is 2^n, as attributes' values do not have any effect on the product and only the inclusion or exclusion of a feature generates a different product. Thus, the original formula can be used when we consider the G-list:

$$Variability\ Factor = \frac{Number\ of\ products\ in\ the\ G-list}{2^n} \quad (4.4)$$

When we consider C-list we see that both the set of features included in the products and the domains of the features' attributes affect the potential number of products as two distinct C-products with exactly the same sets of features but different attribute domains, or two distinct C-products with exactly the same attribute domains but different sets of features may exist in the same C-list. Thus, it is not possible to anticipate the relation between the sets of features and domains of attributes as they are strictly determined by the constraints defined in the particular feature model under analysis.

Therefore, the formula for calculating the number of potential products represented by a feature model would vary from model to model.

Table 4.4 Variability factor of the sample feature models *FM* 1 and *FM 2*

Feature Model	Variability Factor	
	F-list	Corresponding G-list
FM 1	0.268	0.062
FM 2	less than 0.001	less than 0.001

4.3.8 Degree of Orthogonality

Recall that the input of the operation is a feature model and a feature that represents a subtree, and the output is the degree of orthogonality (i.e. the ratio between the total number of products represented by the feature model and the total number of products represented by the subtree). For instance, the degree of orthogonality of the subtree *F2* with respect to the sample feature model given in Figure 4.1 is 14/10 = 1.4. However, as we have discussed previously total number of products that can be derived from a feature model depends on the type of product list to be derived. For instance, for the subtree *Hardware* for *FM 1*, the clp(FD) solver computed the results 19.861 and 16 when we have considered F-list and G-list, respectively (see Table 4.5).

Table 4.5 Degree of orthogonality of the sample feature models *FM* 1 and *FM 2*

Feature Model	Degree of Orthogonality	
	F-Products	Corresponding G-Products
FM 1 (*Hardware*)	19.861	16
FM 2 (*Applications*)	196.139	6.031

As the results imply it is necessary to specify the type of the products that will be taken into consideration. Thus, we suggest revising the definition of the *degree of orthogonality* analysis operation as follows:

***Degree of Orthogonality*:** This operation takes a feature model, a subtree (represented by its root feature), and the type of product list (i.e. F-list, C-list, or G-list) that will be taken into consideration and returns the ratio between the total number of products of the feature model and the number of products of the subtree.

Note that, if we ask to compute degree of orthogonality using a corresponding C-list it may not be possible to answer with a unique value as the outcome of the *number of products* operation for the corresponding C-list would not produce a unique value as discussed in 4.3.2.

4.3.9 Homogeneity

Recall that the input of the operation is a feature model, and the output is the homogeneity degree of the model that is computed with the following formula:

$$Homogeneity = 1 - \frac{\#uf}{\#\ products} \tag{4.5}$$

#uf is the number of features unique in one product and #products denotes the total number of products represented by the feature model. For instance, *homogeneity* of the feature model given in Figure 4.1 is 1, as there are no unique features.

As attributes are also considered in products the definition for *unique feature* would be revised. For instance consider two F-products that are derived from the sample feature model $\mathcal{EM}$. Both of the products P4_1 = ({C, F1, F11, F12}, {F11.a = 4}), and P4_2 = ({C, F1, F11, F12}, {F11.a = 5}) include the fully specialized feature *F11*, however in the former product 4 has been assigned to the attribute *a* of the feature whereas 5 has been assigned in the latter product. As attributes provide extra information about the features one would consider these two fully specialized features different, as they exhibit different characteristics. The same argument applies to partially specialized features as well. For instance consider the two products P2 = ({C, F1, F11}, {F11.a ∈ 1..6}) and P4 = ({C, F1, F11, F12}, {F11.a ∈ 4..6}) that are included in the corresponding G-list for the list of all F-products that can be derived from the feature model $\mathcal{EM}$. Although the feature *F11* is included in two products the specified domains for the attribute *F11.a* are not same. Thus, we suggest revising the definition of *unique feature* as follows:

Unique Feature: A feature is unique if the feature and the A-domains of the attributes of the feature appear in only one product.

For instance, assume that we have the following F-list for a given feature model:

- Product1 = ({X, Y, Z}, {X.a = 1})
- Product2 = ({X, Y, Z}, {X.a = 2})
- Product3 = ({X, Z}, {X.a = 1})

Then the feature *X where X.a=2* is considered to be unique. Similarly, if we have the following corresponding G-list:

- Product1 = ({X, Y, Z}, {X.a ∈ {1, 2})
- Product2 = ({X, Z}, {X.a = 1})

Then the features *X where X.a ∈{1, 2}* and *X where X.a = 1* are considered to be unique.

As we have discussed in 4.3.2 the number of products represented by a feature model depends on the type of the product list represented. Thus, the value that denotes the number of products in the formula depends on the product list type. Thus, we suggest revising the definition of the *homogeneity* analysis operation as follows:

***Homogeneity*:** This operation takes a feature model, and the type of product list (i.e. F-list, C-list, or G-list) that will be taken into consideration and returns a number that provides an indication of the degree to which a feature model is homogeneous. The total number of products represented by the feature model is computed with respect to the input product list type.

Note that, if we ask to compute homogeneity using corresponding C-list it may not be possible to answer with a unique value as the outcome of the *number of products* operation for the corresponding C-list would not produce a unique value as discussed in 4.2.3.

4.3.10 Dependency Analysis

Recall that the input of the operation is a feature model and a partial configuration, and the output is a new configuration with the features that should be selected and/or removed as a result of the propagation of constraints in the model. For instance, if we input the configuration *{{C, F12}, {}}* and the feature model given in Figure 4.1, the operation would output the configuration *{{C, F1, F12}, {F21}}*. Feature *F1* has been added to the configuration to satisfy the parental relationship with *F12*, and the feature *F21* is removed to satisfy the *excludes* relationship with *F12*.

As we have discussed in 4.3.3, with the inclusion of complex feature-attribute and attribute-attribute relationships, attributes have the power to require and/or exclude features. Therefore, the configuration to be input and returned to/from the operation must be in the structure of the new definition. Moreover, as the attributes can restrict the domains of other domains as well, this operation must analyze the effects of the constraints on the domains of the attributes as well. Thus, we suggest revising the definition of the *dependency analysis* operation as follows:

***Dependency Analysis*:** This operation takes a feature model and a partial configuration as input and returns a new configuration with the features that should be selected and/or removed, and the restricted domains of the attributes as a result of the propagation of constraints in the model.

4.3.11 Atomic Lists

Recall that the input of the operation is a feature model, and the output is the set of atomic sets (i.e. groups of features that can be treated as a unit). For instance, the set *{C, F1}* is an atomic set with respect to the feature model given in Figure 4.1, as there is a mandatory relation between the feature *F1* and its parent feature *C*.

However, due to the inclusion of complex feature-attribute and attribute-attribute relationships, one must also consider the specified domains of the attributes that belong to the features in the atomic set. For instance, assume that we have two features *X* and *Y* in an arbitrary feature model, where the feature *X* has an attribute *a* with domain 1..6. Assume that we have the following two cross-tree relationships: *(i) "if X.a < 4 then X requires Y}*, and *(ii) "if X.a ≥ 4 then X excludes Y"*. One cannot consider *{X, Y}* as an atomic set due to the latter constraint. However, we can consider *({X, Y}, {X.a ∈ 1..3})* as an atomic set. Thus, we suggest revising the definition of the *atomic set* as follows:

***Atomic Set*:** An atomic set is a group of features (at least one) and the A-domains of the attributes that belong to the features in the set that can be treated as a unit when performing certain analyses.

4.4 Implementation and Performance

Once the mapping of the extended feature model is complete it is straightforward to implement many useful analysis operations. In this subsection we discuss the implementation and performance of some of the analysis operations using the clp(FD) solver.

We have implemented the following analysis operations:

- All Products *(see 4.3.1)*
- Number of Products *(see 4.3.2)*
- Valid Partial Configuration *(see 4.3.3)*
- Valid Product *(see 4.3.4)*
- Filter *(see 4.3.5)*
- Commonality *(see 4.3.6)*
- Variability Factor *(see 4.3.7)*
- Degree of Orthogonality *(see 4.3.8)*
- Void Feature Model *(This operation takes a feature model as input and returns a value informing whether such feature model is void or not. A feature model is void if it represents no products [7])*
- Core Features *(This operation takes a feature model as input and returns the set of features that are included in all the products in the software product line [7])*
- Variant Features *(This operation takes a feature model as input and returns the set of variant features in the model. Variant features are those that appear in some but not all the products of the software product line [7])*
- Dead Features *(This operation takes a feature model as input and returns the set of dead features in the model. Dead features are those that do not appear in any product of the software product line [7])*
- False Optional Features *(This operation takes a feature model as input and returns the set of false optional features in the model. A feature is false optional if it is included in all the products of the product line despite not being modeled as mandatory [7])*
- Optimization *(This operation takes a feature model and a so-called objective function as inputs and returns the product fulfilling the criteria established by the function [7])*

We have used clp(FD), which is available as a library module for SICStus Prolog, as the implementation tool. SICStus Prolog provides a number of useful predicates such as *labeling*, *findall, statistics*, and so on that we have made use of to implement the operations. We have also used some of the global constraint library predicates such as *all_different*, *global_cardinality*, *sum*, *maximum* and so on provided by the clp(FD) solver to map the global constraints defined on the sample feature model *FM 2.*

For the analysis operations *number of products*, *commonality*, and *variability factor* we have generated the set of all products to find the answer. It would be possible to implement these operations

without actually generating the set of all products; however this is not an issue we are intending to address here.

For the optimization operation we have asked the tool to derive *the product with the greatest value that can be assigned to Video Call.mpc* and *the product with the smallest RAM.size* for the sample feature model *FM 1*, and *the most expensive* and *the least expensive* products for the sample feature model *FM 2*.

Note that, for the operations *filter* and *commonality* the runtime heavily depends on the input configuration. For both of the operations we have used C1 as the input configuration for *FM 1* and C2 as the input configuration for *FM 2*, where:

- C1 = (S1, R1, A1) = ({CPU 2}, {GPS}, {CPU 1.speed = 520})
- C2 = (S2, R2, A2) = ({CPU 2}, {Organizer}, {RAM.size ∈ {512, 1024}})

For the analysis operations *core features*, *variant features*, *dead features*, and *false optional features* we have derived the corresponding G-list for the list of all F-products. Again, it would be possible to implement these operations without actually generating the corresponding G-list; however this is not an issue we are intending to address here.

Performance results for the aforementioned analysis operations are presented in Table 4.6.

Table 4.6 Performance results for the analysis operations

Analysis Operation	≈ Time (seconds)			
	FM 1		*FM 2*	
	F-list	Cor. G-list	F-list	Cor. G-list
All products	0.531	0.008	16.305	0.047
Number of products	0.535	0.008	16.322	0.048
Valid partial configuration	0.001		0.015	
Valid product	0.001		0.005	
Filter	0.047	0.005	3.735	0.031
Commonality	0.563	0.008	20.256	0.060
Variability Factor	0.516	0.007	16.101	0.049
Void feature model	0.001		0.015	
Core features	0.015		0.217	
Variant features	0.015		0.218	
Dead features	0.015		0.218	
False optional features	0.015		0.218	
Optimization *(maximize)*	0.005		0.109	
Optimization *(minimize)*	0.005		0.089	

We have performed the tests on a computer with an Intel Core 2 Duo T5500 1.66 GHz CPU and 2 GB RAM, and running Microsoft Windows XP Professional. Note, however, that although the computer had 2 GB of physical memory, the SICStus Prolog version we have can utilize only 256 MB of memory on 32 bit systems.

4.5 New Analysis Operations

In this section we discuss some new analysis operations that arise with the inclusion of attributes in complex cross-tree relationships.

4.5.1 Generating a Product List from another Product List

As we have discussed in the previous sections different lists of products can be derived from a feature model depending on the type of products to be included. Sometimes it may be desirable to generate a product list from another one. These operations take a product list, and the feature model the list was derived from if necessary, and returns a corresponding product list consisting of products of another type.

- ***Generating Corresponding F-List:*** This operation takes a feature model $\mathcal{M}$ and a list of products $\mathcal{L}$ that is derived from $\mathcal{M}$, and returns *the corresponding F-list* for $\mathcal{L}$ with respect to $\mathcal{M}$.
- ***Generating Corresponding C-List:*** This operation takes a feature model $\mathcal{M}$ and a list of products $\mathcal{L}$ that is derived from $\mathcal{M}$, and returns, if one can be found, *a corresponding C-list* for $\mathcal{L}$ with respect to $\mathcal{M}$.
- ***Generating Corresponding G-List:*** This operation takes a feature model $\mathcal{M}$ and a list of products $\mathcal{L}$ that is derived from $\mathcal{M}$, and returns *the corresponding G-list* for $\mathcal{L}$ with respect to $\mathcal{M}$.

Note that, as we have discussed in 4.2.3, sometimes it would be possible to generate more than one corresponding C-list. Thus, the analysis operation *generating corresponding C-list* would produce "a" list, not "the" list. Next we define an analysis operation to generate "the" corresponding C-list (if it is possible). However, to be able to do that, we must first give a definition:

***Least C-list*:** If there are more than one equivalent C-lists (i.e. the same F-list would be generated from each one of them) then the list including the least number of C-products is called the *least C-list*.

Now we can define the analysis operation:

- ***Generating Corresponding Least C-List:*** This operation takes a feature model $\mathcal{M}$ and a list of products $\mathcal{L}$ that is derived from $\mathcal{M}$, and returns, if one can be found, *the corresponding least C-list* for $\mathcal{L}$ with respect to $\mathcal{M}$.

4.5.2 Product List Relations

These operations take two lists of partially specified product lists and the feature model the lists were derived from as input and return a value informing how the lists are related.

- ***Refactoring:*** A partially specified product list is a refactoring of another one if the same F-list can be generated from both of them while the lists do not include the same set of partially specialized products.
- ***Generalization:*** A partially specified product list, L, is a generalization of another, M, if the F-list that can be generated from L maintains and extends the F-list that can be generated from M.
- ***Specialization:*** A partially specified product list, L, is a specialization of another, M, if the set of products in the F-list that can be generated from L is a subset of the set of products in the F-list that can be generated from M.
- ***Arbitrary Edit:*** There is no explicit relationship between the input lists (i.e. none of the relationships defined above).

4.5.3 Dead Attribute Values

This operation takes a feature model and returns the set of dead attribute values in the feature model. An attribute value is dead if it cannot appear in any of the products that can be derived from the feature model.

Consider the sample extended feature model given in Figure 4.4.

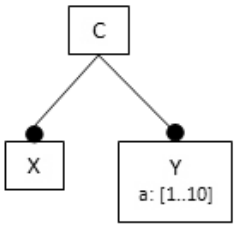

Figure 4.4 A sample extended feature model

Assume that we have a constraint *"X requires Y.a > 6"*. As feature *X* is a core feature (i.e. included in every product) the attribute *Y.a* cannot get values 1..6 in any valid product. These kinds of situations are undesired as they would give the user a wrong idea about the domains of the attributes.

4.5.4 Conditionally Dead Attribute Values

This operation takes a feature model and returns the set of conditionally dead attribute values in the feature model. An attribute value is conditionally dead if it becomes dead under certain circumstances (e.g. with the inclusion of a feature).

Consider the sample extended feature model given in Figure 4.5.

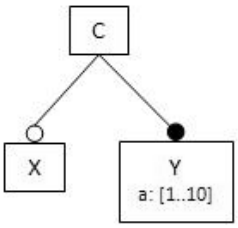

Figure 4.5 A sample extended feature model

Assume that we have a constraint *"X requires Y.a > 6"*. When the feature *X* is included in a product the attribute *Y.a* cannot get values 1..6.

4.5.5 Extra Constraint Representativeness for Attributes (ECR-A)

This operation takes a feature model as input and returns the degree of representativeness of the cross-tree constraints with respect to the attributes in the tree. Extra Constraint Representativeness for Attributes (ECR-A) is the ratio of the number of attributes involved in cross-tree constraints (repeated attributes counted once) to the number of attributes, excluding the implicit attributes, in the feature tree.

For the sample feature model *FM 1*, ECR-A is 6/6 = 1.

4.6 Discussions

In this chapter we have discussed the effects of the inclusion of attributes in cross-tree relationships. We have suggested revised definitions for some of the analysis operations on feature models, and reformulation for some other. We have also introduced new analysis operations on feature models.

New feature categories emerge due to the *variability in terms of attributes*. This new variability type is not at the same level with the variability defined in terms of features, however exhibits similar nature. Thus, the ideas used for removing variability from feature models would be used to remove variability from partially specialized features. For instance, configurations, which themselves may be partial, for partially specialized products may be introduced. Note that configurations have been used successfully in staged configuration of feature models. A similar approach would be adopted for staged configuration of partially specialized products as well.

We have discussed that, when the old definitions of the analysis operations are used it becomes possible to generate different answers for some of the analysis questions. As analysis operations are used to reveal certain characteristics of the feature models this issue would be a problem for the modelers and designers. Thus, we believe that these revised definitions and reformulations would prevent any misjudgments arising due to the new variability type emerged.

The wide variety of tool support for CLP(FD) also adds value to this study. Industrial experiences showed that feature models often grow large with hundreds, or even thousands, of features and complex cross-tree relationships, which closes the door on manual analysis of feature models. Constraint programming is regarded as a mature field of computer science, and the CLP(FD) solvers

have been used for many real-life applications, and evolved to provide efficient implementations for the computationally expensive procedures that had proven to be very effective for modeling discrete optimization and verification problems. Moreover, due to their declarative style, CLP systems lead to highly readable solutions. Thus, coding and running the analysis operations, such as the ones mentioned in 4.4 become straightforward once the mapping of the feature model is complete.

Note that the new analysis operations introduced in 4.5 are just examples of analysis operations arising due to the new ideas presented. Industrial experience would provide motivation on this matter, but it may also be possible to discover new requirements arising from the nature of the analysis needs.

The category of free-choice products presents interesting application areas. For instance, delaying the binding of the variability in feature models is a hot-topic among the communities that work on cross-fertilization of the software product lines and context-aware systems. As free-choice products ensure that any attempt to remove the variability in a partially specialized product of this category will result in a valid fully specialized product, the variability in the products belonging to this category would be delayed as desired. As context-aware systems mostly have access to limited resources (e.g. hand-held devices that do not provide high computation power or big amounts of memory) deploying C-products to run on such platforms would provide the flexibility of a partially specialized product while not demanding high resource usage. As any F-product derived from a C-product will be valid with respect to the feature model; *(i)* constraints in the feature model do not have to be employed in the run-time system, which would decrease memory requirements, *(ii)* any F-product would be derived automatically without checking any constraint, which would decrease computational power requirements.

CHAPTER 5

A SIMPLIFICATION OVER EXTENDED FEATURE MODELS

In Chapter 3 we have discussed a mapping from extended feature models to constraint logic programming over finite domains. The only restriction we have assumed is that the domains of the attributes be finite in that mapping. In this chapter we propose a simplification over extended feature models. For this proposal we allow domains of the attributes to be infinite, or even continuous; however we assume some other restrictions that are presented in the following sections. The simplification that will be presented removes cross-tree relationships involving attributes and introduces an equivalent model with only traditional cross-tree relationships. Thus, this transformation enables using existing automated reasoning approaches on extended feature models.

5.1 Transformation

One of the strengths of extended feature models is enabling to define complex cross-tree constraints involving feature attributes. But this freedom may result in challenging complications while mapping the model to a reasoning base. In this section we introduce a transformation from an extended feature model that has complex feature-attribute and/or attribute-attribute relationships to an equivalent extended feature model which contains no such complex relationships.

It is easy to map a cross-tree constraint that does not involve attribute conditions into a reasoning base. For example consider the following case given in Figure 5.1.

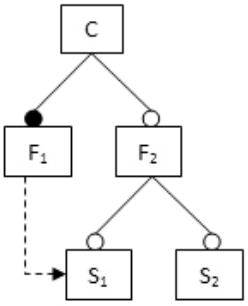

Figure 5.1A simple cross-tree constraint

The constraint "*Feature F_1 requires feature S_1*" depicted in Figure 5.1can easily be mapped as "$F_1 \Rightarrow S_1$" into propositional logic, or as "*if (A>0) B>0*" into CSP.

Generally, what causes trouble is the existence of attributes in constraints. Therefore we shall transform the extended feature model into another extended feature model, which is equivalent to the original one except there shall be no constraints involving feature attributes.

We assume the following restrictions for the feature models to be simplified:

- Only the conditions of the form attribute-constant are allowed.
- Conditions may involve only the attributes of the two features that figure in the cross-tree relationship.
- Conditions may be combined with only logical conjunction.

The transformation consists of three steps:

1. Find all cross-tree constraints that involve feature attributes.
2. Determine variant features (and their attributes' values) to be introduced.
3. Simplify all the constraints that were found in Step 1, and introduce new constraints that do not involve feature attributes.

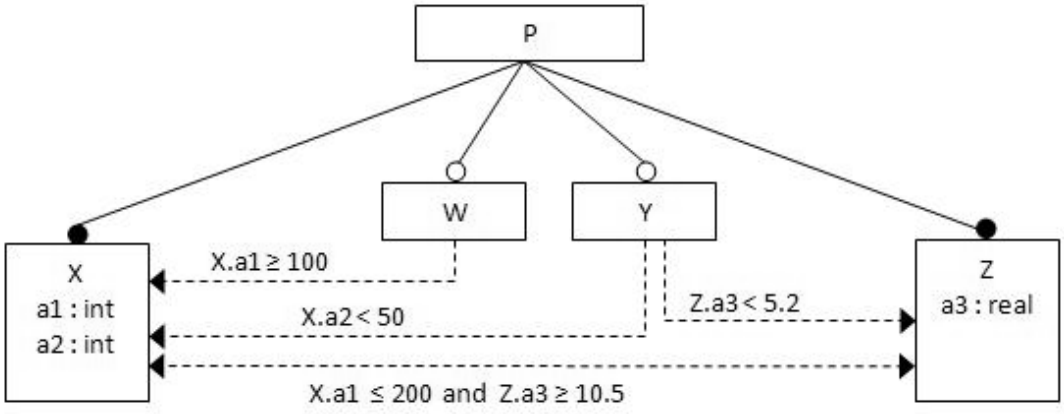

Figure 5.2 A sample extended feature model

Assume that we have the sample extended feature model given in Figure 5.2. We have one concept feature, two mandatory features, two optional features, and four cross-tree constraints involving attributes. Note that the excludes relationship between the feature *X* and feature *Z* is in the form of a *guarded constraint*, which should be interpreted as *"if X.a1≤ 200 and Z.a3 ≥ 10.5 then X excludes Z"*. Also note that we have an attribute, *Z.a3*, with a continuous domain.

We have the following cross-tree constraints:

- (C1) Feature W requires X.a1 ≥ 200
- (C2) If X.a1 ≤ 200 and Z.a3 ≥ 10.5 then feature X excludes feature Z
- (C3) Feature Y requires X.a2 < 50
- (C4) Feature Y requires Z.a3 < 5.2

In the first step all cross-tree constraints that include feature attributes are found, and classified according to the feature attribute they involve. For the sample case in Figure 5.2 we have the following listings:

Constraints on attribute a1 of feature X:

- Feature W requires X.a1 ≥ 200 (C1)
- If X.a1 ≤ 200 and Z.a3 ≥ 10.5 then feature X excludes feature Z (C2)

Constraints on attribute a2 of feature X:

- Feature Y requires X.a2 < 50 (C3)

Constraints on attribute a3 of feature Z:

- Feature Y requires Z.a3 < 5.2 (C4)
- If X.a1 ≤ 200 and Z.a3 ≥ 10.5 then feature X excludes feature Z (C2)

Next step is determining variant features to be introduced. To be able to do that we first calculate the value intervals for attributes that satisfy and/or activate various constraints. Continuing the sample case we have the following results.

For attribute a1 of feature X:

- If X.a1 < 100, then C1 is not satisfied, C2 may become effective
- If 100 ≤ X.a1 ≤ 200, then C1 is satisfied, C2 may become effective
- If X.a1 > 200, then C1 is satisfied, C2 cannot become effective

For attribute a2 of feature X:

- If X.a2 < 50, then C3 is satisfied
- If X.a2 ≥ 50, then C3 is not satisfied

For attribute a3 of feature Z:

- If Z.a3 < 5.2, then C4 is satisfied, C2 cannot become effective
- If 5.2 ≤ Z.a3 < 10.5, then C4 is not satisfied, C2 cannot become effective
- If Z.a3 ≥ 10.5, then C4 is not satisfied, C2 may become effective

We iteratively combine calculations for the attributes of the same feature until all cases are combined. Therefore, we combine the calculations for the feature X in our example iteratively.

For attributes a1 and a2 of feature X:

- If X.a1 < 100 and X.a2 < 50, then C1 is not satisfied, C2 may become effective, and C3 is satisfied
- If X.a1 < 100 and X.a2 ≥ 50, then C1 is not satisfied, C2 may become effective, and C3 is not satisfied
- If 100 ≤ X.a1 ≤ 200 and X.a2 < 50, then C1 is satisfied, C2 may become effective, and C3 is satisfied
- If 100 ≤ X.a1 ≤ 200 and X.a2 ≥ 50, then C1 is satisfied, C2 may become effective, and C3 is not satisfied
- If X.a1 > 200 and X.a2 < 50, then C1 is satisfied, C2 cannot become effective, and C3 is satisfied
- If X.a1 > 200 and X.a2 ≥ 50, then C1 is satisfied, C2 cannot become effective, and C3 is not satisfied

Next we determine the variant features. Number of cases for each feature gives us the number of variant features to be introduced. For our example case we shall need 6 variant features for the feature

X, and 3 variant features for the feature Z. These variant features have exactly the same number of attributes with the feature that they represent, and each attribute's value is restricted to reside in the given interval that was calculated for that attribute in the corresponding case. All these variant features shall be mandatory and connected to the feature they represent with an *alternative relation*.

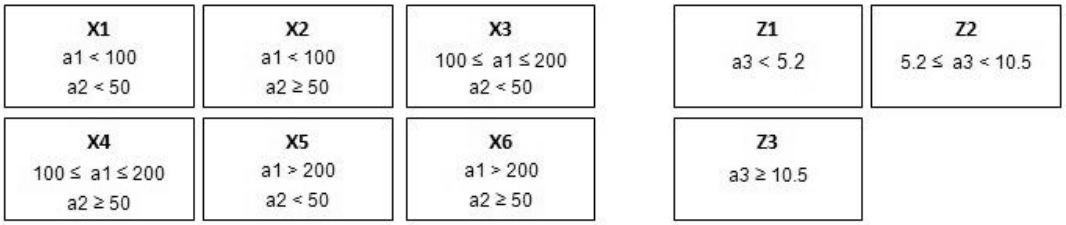

Figure 5.3 Variant features to be introduced

In the third step we remove the conditions on attributes from the constraints that were found in step 1 and introduce new constraints that can easily be mapped to the reasoning base. While introducing new constraints we apply the following rules.

For the sake of clarity we will use the following terminology while giving the rules:

- If an attribute of a feature is involved in the constraint we shall call that feature *A-Feature*.
- If a feature is a part of a constraint but no attribute of that feature is involved in that constraint we shall call that feature *S-Feature*.

Rules for *requires* relationships:

1. Every original *requires* relationship is kept, but simplified to a basic feature-feature constraint.
2. For each original *requires* relationship that is a constraint between an S-Feature and an attribute of an A-Feature; we add a basic *excludes* constraint between the S-Feature and the newly introduced variant features that represent intervals where the original constraint is not satisfied.
3. For each original *requires* relationship that is a constraint between attributes of A-Features; we add a basic *excludes* constraint between the variant features that activates the constraint, and the variant features that represent intervals where the original constraint is not satisfied.

Rules for *excludes* relationships:

4. We remove from the model every original *excludes* relationship that involves attributes.
5. For each original *excludes* relationship that is a constraint between an S-Feature and an attribute of an A-Feature; we add a basic *excludes* constraint between the S-Feature and the newly introduced variant features that represent intervals where the original constraint is satisfied.
6. For each original *excludes* relationship that is a constraint between attributes of A-Features; we add a basic *excludes* constraint between the variant features that activates the constraint, and the variant features that represent intervals where the original constraint is satisfied.

Returning to our illustrative example we do the following as the third and final step of transformation:

Constraint C1:

- We keep the original constraint but simplify it to "*Feature W requires Feature X"* (by Rule 1).
- We add the following simple constraints (by Rule 2):
  - Feature W excludes Feature X1
  - Feature W excludes Feature X2

Constraint C2:

- We remove the constraint C2 from the model (by Rule 4).
- We add the following simple constraints (by Rule 6):
  - Feature X1 excludes Feature Z3
  - Feature X2 excludes Feature Z3
  - Feature X3 excludes Feature Z3
  - Feature X4 excludes Feature Z3

Constraint C3:

- We keep the original constraint but simplify it to "*Feature Y requires Feature X"* (by Rule 1).
- We add the following simple constraints (by Rule 2):
  - Feature Y excludes Feature X2
  - Feature Y excludes Feature X4
  - Feature Y excludes Feature X6

Constraint C4:

- We keep the original constraint but simplify it to "*Feature Y requires Feature Z"* (by Rule 1).
- We add the following simple constraints (by Rule 2):
  - Feature Y excludes Feature Z2
  - Feature Y excludes Feature Z3

The transformed model is given in Figure 5.4, which is equivalent to the original model but contains no cross-tree relationships involving attributes.

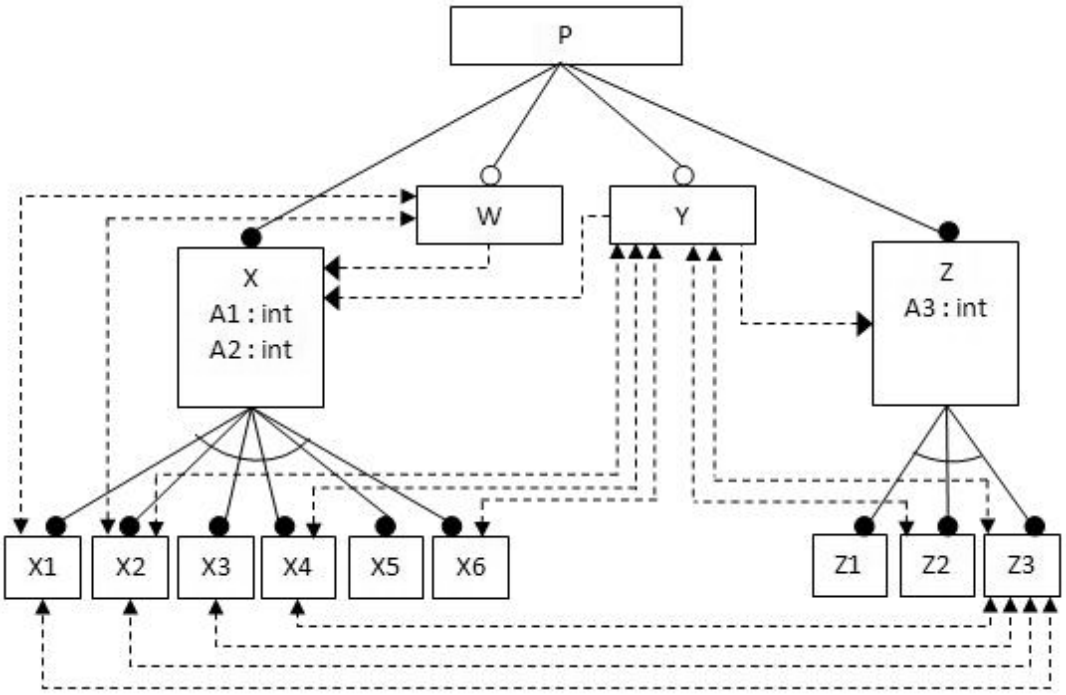

Figure 5.4 Feature model after the transformation

5.2 Complexity Issues

In this section we discuss the complexity issues. First we discuss the change in the complexity of the feature model in terms of number of features and constraints in 5.2.1. Then we give the computational complexity of the transformation process in 5.2.2.

5.2.1 Changes in the Complexity of the Model

In the illustrative example we have used to present the transformation, the extended feature model had 5 features and 4 cross-tree relationships before the transformation, and 14 features and 14 cross-tree relationships after the transformation. Since the transformation algorithm "enumerates" possibilities, an increase in the number of features and constraints is not a surprise. But the change in the complexity of the model becomes more dramatic in the worst case.

Assuming an extended feature model with *n* features and *k* cross-tree relationships each involving conjunction of 2 conditions, the worst case occurs when two features, say feature X and feature Y, have *k* (or more) attributes and all of the *k* cross-tree relationships are constraints that are defined between different attributes of these two features.

In this case, the transformation algorithm will detect one constraint for each of the *k* attributes of the features X and Y. Each constraint shall introduce at least two intervals (one activating the constraint and one deactivating the constraint; or one satisfying the constraint and one failing the constraint) for each of the *k* attributes. Since each constraint is assumed to be on different attributes iterative combination of these intervals will increase the number of possibilities by a factor of 2, which results in totally 2^k variant features for each of the features X and Y. Adding the number of

variant features introduced for X and Y we find 2^{k+1} variant features to be introduced in total. Hence the growth in the number of features will be exponential.

Similarly, the growth in the number of constraints will be exponential as well. For the sake of clarity we assume that the condition to decide whether a specific cross-tree relationship is activated or not is determined by a condition on an attribute of the feature X, and the condition to decide whether a specific cross-tree relationship is satisfied or not is determined by a condition on an attribute of the feature Y. Note that this assumption does not change the calculations, and the results would be same in case of any mixture of activation/satisfaction conditions distributed on the attributes of X and Y randomly.

In this case half of the variant features introduced for the feature X, which is equivalent to 2^{k-1} variant features, will activate a specific constraint, and half of the variant features introduced for the feature Y, which is equivalent to 2^{k-1} variant features, will satisfy (or fail) that specific constraint. Therefore the transformation algorithm shall add 2^{k-1} excludes constraints for each original cross-tree attribute-attribute relationship. Since we assumed that there are k relationships in the model total number of excludes relationships to be added is $k * 2^{k-1}$.

5.2.2 Computational Complexity of the Transformation

The complexity of the transformation algorithm is tightly related to the change in the complexity of the model. Again considering the worst case discussed in 5.2.1, It will take at least 2^{k+1} steps to introduce the variant features, and $k * 2^{k-1}$ steps to add the basic cross-tree constraints. Thus complexity of the transformation process is exponential as well.

5.3 Discussions

In this chapter we proposed a simplification over extended feature models. The transformation presented enables the use of existing automated reasoning proposals that are capable of handling extended feature models even in the case where cross-tree relationships involving attributes are present in the feature model, under certain restrictions discussed in 5.1.

As the transformation algorithm is well-defined, it would be easy to develop a tool that will perform the transformation proposed automatically. The calculations presented show that it is possible to express a feature model much more elegantly by using cross-tree constraints involving attributes while keeping the complexity of the model fairly low and small in size, which would otherwise grow exponentially. Therefore we believe that this proposal would be useful for the studies intending to decrease the complexity of feature models, which is an important and challenging issue.

Also note that it would be interesting to see how the transformation would perform on realistic feature models, besides worst-case scenarios.

CHAPTER 6

USING CONTEXT INFORMATION FOR STAGED CONFIGURATION OF FEATURE MODELS

As we have discussed in the previous chapters feature models may become too large easily, which complicates the management of the model. One of the reasons of complexity arises from the constraints imposed by the context on the product family. In this chapter we discuss a proposal [30] by the author of this dissertation that presents a strategy that uses context information for performing staged configurations on feature models. We introduce an organization structure for the context variability model, and use this organization to determine the configurations for specification stages. This approach enables the elimination of context-related variability from the feature model, thus results in a reduction in the complexity of the model and enables to focus on the functional features of the product family.

6.1 Introduction

Each binding on a variation point in a feature model decreases the number of variations and results in a more specialized model with a smaller size. Such bindings decrease the complexity of a feature model but come with a cost: each binding removes a number of possible products from the product family. Selecting the variation points to be bound randomly would eliminate the possibility to build some high quality products, would result in a fully specialized feature model representing a product that does not satisfy the stakeholders' goals, thus can never appear on the market, or even would cause inconsistencies between cross-tree constraints and cause a deadlock. Therefore it is important to implement a strategy that will not eliminate desirable products while selecting the variation points to be bound.

In this chapter we propose a strategy that suggests binding the variations in the context before actually starting the specialization of the feature model. Our strategy is based on identifying the context entities that impose constraints on the feature model of the product family and the relations between these entities, and then using this context information to perform staged configurations on the feature model. We provide guidelines for deciding on the selection order of the context variations to be bound during the staged configurations. By this strategy it becomes possible to remove variability related to the context from the feature model, and decrease the complexity of the model.

6.1.1 Context

Every product is designed to carry out certain tasks. But while performing its main functions the product must be able to exist in harmony with its context. There is no absolute isolation for a product, everything is surrounded by an environment and there are many factors affecting the behavior of the product. For example a computer in a network must work in cooperation with other computers therefore must run some networking protocols. Software to be developed must be able to efficiently use the hardware resources it is running on, or a missile must be capable of flying and hitting its target in different weather conditions.

Context have already played an important role in a number of domains like software product lines, AI, databases, communication, knowledge acquisition, machine learning, navigation and electronic documentation, and vision for a long time. Context permits to define which knowledge should be considered, what are its conditions of activation and limits of validity and when to use it at a given time. Therefore understanding, defining, and using the context is especially important to be able to build a successful product.

Although context is a very important term there is no clear definition for it. Different authors use different definitions to this term for different domains. Some of the suggestions are *"a set of preferences and/or beliefs"*, *"an infinite and only partially known collection of assumptions"*, *"a list of attributes"*, *"possible worlds"*, *"assumptions under which a statement is true or false"*, *"the characteristics of the situation"* [11].

In this work we define context as everything that is not a part of the product family but directly affects the products in the family. Therefore context represents the outside world, the surrounding environment of the product that has direct relations with the products and poses constraints on the features of the products.

6.1.2 Context Variability Model

Separation of context information from the feature model and modeling in a variability model of its own, which the authors call the *Context Variability Model (CVM)*, was introduced by Hartmann *et al.* in [24]. They proposed to model the general classifiers of the context, which stand for a set of requirements or constraints for that context, in a separate model that captures the commonality and the variability of the context. Further, three types of dependencies; namely, *requires*, *excludes* and *sets cardinality*, are defined between the CVM and the feature model. *Requires* and *excludes* relations have the conventional meaning as in the feature models, and *sets cardinality* is used to narrow the cardinality of either a feature or a group.

6.1.3 Staged Configurations

The idea of staged configurations was first proposed by Czarnecki *et al.* in [16]. In this work authors proposed to divide the specification process of product family members into several stages. At each stage some variability in the feature model is eliminated by configuration choices, yielding a more

specialized feature model. At the final stage the transformation process results in a fully specialized feature model of a product.

In a later work [15] the authors extended the notion of staged configurations to support software supply chains. They have also suggested using multi-level configuration, where high level requirements such as those address different market segments or geographical regions are represented by a level-0 feature model.

6.2 Staged Configuration of Context Entities

As mentioned in the previous section, it is very important to correctly identify, understand and define the context. Only by this way it is possible to add the necessary features to the product that shall enable it to operate in a predictable way in the target context. Therefore, we will categorize context entities into four classes:

- Forces of nature
- Laws and standards
- External interfaces
- Resources

Selection of features in a product family may be directly constrained by some context entities coming from all or some of the categories listed above. In the following subsections we describe the properties of the entities in each of the categories, and then we propose an organization structure for the context variability model.

6.2.1 Forces of Nature

Context entities in this category are the entities that are representing information related to the forces of nature, i.e. laws of physics. These entities can directly affect the selection of features to be included in a product.

As an example, consider a product line constructed to build a family of research submarines. If a submarine is intended to operate in deep oceans it will require a strong body structure, possibly constructed by using strengthened special alloys, which can endure very high pressures. But for another submarine designed to operate in relatively shallow waters, e.g. a touristic excursion submarine, a cheaper and relatively weaker body which can employ big windows for sightseeing would suffice.

As another example consider a helicopter product line. If the requirements state the necessity to operate during nights the products would require employment of special hardware to enhance the vision of the pilots, which would require special software to run on these hardware.

6.2.2 Standards and Laws

If a product does not comply with the standards and/or laws of the target country and market then that product cannot be offered to the market. Clearly the standards or laws are not features of the product,

but the products must comply with them. Context entities in this category represent information about the laws and standards.

For example, a car to be used in United Kingdom should have its steering wheel on the right whereas a car to be used in United States should have it on the left. As another example consider the selection of the measurement units to be used in software. If the software is to be used in United States it should use feet and miles whereas software to be used in continental Europe should use meters and kilometers.

6.2.3 External Interfaces

If a system is required to interact with another system the requirements dictated by the other (and maybe already operating) system would affect the bindings of the variations. Context entities in this category represent information about the interaction requirements *(i.e. communication protocols, data structures, implementation algorithms)* with the other systems.

For example, a product that shall record music on a magnetic tape should implement analog recording algorithms whereas a product that shall use CDs should include a component that can make sampling and record digitized data. As another example, consider a mobile phone, a mobile phone that is required to make Bluetooth connections with other Bluetooth enabled devices should employ appropriate hardware and should be able to implement Bluetooth communication protocols.

6.2.4 Resources

Every product must comply with the hardware it is running on, further it should take the advantage of the resources it is to use. Context entities in this category represent information about the resources (hardware and/or software) to be used by the product.

Consider the case of development of software with fancy user interfaces to be run on a hand-held device. The amount of memory in the device may cause changes in the features of the software. As an example a device with 64 MB of memory can show only 2D graphics where a device with 128 MB of memory may be capable of showing both 2D graphics and also 3D graphics. Here the amount of memory the device has is a context entity (not a feature of the product) but directly affects the binding of a possible variation point in the feature model. Continuing the previous example, a component is needed for drawing graphics; one component draws 2D graphics while another component draws 3D graphics.

As another example let us again consider a mobile phone. Existence of a camera on a mobile phone may cause changes in the features of the software to operate on the phone. If there is a camera, a component to manage camera functions should be a part of the product. However, if the phone does not have a camera such kind of a component may be omitted.

6.3 An Organization Structure for the Context Variability Model

Since the CVM is used to model the commonalities and variations in the context it is convenient to represent the variability in the entities belonging to the four context classification categories in this model.

Following Hartmann *et al.* [24], we keep the root classifier context as in the original model, but suggest decomposing the root into four sub-classifiers: “Forces of Nature”, “Standards and Laws”, “External Interfaces”, and “Resources” where each classifier represents a context entity category. The decomposition should be an *and-decomposition* and each sub-classifier must be *optional* as shown in Figure 6.1.

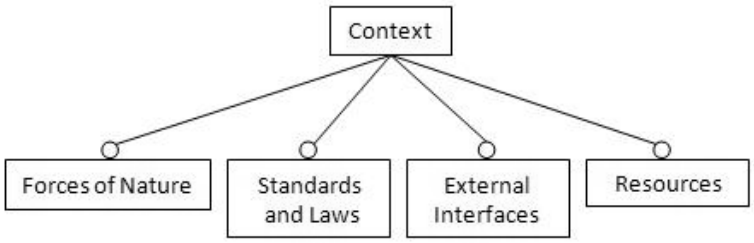

Figure 6.1 Organization of the CVM

Lower levels are organized in the following way: A variability point in context, which is related with a context classification category C and includes variants from the entities from this category C, will be placed under the context classifier representing the context classification category C.

We also allow cross-tree constraints that are *requires* and *excludes* to be used among context classifiers as long as the following principle holds: Assume we have two context classifiers X and Y, where X belongs to the context classification category C1, and Y belongs to the context classification category C2:

- X can impose a constraint on Y; only if the yieldingness of C1 is smaller than the yieldingness of C2 *(see the following subsections for a discussion of yieldingness degrees of the context classification categories)*.

6.4 Yieldingness of the Categories

In this section we propose a strategy based on using context information for determining configuration choices to perform staged configurations for the specification of family members. First we discuss the relation between the context variability and the feature model, and the relations between the classes of context information that was presented in the previous section. Then our approach on determining the configurations is presented, to be applied in a suggested order.

6.4.1 Relation between the Context and the Product Family

There is commonality and variability both in the context and the product family. Both sides may have expectations about the other side, but they do not have the equal power to force expected binding on the variation points of the other model, therefore this is not a relation between the equals.

Before moving further we present an example. Consider the case of a software product line designed to build a family of software that will run on mobile phones. Here the properties of the device, namely mobile phone, form some of the context information. An expectation from the product family may be including digital zooming algorithms in a product that will run on phones with a camera. Here context forces to employ a feature, a digital zooming component, in the product, which is within the abilities of the developers of the software.

Product family may have expectations on the context as well. For example a real-time computer graphics component may expect to use a special graphics chip located in the main board of the mobile phone. But the developers cannot force such a request; they have to simply give up using that component if the phone does not have that special graphics chip.

Generally it is the case that the context *"requires"* or *"excludes"* some features from the feature model, but the feature model may just *"assume"* that a condition holds in the context or not. It is not possible to change the context to fit the product; the product must change itself to fit into the context.

Therefore our approach suggests using the context information and make necessary binding on the feature model to satisfy the constraints imposed by the context as the first step of specialization. This step eliminates the variability caused by the context and enables to focus on the functional features in a smaller feature model. We also propose to divide this step into four stages. Why and how to divide this step is discussed in the following subsection.

6.4.2 Relations between the Categories of the Context Entities

Similar to the relation between the context and the product family, relations between the different categories of context entities are not relations between equals. A context entity belonging to a category may have expectations about some binding on the variants of another context entity that is in another category and vice versa. In order to determine the dominant category that may force constraints on the other categories we will first discuss interactions between the categories.

Yieldingness for allowing alternative variant introduction is an important characteristic of context categories. Degree of the provided yieldingness determines the number of possible adaptation choices to satisfy a constraint. More yieldingness means it is more likely to be able to introduce a variant to satisfy an imposed constraint, and more unyieldingness means it is hard or maybe even impossible to introduce a variant to satisfy a forced constraint.

6.4.3 A Motivating Example

For instance assume a software product line designed to develop control software for research submarines, and assume that we are trying to specify a context where the submarine can operate in the debris of the famous wreck of Titanic. In the context, among many others, there are two variability

points: one belonging to the forces of nature category, named *operation depth* with alternative sub-classifiers *deep*, *medium*, and *shallow*, one belonging to the resources category, named *body structure* with alternative sub-classifiers *weak*, *strong*, and *ultra strong*. And assume that the body structure imposes a constraint on the alternative engine types, where each engine type requires different software components as shown in Figure 6.2. The picture here means the specified context will activate different constraints on the features of our feature model, thus directly affects our product family.

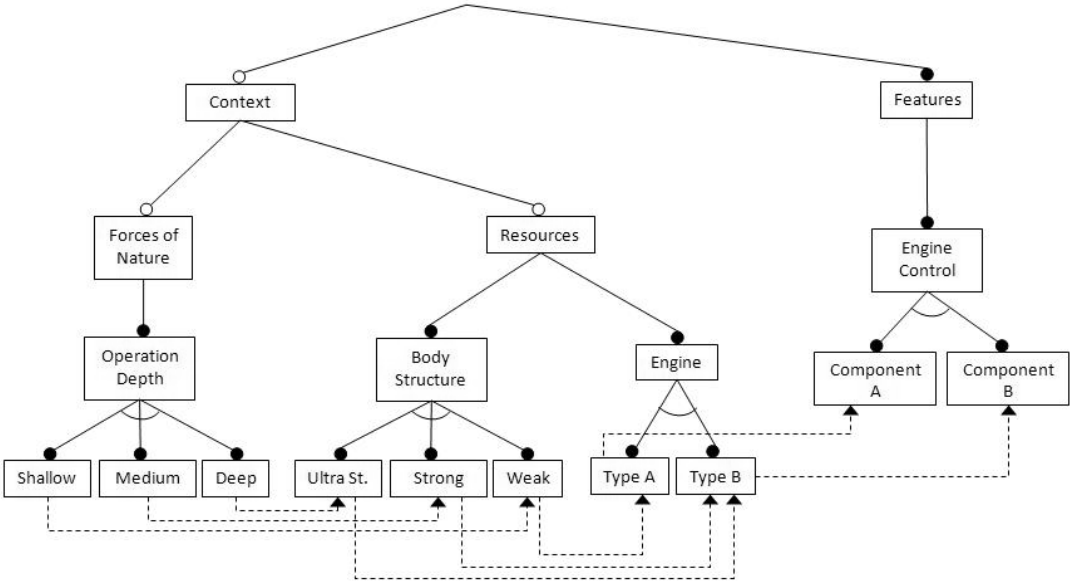

Figure 6.2 A sample model for the motivating example

If we bind the *body structure* first and chose features from our feature model in accordance with the constraints supplied by the context classifier *weak* the product we will have will be totally useless. A submarine with weak body structure will not be able to endure high pressures in deep ocean surface, and it is impossible to find a way to alter the laws of physics and change the water pressure, therefore the specialized context we have will not be realistic and consistent. And unrealistic and inconsistent contexts will lead to useless products.

On the other hand assume that we have chosen to bind a variant of the *operation depth* context variability point first, and have chosen the context classifier *deep*. This classifier would require a *body structure* that is capable of enduring high pressures (e.g. water pressure in the debris of Titanic is around 5500 lbs/inch^2). Now even if it is the case that no variants of the context variability point *body structure* is capable of enduring such pressures we may introduce a new body structure variant (e.g. body structure of a submarine that had actually dived to the grave of Titanic) and add the constraints imposed by this new body structure on the context entities (e.g. type of engine to be selected) and the feature model (e.g. features necessary to operate the newly introduced engine), and get a realistic context which would lead to a realistic and usable product.

Motivated by the example given above, we have identified some characteristics of the yieldingness levels corresponding to the categories of class entities. These characteristics are presented in the following subsections.

6.4.4 Yieldingness of the Entities in the Forces of Nature Category

Considering the context entities representing information about forces of nature, the yieldingness to introduce new variants or to alter the properties of the entities is absolutely equal to zero. Since it is beyond mankind's powers to alter the laws of physics no exception or alteration can be made on these entities.

For example a mobile phone that must be capable of directly communicating with a satellite has to employ an antenna that is powerful enough to send radio signals to penetrate through different layers of the atmosphere to reach the satellite. It is not possible to employ a weak antenna and change the properties of the atmosphere (to thin the atmosphere, or guarantee to prevent rough weather conditions that will disrupt transmission) or introduce a suitable atmosphere variant. Simply you have to equip your product with an antenna that can generate enough power to send signals to the space.

6.4.5 Yieldingness of the Entities in the Standards and Laws Category

Although laws and standards made by man are not as unyielding as the laws of physics, laws and standards may evolve in time, they cannot be altered in practice. Therefore the yieldingness to introduce new variants or to alter the properties of the entities in this category is very close to zero.

For instance it is not possible to market a car that is not equipped with safety belts in United States where the safety regulations strictly forbid such a situation. You cannot introduce your own regulations as a new variant to create a consistent and realistic context.

6.4.6 Yieldingness of the Entities in the External Interfaces Category

The third class of context entities, external interfaces, is not as unyielding as the former categories. Sometimes it may be possible to introduce new variants, or to alter the properties of the existing variants to create a consistent and realistic context instance.

As an example, consider the case that the software product to run on a hand held device must connect to the internet to upload and download data. Assume that the external interface variability connection has only two context classifiers; *Wi-fi* and *Bluetooth*. But the device will operate in a military zone and the regulations strictly forbid all kinds of transmission using air medium. We can simply add a new variant *cable* to fit in the regulations and create a realistic context.

6.4.7 Yieldingness of the Entities in the Resources Category

The fourth class, resources, is the most yielding category of context entities among the all four categories. This is not surprising since the number of possible variants that can be introduced in this category is much bigger than the number of possible variants that can be introduced in other categories. And this is not surprising since resources are manmade entities, and the producers of

resources mostly seek to make profit by the production of these entities. More adaptable resources can be used in a wider spectrum, which enables to enter more markets, which will eventually return more profit. Therefore producers of the resources aim to make their products more adaptive. For example if the forces of nature forces to use a strong antenna in order to get a range wide enough, in most cases it will be possible to introduce a competitor's product as a new variant to fulfill the requirement.

6.5 Specialization Stages

Constraint solving can be seen as a combination of two processes: constraint propagation and labeling. Labeling process is basically choosing a variable and assigning to the variable a value from its domain. There are many labeling strategies used by the constraint solvers, and the *first fail* labeling method is one of them.

The *first fail* labeling is based on a principle proposing the following strategy: *to succeed it is best to try the case you are most likely to fail first*. To implement this strategy the principle recommends choosing the variables with the smallest domains *(most constrained variables)* before the variables with larger domains. We have adopted the same strategy to order the stages of the specialization process.

Variability points in the context variability model correspond to the variables in a constraint problem, and the set of variants for a variation point correspond to the domain of the variable. If a context entity category has a high degree of unyieldingness it means the variability points in this category has a small number of variants that can be used, which we may be interpreted as: the variables in this category have smaller domains. On the other hand, a high degree of yieldingness means bigger domains.

We have the following observations:

- Yieldingness of forces of nature <
  Yieldingness of standards and laws <
  Yieldingness of external interfaces <
  Yieldingness of resources

Therefore it is best to start eliminating variability caused by forces of nature and move to resources. Thus we have the following stages:

- *Stage 1*: Eliminate variability in the sub-tree of the context classifier "Forces of Nature"
- *Stage 2*: Eliminate variability in the sub-tree of the context classifier "Standards and Laws"
- *Stage 3*: Eliminate variability in the sub-tree of the context classifier "External Interfaces"
- *Stage 4*: Eliminate variability in the sub-tree of the context classifier "Resources"

6.6 An Illustrative Example: Research Submarine

As a motivating example we have chosen to consider a software product line designed to build a family of software for research submarines. The context variability model is given in Figure 6.3, and

the feature model is given in Figure 6.4. Note that in reality these two models would be much more complicated; we have simplified the models for illustrative purposes.

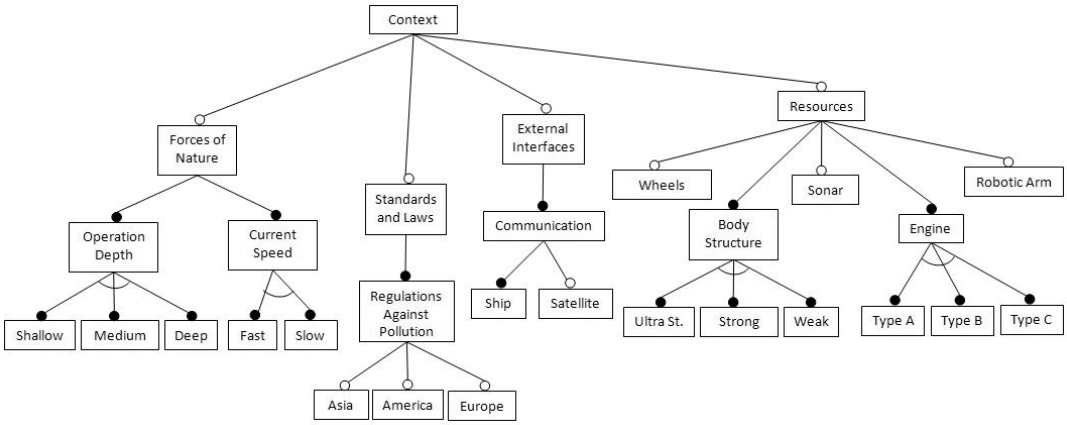

Figure 6.3 The CVM for the SPL of research submarines

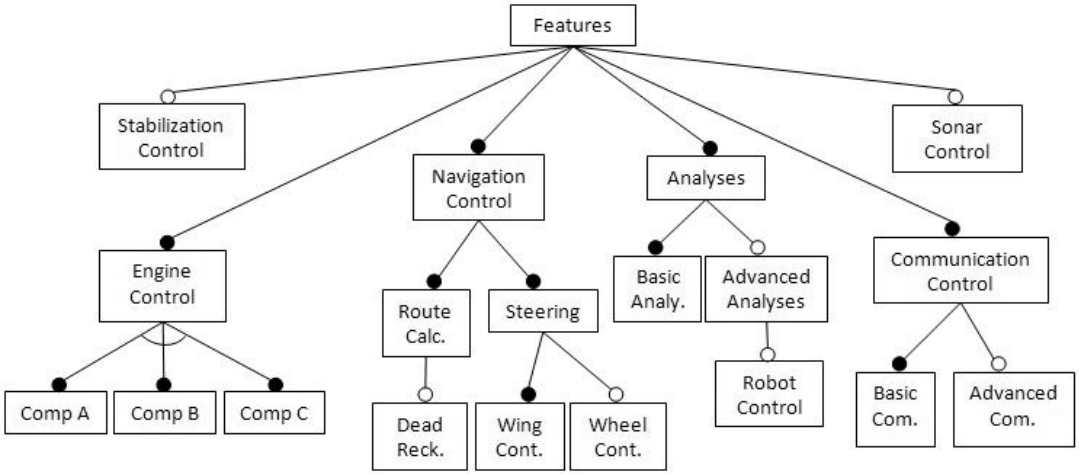

Figure 6.4 The feature model for the SPL of research submarines

There are many cross-tree constraints between the categories of the context classifications and between the context variability model and the feature model. Due to presentation concerns we prefer to list the cross-tree constraints in the following tabular forms instead of showing the constraints graphically.

Table 6.1 Cross-tree constraints among context classifiers

Context Classifier	Constraint	Context Classifier
Shallow	requires	*Weak*
Medium	requires	*Strong*
Deep	requires	*Ultra Strong*
Asia	excludes	*Type A*
America	excludes	*Type B*
Europe	excludes	*Type B*
Ultra Strong	excludes	*Type B*
Ultra Strong	excludes	*Type C*
Strong	excludes	*Type C*

Table 6.2 Cross-tree constraints between the context variability model and the feature model

Context Classifier	Constraint	Feature
Fast	requires	*Stabilization Control*
Satellite	requires	*Advanced Com.*
Wheels	requires	*Wheel Cont.*
Robotic Arm	requires	*Robot Control*
Type A	requires	*Comp A*
Type B	requires	*Comp B*
Type C	requires	*Comp C*
Sonar	requires	*Sonar Control*

As our proposal suggests the first stage in specializing is the elimination of context variability under the context classifier *Forces of Nature*. There are two variability points in this sub-tree, *Operation Depth* and *Current Speed*. Assume that we will derive a product for a submarine that can operate in medium depths where the current will be strong and fast. We chose *Medium* for the former variability, and *Fast* for the latter one. Once the mentioned bindings are done we have the transformed models given in Figure 6.5, and Figure 6.6:

- *Strong* is selected automatically due to the constraint *"Medium requires Strong"*.
- *Type C* is removed from the model due to the constraint *"Strong excludes Type C"*.
- *Stabilization Control* becomes a mandatory feature due to the constraint *"Fast requires Stabilization Control"*.

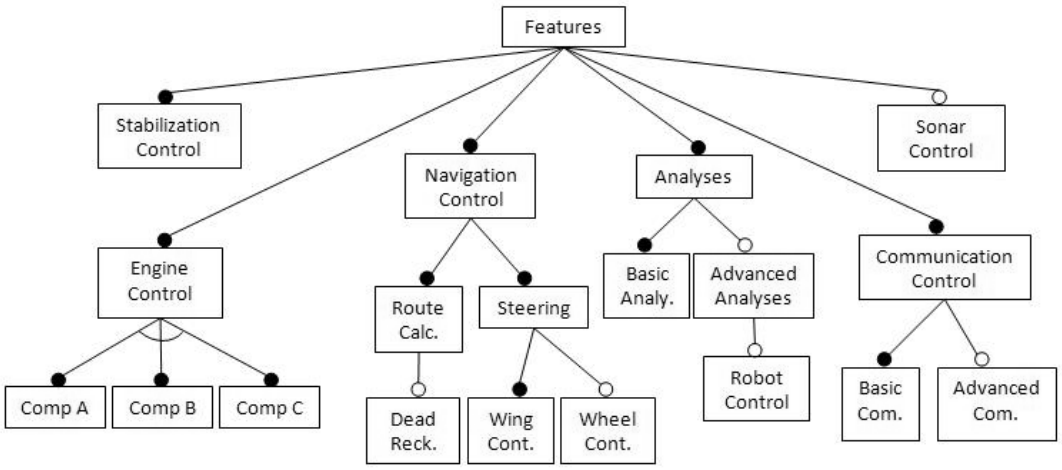

Figure 6.5 The CVM after Stage 1

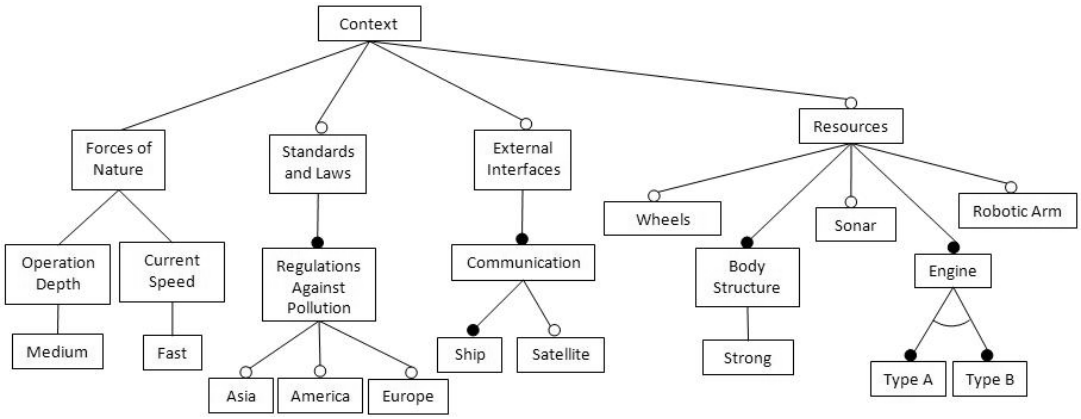

Figure 6.6 The feature model after Stage 1

Next stage is eliminating the variability under the context classifier *Standards and Laws*. Here we assume that our submarine will have to comply only with the anti pollution regulations published by Europe, so we chose only the classifier *Europe*. Once the mentioned bindings are done we have a transformed context variability model given in Figure 6.7:

- *Type B* is removed from the model due to the constraint *"Europe excludes Type B"*.

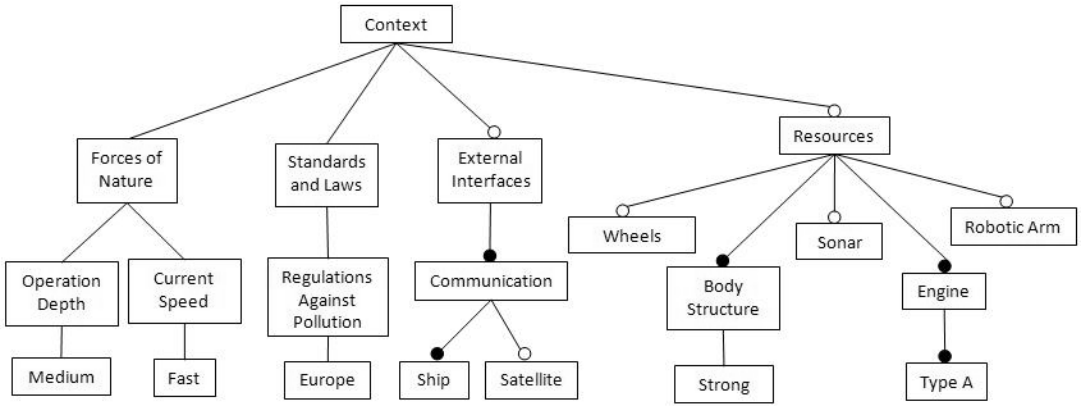

Figure 6.7 The CVM after Stage 2

Third stage is eliminating the variability under the context classifier *External Interfaces*. We assume that a dedicated guiding ship will be following the submarine all the time, so there is no need for the submarine to make communication over the satellite. Once the mentioned bindings are done we have a transformed context variability model given in Figure 6.8.

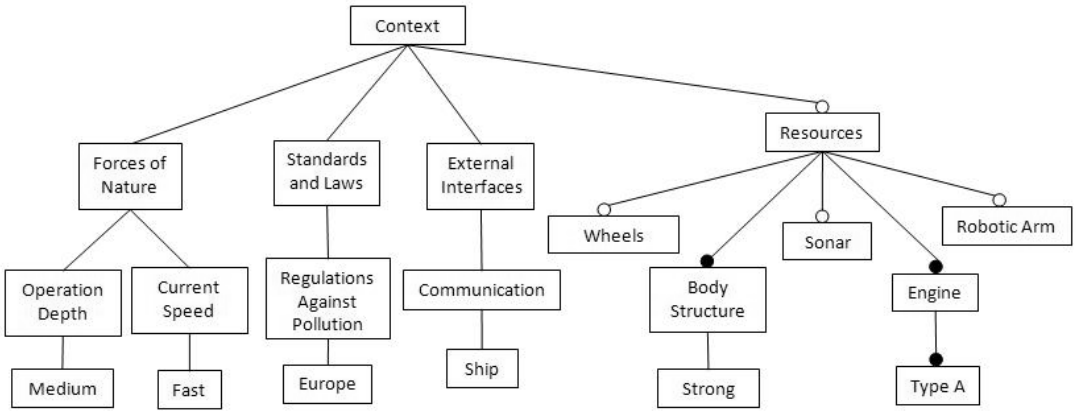

Figure 6.8 The CVM after Stage 3

Final stage is eliminating the variability under the context classifier *Resources*. We assume that our submarine will have *Wheels*, which enables to move on the bottom, has an engine *Type A*, and employs a *Robotic Arm* used to collect samples. Fully specialized context variability model, which represents a realistic and consistent context instance, is given in Figure 6.9, and the specialized feature model is given in Figure 6.10:

- *Comp A* becomes the only choice in the alternative decomposition due to the constraint *"Type A requires Comp A"*.
- *Robot Control* and *Advanced Analyses* become mandatory due to the constraint *"Robotic Arm requires Robot Control"*.

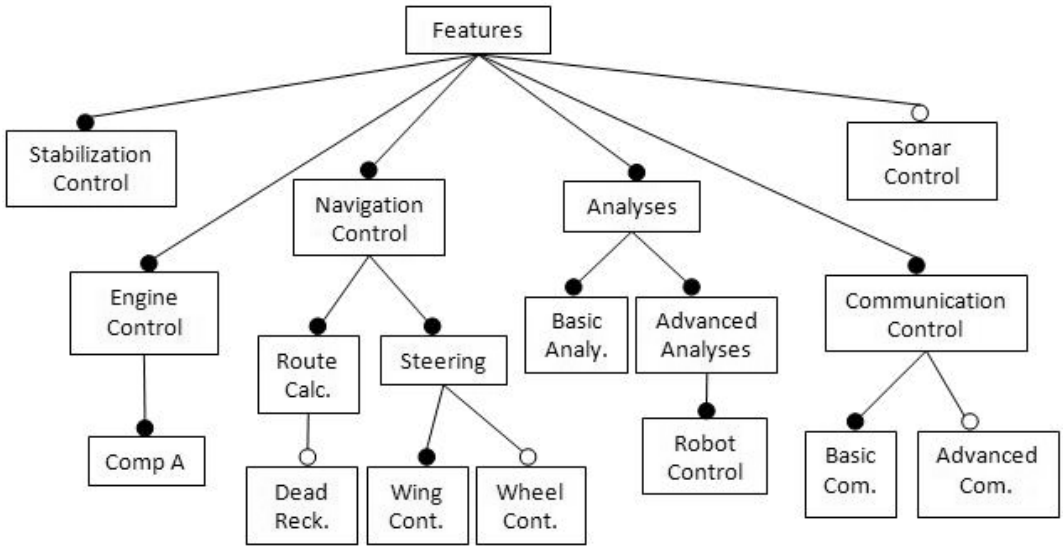

Figure 6.9 The feature model after Stage 4

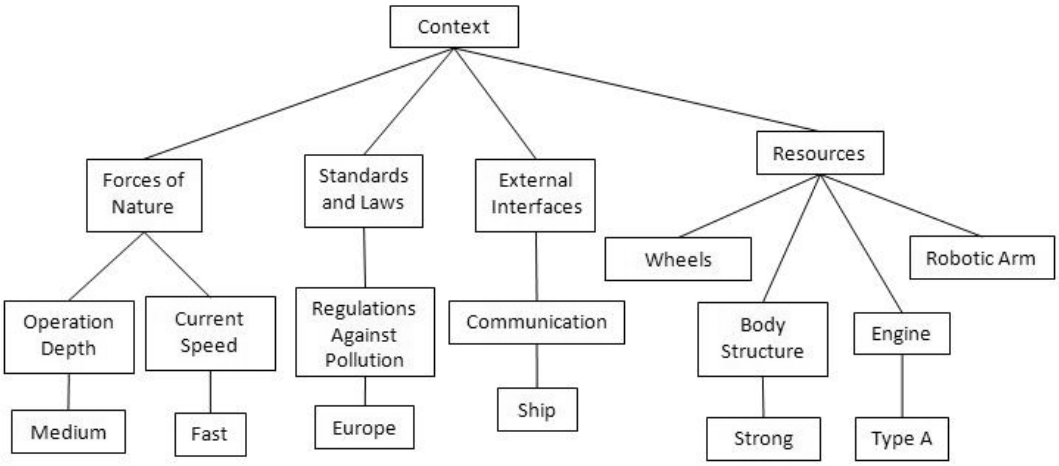

Figure 6.10 The context instance after Stage 4

6.7 Discussions

In this chapter we proposed to use context classification information to build a context variability model and perform staged configurations. A feature model with a high degree of variability would generate a large number of product instances, but not all of the products will be meeting the stakeholders' goals. The strategy presented provides some guidelines to create realistic and consistent context instances that eliminate some of the useless product instances and decreases complexity of the feature model.

Another classification on context entities can be given with respect to the dynamism they have. Binding time of a feature is often an important consideration. Feature binding time choices available to a product engineer typically include design time, compilation time, deployment time, and run time. This choice should consider the dynamism of the relevant context. A relatively stable context element, such as country of operation for a car, should be considered for design time bound product features, whereas a more volatile context element, such as weather situation, should be considered for dynamically bound product features. In summary, the more dynamic the context element is the more dynamic the binding time of the constrained features should be. However, this topic needs further research.

CHAPTER 7

DISCUSSIONS AND CONCLUSIONS

In this dissertation we have laid the groundwork of automated analysis of extended feature models with constraint programming. First we have presented a mapping from extended feature models, which may include complex feature-feature, feature-attribute and attribute-attribute cross-tree relationships as well as global constraints, to constraint logic programming over finite domains. Then, we have discussed the effects of including complex feature attribute relationships on various analysis operations defined on the feature models. Then, we have proposed a transformation from extended feature models to basic/cardinality-based feature models that may be applied under certain circumstances and enables using SAT or BDD solvers in automated analysis of extended feature models. Finally, we have discussed the role of the context information in feature modeling, and propose to use context information in staged configuration of feature-models.

The mapping presented from extended feature models to CLP(FD) is capable of effectively handling every decomposition and complex cross-tree relationship that may be defined on extended feature models, which was not possible before. The only restriction we assume is attributes' domains be finite. Moreover, basic and cardinality-based feature models are treated uniformly under the scheme presented, which means the mapping can be used to translate every feature model reported in the literature so far to CLP effectively. Thus, it enables using CLP as a common reasoning base for any feature model as long as the domains of the attributes in the model are finite, which would be a standard for the automated analysis of feature models.

Besides its power, the mapping also offers many benefits. First of all, it enables using an off-the-shelf CLP(FD) solver for the analysis tasks. It also enables benefiting from the global constraints in the automated analysis. It translates a feature model to CLP, which is regarded as a mature field of computer science, and enables benefiting from the advantages provided by the CLP systems such as production of highly readable and maintainable code due to the highly declarative form of programming provided by the CLP systems, make use of discrete optimization and verification algorithms that have proven to be very effective, wide variety of tool support, and so on.

Its well-defined structure also increases the value of the mapping, as it is possible to incorporate it into existing feature modeling and analysis tools or develop new tools. Such a tool would employ the following components:

- A *file i/o component* to read in the feature models represented in an input language. Such a language must be powerful enough to express extended feature models that may include complex feature-feature, feature-attribute and attribute-attribute cross-tree relationships as well as global constraints. This input language may be designed from scratch or existing feature modeling languages may be extended.
- A *preprocessor* to carry out tasks to be performed before the mapping initiates. For instance, introducing implicit attributes, or searching for possible common neutral values for attributes involved in global constraints, possibly with user guidance, are examples of such tasks.
- A *mapping component* to perform the translation to targeted CLP(FD) system code.
- An *analysis component* that will utilize an off-the shelf constraint solver for the analysis operations.
- A *postprocessor* to output and comment on the results produced by the solver.

As the inclusion of attributes in complex cross-tree relationships create a new type of variability, *variability in terms of attributes*, we have reevaluated the analysis operations on feature models in the light of this new variability introduced. We have shown that features, and as a result products can be classified with respect to the variability they include.

Analysis operations are performed on feature models to reveal certain characteristics that will help the designers to understand, manage, and update the models. Designers seek the answers to the questions in their mind by making use of the analysis operations. However, if one asks the wrong or incomplete questions will get wrong answers. Thus, we have revised the definitions of some of the analysis operations, and reformulated some other, so that correct answers can be provided to the correct, incomplete questions.

As every invention, the invention of a new variability type brought into surface some new questions. Thus, we have proposed some new analysis operations to seek the answers to some of these questions. Note that there may be many other analysis operations that will originate due to the effects of this new variability type. This is an open research area now.

The classification of products into separate categories led to the classification of lists of products. We have shown that these lists exhibit different behaviors but they are not completely apart, there are relations between these newly introduced lists of products. We believe that it is worthwhile to investigate the properties of these lists and the relations among them in more detail.

Especially one product category, the free-choice products, would gather attention from other computer science fields as well. Recall that C-products include some unresolved variability in terms of attributes; however one can make choices free from the constraints imposed by the feature model to produce F-products. Thus C-products may also be considered as a compact representation for a number of products that can be derived. This idea may be useful in many areas. For instance, C-products would be good candidates for deployment to context-aware software that typically runs on platforms offering low resources (e.g. hand held devices). As C-products are free from the constraints of the feature model one may delay binding of the attributes as long as s/he wants. And dynamic binding would be performed effectively as there would be no necessity to check whether the resulting

product conforms to the requirements or not, which would save computational power, and as the C-products are free from constraints there would be no necessity to deploy extra information about the feature model or requirements, which would decrease memory consumption.

Recall that the only restriction we assume about the extended feature model to be translated is that domains of the attributes be finite. For the situations that this restriction would not hold we have proposed a simplification over extended feature models. Under some other certain restrictions we have shown that it is possible to transform an extended feature model into an equivalent basic or cardinality-based feature model. Thus, this transformation would be used in the cases that the mapping proposed cannot be applied.

As this simplification transforms an extended feature model into a basic or cardinality-based feature model, it renders using the automated analysis approaches reported in the literature for such feature models possible. It has been reported that using a multi-solver approach for automated reasoning on feature models would provide some benefits. Thus, this simplification opens the door for multi-solver approaches for extended feature models under certain restrictions.

This simplification also reveals an important characteristic about the extended feature models. As we have discussed the space complexity of the transformation is exponential in the worst case. This implies that extended feature models offer a compact representation for some product families, which would otherwise be very large and complicated to manage.

Note that the transformation algorithm has a well-defined structure as well. Thus, it may be possible to incorporate it into existing feature modeling and analysis tools or develop new tools, as it was the case for the mapping. Such a tool would employ similar components as the envisioned tool for the mapping.

Feature models may become too large easily, which complicates the management of the model. There are many causes responsible from the complication of the feature models, and context is one of them. We have discussed the role and effects of context variability in feature modeling. We have proposed a restructuring for the models of context variability that is built on previously reports proposing to separate feature models and context models while keeping the relationships between. We have investigated the entities in the context, classified context entities into categories, and discussed the relations between these categories.

The same study also discusses the relations between context and staged configurations using feature models. As it may be desirable to remove variability from a feature model in stages, rather than to perform at once, we have discussed how context information can be used during these stages. We have also provided guidelines for the ordering of stages.

This study aims to lay the groundwork for the automated analysis of basic, cardinality based, and extended feature models, which may include complex feature-feature, feature-attribute and attribute-attribute cross-tree relationships as well as global constraints. We believe that the proposals presented in this dissertation will fill some of the missing parts in the literature, and leverage the automated reasoning efforts on feature models.

REFERENCES

[1] L. Abo, F. Kleinermann, and O. De Troyer. “Applying semantic web technology to feature modeling”, SAC, pages 1252–1256, 2009.

[2] D. Batory. “Feature models, grammars, and propositional formulas”, Software Product Lines Conference, LNCS 3714, pages 7–20, 2005.

[3] N. Beldiceanu, M. Carlsson, and J.X. Rampon. “Global Constraint Catalog”, SICS Technical Report T2005:08, ISRN: SICS-T-2005/08-SE, 2005.

[4] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. “Automated reasoning on feature models”, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005, volume 3520 of Lecture Notes in Computer Sciences, pages 491–503. Springer–Verlag, 2005.

[5] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. “Coping with automatic reasoning on software product lines”, Proceedings of the 2nd Groningen Workshop on Software Variability Management, November 2004.

[6] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. “Using constraint programming to reason on feature models”, The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE 2005, pages 677–682, 2005.

[7] D. Benavides, S. Segura, A. Ruiz-Cortés. “Automated analysis of feature models 20 years later: A literature review”, Information Systems, Volume 35, Issue 6, pages 615-636, 2010.

[8] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. “A first step towards a framework for the automated analysis of feature models”, Managing Variability for Software Product Lines: Working With Variability Mechanisms, 2006.

[9] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. “FAMA: Tooling a framework for the automated analysis of feature models” Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS), pages 129–134, 2007.

[10] D. Le Berre. “SAT4J solver”, www.sat4j.org. (published on line – 30.06.2010).

[11] P. Brézillon, “Context in human-machine problem solving: A survey.”, Technical Report 96/29, LAFORIA, 1996.

[12] M. Carlsson, G. Ottosson, and B. Carlson. "An Open-Ended Finite Domain Constraint Solver", Proc. Programming Languages: Implementations, Logics, and Programs, 1997.

[13] CHOCO Developers. “CHOCO solver”, http://www.emn.fr/z-info/choco-solver/ (published on line – 30.06.2010).

[14] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker. “Generative programming for embedded software: An industrial experience report”, Proceedings of the ACM SIGPLAN/ SIGSOFT Conference on Generative Programming and Component Engineering (GPCE’02), Pittsburgh, LNCS 2487, Springer-Verlag (2002) 156–172, October 6–8, 2002.

[15] K. Czarnecki, S. Helsen, and U. W. Eisenecker, “Staged Configuration through specialization and Multi-Level Configuration of Feature Models”, Software Process Improvement and Practice, 10(2), 2005.

[16] K. Czarnecki, S. Helsen, and U. Eisenecker. “Staged Configurations Using Feature Models”, Software Product Lines: Third International Conference, SPLC 2004, Proceedings, Vol. 3154 of Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, Germany, pp. 266–283, Boston, MA, USA, August 30-September 2, 2004.

[17] K. Czarnecki and C.H.P. Kim. “Cardinality-based feature modeling and constraints: a progress report”, International Workshop on Software Factories, San Diego, California, Oct 2005.

[18] M. Dean and G. Schreiber. “OWL web ontology language reference”, W3C recommendation, W3C, February 2004.

[19] A. van Deursen and P. Klint. “Domain–specific language design requires feature description”, Journal of Computing and Information Technology, 10(1):1–17, 2002.

[20] S. Fan and N. Zhang. “Feature model based on description logics”, Knowledge-Based Intelligent Information and Engineering Systems, 2006.

[21] A. Fernandez and P. M. Hill. “A comparative study of eight constraint programming languages over the Boolean and finite domains”, Journal of Constraints, 5: 275– 301, 2000.

[22] GNU Prolog developers. “GNU prolog”, www.gprolog.org/ (published on line – 30.06.2010).

[23] M. Griss, J. Favaro, and M. dlAlessandro. “Integrating Feature Modeling with the RSEB”, Proceedings of the Fifth International Conference on Software Reuse, pages 76-85, Vancouver, BC, Canada, June 1998.

[24] H. Hartmann and T. Trew. “Using feature diagrams with context variability to model multiple product lines for software supply chains”, 12th International Software Product Line Conference, IEEE Computer Society, 2008.

[25] W. van Hoeve and I. Katriel. “Global Constraints”, Book Chapter in Handbook of Constraint Programming, Elsevier, 2006.

[26] ILOG. “OPL studio”, www.ilog.com/products/oplstudio/ (published on line – 30.06.2010).

[27] JaCoP development team. “JaCoP solver”, http://jacop.osolpro.com/ (published on line – 30.06.2010).

[28] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. “Feature-Oriented Domain Analyses (FODA) Feasibility Study”, Technical Report CMU/SEI-90-TR-21, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, 1990.

[29] K. Kang, S. Kim, J. Lee, and K. Kim. “FORM: A feature-oriented reuse method with domain-specific reference architectures”, Annals of Software Engineering, volume 5, pp 143-168, 1998.

[30] A. S. Karataş, A. Doğru, H. Oğuztüzün, and M. Tolun. “Using Context Information for Staged Configuration of Feature Models”, Proceedings of the 13th International Transdisciplinary Conference-Workshop on Integrated Design & Process Science, SDPS 2010, Dallas USA, 2010.

[31] A. S. Karataş, H. Oğuztüzün, and A. Doğru. “Global Constraints on Feature Models”, To be presented in the 16th International Conference on Principles and Practices of Constraint Programming, CP 2010, Scotland, 2010.

[32] A. S. Karataş, H. Oğuztüzün, and A. Doğru. "Mapping Extended Feature Models to Constraint Logic Programming over Finite Domains", To be presented in the 14th International Software Product Line Conference, SPLC 2010, South Korea, 2010.

[33] A. S. Karataş, H. Oğuztüzün, and A. Doğru. "Analysis Operations on Extended Feature Models", Submitted for publication (Under review).

[34] P. Klint. "A meta-environment for generating programming environments", ACM Trans. Softw. Eng. Methodol., 2(2):176–201, April 1993.

[35] M. Mannion. "Using first-order logic for product line model validation", Proceedings of the Second Software Product Line Conference (SPLC'02), LNCS 2379, pages 176– 187, San Diego, CA, 2002. Springer.

[36] M. Mannion and J. Camara. "Theorem proving for product line model verification", Software Product-Family Engineering (PFE), volume 3014 of Lecture Notes in Computer Science, pages 211–224. Springer Berlin / Heidelberg, 2003.

[37] Racer Systems GmbH Co. KG. "RACER", www.racer-systems.com (published on line – 30.06.2010).

[38] M. Riebisch, K. Bollert, D. Streitferdt, and I. Philippow. "Extending Feature Diagrams With UML Multiplicities", 6th Conference on Integrated Design & Process Technology (IDPT 2002), Pasadena, California, USA, 2002.

[39] P. Schobbens, J.C. Trigaux P. Heymans, and Y. Bontemps. "Generic semantics of feature diagrams", Computer Networks, 51(2):456–479, Feb 2007.

[40] S. Segura. "Automated analysis of feature models using atomic sets", First Workshop on Analyses of Software Product Lines (ASPL 2008), SPLC'08, pages 201–207, Limerick, Ireland, September 2008.

[41] SICStus Prolog developers. "SICStus prolog", http://www.sics.se/isl/sicstuswww/site/index.html (published on line – 30.06.2010).

[42] M. Simos *et al.* "Software Technology for Adaptable Reliable Systems (STARS) Organization Domain Modeling (ODM) Guidebook Version 2.0", STARS-VC-A025/001/00, Manassas, VA, Lockheed Martin Tactical Defense Systems, 1996.

[43] J. M. Spivey. "Introducing Z: a Specification Language and its Formal Semantics", Cambridge University Press, 1988.

[44] J. Sun, H. Zhang, Y.F. Li, and H. Wang. "Formal semantics and verification for feature modeling", Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 2005.

[45] Various developers. "Alloy analyzer", http://alloy.mit.edu/ (published on line – 30.06.2010).

[46] Various developers. "SMV system", www.cs.cmu.edu/~modelcheck (published on line – 30.06.2010).

[47] H. Wang, Y. Li, J. Sun, H. Zhang, and J. Pan. "A semantic web approach to feature modeling and verification", Workshop on Semantic Web Enabled Software Engineering (SWESE'05), November 2005.

[48] J. Whaley. "JavaBDD", http://javabdd.sourceforge.net/ (published on line – 30.06.2010).

APPENDIX A

SAMPLE FEATURE MODEL *FM 1*

The first sample extended feature model we will use is a sample feature model for a mobile phone, given in Figure A.1.

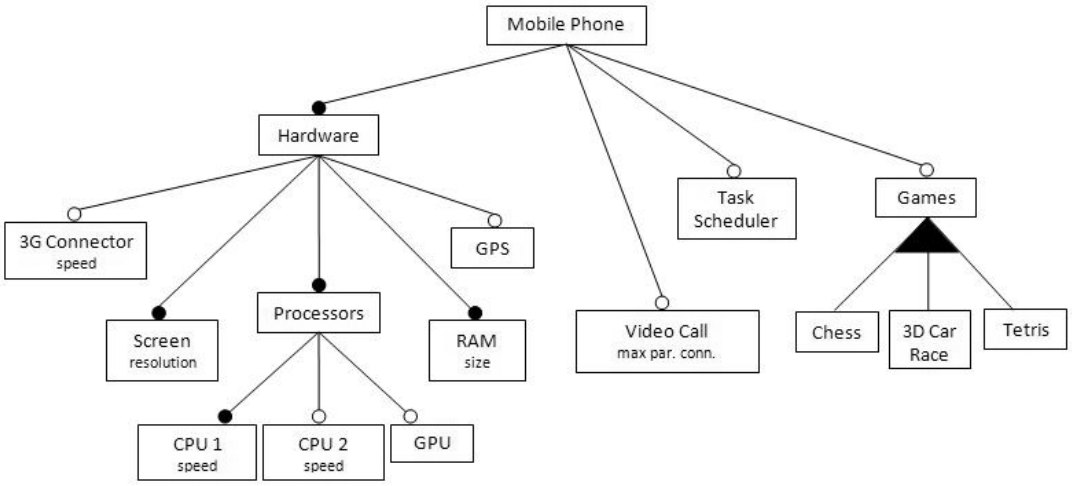

Figure A.1 Sample feature model *FM 1*

This feature model, *FM 1*, has 16 Features *(one concept feature, 5 mandatory features, 7 optional features, and 3 or features)*, and 5 complex cross-tree constraints. The cross-tree constraints are as follows:

- When there are two CPU's on board, Task Scheduler must be a part of the product (*CPU 1* and *CPU 2* implies *Task Scheduler*)
- If *Video Call.mpc* ≥ 4 then *Video Call* requires *Screen.resolution* ≥ 320x640 and *3G Connector.speed* ≥ 6
- *3D Car Race* requires (*GPU* and *RAM.size* ≥ 256) or (*RAM.size* ≥ 512)
- If *Screen.resolution* < 320x640 then *Screen* excludes *GPS*
- *Task Scheduler* requires *CPU 1.speed* ≥ *CPU 2.speed*

We assume that the domains of the attributes are as follows:

- *3G Connector.speed* ∈ {2, 4, 6, 14}

- *Screen.resolution* ∈ {128x160, 240x320, 320x640, 360x640}
- *RAM.size* ∈ {128, 256, 512, 1024}
- *Video Call.mpc* ∈ {2, 4, 8}
- *CPU 1.speed* ∈ {416, 520, 800}
- *CPU 2.speed* ∈ {416, 520, 800}

As the domain of *Screen.resolution* does not consist of integers, we introduce the following conversion before we start: {128x160 → 1, 240x320 → 2, 320x640 → 3, 360x640 → 4}, hence the domain of the attribute *Screen.resolution* becomes {1, 2, 3, 4}.

APPENDIX B

SAMPLE FEATURE MODEL *FM 2*

The second sample extended feature model we will use is a sample feature model for a computer family, given in Figure B-1. This feature model, *FM 2*, has 52 Features *(one concept feature, 28 mandatory features, 12 optional features, 8 alternative features, and 3 or features)*, 5 global constraints, and 35 complex cross-tree constraints.

The global constraints are as follows:

- (GC1) Each PCI Card that is a part of the product must be installed on a different PCI slot.
- (GC2) Each data collector, 1 through 4, must be assigned to a data communication channel, 1 through 3. Each data communication channel has a designated capacity and these capacities must not be exceeded.
- (GC3) Total power consumption of the hardware parts cannot exceed the capacity of the power supply.
- (GC4) Total cost of a product cannot exceed a designated budget.
- (GC5) Memory size must be greater than the total memory consumption of the system software plus memory consumption of the application with the highest memory requirement among the applications chosen to be a part of the product.

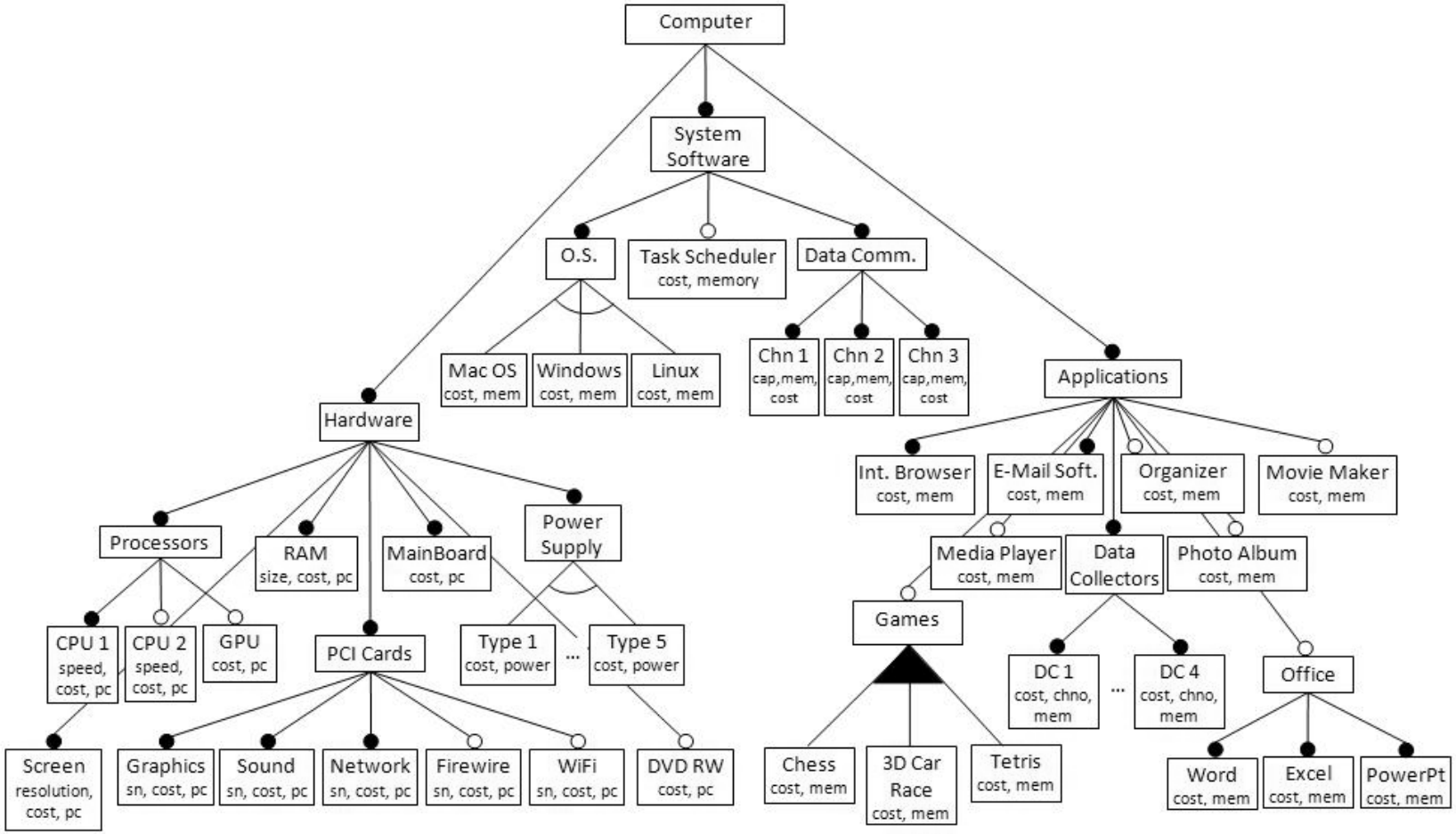

Figure B.1 Sample feature model *FM 2*

The complex cross-tree constraints are as follows:

- When there are two CPU's on board, Task Scheduler must be a part of the product (*CPU 1* and *CPU 2* implies *Task Scheduler*)
- *Task Scheduler* requires *CPU 1.speed* ≥ *CPU 2.speed*
- *3D Car Race* requires ((*GPU* and *RAM.size* ≥ 1024) or (*RAM.size* ≥ 2048)) and *DVD* and *Screen.resolution* ≥ 1024x768 and *CPU 1.speed* ≥ 1800
- If *Screen.resolution* < 1200x800 then *Screen* excludes *Movie Maker*
- If *Screen.resolution* < 1024x768 then *Screen* excludes *Games*
- *Movie Maker* requires *FireWire*
- *Power Supply Type 1* excludes *DVD*
- *Power Supply Type 2* excludes *DVD*
- *Power Supply Type 1* excludes *FireWire*
- *Power Supply Type 2* excludes *FireWire*
- *Power Supply Type 3* excludes *FireWire*
- If *CPU 1.speed* = 1600 Then *CPU 1* excludes *Games*
- *Mac OS* requires *Memory.size* ≥ 2048
- *Windows* requires *Memory.size* ≥ 1024
- *Linux* excludes *3D Car Race*
- *Organizer* requires *Windows*
- *FireWire* requires *CPU 2*
- *Chess* requires *Linux*
- *Movie Maker* requires *GPU*
- *Organizer* excludes *Photo Album*
- *Task Scheduler* excludes *Mac OS*
- *MainBoard* excludes *Power Supply Type 4*
- *Games* requires *DVD*
- *Windows* requires *Media Player*
- *Office* requires *Windows* or *Linux*
- *MainBoard* requires *DVD*
- *Graphics Card* must be in one of the first two PCI slots
- *Sound Card* cannot be installed before *Graphics Card*
- *FireWire Card* cannot be installed after *Network Card*
- *Network Card* cannot be installed in PCI slot *4*
- *Data Collector 1* cannot be assigned to *Data Channel 1*
- *Data Collector 2* cannot be assigned to *Data Channel 2*
- *Data Collectors 3* and *4* has to be assigned to the same *Data Channel*
- *Data Collectors 1* and *2* cannot be assigned to the same *Data Channel*
- If there is at least one *Data Collector* then *Data Channel 1* cannot be unused

We assume that the domains of the attributes are as follows:

- *CPU 1.speed* ∈ {1600, 1800, 2000}
- *CPU 2.speed* ∈ {1600, 1800, 2000}
- *RAM.size* ∈ {512, 1024, 2048, 4096}
- *Screen.resolution* ∈ {640x480, 800x600, 1024x768, 1200x800}
- *Type 1.power* ∈ {475}
- *Type 2.power* ∈ {400}
- *Type 3.power* ∈ {520}
- *Type 4.power* ∈ {525}
- *Type 5.power* ∈ {550}
- *Graphics.sn* ∈ {1, 2, 3, 4}
- *Sound.sn* ∈ {1, 2, 3, 4}
- *Network.sn* ∈ {1, 2, 3, 4}
- *FireWire.sn* ∈ {1, 2, 3, 4}
- *WiFi.sn* ∈ {1, 2, 3, 4}
- *DC 1.ch* ∈ {1, 2, 3}
- *DC 2.ch* ∈ {1, 2, 3}
- *DC 3.ch* ∈ {1, 2, 3}
- *DC 4.ch* ∈ {1, 2, 3}
- *Chn 1.cap* ∈ {0, 1}
- *Chn 2.cap* ∈ {0, 1, 2}
- *Chn 3.cap* ∈ {0, 1, 2}

As the domain of *Screen.resolution* does not consist of integers, we introduce the following conversion before we start: {640x480 → 1, 800x600 → 2, 1024x768 → 3, 1200x800 → 4}, hence the domain of the attribute *Screen.resolution* becomes {1, 2, 3, 4}.

We also assume that the following relations exist:

- If *CPU 1.speed* = 1600 then *CPU 1.cost* = 62, *CPU 1.pc* = 18
- If *CPU 1.speed* = 1800 then *CPU 1.cost* = 76, *CPU 1.pc* = 20
- If *CPU 1.speed* = 2000 then *CPU 1.cost* = 99, *CPU 1.pc* = 24
- If *CPU 2.speed* = 1600 then *CPU 2.cost* = 62, *CPU 2.pc* = 18
- If *CPU 2.speed* = 1800 then *CPU 2.cost* = 76, *CPU 2.pc* = 20
- If *CPU 2.speed* = 2000 then *CPU 2.cost* = 99, *CPU 2.pc* = 24
- If *RAM.size* = 512 then *RAM.cost* = 13, *RAM.pc* = 11
- If *RAM.size* = 1024 then *RAM.cost* = 25, *RAM.pc* = 22
- If *RAM.size* = 2048 then *RAM.cost* = 48, *RAM.pc* = 30
- If *RAM.size* = 4096 then *RAM.cost* = 80, *RAM.pc* = 37

- If *Screen.resolution* = 640x480 then *Screen.cost* = 47, *Screen.pc* = 121
- If *Screen.resolution* = 800x600 then *Screen.cost* = 58, *Screen.pc* = 125
- If *Screen.resolution* = 1024x768 then *Screen.cost* = 69, *Screen.pc* = 129
- If *Screen.resolution* = 1200x800 then *Screen.cost* = 75, *Screen.pc* = 132

Table B.1 Domains of some attributes in *FM 2*

	cost	**mem**	**pc**
CPU 1	{62, 76, 99}	-	{18, 20, 24}
CPU 2	{62, 76, 99}	-	{18, 20, 24}
GPU	{41}	-	{12}
RAM	{13, 25, 48, 80}	-	{11, 22, 30, 37}
MainBoard	{85}	-	{53}
Screen	{47, 58, 69, 75}	-	{121, 125, 129, 132}
DVD	{35}	-	{153}
Graphics	{62}	-	{17}
Sound	{37}	-	{12}
Network	{14}	-	{19}
FireWire	{114}	-	{24}
WiFi	{21}	-	{81}
PS Type 1	{28}	-	-
PS Type 2	{33}	-	-
PS Type 3	{35}	-	-
PS Type 4	{38}	-	-
PS Type 5	{40}	-	-
Task Scheduler	{11}	{8}	-
Mac OS	{94}	{372}	-
Windows	{60}	{400}	-
Linux	{33}	{190}	-
Chn 1	{22}	{48}	-
Chn 2	{22}	{48}	-
Chn 3	{22}	{48}	-
Int. Browser	{24}	{150}	-
Email Software	{28}	{72}	-
Organizer	{50}	{92}	-
Movie Maker	{95}	{195}	-
Media Player	{8}	{50}	-
Photo Album	{25}	{180}	-
Chess	{12}	{48}	-
3D Car Race	{35}	{230}	-
Tetris	{7}	{34}	-
DC 1	{22}	{20}	-
DC 2	{22}	{20}	-
DC 3	{22}	{20}	-
DC 4	{22}	{20}	-
Word	{18}	{42}	-
Excel	{18}	{42}	-
PowerPoint	{18}	{42}	-

We also assume that the following common neutral values exist for the attributes involved in global constraints:

- 0 for the *cost*, *mem*, *pc*, and *ch* attributes
- 101 for *Graphics.sn*, 102 for *Sound.sn*, 103 for *Network.sn*, 104 for *FireWire.sn*, 105 for *WiFi.sn*.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Karataş, Ahmet Serkan

Nationality: Turkish (TC)

Date and Place of Birth: 24 May 1975, Nazilli - AYDIN

Marital Status: Single

Email: karatas@ceng.metu.edu.tr

EDUCATION

Degree	Institution	Year of Graduation
PhD	METU Computer Engineering	2010
MS	METU Computer Engineering	2000
BS	METU Computer Engineering	1998
High School	Nazilli Lisesi	1993

WORK EXPERIENCE

Year	Place	Enrollment
2008-Present	K&K Technologies	Company Owner
2005-2007	S.P.A.C.	Project Manager
2003-2005	Aselsan A.Ş.	Software Engineer
1998-2003	METU Computer Engineering	Research Assistant

FOREIGN LANGUAGES

Advanced English, Poor German