Operationalizing Feature Models for Concern-Oriented Reuse

Nishanth Thimmegowda

Master of Science

School of Computer Science

McGill University

Montreal,Quebec

2015-11-01

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

DEDICATION

This thesis is dedicated to all the members of Software Engineering Lab, McGill University without whom this thesis would not have been possible

ACKNOWLEDGEMENTS

This thesis as it stands would not have possible without the help of a lot of people. I would like to express my gratitutde towards them.

First of all, my deepest respect and acknowledgement to Prof. Joerg Kienzle who guided me throughtout my journey of Masters at McGill. Be it acadaemic or non-acadaemic, his guidance has made me more ambitious, more dedicated and simply a better human being. No amount of words can express the respect which I shall always have for him. Thank you for all the opportunities which you provided me. Keep rocking Prof.

I would also like to thank Gunter Mussbacher for this continous support throughout my thesis and answering any queries arised regarding jUCMNav technicalities.

Matthias, Berk and Celine - My three awesome companions at SEL, McGill who because of their work, passion, beer and pure awesomeness made my time at McGill nothing but extraordinary.

100 Gamaads, my group of friends from MSRIT, Bangalore, whose constant jokes, messages and support motivated me during my thesis.
Doda : "Lol"
Satvik : "Enro nimdhu"
Avinash : 9gag video

Sam : Image

Akash : Classy pic

Sridhar : "Rod macha"

Vikas : "En macha"

Tejesh : Non-existent

KT : "Yaakro"

Tejas : "Platinum gym"

Special mention to Hucha Venkat Sene Montreal Chapter, honarary director Vinnay Mayya, Chief Treasurer, Kumar Vishwanath, Executive President, Kishan Pai, who enforced my belief that anyone can make it big if he beleived in himself and the power of the sun.

All my friends from Montreal - Akhilesh, Vidhya, Adithya, Gaurav, Debal, Srishti, Nithin, Nandu, Nikhil, Vanessa, Edward, Yaron, Muntasir, Navjot. Thank you for all your patience and the support you people provided for me.

Lastly Vishnu, Manisha, Kavya, Divya, Aditya, Rajath, Harit you people were as much as an integral part of my journey as me. This thesis would not have been possible if not for you guys.

ABSTRACT

Concern-Oriented Reuse (CORE) is a reuse-focussed software development approach that builds on the disciplines of model-driven engineering, software product lines and aspect-orientation. CORE defines broad units of reuse called concerns, in which feature models play a central role. They express the variability encapsulated within a concern, are used as a basis for calculating the impact on non-functional properties and qualities of specific concern configurations, and establish a link to the realization models that implement the functionality of each feature. With the aim of creating the first CORE-based modelling tool, this thesis investigated two different ways of operationalizing feature models within CORE: a) by means of a well-defined interface that links an external implementation of feature models with the CORE tool, and b) by directly integrating feature models into the CORE metamodel and providing operations to perform edit operations. To simplify and streamline the interaction between tool users and the feature models, this thesis also presents different feature model visualization algorithms that are optimized for concern designers (i.e., modellers that create or modify a concern) or for concern users (i.e., modellers that reuse an existing concern as is).

ABRÉGÉ

Concern-Oriented Reuse (CORE) est une approche de développement de logiciel axée sur la réutilisation qui se fonde sur des disciplines de l'ingénierie dirigée par les modèles,les lignes de produits logiciels et des techniques orientées aspect. CORE définit des larges unités de réutilisation appelées des préoccupations où les modèles de fonctions y jouent un rôle central. Ils représentent la variabilité encapsulé dans une préoccupation. Ils sont utilisés comme base pour le calcul de l'impact sur les qualités et les propriétés non-fonctionnelles de configurations spécifiques. Ils établissent aussi un lien vers les modèles de réalisation qui mettent en œuvre la fonctionnalité de chaque caractéristique. Avec le but de créer le premier outil de modélisation basé sur le concept de CORE, dans cette thèse j'étudie deux manières différentes de rendre opérationnels des modèles de fonction au sein de CORE: a) au moyen d'une interface bien définie qui relie une mise en œuvre externe de modèles de fonction avec l'outil de CORE, et b) en intégrant directement les modèles de fonction dans le méta-modèle de CORE et en fournissant des opérations pour effectuer des opérations de modification. Pour simplifier et rationaliser l'interaction entre les utilisateurs de l'outils et les modèles de fonction, cette thèse présente également différents algorithmes de visualization pour les modèles de fonction qui sont optimisés pour les concepteurs, à savoir les modélisateurs qui créent ou qui modifient une préoccupation, ou pour les utilisateurs de préoccupation, à savoir les modélisateurs qui réutilisent une préoccupation telle qu'elle a été conçue.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
Introduction

Over the years, engineering disciplines have matured to a point where official organizations now exist that govern and regulate how engineers practice their professions. Different engineering disciplines, e.g., civil engineering, provide standards and manuals for their processes, practices, and even safety guidelines. These manuals guide the engineer in making proper decisions and choosing the best solution that satisfies stakeholder's requirements. Similar to other engineering disciplines, software engineering aims at systematic production of software, by choosing the best solution, and by applying the best practices that satisfy the requirements, are cost-effective, and minimize time-to-market. Producing complex software systems involves many stakeholders such as developers, scientists, engineers of other disciplines, the customer, and end users with specialized domain knowledge in their respective fields. Bridging the gap between the specialized domain knowledges and the development technologies (e.g., programming languages, technical infrastructure, testing methods) becomes a major challenge when different stakeholders work together in a software project. Previous research has shown that manual efforts to bridge this gap result in accidental complexities [18].

Model-Driven Engineering (MDE) [38] provides means to represent domain specific knowledge within models. MDE aims at developing software through model creation, refinement,

and composition. If done with automated tool support, MDE helps reduce accidental complexities and bridge the domain-implementation gap. MDE advocates using the best modelling formalism that expresses the relevant properties of the system under development at each level of abstraction, for a given stakeholder group. A formalism used at the requirement level for scientists is different from the formalism used at the design level for developers. Through model transformations, models of higher level of abstraction are integrated with lower-level models that are closer to the solution space, such as algorithms, data structures, networking, etc. This process continues until an executable model (which can be code) is generated.

However, MDE has challenges of its own. The crosscutting nature of most models makes it difficult to apply in a modular way software engineering techniques such as information hiding, decomposition, interfaces, and abstraction. In addition, modellers usually create models from scratch as there is limited support for reusable model libraries. Model reuse is a challenge in MDE, despite the success stories in programming languages as exemplified by, e.g., class libraries, services, and components. Typically, there exist different reusable solutions for a particular problem, with each solution having different impacts on high-level goals and system properties. When reusing existing artefacts, software practitioners usually rely on their experience to assess the advantages and disadvantages, or have to consult lengthy and informal documentation, books, blogs, tutorials, etc.

To address the aforementioned issues, a novel reuse paradigm called Concern-Oriented Reuse (CORE), that builds on the ideas of MDE, advanced Separation of Concerns (SoC) [16], Software Product Lines (SPL) [35], and goal modelling [15, 43] was introduced in [3]. CORE

introduces a new unit of reuse called *concern*, that enables broad-scale model reuse. A concern groups together software artifacts, in particular models, related to a domain of interest to a stakeholder or developer. Typically models defined inside a concern span multiple phases of the software development cycle, and go through different layers of abstraction. Furthermore, and most importantly for this thesis, a concern encapsulates all relevant variations/choices that are available for reuse in the captured domain, together with guidance on how to choose among those variations. The variations are specified in a variation interface that takes the form of a feature model.

At the start of my thesis, the idea of CORE existed, but there was no modelling tool that supported the concepts of CORE. Researchers had created feature models for example concerns, e.g., Association, Observer, Workflow, on paper without proper tool support. The detailed, generic design model realizing each feature or variation were separately designed using stand-alone modelling tools with support for aspect-oriented composition. In order to reuse a specific concern configuration, researchers had to use the stand-alone modelling tools to compose the realization models of the desired features in the correct order.

This situation was not only cumbersome, but also highly error-prone, because the modeller could easily decide to compose realization models of conflicting features, or forget to compose realization models that take care of feature interactions. Since the main idea of CORE is to make reuse simple and straightforward for the modeller, it was imperative to operationalize CORE by creating a tool that would support the concepts of CORE and the reuse process. For concern designers, this includes supporting the definition of feature models and assigning realization models to features, and for concern users to streamline the concern

reuse process (automated evaluation of feature selections, evaluation of impacts, generation of customized models, composition with application models).

1.1 Thesis Contributions

This master thesis focusses on operationalizing the fundamental concepts of concern-oriented reuse: concerns, feature models and features, and how they are linked to realization models. To demonstrate the usability of CORE, the existing aspect-oriented software modelling tool TouchRAM was converted to concern-orientation. With TouchCORE, as the tool is now called, it is now possible to build and reuse design concerns. At the beginning of the thesis, we envisioned two possible ways to operationalize CORE, either by reusing an existing implementation of feature and impact models to work with CORE or by directly integrating features and impacts into CORE. One of the goals of the thesis was to evaluate which appoach would be better.

Specifically this thesis makes the following contributions:

- Investigate the operationalization of feature models and impact models for CORE by reusing an existing implementation
  - Initially it seemed like the best idea to operationalize feature and impact models for CORE was to reuse an existing, external implementation. To this aim, the features and impacts that were only abstractly defined in the CORE metamodel were linked to an external implementation by means of an interface. This interface defined enumeration types and function signatures which the external implementation had to implement to interact with CORE. This external interface included all the possible feature model editing operations which the modeller might need. The interface and integration was implemented by building a bridge

between TouchCORE and jUCMNav, an Eclipse-based tool supporting feature and goal modelling as defined by the User Requirements Notation [22].

  - The main advantage of an external implementation is that it already exists, which reduces the implementation effort. Additionally, by defining an interface, it is possible to decouple CORE from a specific external implementation, and even use different implementations, if available. Unfortunately, this approach also revealed several serious limitations, e.g., problems with navigability, interface evolution, cross referencing of model elements, etc.

- Operationalize feature models by directly integrating them into CORE
  - The CORE metamodel was extended to directly contain the feature model. Classes and enumeration types were added to the CORE metamodel to encode features and feature dependencies. The concept of feature configuration was added to store feature selections and feature re-exposure.
  - Algorithms describing the edit operations for CORE feature models are described in detail, and were implemented in TouchCORE. An algorithm that checks for the validity of a given feature configuration is also presented.
  - The integration of an external SAT solver with CORE in order to perform advanced validity checks of feature configurations is described, and was implemented for TouchCORE.
- Propose different feature model visualization algorithms for concerns depending on whether a modeller is *designing* a concern or *reusing* a concern
  - When a *concern designer* works on realizing a feature within a current concern, our proposed visualization algorithm shows any reused concerns and their selected

features as mandatory child features of the feature that made the reuse in the current concern. This gives the concern designer all the information he needs to properly integrate the models he develops with the structure and behaviour of the current concern, in particular the parent features and the reuses they made.

- When a concern user reuses a concern, our proposed visualization algorithm presents a condensed feature model that includes all the choices that the user has to make, including reexposed choices of internally reused concerns, if any. Any features that have to be reused (mandatory and already selected ones) are hidden to simplify the reuse process.
- Furthermore, we propose two different selection orders to the concern user. Full display involves showing the entire feature model of the concern that is being reused including re-exposed features of lower-level concerns to the user, which provides him with a condensed view of all options and allows him to try out the different selections to compare their impacts. Top-down display presents the reuse choices to the concern user layer-by-layer, starting with high-level decisions. Only when the user has made a decision by selecting a feature, the next layer, i.e., the subfeatures, are visualized. In this mode, the user is confronted only with the minimal set of decisions that need to be made, thus reducing cognitive load.

1.2 Thesis Outline

This master thesis is structured as follows. Chapter 2 presents the main concepts of concern-oriented reuse and the concern reuse process, as well as the old CORE metamodel. Chapter 3 presents the background of URN and jUCMNav, a tool that already supported feature models and impact models, and then describes how we attempted to reuse this existing

feature and impact model implementation to operationalize CORE. Finally, the limitations of this approach are summarized.. Chapter 4 presents a stand-alone implementation of feature models. It introduces the new CORE metamodel with integrated feature modelling together with algorithms describing the behavior of the operations on feature models required by CORE. Furthermore, the integration of an external SAT solver to validate partial feature selections is also presented. Chapter 5 presents different visualization algorithms for the feature model of a concern optimized for the type of modeller using the CORE tool, i.e., the concern designer or the concern user. Chapter 6 presents the related work in the field, and last chapter concludes the thesis.

Chapter 2
Background

Model Driven Engineering (MDE) offers a new approach to address platform inabilities and express the domain concepts effectively with the help of transformation engines and generators which analyse certain aspects of models. Classic MDE focusses on models (design models , sequence diagrams...) to be the essential unit of abstraction, design, construction and reasoning. In sharp contrast to this classic approach of MDE, in Concern-Oriented Reuse (CORE), the main focus is on concerns.

CORE aims to combine the best practices and principles from MDE, advanced modularization techniques and Software Product Lines (SPL). Reuse is the main focus in the context of SPL development [12]. The reuse concepts of SPL offers limited scope. i.e. the reuse are not intended to cross boundaries. CDD covers reuse in a broader sense. It enables reuse across concerns. Concerns are designed individually and reuses are enabled by the means of well defined interfaces.

Aspect Oriented Modelling (AOM) aims to increase modularity and allows separation of concerns. Many aspect-oriented modelling approaches have been proposed [26]. AOM provides generic methods to package concerns which enhances reusability. This reuse can be utilized by the detailed specification of the concern using independent aspect oriented modelling notations.

2.1 Variability Modelling in Software Product Lines

Software Product Lines (SPL) refers to software engineering methods used for creating similar set of software products. SPL can exist in many forms, namely programs, libraries, software artifacts..etc. To express the commonalities and variabilities which exists in a software product lines, various modelling notations have been suggested. The prominent modelling notations are presented below.

2.1.1 Feature Modelling

Feature Model was first introduced by Kang et al in 1990 [25]. Since then the topic of feature models has been subjected to extensive research across various domains. A feature is an unique characteristic of a software system which has particular importance to the stakeholder or in the functioning of the software system. A feature model has a tree-like structure with parent-child relationships between the features.

2.1.2 Decision Modelling

Decision Modelling(DM) originated from the Synthesis method [10] and has been existing since the time of feature models. The origin of DM was mainly due to practical applications or applications closely tied to industrial use. DM can be defined as "*a set of decisions that are adequate to distinguish among the members of an application engineering product family and to guide adaptation of application engineering work products*".

2.1.3 Orthogonal Variability Management

Orthogonal Variability Management (OVM) [36]was introduced to document variability in a dedicated model. OVM documents only the variability of the product line. OVM offers significant advantages which includes smaller and less complex models, consistency

across models, and improved communication [30]. The information about the variability is documented in the following format.

- Variation point : What varies ?
- Variant : How does it vary ?
- Variability constraints : The constraints on the variation
- Visibility of variability : Who is it documented for ?

2.2 Concern Representation

The concern reuse unit defines three interfaces which specify the details on how it is supposed to be reused. The three types of interfaces are described as below :

- Variation interface: The variation interface describes the available variations (and commonalities) provided by the concern and its encapsulated models. It also specifies the impact that different variations have on non-functional requirements. These variations are expressed at a high level of abstraction, i.e., as user-perceivable features, and the relationships between those features. In concerns, feature models and goal models are used to specify the variation interface.

Feature Models: Feature model can be defined as a model which expresses the variabilities and commonalities between objects called feature in a hierarchical tree like format. The features in between them can have relationships or constraints. The possible type of relationship with a child feature can be Optional, Mandatory, XOR or OR. Kong introduced feature models in 1991, in order to explain the variabilities and commonalities which existed in Software Product Lines.

Goal Modelling : Goal models are used to capture actors, stakeholders, business requirements and objectives during the early requirements phases. Goal models indicate ways of

satisfying objectives and the positive and negative impacts on higher level alternate goals and objectives. Goal models can provide quantitative and qualitative analysis on alternative solutions which can be used to solve a problem.

- Customization interface :The customization interface specifies how a chosen variation of the solution should be adapted according to the specific needs of the application under development. Every variant of the concern is described as abstractly to enhance reusability. The main principle of the customization interface is to highlight the model elements that needs to be adapted to the application domain. The CORE tool can be utilized to provide these highlighting. Some model elements in the concern, are partially specified and needs to be mandatorily related or adapted to the application's elements that intends to use the generic elements. Hence the customization interface is used when the chosen variant from the concern needs to be adapted to the application domain.
- Usage interface: The usage interface for the concern is similar to what the public operations are to a class in object-oriented design. I.e., the public operations define how to access the functionality provided by the class from the outside world. In a similar manner, the usage interface of the concern describes how the application can access the functionality provided by the concern unit. The functionality is the structural and behavioral properties of the underlying generic concern design. In other words, it abstractly presents the functionality encapsulated by the concern to the developer.

2.3 Designing a Concern

Designing a concern is a non-trivial task, which can be done only by a designer who is an expert of the domain for which the concern is to be created. This domain expert should

know all the low level details of the concern being designed and how it can be adapted to the various application domains which can use the current concern. The concern designer will try to adapt the design to make it as generic as possible which will facilitate easy reuse and faster adaptation to the reusing context.

The concern designer starts off designing the concern by identifying the key features of the concern. The features represent the distinctive aspects of the software systems. These features can be characterised with deep understanding of the system. These features are grouped together in a feature model. Feature model capture the relationships and dependencies that exist between these unique features and are represented in a hierarchical tree like format with parent - child relationships between each pair of feature defined appropriately. Some set of features can also have extra dependencies which cannot be easily represented with the parent - child format. The four possible types of relationships which can exist in feature model between a parent and a child are :-

- Mandatory - The child feature is mandatory to the parent
- Optional - The child feature is optional to the parent
- XOR - Exactly one child among all the children should be selected
- OR - At least one child among all the child must be selected

In addition to relationships mentioned above, certain cross dependencies can also exist between certain pairs of features. The two types of cross dependencies are :

- Excludes This indicates that one feature cannot work with another feature, i.e., Feature A excludes Feature B
- Requires: This indicates that one features needs another feature to function appropriately, i.e., Feature A requires Feature B.

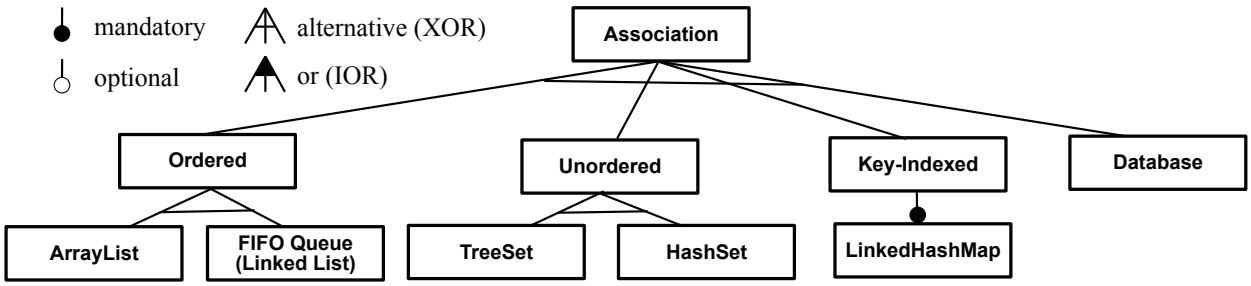


Figure 2–1: Association Feature Model

The 2–1 represents the Association concern. The root level association has an XOR constraint, indicating only one child can be selected. It is followed by their children and their respective relationships.

After designing the feature model for the concern, the concern designer proceeds to model an impact model for the concern. These impact models represents the impacts of certain features on high level goals and non-functional properties. The concern designer will appropriately make use of sub level goals and contribution links to design the impact model for the concern. These impact models can represent how different variants of the solution / selection of a combination of features impact the high level goals.

After designing the impact and feature models for the concern, the concern designer needs to provide the appropriate structural and behavioral properties for the features defined in the model. The concern designer realizes each feature via a model which encapsulates the properties at all relevant levels of abstraction. These features and models are interlinked to make the concern complete.

2.4 Reusing a Concern

After the concern designer completes designing a concern, this concern can be used by any concern user to adapt to any application domain. Even though designing a concern, is

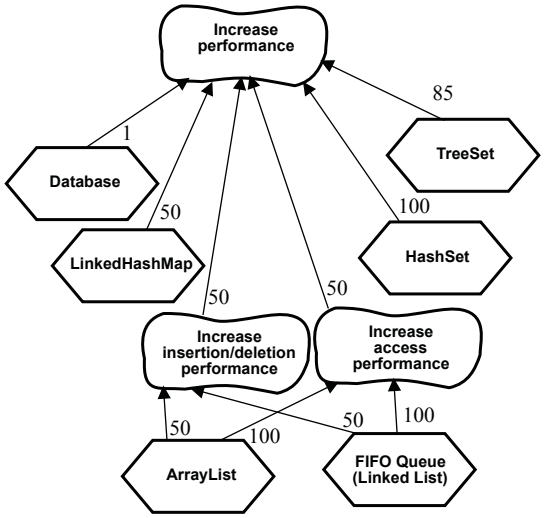


Figure 2–2: Association Impact Model

time - consuming, challenging and requires the concern designer to have in-depth knowledge of the system and the functionality, the benefits of the process lie in the ease of the reuse process. Any user who wants to reuse an existing concern needs to perform 3 simple steps :

- The concern user who wants to reuse a concern, must first select all the required features he desires based on the impact of each variant on the non functional requirements and high level goals. After the concern user chooses a specific variant from the feature model, the CORE tool merges all the realizing models of those features to produce a new model according to the user selection.
- The second step in the process is to adapt the produced model to the application context of the user. Some elements in the produced model might have been purposefully left partial which can later be completed by mapping the element to the application context. The user maps all the elements from the produced detailed model through an interface with the application's model elements.

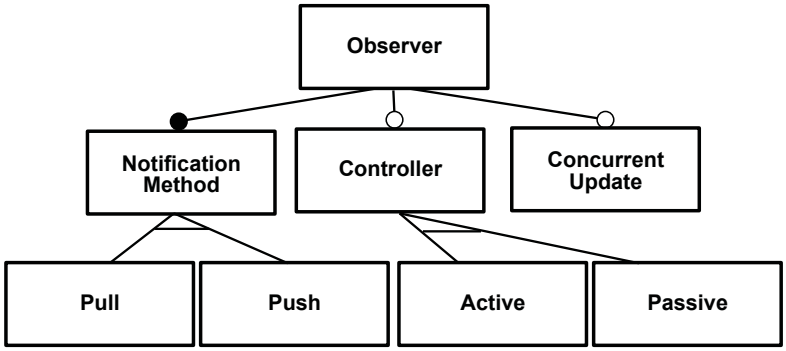


Figure 2–3: Ordered concern reusing association

- The concern user can use all the functionality provided by the selected configuration model through the exposed usage interface.

Furthermore, it happens more often than not, that no concrete selections can be made at that stage of process on what the user wants. This can result from the fact that the current user of the concern is an intermediary concern designer for the next user. Hence this intermediary concern designer aims to reduce the concern design specifically to a smaller sub sets of application. For example : An authentication concern can be used by a designer to design only a generic High - Level Security concern makes use of only biometric means to provide the security measures. In such cases, the intermediary concern designer would want to "re-expose" certain features, implying that the decision about such features is deferred until the next user. All other features which are not selected by the user (intermediary concern designer) are not shown to the next user nor included in the combined models. By re-exposing the features, the concern designer defers the decision to the time of reuse of the current concern. The reexposed features are inherently added to the variation interface of the current concern.

In the ordered feature model presented, the concern reuses association reexposing treeset and hashset features

2.5 Existing CORE Metamodel

Figure 4 represents the CORE metamodel. In the heart of the meta model lies the COREConcern class which conceptually represents the whole concern.

This concern has two models to represent the variation interface, the feature model, represented by COREFeatureModel and the impact model, represented by COREImpactModel. Both of these are subclasses of COREModel. The COREModel also references COREModelElements which contains different model elements from the feature model, impact model and the realization models. The features and impact models elements are represented by COREFeature and COREImpactModelElements which are subclasses of COREModelElements.

The structural and behavioral properties represented by the feature is realized by a model. These models are subclasses of COREModel. The composed detailed model produced from the selection of features is also subclassed under COREModel.

The COREFeature has a reference to COREReuse, which conceptually signifies the reuse of another concern by a feature. The COREReuse points to the concern reused. In this reuse, CORECompositionSpecification is defined for the customization interface to map the different model elements from the produced detailed model to the application model. This is achieved by the to and from references from the COREMapping. The CORECompositionSpecification also has a reference to the detailed model produced from the selection of the user.

The concern has a COREInterface , referred by the COREConcern, which signifies the usage interface. This interface has links to the selectable feature, the impacted features and the customizable model elements for the user of the concern.

All the elements in the metamodel are subclassed under COREElement which has the attribute of name and id. The id's are used to uniquely identify a model element.

This metamodel is defined as abstractly as possible while maintaining the logical structure of Concern-Oriented Reuse. Any implementation platform can implement their own feature model techniques / operations combining them with their modelling notation and realizing the same models. The idea of keeping the metamodel as abstract as possible was to encourage and provide flexibility to the underlying implementations.

2.6 TouchCORE

TouchCORE is a multi-touch enabled tool for Concern-Oriented Reuse aimed at developing scalable and reusable software design models [42]. TouchCORE is based on TouchRAM, which provides aspect-oriented modelling support as defined by the Reusable Aspect Models (RAM) [27] to express software design models using class, sequence and state diagrams. At the start of my thesis, the first version of TouchCORE had just been released, which extended TouchRAM with abstract support for concern-orientation.

TouchCORE consists of the front end, i.e., the graphical user interface (GUI), and the backend, which contains the abstract CORE metamodel, the RAM metamodel, the controllers for manipulating RAM models, and a RAM model weaver to compose RAM models. The GUI for TouchCORE is realized using the open source Java framework Multitouch for Java (MT4j) [28]. MT4j is an open source, cross platform Java framework, created for rapid and easy development of visually rich 2D or 3D applications and is designed to support

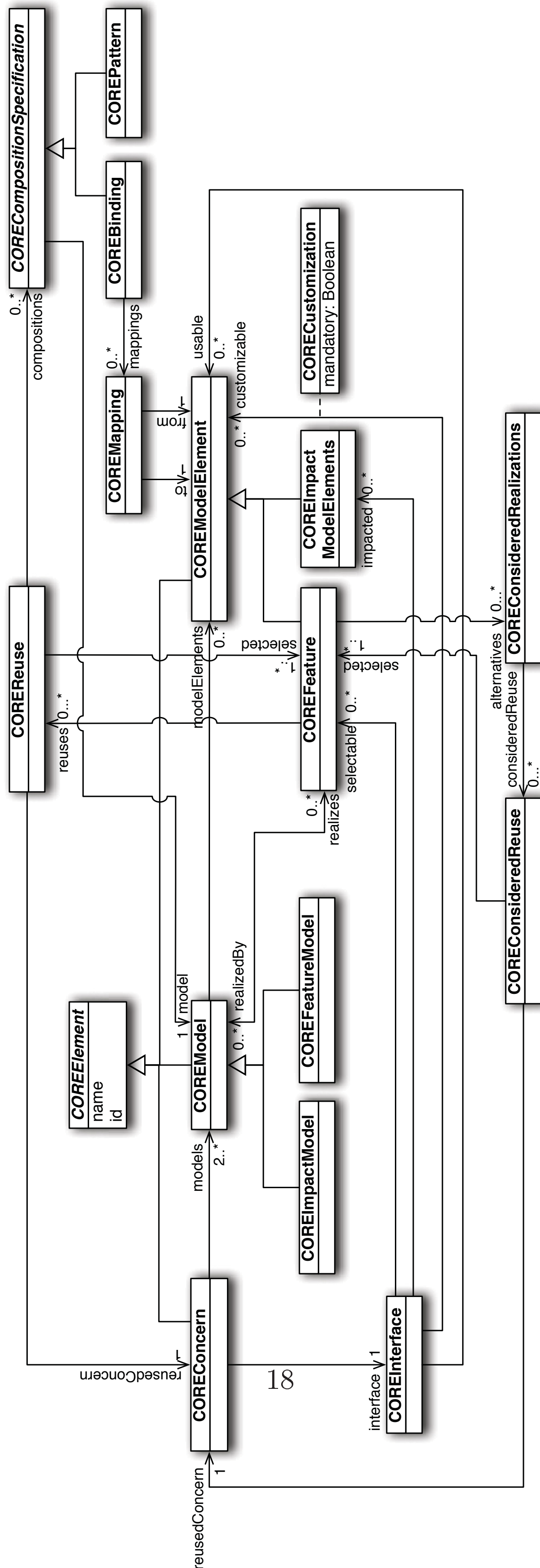


Figure 2–4: CORE metamodel

different kinds of input devices with a special focus on multi-touch support. Many standard multi-touch gestures are already built in and new gestures can easily be defined. TouchCORE has been tested on many popular platforms such as MacOSX, Windows etc. Its dependence on Java and TUIO implies that it should work on any environment where both of these are supported.

Although the front end of TouchCORE is dependent on MT4j, all other components of the tool are decoupled from the GUI based on the Model-View-Controller design. Hence the front end and the back end can have a separate evolution path without being dependent on each other.

The back end of TouchCORE can be categorised into two parts: the CORE part, whose foundation is the abstract CORE metamodel, and the realization part, whose foundation is currently the RAM metamodel, the RAM controllers and the RAM weaver. Both metamodels are defined using the Eclipse Modelling Framework (EMF) [19]. EMF allows generation of code from the defined structured data model which is further used by TouchCORE. The models are serialized by XMI (XML Metadata Interchange) format corresponding to the metamodel. The Object Constraint Langauge (OCL) is used for specifying constraints on the meta-model specification of derived properties.

In this thesis, we used TouchCORE as a testbed for evaluating the different ways of operationalizing CORE, i.e., adding concrete CORE structure and behaviour to the backend (metamodel, operations for manipulating feature models, feature model composition algorithms), as well as the front end (GUI and algorithms for feature model visualization using MT4J).

As the abstract CORE metamodel presented in section 2.5 has no concrete structural support for feature and impact models, the first version of TouchCORE also did not support creating, manipulating and visualizing CORE-related concepts. With no tool support, a user who wanted to define or (re)use concerns was not able to do so. The goal of my thesis was to make CORE practical and therefore as first step to the process, I investigated reusing an existing feature model implementation just like the abstract CORE metamodel suggested to do. The integration of an external feature model implementation to work with the original, abstract CORE metamodel is presented in the next chapter.

Chapter 3
Reuse of Existing Feature and Impact Model Implementation

The unit of modularization, concern lays emphasis on reusability and adaptability. To support this, concern metamodel has certain concepts namely, Feature Models and Impacts Models abstractly defined. In case of Feature Model, CORE only specifies the Feature Models and Feature, without specifying how they are linked or the relationships which exists between them. The same follows with Impact Models. The main objective of this first part of the thesis is to provide a concrete realization for Feature and Impact Models. While exploring the different frameworks / languages / modelling notations which supported this, we came across URN which supported all of this in the form of goal models. These were integrated to work with the existing CORE structure. Based on the result of this integration, several limitations of this approach were observed.

The structure of this chapter is as follows. Section 1 gives a brief background of URN with subsections describing components of URN and its support for feature models. Section 2 describes the jUCMNav tool with support for feature and impact models. Section 3 describes the interface structure which was defined to interact between jUCMNav and CORE followed by the limitations of using an interface.

3.1 User Requirements Notation (URN)

URN is a lightweight graphical language used for modelling and analyzing requirements in the forms of goals and scenarios and links between them.

URN is a part of the ITU family of languages, was standardized in 2008 [23], and was the first standard which explicitly addressed functional and non-functional requirements under the umbrella of one unified language. It was developed with the objective of focussing on the early stages of software development, particularly the requirements stage. It aims at supporting reusability, which is one of the top goals of the standard, along with providing early performance analysis along with traceability and transformations to other languages. It makes it possible to express specifications for stakeholders, non-functional requirements, goals, rationales, behaviours and actors.

URN combines two modelling notations, namely, Use Case Maps and the Goal-oriented Requirements Language.-

3.1.1 Use Case Maps (UCM)

UCM is a formal way of expressing use cases / scenarios / workflows and their relationships Use Case Maps. The notation is most useful in the early stages of software development and can be used in elicitation, validation and in high-level architectural design and generation of test cases. UCM's can help bridge the gap between requirements and design greatly by the flexible allocation of responsibilities to architecture decisions. UCMs can also express dynamic variations of scenarios, and provides features for incremental development and integration of complex scenarios.

3.1.2 Goal-Oriented Requirements Language (GRL)

The Goal-oriented Requirements Language is intended to be used to capture the business and system goals, and how the goals influence each other. This notation can be applied

to specify functional and non-functional requirements. GRL contains various graphical elements, which are used to express different kinds of concepts that are relevant during requirements specification. The four main categories of concepts are: the intentional elements, the links, the actors and non-intentional elements.

The intentional elements contains elements like goals, tasks and softgoals. They are used to allow answering questions for models pertaining to aspects of behaviors, informational and structural aspects, and how and why they were included in the system requirement, and what possible alternative options can be used instead of the other and the reasons for them.

The actors are the stakeholders for the goals. These actors are a subset of the stakeholders of the system. Each goal model can contain one or more actors. Specifying actors in the goal model can help understand in the GRL which stakeholders are affected by a particular goal.

The links represent the contribution between two intentional elements, which specify how much each intentional element influences or "contributes" to some other intentional element. Each link has a contribution value. These contribution value can be absolute, indicating whole values, or can be relative with respect to the contribution values on links coming from other siblings. The links can be thought of as a means of connecting the intentional elements. The links can also be of three decomposition types, namely, OR, XOR and AND. For AND decomposition link, the minimum of the satisfaction value of the child elements is taken, while for OR and XOR, the maximum is taken. GRL also provides different evaluation mechanisms to calculate the quantitative and qualitative satisfaction of intermediary and high level goals. It is done by assigning a value (satisfaction) to the tasks and goals at the leafs of the goal graph. The values of the leafs are then propagated towards the top, to

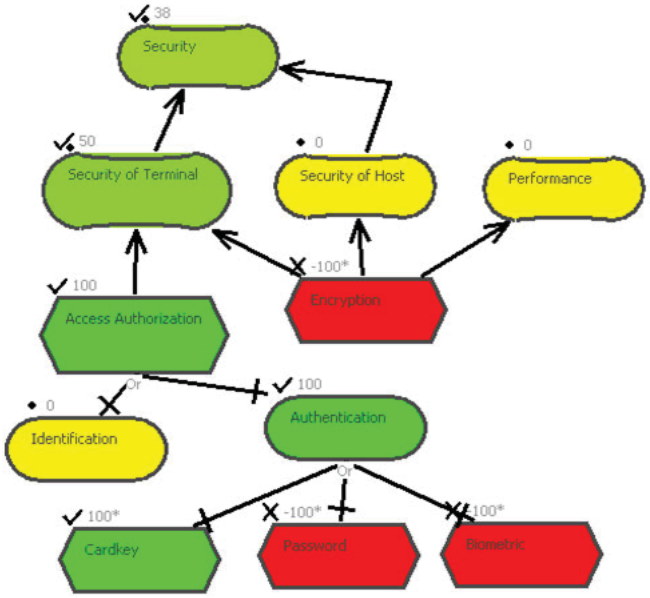


Figure 3–1: An Example Goal Graph

successively determine all satisfaction values. Many evaluation mechanisms exists and the user can apply a mechanism of his choice for the evaluation.

3.1.3 Requirements Elicitation and Specification with URN

GRL addresses both the functional and non-functional requirements, and exclusively focuses on answering the "why" questions. UCM on the contrary tries to address the "what" questions which may exist in the design. By defining both GRL and UCM models, together they address the "how" questions. GRL establishes traceability by linking the intentional elements in the GRL model with the non-intentional elements in the UCM models. Specifying both models aids in identifying further goals, and helps in developing more detailed and accurate scenarios. It is recommended to start with high level goals, which are further

resolved as sub-goals and scenarios. As scenarios are refined they may impact the goals, which leads to an iterative software development cycle.

3.1.4 How URN can Support Feature Models

In order to support feature modelling in URN, the metamodel of GRL was extended to support feature modelling concepts. A feature is conceptually close to GRL tasks. Both represent the behavioral aspects or properties of a system/product. Hence *Feature* is subclassed under the GRL intentional element. Keeping the existing GRL propagation mechanism, to preserve the semantics of feature selections, the optional link (*OptionalFMLink*) and mandatory link (*MandatoryFMLink*) are subclasses under of the contribution links in GRL. The concepts of XOR / OR are reused from GRL as it is as they conceptually had the same meaning. OCL constraints are used to express the *requires* and *excludes* constraints, mentioned in the feature model. Storing a set of features for configuration was performed by subclassing feature configuration from GRL strategy concept which allowed to store a set of model elements for analysis purposes.

The satisfaction values in a GRL graph is in the range between -100 (element not being satisfied) to +100 (element is completely satisfied). However for feature models, the satisfaction values can be either 0 (element not being selected) to +100 (element is selected). The evaluation algorithm already defined in GRL handles cases in feature model for feature/intentional elements, decomposition links/feature groups, feature configuration/ strategies and integrity/OCL constraints. Hence the evaluation algorithm defined can be used for evaluation of feature models with special rules for optional and mandatory links. Each optional link have contribution link specified by a value of 100. In case of mandatory link, each value of contribution link is specified such that the parent reaches a value of 100 when all children

are selected. With these changes, the existing GRL evaluation algorithm can be used to adapt to evaluate feature models.

3.2 jUCMNav

JUCMNav [33]is an Eclipse-based [37] tool for URN which provides model editing and analysis capabilities. As such, jUCMNav provides all the necessary support for requirements elicitation, specification and validation expressed using UCM and GRL. jUCMNav is open source.

3.2.1 Feature Model Support in jUCMNav

When I started my master thesis, jUCMNav already provided an implementation of feature models. As outlined in subsection 3.1.4, a goal model can be adapted to represent a feature model . By treating tasks as features, and by changing the decomposition links to accommodate "Optional" and "Mandatory" links, a goal model can be used as a fully functioning feature model. Hence we decided to use the the jUCMNav features (i.e., the tasks) as a concrete realization of the abstract CORE feature, and use the feature model evaluation strategy provided by jUCMNav to evaluate the correctness of feature model selections.

3.2.2 Impact Model Support in jUCMNav

The concepts of the impact model that are needed to specify the variation interface of a concern in CORE are actually a subset of the concepts found in goal models. The impact model evaluation is also equivalent to the evaluation used in goal models, except that the leaf goals are replaced by features (which are actually GRL tasks as described in subsection 3.2.1), contributing to upper level goals. Selecting any feature in the feature model would set the satisfaction value of the feature to 100.

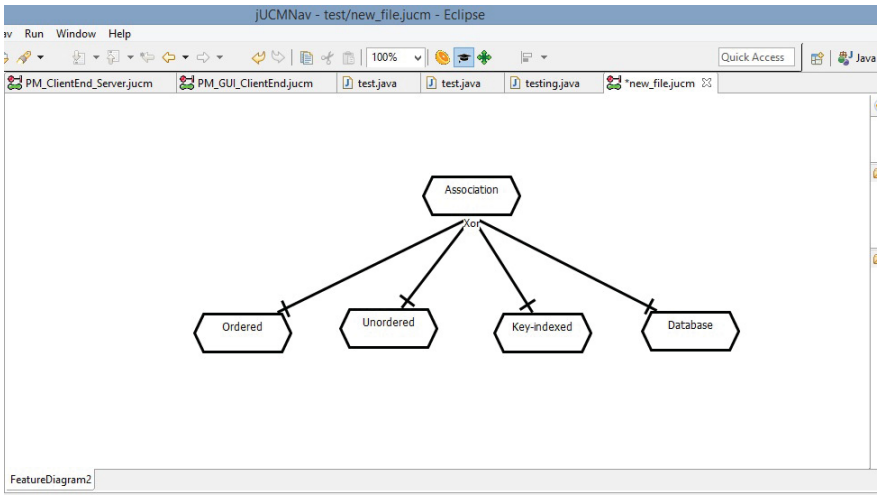


Figure 3–2: A Feature Model in jUCMNav

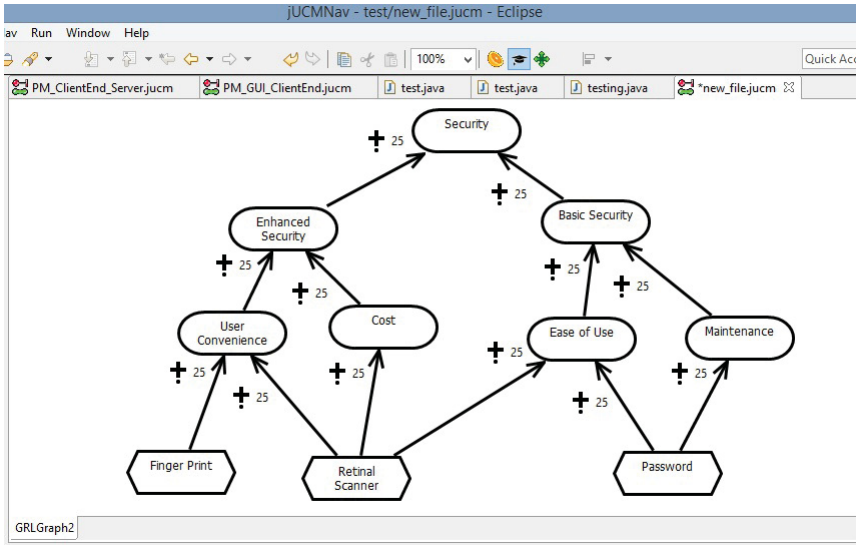


Figure 3–3: An Impact Model in jUCMNav

indicating accordingly how much selecting each of the sub-task contributes to the propagation to the parent

Since the feature and the impact model implemented in jUCMNav matched the requirements for expressing the variation interface for concerns as defined in CORE, we decided to reuse the existing jUCMNav implementations in the context of CORE.

3.3 Interface between CORE and Existing jUCMNav Implementation

Every modelling language that embraces concern-orientation and integrates with the CORE metamodel – a process called corification – needs to be able to use feature models and impact models to specify the variation interface of the concerns that are being built. In order to keep the implementations of modelling languages independent from a particular implementation of feature and impact models such as jUCMNav, we needed to define a global interface which would be used to communicate between the two implementations. This would allow for all corified implementations to be independent from the underlying feature model and impact model implementation in case in the future the jUCMNav implementation might be replaced by a different one, e.g., one that uses Familiar [1]. The interface must cover all the possible types of operations which the user might need while trying to construct or edit or evaluate a feature model.

3.3.1 Structural Additions to the CORE Metamodel

To support different implementations, two new enumeration types had to be defined in CORE to express the kind of relationships that exist between features, and to describe the selection status of features.

Enum COREAssociation - {OR, XOR, Optional, Mandatory, None}

The five values defined by this enumeration type encode the four possible relationships a child feature can have with its parent. An attribute indicating accordingly how much selecting each of the sub-task contributes to the propagation to the parent of type COREAssociation was added to the *Feature* class in the CORE metamodel, representing the relationship kind that this feature has with respect to its parent. The values are interpreted as follows:

- OR - With this constraint, one or more children have to be selected for the feature model to be valid. Note that when constructing a feature model, the tool must ensure that all the children of a parent have their attribute value set to OR or none has.
- XOR - With this constraint exactly one child has to be selected from all the children under the parent. Note that when constructing a feature model, the tool must ensure that all the children of a parent have their attribute value set to XOR or none has.
- Optional - The current child of the parent can be selected or not selected for the feature model to be valid.
- Mandatory - The current child of the parent has to be selected for the feature model to be valid.

Enum <Selection status> { NOT_SELECTED_ACTION_REQUIRED, NOT_SELECTED_NO_ACTION_REQUIRED, WARNING , USER_SELECTED , AUTO_SELECTED , USER_RE_REXPOSED, USER_AUTO_RE_EXPOSED }

The following 6 enumeration values were defined to encode the selection status of any particular feature in the feature model. jUCMNav evaluates the feature model and assigns the evaluation result to the features. The evaluation result is determined internal to jUCMNav. This enumeration type is needed to read the result of each individual result to the

user. Based on the value, the tool can adapt the way it displays the feature. The meaning of each of the values is as follows:

- NOT_SELECTED_NO_ACTION_REQUIRED: The following feature has not been selected by the user and during the evaluation no action is required by it. It usually is for an optional feature or for a XOR child, where another sibling has already been selected.
- NOT_SELECTED_ACTION_REQUIRED: This enumeration value indicates that the parent feature of the feature has been selected and an action is required either on it or one its siblings. This enumeration is usually returned for XOR / OR relationships where no children of the parent have been selected.
- WARNING: This enumeration value is assigned when there is a violation of constraints of the feature model . Having this enumeration type in the feature model, indicates the selection made by the user is invalid. It can arise in two situations:
  - When a *requires* or *excludes* constraint in the feature model has been violated.
  - When multiple children have been selected from an XOR parent.
- USER_ SELECTED : This enumeration value is assigned for features which are selected exclusively by the user. These selections are made exclusively by the user choice, indicating that the current feature is being preferred by the user. This enumeration can be overwritten by other enumeration values based on implementation logic or by the resulting evaluation.
- AUTO_SELECTED : This enumeration is assigned when a feature is automatically selected when a user makes another selection. Common cases where this occurs is when a constraint exists in a feature model, selecting a feature will in turn select another

feature in the feature model. This enumeration can also be resulted when parents of a particular feature are not selected by the user, hence during evaluation all the parents from the current feature to the root feature are auto-selected if not selected by the user. If any mandatory association exists between a child feature and a parent which is either "auto_selected " or "user_selected" and the feature is not selected, then the child feature is assigned the enumeration value "auto_selected".

- USER _RE_EXPOSED : This enumeration is used when a user wants to reexpose a feature. Reexposing a feature implies that a feature selection is deferred by one level. The enumeration value is determined by the feature set which were passed during the function and then assigned to it. This enumeration is used only when the user wants to exclusively re-expose a feature.
- USER_AUTO_RE_EXPOSED : This enumeration is used when the evaluation determines that some features have been re-exposed by the user, and because of constraints the feature has to be auto-re exposed in order for the feature model to be valid. This enumeration cannot override any other enumeration and is only used when the auto re-expositions is deemed necessary.

3.3.2 Interface Operations

The proposed interface contains 11 operations, one for concerns (initialize), and 10 for features (getRoot, addFeature, ...). The rest of this subsection describes the functionality / behaviour to be provided. Based on the type of the behaviour of the operation, the operations can be classified to be under one of the categories of initialization, getters, setters or edit.

COREConcern.initialize(String rootFeatureName)

This function is used to create the concern, by initializing everything by the constraints of the meta model (one feature model and one impact model and one root feature). The name of root feature will be the concern name. The name of the root feature is passed as string constant to the underlying implementation so it can instantiate the feature class implementation that subclasses the feature class of the CORE metamodel. The initial impact model has initially no impact model elements in it. This operation can be classified under the category of Initialization.

COREFeature = COREConcern.getRoot ()

This getter function returns the root feature of a concern and is mainly used to traverse the feature model starting at the root. This operation can be classified under the category of Getter.

COREFeature.addFeature(String childFeatureName,COREAssociation parentAssociationKind) :

This interface function adds a child to the parent in the feature model. The function is to be called on the parent. There are two parameters which are passed along with this: the name of the child feature and the association kind of the child with respect to the parent. With the child feature name being passed to the function, the function first creates a COREFeature with the specified name and then sets the parent constraint as indicated in the second parameter. Since the order of the children within their parent has no semantic meaning, the position of the child is not included in the signature of the operation. This operation can be classified under the category of Edit.

COREFeature.delete() :

This interface call is used to delete a feature in a feature model. The delete function is directly called on the COREFeature, which deletes the COREFeature from the feature model and any references from the parent to the feature. The interface function ensures that it is called on leaf nodes only, as it was decided that cascading deletion of features should not be supported in the interface. In case such a functionality is desired, it would be the responsibility of the CORE tool to implement it. This operation can be classified under the category of Edit.

COREFeature.rename(String newName)

This interface call is used to change the name of a feature. The new name of the feature to be renamed is passed as a parameter with the function call. This operation can be classified under the category of Setter.

COREFeature.changeLink(COREAssociation newAssociation)

This interface call is used to change the association type with its parent. Based on the existing relationship / association with its parent, the implementation of this function changes the association of the rest of siblings if needed. If the current association is Mandatory / Optional and is being changed to XOR / OR, the rest of siblings association with respect to the parent are also changed. If the current association is OR/ XOR and is being changed to Optional / Mandatory, then the rest of the sibling association types are changed to Optional. However switching between XOR and OR, or between Optional and Mandatory do not result in any change of association type of the siblings with its parent.This operation can be classified under the category of Edit.

COREFeature.changeParent(COREFeature feature, COREAssociation newAssociation)

This interface call sets the parent of the feature to a new feature. A parameter designating the new feature along with the new association type are passed in the call. The implementation removes the feature from the children set of the current parent and adds the feature to the children set of the new parent. Based on the existing relationship / association with its parent, the implementation of this function changes the association of the rest of siblings if needed. If the current association is Mandatory / Optional and is being changed to XOR / OR, the rest of siblings association with respect to the parent are also changed. If the current association is OR/ XOR and is being changed to Optional / Mandatory, then the rest of the sibling association types are changed to Optional. However switching between XOR and OR, or between Optional and Mandatory do not result in any change of association type of the siblings with its parent. This operation can be classified under the category of Edit.

COREFeature.requires(COREFeature feature)

The interface call is used to to assign a 'requires' constraint to a feature, indicating that it requires another feature passed as a parameter. The requires constraint is uni-directional. This operation can be classified under the category of Setter.

COREFeature.excludes(COREFeature feature)

This interface call is used to assign an 'excludes" constraint to a feature, indicating that it requires another feature to not be selected in the feature model to be valid during evaluation. The excludes constraint is bi-directional. This operation can be classified under the category of Setter.

COREFeature.removeConstraint(COREFeature feature)

The removeConstraint interface call is used to remove a constraint which may exist between two features. The constraint can either be a 'requires' or 'excludes' constraint. The function is called on the feature which contains the constraint. This operation can be classified under the category of Edit.

COREFeature.addRealizedBy((COREModel) Aspect)

This function establishes a realization link between a COREModel and a feature. Each feature can have zero to many realizing models, which can be added sequentially. The implementation of this function takes care of the bi-directionality of the association, i.e., it also updates the *realizes* link of the COREModel that refers to the current feature. This operation can be classified under the category of Setter.

{ Map<COREFeature feature, COREFeatureSelectionStatus selectionStatus> , Map<COREFeature feature, int satisfactionValue> } static select(List<COREFea selected, List<COREFeature> reexposed)

This function is used to evaluate a feature model based on the selected and the reexposed features passed by the user as parameters. The implementation of the function takes into consideration the constraints (parent-child relationships, requires and excludes constraints) in the feature model to evaluate if the current selection is a valid one. The function returns two hashmaps: one pertaining exclusively to the feature model and the other to the impact model.

In the first map, all the features of the concern are returned with their appropriate COREFeatureSelectionStatus value.

In the second map, all the impacts (top-level or intermediary) of the current concern are returned along with their satisfaction values determined using the given feature selection.

The lower layer of the impact models are contributed by the features. A user selected feature is considered as a full selection and contributes accordingly to the higher level parent and then concurrently to all the upper level parents / goals. The contribution of all the high level goals / impacts or all the intermediary impact / goals are then grouped together and then passed back from the function.

3.3.3 Limitations of the Proposed Interface

The interface defined above provides a lot of benefits which enable cross compatibility across different implementations of feature models and impact models to work in the context of CORE in combination with other modelling notations that are corified. However, we discovered several major limitation when we started using the interface.

- Difficulty in navigating the models: Creating an interface for the operations of feature models allowed the flexibility to have a separate implementation in jUCMNav. However to interact with thefeature model in CORE, knowledge of the concrete metamodel of the external implementation was necessary. This creates a dependency on the external implementation, which defeats one of the main reasons for having a separate implementation in the first place.To render CORE completely independent, the interface towards the external implementation would have needed to be extended by adding more helper functions, in particular getter operations.
- Significant implementation effort adapting the external implementation: The decision to interface with an external feature and impact model implementation was motivated primarily because of the expected reduced implementation effort. We discovered in the end that it is not trivial to adapt the external implementation to connect with the proposed interface. In our concrete case, the command pattern library that jUCMNav

used to perform edit operations on feature and goal models was incompatible with a library that our TouchCORE tool depended on. In the end, in order to be able to use the jUCMNav implementation from TouchCORE through the interface, the jUCMNav source code had to be refactored, and most of the editing code rewritten.

- Difficulty in evolving the interface: Model evolution and maintaining consistency between evolving models is a challenge for MDE. Since concern-orientation is still a young development paradigm, the evolution of the CORE metamodel is something which cannot be avoided, e.g., to eliminate existing bugs or incorporate new ideas and trends in concern-oriented software development. With each change in the CORE metamodel for whatever reason, it would most likely require all the underlying implementations for feature models and impact models to be adapted. This is not a viable situation, as it involves escalating costs with each change.
- Cross-referencing between models: With the implementation of feature and impact models being separate, the models are also separately stored when they are persisted. With the current structure of the metamodel, there exist mappings from concrete classes of CORE to concrete classes in the feature model implementation. This results in cross-model references and as a result yields poor portability of concerns. As long as the implementation does not change, no problems occur. However, if users want to load their concerns in a tool that is based on a different feature model / impact model implementation, problems occur. To solve this issue, concerns would have to be restricted to be used only on a specific implementation, or multiple versions of each concern – one for each implementation – would have to be shipped.

Based on all the limitation encountered above, we decide to follow a different approach, i.e., to integrate both the feature model and impact models into the CORE metamodel. By placing the common implementation of feature and impact models inside the CORE metamodel, all modelling tools and notations could share this easily to navigate the feature and impact models. The details of this common implementation is described in the next chapter.

Chapter 4
Feature Model Integration

Based on the limitation described in the previous chapter, it became very evident that having an external implementation of feature models results in a contrived, sometimes even non-efficient solution. Hence, we decided to integrate feature models into CORE. This involved two steps: defining the structure for feature models and integrating it into the CORE metamodel, together with any constraints if needed (see section 4.1), and defining behaviour / operations that implement the basic functionality needed to create and manipulate feature models (see section 4.2).

4.1 Feature Model Structure

4.1.1 Feature Model and Feature Relationships / Constraints

To move the logic of feature models to CORE, the CORE metamodel had to be adapted. The part of the CORE metamodel dealing with reuses was kept untouched.

The COREFeatureModel was added to be concrete class that derives from the abstract class COREModel. A containment link from the COREConcern pointing to FeatureModel was added. With this link, the feature model can be directly accessed from the COREConcern.

COREFeature, a new concrete subclass of COREModelElement, was added, together with a containment link from the FeatureModel to the COREFeatures it contains. To access the root feature of the concern, a new unidirectional link named *root* with multiplicity 1 leads from the COREFeatureModel to the root feature of the feature model

Four additional, reflexive links point from the COREFeature class to itself:

- requires (constraint): This is used to specify the *requires* constraints. The link references all the other COREFeatures which are required by the current COREFeature. Since this can be many COREFeatures, the multiplicity is 0 to many.
- excludes (constraint): This is used to specify the *excludes* constraint. The link references all the other COREFeature which are not to be selected along with the current COREFeature. It can contain many COREFeatures, hence the multiplicity is 0 to many.
- parent: This is used to specify the parent of the feature in the feature model. The parent link is of multiplicity 0 .. 1, indicating that a feature can either have a parent or not, which is respecting the structure of feature model. A feature not having a parent is when the feature is a root feature and does not point to any parent.
- children: This is used to specify the children of the feature in the feature model. The children multiplicity is zero to many.

The parent and children is represented as a bidirectional relationship, indicating that the relationship is traversable in both the directions.

The following attributes were defined for the COREFeature class:

- parentRelationship: The parentRelationship represents the relationship of the child with respect to the parent. It is of type COREFeatureRelationshipStatus, defined in the enumeration above. If the relationship is of None, then it indicates that it is the root feature of the feature model. Optional and Mandatory relationship status can exist independently regardless of the status of the other sibling features. However,

for XOR and OR relationships all siblings under the same parent must have the same value.

The enumeration class of COREFeatureRelationship was added to the metamodel containing the five possible values of NONE, XOR, OR, OPTIONAL and MANDATORY. This relationship defines the association a child holds with its parent. The indiviudal realtionship meaning is as follows :

- NONE - Indicates that the current feature is the root feature of the feature model, and it does not have any parent.
- OR - In this relationship, one or more children have to be selected for the feature model to be valid.
- XOR - In this relationship, exactly one child has to be selected from all the children under the parent.
- Optional - In this relationship, the current child of the parent can be selected or not selected for the feature model to be valid.
- Mandatory - In this relationship, the current child of the parent has to be selected for the feature model to be valid.

The above changes reflect all the adaptions made to the CORE metamodel, which were necessary to fully support feature models

4.1.2 Feature Model Configurations

A COREConfiguration is an object which contains a set of selected and reexposed features of a particular concern.

Each configuration is defined for a particular concern, but if the concern reuses other concerns then it may also contain selections of features in the reused concerns.

The COREConfiguration contains two sets under it, one to represent all the selected features and the other to represent all the reexposed features. The selected and reexposed features can contain features defined in more than one concern, i.e., features which are contained in the reused concerns.

During the concern reuse process, a new configuration is created and stored with the reuse. The metamodel provides support for a user to define and store several configurations for a reuse, e.g., to evaluate different trade-offs. However for a particular reuse, the user can ultimately select only one configuration, which is then used for customizing all the underlying models.

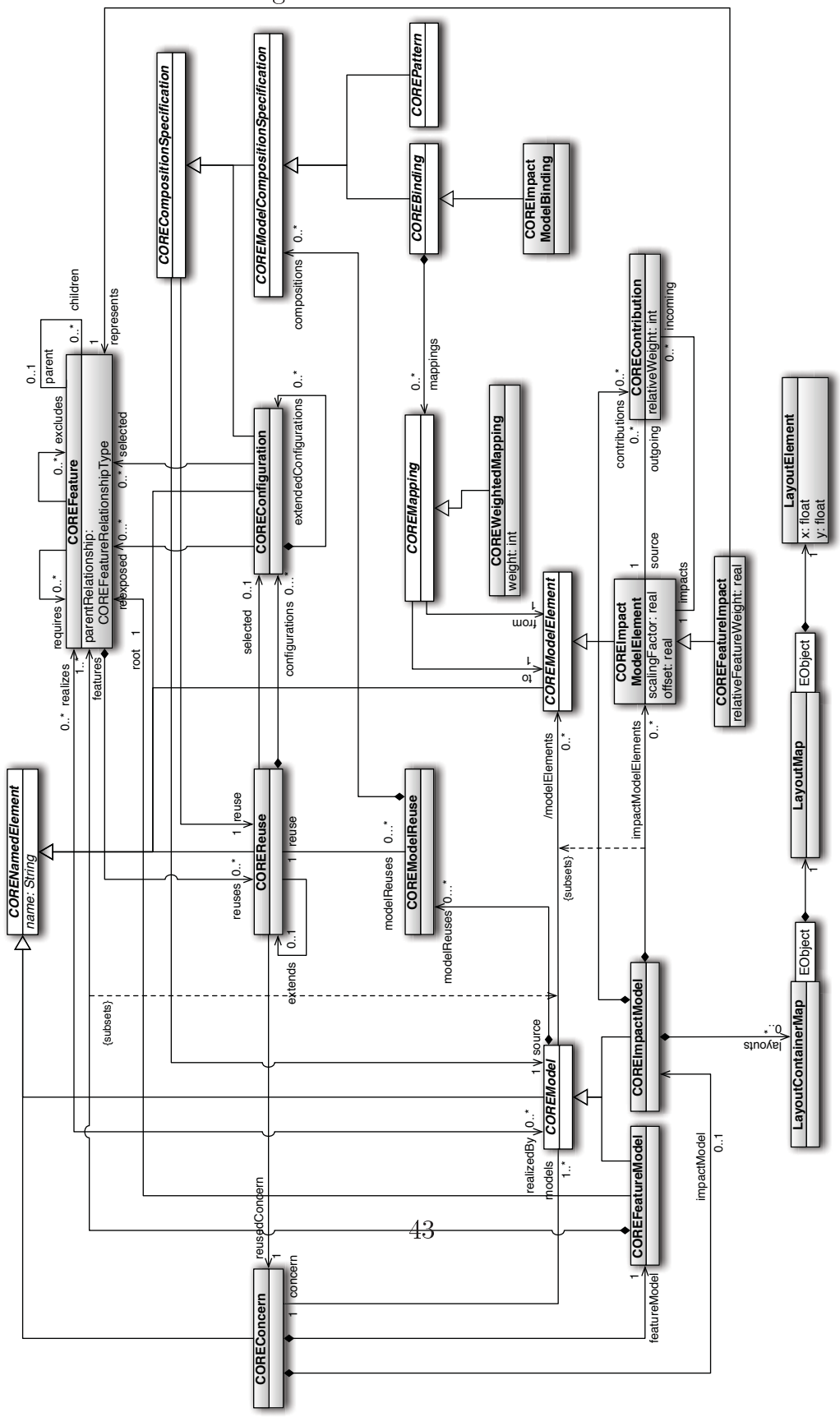
In addition to configuration being defined and stored in the reuse, the concern designer also has a choice to define configuration when the concern is being designed or created. The configuration is defined based on non-functional requirements as the concern designer would have the appropriate knowledge about the concern and its uses. This set of configuration defined during the concern design process is a property of the concern and is accessible via the interface as "default" configurations. These default configurations are also available to the user during the reuse process if the user wants to it use it in his application domain.

For example, a concern which offers high security as a non-functional requirement can offer it in a particular default configuration. It can contain a particular set of selected and reexposed features. Different configurations that optimize other non-functional requirements, such as cost, may also exist for a particular concern.

4.2 Feature Model Behaviour

With the CORE meta model adapted to reflect the required structural changes, the operations defined in the interface in the previous chapter had to be implemented. The

Figure 4–1: CORE Metamodel



following descriptions of the operations abstractly describe the model changes required by the operations.

4.2.1 Feature Model Editing Operations

- Changing the parent of feature: If the user desires to change an existing feature in the feature model to have a different parent, changes to the model have to be performed in order to achieve this. The changes lie in the two links of the COREFeature parent link and the children link. The current feature's parent link is changed to point to the new parent feature. Following this, from the previous parent children set, the current feature is removed. Now the current feature needs to be added to the new parent's children set. By performing these three model changes, the parent of the feature is changed. Additional checks are performed that while changing the parent of a feature, the parent relationship constraint is not violated. For instance, a feature having an XOR parent relationship can be moved to a parent whose children have XOR parent relationship or a "Mandatory" or an "Optional" child can be added to a parent whose children have only "Optional" or "Mandatory" parent relationship.
- Adding a *requires* constraint: If a feature has a requires constraint on any other feature, it is stored in the reflexive "requires" association of COREFeature.
- Adding a *excludes* constraint: To add an *excludes* constraint from a feature to another feature, the other feature has to be added to the reflexive *excludes* association of COREFeature.
- Removing a constraint: A constraint may be of two types, a requires constraint or an excludes constraint. It is impossible to have two constraints between the same two features. Therefore, in order to remove a constraint between a feature and another

feature, the implementation must check if the other feature is contained in either the *requires* or the *excludes* association, and if found, remove it.

- Adding a realization model: Each COREFeature is realized by one or many COREModels, which specify the behavioral and structural properties of the feature implementation. The realization models are stored in the *realizedBy* association between COREFeature and COREModel. Any realization model has to be a subclass of COREModel. Adding a realization model therefore simply requires adding the model to the *realizedBy* set.
- Adding a feature as a child: If a new feature is to be added under a feature as a child, a newly created COREFeature must first be added to the containment association of the COREFeatureModel and then added to the *children* set of the intended parent. In addition to this, the feature's attribute *parentRelationship* must be set to reflect the relationship that it has with its parent. The COREFeatureRelationshipStatus of "NONE" cannot be assigned to any child while adding the feature as a child, as this relationship is only reserved for the root feature and can only be used when the feature model is created. If all the children under the parent either have the relationship with respect to parent as either "Optional" or "Mandatory", a new feature with relationship "Optional" or "Mandatory" is possible. Similarly, if all the children under the parent have the relationship to be either "XOR" or "OR", adding a new feature with relationship "XOR" or "OR"is allowed. However , if an "XOR" or "OR" relationship is added to a parent whose children are either "Optional" or "Mandatory", all the existing children's relationships with their parent are changed to an "XOR" or "OR" relationship, in order to ensure consistency of the feature model. Similarly, if an "Optional" or "Mandatory"

relationship is added to a feature whose children are either "XOR" or "OR", all children are either changed to "Optional" or "Mandatory" based on user preferences.

- Deleting a feature: A COREFeature can partake in several associations. In order to delete a feature, it has to be removed from all associations. If a feature is not a leaf feature, , i.e., the feature is in the mid-section of the feature tree, deleting it triggers a cascading delete of all the children under it. To delete a leaf feature, the feature is removed from the containment association in the COREFeatureModel. The next step is to remove this reference from the children set. After performing these steps, the all the features in the feature models are checked against if the current feature was in the *requires* or *excludes* set of the feature. If the current feature is present in any of the constraint, it is removed from the set.

4.2.2 Feature Model Selection / Evaluation Operations:

The above section presented all the operations which are used when editing a feature model. This subsection presents the operations and algorithms that are needed to evaluate the validity of selections of feature model. This is equivalent to the "select" operation defined in the interface of chapter 3 3.3.2.

The enumeration type <Selection Status> {NOT_SELECTED_ACTION_REQUIRED, NOT_SELECTED_NO_ACTION_REQUIRED, WARNING, USER_SELECTED, AUTO_SELECTED USER_RE_EXPOSED, USER_AUTO_RE_EXPOSED} is used to determine the selection status of a particular feature.

Before the start of the algorithm, all features in the feature model are assigned the value of NOT_SELECTED_NO_ACTION_REQUIRED. This implies that no feature is currently

Algorithm 1 Feature Model Validation

```
evaluate (COREFeatureModel FeatureModel, Set<COREFeature> selectedFeatures, Set<COREFeature>
     reexposedFeatures) {

    evaluateOptionalMandatory(selectedFeatures)
    evaluateORXOR(FeatureModel)
    evaluateReexposedFeatures(reexposedFeatures)
    evaluateConstraint(selectedFeatures)

}
```

Algorithm 2 Optional / Mandatory Evaluation

```
evaluateOptionalMandatory(Set<COREFeature> selectedFeatures) {
    for all feature in selectedFeatures:
        feature.select(USER_SELECT)
}

void select(Selection_Status type){
    if(type == AUTO_SELECT and this.currentStatus == AUTO_SELECT)
        return;
    if(type == USER_SELECT and this.currentStatus == AUTO_SELECT)
        this.currentStatus = USER_SELECTED
        return;
    if (type == USER_SELECT)
        this.currentStatus = USER_SELECTED;
    else
        this.currentStatus = AUTO_SELECTED;
        this.parent.select(AUTO_SELECT);

    for all (child in this.children)  :
        if child.parentRelationship == Mandatory then
           child.select(AUTOSELECT)
}
```

selected and no action is required on any of the features. The evaluation function receives as a parameter the set of features selected by the user.

The evaluation algorithm 2 represents the first stage of the simple evaluation of the feature model. This evaluation is very basic and relies on iterating through each feature and assigning them the user selected values. In this step, the features are determine whether they are

USER_SELECTED and whether they need to be AUTO_SELECTED. At the end of the algorithm, the possible three values for the feature can are NOT_SELECTION_NO_ACTION, USER_SELECTED and AUTO_SELECTED.

The function loops through all the features in the selected feature set and sets their status to USER_SELECTED by calling the inner function *select*. In the inner helper function, based on the value passed as a parameter, sets the feature's selection status, following which two operations are invoked in sequence. The first sequence assigns all the features from the parent up to the root to be AUTO_SELECTED. In the second sequence, the function takes care of all the descendant features of the current feature. If any child of the feature is a mandatory child, it is assigned as AUTO_SELECTED using a recursive call.

In order to avoid repetitive traversal of the feature models, certain break conditions are applied at the start of the *select* function. Two break conditions can be applied: the first one, if the current selection status is AUTO_SELECTED and the current status to be assigned is AUTO_SELECTED, the function is stopped, as all the features above the current feature have already been handled previously. The second break condition is encountered when the current status is AUTO_SELECTED with the current status to be assigned is USER_SELECTED, this signifies that the user has exclusively selected this feature, but it has already been auto selected in a previous traversal, hence only the current selection status of it is changed to USER_SELECTED and the function is terminated.

The second part of the evaluation determines whether if the current user selections is valid, or whether there are any WARNING or SELECTED-NO-ACTION-REQUIRED status to be assigned to any of the features. The below algorithm loops through all the features

Algorithm 3 OR / XOR evaluation

```
1  void evaluateORXOR(COREFeatureModel fm) {
2      for all (feature in FeatureModel):
3          if(feature.currentStatus != NOT_SELECTED_NO_ACTION_REQUIRED): [Can be auto-selected
               or user-selected]
4              continue the loop.
5
6          if(feature.parentRealationship == XOR):
7              if(number of children selected under feature.parent > 1):
8                  for all (child in feature.parent.children ):
9                      child.currentStatus = WARNING
10             if(number of children selected under feature.parent == 0):
11                 for all (child in feature.parent.children):
12                     child.currentStatus = NOT_SELECTED_ACTION_REQUIRED
13         if(feature.parentRelationship == OR) :
14             if(number of children selected under feature.parent == 0) :
15                     for all (child in feature.parent.children):
16                         child.currentStatus = NOT_SELECTED_ACTION_REQUIRED
17 }
```

in the feature model to determine if the current feature model selection violates any OR or XOR constraints.

The above algorithm represents the second part of the evaluation algorithm which determines if the current feature model selection violates any XOR or OR constraints. The function loops through all the features in the feature model, and only proceeds with the functionality if the feature is selected, i.e., user-selected or auto-selected. For the first sequence, the algorithm checks for XOR relationships. An XOR relationship implies that only one child can be selected under the parent. It checks the number of selected children, and if it is more than one, all the selected children are assigned the WARNING status, indicating the feature model selection is currently invalid. If the number of selected children is zero, all the children are assigned the status of NOT_SELECTED-ACTION-REQ, indicating that the user needs to mandatorily make a decision on the children. A similar check is done in case of the OR relationship.

Algorithm 4 Reexposed feature selection validity checker

```
1  void evaluateReexposedFeatures(Set<COREFeature> reexposedFeatures) {
2      for (each feature in reexposedFeatures) :
3          feature. reexposed (USER_REEXPOSE)
4  }
5
6  void reexposed (Selection_Status type) {
7      if(feature already has a status other than AUTO-REEXPOSED || USERREEXPOSED) :
8          return;
9
10     if(type == AUTO-REEXPOSED and feature == AUTO-REEXPOSED ):
11         return;
12
13     if(type == USER_REEXPOSEDand feature == AUTO-REEXPOSED):
14         feature = USER_REEXPOSED
15         return;
16
17     if(type == USER_REEXPOSED) :
18         feature = USER-REEXPOSED
19     else :
20         feature = AUTO-REEXPOSED
21     parent.reexposed(AUTO-REEXPOSED)
22
23     if(any children under feature is Mandatory)
24         mandatoryChild.reexposed  (AUTO-REEXPOSED)
25 }
```

Following the two algorithms, the completeness and the validity of the feature model is checked next. A feature model is considered to be valid, if there are no WARNING or SELECTED-NO-ACTION-REQ status in any of the features in the feature model. If there are any, the feature model is considered to be invalid, which means that no impact model analysis is performed for this invalid selection, as it would most likely return invalid results.

The above two algorithms presented do not take into account the selection validity of reeexposed features. When the user of the concern wants to reexpose certain features whose selection decision is deferred to one level down, these features are passed as a set of reexposed features. . The algorithm presented below is used for the evaluation of features reexposed by the user.

Algorithm 5 Requires and Excludes Constraint Checker Algorithm

```
1  void evaluateConstraint(Set<COREFeature> selectedFeatures){
2      for(each feature in selected ) :
3              for(each requiredFeature in feature) :
4                  if(requiredFeature not present in selectedSet) :
5                  return false;
6
7              for (each excludedFeature in feature) :
8                  if(excludedFeature present in selectedSet) :
9                  return false;
10 }
```

The algorithm 4 is the same as the selection algorithm and performs the same actions, except the first condition is used to break the function if a status already exists for a feature, meaning it has already been evaluated and there is no need to evaluate it again now.

The algorithm 5 presents a simple constraint solver example, which checks exclusively if any of the constraints specified in the feature model are violated.

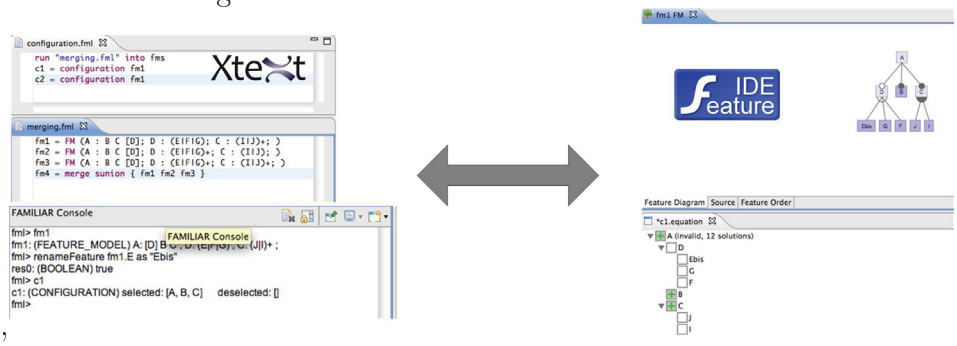
All the features which are selected and assigned, AUTO_SELECTED by the above evaluation algorithms are collected together in a set and then looped through to check for requires and excludes dependency.

4.3 Integration of an External Constraint Solver

The simple feature model selection validation algorithm presented in the previous section 4.2 can check for constraint violations of the features selected by the user. However if a user reexposes some features, the validation algorithm is not be able to check if in the end, when the final selection is made, it is always possible to make at least one valid selection.

This stems from the fact that when a concern designer of concern A reuses a concern B by making a selection of features and reexposing others, the selection decision for the reexposede features is deferred to the next level. The concern A designer can depend on the already selected features of concern B, and he can define constraints between the selected

Figure 4–2: Screenshot of FAMILIAR tool



,

features and optional features of A, if needed. However, at the next level up, when the concern user of A is selecting the features for of A which includes reexposed features from B, making a selection on the reexposed features might result in a constraint conflict with an already selected feature of A. These type of complex constraint conflict detection cannot be detected by our simple verification algorithm presented above, which uses.an iterative algorithm to traverse the feature model. Although it is well-adapted to check for OR and XOR constraints, it is highly inefficient for cross-tree constraints (i.e., includes and excludes).

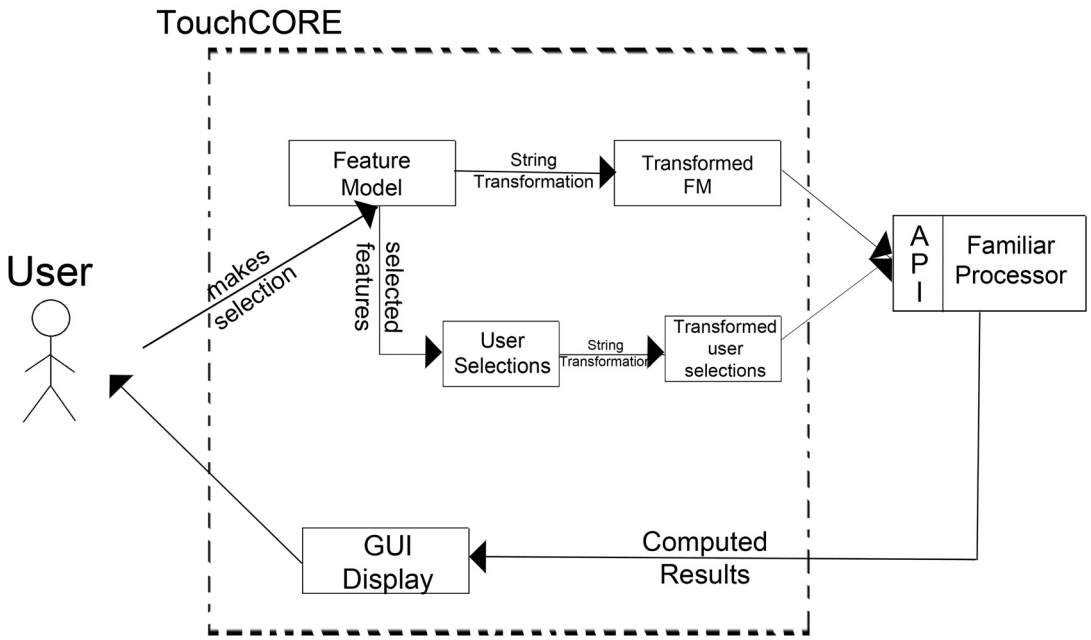
In the context of feature model evaluation, SAT solvers have been used extensively to solve the problem of evaluating complex cross-tree constraints. Typically, techniques using SAT solvers reduce the feature model that is to be analyzed to a propositional formula , which is then analyzed using standard SAT solving techniques.

4.3.1 Integration of FAMILIAR with TouchCORE

As a part of my thesis, the CORE tool which I worked on, TouchCORE, was integrated with a domain specific language called FAMILIAR [1]. .

FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning) is a domain specific language used for a variety of functions with respect to feature

Figure 4–3: Interaction of TouchCORE with FAMILIAR



models. These functions range from composing, decomposing, editing, configuring, etc., multiple feature models. The functions can be used for analysis of feature models and in particular variability management tasks. FAMILIAR also provides range of complex algorithms specified for the validation of constraints in feature model based on SAT solving techniques.

Flow of Control and Exchange of Information

The CORE tool is responsible for visualizing and editing operations of the feature model. The interaction between the CORE tool, TouchCORE and FAMILIAR is as shown in 4–3.

During the concern reuse process, the feature model of the reusing concern is shown to the user. The user makes the selection according to his requirements.

All the feature selections made by the user are composed together and are sent to the evaluation algorithm of FAMILIAR. To this aim, the feature model that in memory is organized according to the CORE metamodel is first converted to the FAMILIAR format, i.e., flattened into a sequence of characters. A similar conversion is done to specify the set of selected features. Then, FAMILIAR is invoked. FAMILIAR provides an API which accepts the feature model and the selected features in a flattened format and checks for constraint violations. It first checks for any parent relationship violation. Any violations would immediately result in stopping the evaluation and returning the violations. Following this, FAMILIAR then transforms the feature model into propositional logic and runs the SAT solver algorithm.

If no solution is found, i.e., some constraints are violated, the violated constraints are returned to CORE. To achieve this, we had to define a data exchange format that FAMILIAR would produce and TouchCORE could read, so that subsequently the detected constraint violations can be displayed in the GUI.

Chapter 5
Feature Model Visualisation

Feature Models are at the heart of Concern-Oriented Reuse, since they capture the relationships and dependencies that exist between distinctive user-visible characteristics of the software that a concern modularizes and encapsulates.. As such, they are extensively used by both the *concern designer* and the *concern user*.

A concern designer is the user who is responsible for creating the generic reusable concern. A concern designer has the most up-to-date knowledge of the domain and has all the necessary background to understand the concern completely. He decomposes the concern into features, thus creating the tree structure of the feature model, and then elaborates the realization models for each feature.

The concern user on the other hand uses an existing concern by selecting features of interest, performing trade-off analysis, and finally by customizing it to his specific application context. Hence at this point it is advantageous to display the most important part of the feature model pertaining to the user, i.e., the decisions that need to be made.

In order to maximally support the task of the modeller, depending on the type of user, certain parts of the feature model become more important than others. Therefore, adapting the way the concern feature model is displayed depending on the user greatly simplifies the model creation process for the concern designer as well as aids in the concern user's decision making process.

Figure 5–1 shows three different concerns and their associated feature models: *Smart Home*, *Authentication*, and *Storage*.

Storage concern presents the various options the user can use for storing information. The presented features are ordered, unordered or by key indexed which can further be specified by a remote lookup or a database *Storage* concern is used at the highest level of abstraction. This concern can be used in any domain / application which needs storage mechanism.

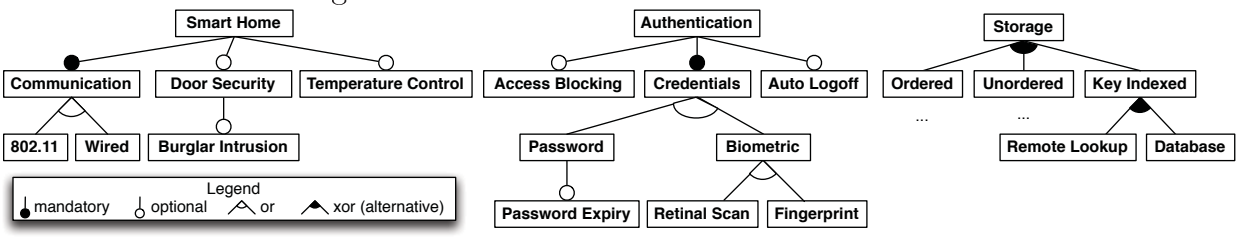
Authentication concern presents the various options that the designer can utilize for authentication purposes. The concern design makes a distinction between how the credentials are stored either by *Password* or by *Biometric* means. Over these credentials, additional mechanisms such as *Access Blocking* and *Auto Logoff* can be attached to it. The *Authentication* concern is used at the middle level of abstraction. This concern can use other concerns and in turn some other concern can use *Authentication* concern in their design.

Smart Home concern presents the design of a modern home system with various option for *Door Security*, *Temperature Control* and the method of communication. This concern is at the lowest level of abstraction as it is defined for a particular domain. These three concerns are not independent, but form a reuse hierarchy. In fact, the realization model of the *Smart Home* concern's feature *Door Security*,reuses the *Authentication* concern, which in turn reuses the *Storage* concern.

In the following sections we describe how we propose these feature models to be visualized depending on who is working with them[1] .

[1] Parts of this chapter of the thesis have been published as a paper entitled "Visualization Algorithms for Feature Models in Concern-Driven Software Development" [41] in the "14th

Figure 5–1: Concern and their Feature Models



5.1 Concern Designer Visualization Algorithm

During the design of the concern, for the concern designer, the feature model can be thought of as a table of contents which organizes all the features being designed, which in turn group all the reuses that this feature makes and all the models that realize this feature. This table of contents encodes in a precise, hierarchical way how features are organized, i.e., how child feature relationships are classified and organized under a parent. In a feature model, a child feature under a parent defines the additional structure and functionality which needs to integrate with the parent's feature structure and behavior. The child feature can contain extra additional constraints which need to be satisfied to work with the parent. Hence, while designing the realization model(s) of a feature in the concern, the concern designer uses the feature model to be aware of what functionality of the parent and ancestor features he can depend on or he needs to integrate with. To elaborate a coherent design, having such a complete picture of what lies "above" a feature within the concern is crucial.

For example in Figure 5–1, the *Burglar Intrusion* feature in the *Smart Home* concern can depend on the structure and behaviour of its parent feature *Door Security* .

~~International Conference on~~ Modularity", co-authored by myself and my supervisor Jörg Kienzle.

In the concern reuse process, whenever a feature reuses another concern, all the child features also have access to the functionality provided by the reuse, as the reuse is considered to be an integral part of the feature that requested it. Therefore it is imperative for the concern designer that he is aware of all the concern reuses of the parent and ancestor features, and hence they should be visualized in the feature model during the design process. In essence, the feature making the reuse imports and groups all the structural and behavioral properties of all the selected features of the reused concern. Hence, to document the fact that any descendant features can depend on the selected features of a reuse made in a feature, the selected features of the reused concern are visualised as mandatory children of the feature that made the reuse. As a result, the concern designer knows precisely which features the descendants can depend on.

The reexposed features are not shown in this mode of display. The main reason for this is that the designer of the concern cannot depend on any of the structure and behaviour of the reexposed features of a reuse, since whether or not this functionality is available in the end has not been decided yet. Furthermore, hiding reexposed features of a reuse in the concern designer visualization mode reduces complexity and visual cluttering.

The *DisplayFMDuringDesign* visualization algorithm that implements the concern designer visualization mode described above is presented in Algorithm 6. The following abbreviations are used:

- fm : Feature Model
- f : Feature
- cf : Child Feature
- c : Concern

Algorithm 6 Concern Designer Visualization Algorithm

```
1  DisplayFMDuringDesign(fm:FM) {
2     fm.root.display()
3     DisplayFDesign(fm,fm.root)
4  }
5
6  DisplayFDesign(fm:FM,f:F) {
7
8    forAll (Reuse r of concern c within f) {
9      DisplayFSelected(f, c.fm.root, Sel(r))
10   }
11   forAll(cf, child of f) {
12     f.display(cf)
13     DisplayFDesign(fm,f,cf)
14   }}
15
16 DisplayFSelected(parent:F,child:F,Selected:Set{F}) {
17   parent.displayMandatory(child)
18   forall(children f of child) {
19     if (f in Selected)
20       DisplayFSelected(child,f,Selected)
21   }}
```

- r : Reuse
- Sel(r) : Set of selected features for the reuse r

In the above algorithm, it is assumed that the root feature of a feature model can be accessed via the property `.root` . The operation `f.display(cf)` displays the feature cf (which is a child) and the relationship link to its parent, and the function `f.displayMandatory(cf)` displays a mandatory dependency link between the feature `cf` and the parent feature `f`.

The above algorithm *DisplayFMDuringDesign* takes as input the feature model to be displayed. This feature model is the feature model of the concern that the concern designer is currently editing. After displaying the root feature of the feature model, the algorithm calls the recursive *DisplayFDesign* operation, passing the feature model and the root feature as a parameter..

DisplayFDesign first loops through all the reuses of the feature passed as parameter, and calls the *DisplayFSelected* operation to display the selected features in the reuse, passing the

current feature, the root feature of the reused concern as well as the set of selected features in the reuse configuration as parameter (see description of *DisplayFSelected* below). Once this is done, the operation continues to display the current feature model by iterating through the children and calling itself recursively.

Inside the *DisplayFSelected* operation, the root feature of the reused concern is first displayed as a mandatory child of the feature that makes the reuse. Then, each child feature of the root feature of the reused concern is checked to determine whether it is also part of the selected features. If yes, the DisplayFSelected operation calls itself recursively. This ensures that all selected features in a reuse are displayed as a mandatory children under the feature that made the reuse.

The current visualization algorithm only crosses the concern boundaries *once*, which essentially shows reuses made only by the models that realises features of the concern are shown. Inner reuses made by the features selected by the concern are not shown. This is consistent with the information hiding principles, which dictate that a concern designer should not depend on the internal design of a concern that is being reused, but only its interface.

Figure 5–2 presents the application of the concern designer visualization algorithm on the *Smart Home* concern. The selected features of the configuration of reuse of the *Authentication* concern are *Access Blocking*, *Credentials* and *Biometric*. These selected features show up as mandatory sub features under *Door Security,* which is reusing the *Authentication* concern. The *Authentication* concern reuses the *Storage* concern, but this is not shown in the designer view, nor the selected configuration of *Storage,* which is the feature *Key-indexed*, because the algorithm crosses concern boundaries only once. As a result, the knowledge of

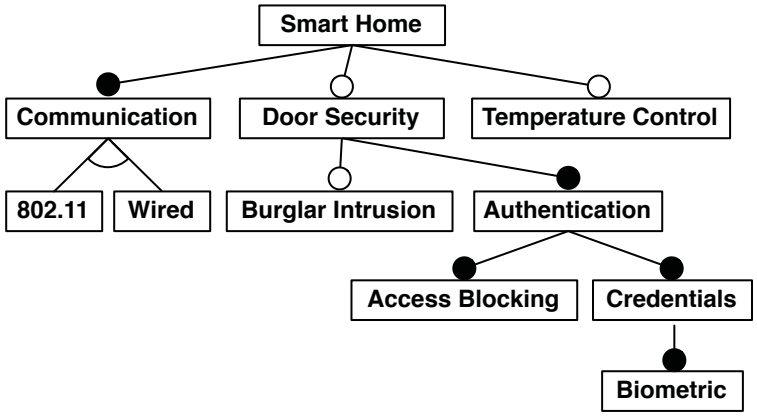


Figure 5–2: Concern Designer Visualization Algorithm applied to the Smart Home Feature Model

Authentication reusing *Storage* remains internal to the *Authentication* concern, and is not be available to the concern designer of *Smart Home*. This is inline with the information hiding principles enforced by concern-orientation.

5.2 Concern User Visualization Algorithm

One of the main goals of concern-orientation is to hide the complexity which exists within a concern from the concern user whenever possible. The internal design and the working of the concern should be abstracted away from the user as much as possible. For instance, a concern user does not need to be aware of the extension hierarchy that exists between realization models that makes it possible to share structural and behavioral design elements among concern features. Also, feature interactions identified by the concern designer should be dealt with in a way that is transparent from the concern user. During the concern reuse process, the concern user needs to focus on the available user features and their impacts, perform trade-off analysis and select features. Ultimately, what is important to be shown to the concern user during the reuse is the available functional variants and the design choices

that the concern has to offer, and to evaluate and display the different impacts that the different available choices have on non-functional properties.

5.2.1 Compact User Visualization Mode

Since the choices are most important during the reuse process, the visualization of the feature model of the reused concern during reuse should be "condensed" to only show the choices that are available to the concern user. For example, in the feature model, it is not necessary for the user to see the mandatory children, but is very essential that the optional, or the XOR or the OR children are shown. If the features inside the reused concern have made reuses that reexpose features, those are also choices that the concern user needs to consider, and hence they should be visualized as well. This visualization should continue down the reuse hierarchy as long as there are reexposed features, since reexposition can cross an unlimited number of reuse layers. To summarize, to maximally focus the concern user on the choices he needs to make, mandatory as well as features of a reused concern that were selected by the concern designer should be omitted from the feature model that is shown to the concern user during the reuse process..

The *DisplayFMDuringReuse* algorithm described in Algorithm 7 receives the feature model and the set of currently selected features as parameters. The root of the feature model is displayed and then an inner function is called.

The inner function receives four parameters as follows :

- attachTo: This parameter refers to the feature under which the features are going to be visualized.
- consider: This parameter indicates which feature is currently being examined for possible selections and reexpositions.

Algorithm 7 Compact User Visualization Algorithm

```
1  DisplayFMDuringReuse(fm:FM,selected:Set{F}) {
2     fm.root.display()
3     DisplayFRexOnly(fm.root,fm.root,selected,{})
4  }
5
6  DisplayFRexOnly(attachTo:F,consider:F,
7      selected:Set{F},reexposed:Set{F}) {
8    selected_flag = false; reexpose_flag = false;
9    if (consider.relationship == OR) {
10     forAll (cf:child of consider) {
11       if (cf in selected) selected_flag = true;
12     }
13     forAll (cf:child of consider) {
14       if(cf in reexposed) {
15         if (selected_flag} {
16           attachTo.displayOptional(cf);
17         } else {
18           attachTo.display(cf);
19         }
20         reexposed_flag = true;
21       }
22     }
23   } else {
24     forAll (cf:child in consider) {
25       if(cf in reexposed){
26         attachTo.display(cf);
27         reexposed_flag = true;
28       }
29     }
30   }
31   forAll (cf:child in consider) {
32     if(cf or descendant of cf in reexposed)) {
33       if(!(cf in reexposed) && reexpose_flag) {
34         attachTo.displayMandatory(cf);
35         DisplayFRexOnly(cf,cf,selected,reexposed);
36      } else {
37         DisplayFRexOnly(attachTo,cf,selected,reexposed);
38       }
39     } elseif (cf in selected or cf.relationship == mandatory) {
40       DisplayFRexOnly(attachTo,cf,selected,reexposed);
41     }
42   }
43   forAll (Reuse r of concern c within consider) {
44     if (descendant of c.root in reexposed) {
45       if (reexpose_flag) {
46         consider.displayMandatory(c.root)
47       }
48       DisplayFRexOnly(c.root, c.root, selected+Sel(r), reexposed)
49     }}}
```

- selected: Set {F}: This indicates the set of selected features in the configuration,
- reexposed: Set {F}: This indicates the set of reexposed features.

It is more elaborate than the feature model visualization algorithm during concern design. In particular, it must detect OR relationships where some children are reexposed and some are selected and convert the selected features to mandatory features, and the reexposed ones to optional features (lines 8 to 24). If a child feature has reexposed descendants and a sibling is reexposed as well, then it needs to be displayed in order to guarantee that the feature model is syntactically correct (lines 35 - 37). Furthermore, if a subfeature is mandatory or selected, then it should not be displayed and can be skipped (lines 39 - 43). Finally, the feature models of reused concerns have to be visualized as well, which means that concern reuses have to be handled recursively as long as they contain any reexposed features (lines 45 - 52).

The algorithm for feature model visualisation for the concern user crosses the concern boundary as many number of times as necessary to show all available choices to the concern user, i.e., as long as there are reuses that reexpose features. Only when a reuse makes a complete selection, i.e., if there are no open choices left, then the algorithm does not need to enter the reused concern.

In Figure 5–3, the *Smart Home* concern reuses *Authentication* which in turn reuses *Storage*. The reuse of *Authentication* reexposed the features *Retinal Scan* and *Fingerprint*. The reuse of *Storage* inside *Authentication* reexposes the features *Remote Lookup* and *Database*, with *Key-indexed* being selected in the configuration. Hence when showing the selections to the user when a user reuses the *Smart Home* concern, *Authentication* is shown as a mandatory subfeature under *Door Security* as described in the algorithm. *Credentials* is shown

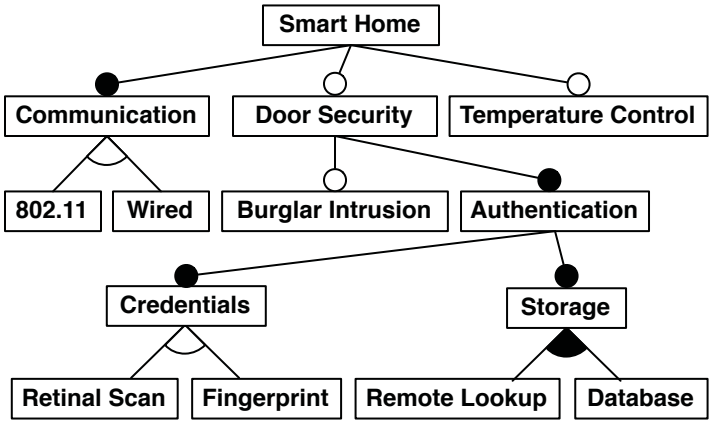


Figure 5–3: Compact User Visualization of the Smart Home Feature Model

as a mandatory child, under which the reeexposed features *Retinal Scan* and *Fingerprint* are shown under the OR relationship. Similarly with *Authentication* reusing *Storage*, the *Storage* feature is shown as mandatory subfeature under *Authentication,* followed by reexposed features *Remote Lookup* and *Database* with an XOR relationship. For this example, the algorithm crosses the concern boundaries twice, since the reuse in *Authentication* has reexposed features of *Storage*, which are again reexposed in the reuse of *Authentication* in *Door Security.*

5.2.2 Verbose User Visualization Mode

The aim of the compact user visualization algorithm presented in the previous subsection is to minimize the cognitive load on the concern user by only showing the features for which decisions need to be made during the reuse process. As a result, the compact user visualization algorithm often does not display many mandatory or already selected features. This happens even if these mandatory or selected features are parents of features that are reexposed. In that case, experience has shown that it is sometimes not obvious for the concern user to understand to which part of the system the features apply. This in turn negatively

Algorithm 8 Verbose User Visualization Algorithm

```
1  forAll (cf:child in consider) {
2      if(cf or descendant of cf in reexposed)) {
3          ancestor.displayMandatory(cf);
4           DisplayFRexOnly(cf,cf,selected,reexposed);
5      } elseif (cf in selected or cf.relationship == mandatory) {
6        DisplayFRexOnly(attachTo,cf,selected,reexposed);
7      }
8    }
```

affects the decision making process. To remedy this situation, we propose a verbose user visualization mode which does not hide the intermediary mandatory nodes from the user.

A minor change to the algorithm in Algorithm 7 in lines 32-37 achieves the desired effect. The condition in the `elsif` statement at line 37 has to be removed, transforming it into an `else` statement. This ensures that all features up to the reexposed feature are shown as mandatory.

Algorithm 8 shows the modified section of the previous algorithm that enables the verbose user visualization mode.

Figure 4 shows the *Smart Home* concern visualized during a reuse with the verbose user vosialization mode. As seen from the figure, all the sub features up to the reExposition are visualized as mandatory children. In this case *Biometric* and *Key-indexed* now appear as children of *Credentials* rsp. *Storage*.

5.3 Order of Visualization

Both modes of the *DisplayFMDuringReuse* algorithm maximally focusses the attention of the concern user to the decisions that must be made. In both modes, the set of decisions that the concern user has to make is shown explicitly. This section proposes two ways of interactively presenting the feature model that constrain the *order* in which a concern user makes decisions as part of the reuse process.

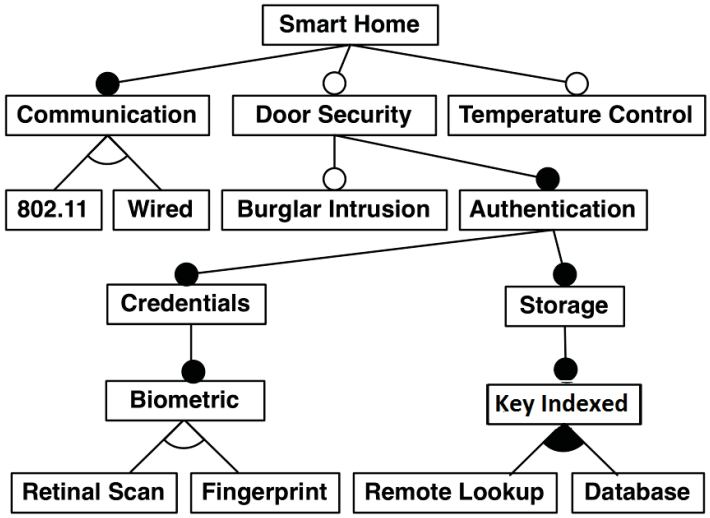
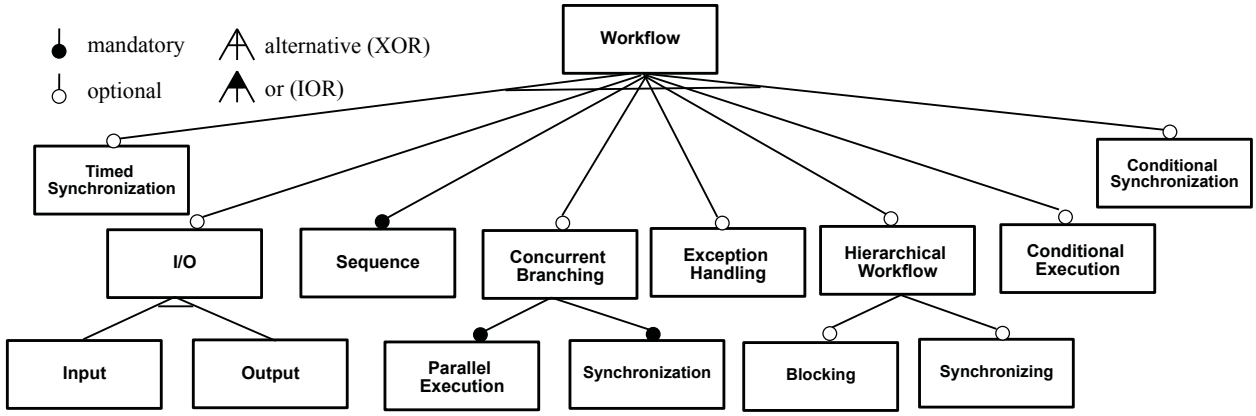


Figure 5–4: Verbose User Visualization of the Smart Home Feature Model



Workflow *reuses* Association<Unordered *reexpose* TreeSet, HashSet>
,
Synchronization *reuses* Association<Ordered *reexpose* ArrayList, FIFOQueue>

Figure 5–5: Workflow Feature Model

,

5.3.1 Top-Down Display

In this mode, the CORE tool prompts the concern user to make decisions in a top down manner. The idea of this mode is to display to the user the choices to be made level by level. In other words, optional children are only displayed once the decision about their parent has been made. This forces the user to make a decision at a particular level, and only then the options for the next level will be shown. This imposes a decision order aligned with the abstraction level of features. General decisions are made before the concern user has to worry about detailed decisions.

The reduction of cognitive load is easily demonstrated by means of the feature model of the Workflow concern shown in Figure 5–5. When reusing *Workflow*, the first level of decisions that the concern user needs to make is to determine which kind of workflow control flow elements he needs: he needs to decide whether *Timed Synchronisation*, *I/O*, *Concurrent Branching*, *Exceptional Handling*, *Hierarchical Workflow*, *Conditional Execution* and/or *Conditional Synchronisation* is needed (the mandatory feature Sequence is always selected, and hence not displayed in the *DisplayFMDuringReuse* algorithm). Only once a feature such as I/O is selected, the sub features that require decisions, in this case *Input* and *Output*, are shown to the user. The same situation occurs for the *Blocking* and *Synchronizing* features under *Hierarchical Workflow*.

5.3.2 Full Display

In this mode of display, the CCD tool from the beginning displays all the choices that need to be made by the concern user. In other words, the tool does not impose an order on the decision making to the concern user. As a result, the concern user can inspect the

feature model from top to bottom, together with all the relationships and cross constraints that exists between features.

In the *Workflow* example shown in Figure 5, this means that the concern user is confronted immediately with all the choices, i.e., *Timed Synchronisation*, *I/O*, *Input, Output, Concurrent Branching*, *Exceptional Handling*, *Hierarchical Workflow*, *Blocking, Synchronizing, Conditional Execution,* and *Conditional Synchronisation.* Additionally, there are also decisions that need to be made for the reexposed features *TreeSet*, *HashSet*, *ArrayList* and *FIFOQueue.*

The advantage of the top-down display is that it again reduces cognitive load. The concern user does not have to be aware of the existence of variants that determine low-level details when it is undecided if those details are relevant for his particular use of the concern. For *Workflow*, the concern user is initially confronted with 9 independent choices when using top-down display, instead of a total of 15 in full display mode.

The advantage of full display is that the concern user has a complete overview of the features that the concern offers before making any decisions. For example, the concern user would see that the *Hierarchical Workflow* feature offers hierarchical workflow decomposition with synchronization, which is an alternative to Conditional Synchronization and additionally provides structural decomposition benefits when creating workflows.

Chapter 6
Related Work

This chapter presents an overview of research and tools that are related to this thesis. Section 6.1 summarizes the modelling notations and modelling approaches related to CORE. Section 6.2 presents the different variations of feature models that have been proposed in the literature, describes the modelling tools with support for feature modelling and comments on their ways of visualizing variability.

6.1 Concerns and Aspect-Oriented Modelling Approaches

Multi-dimensional separation of concerns (MDSoC) was first introduced conceptually by Tarr et al. in [40]. The units of reuses (e.g., classes) were grouped together in modules to address a particular concern. The programming language HyperJ [34] is a prototype extension of the Java that realized the MDSoC vision at the programming language level. The MDSoC inspiration lead to the development of ModelSoC [24], an abstract MDE framework in which development is organized around concerns, and models are developped that specify the concerns by means of orthogonal views.

There are many other MDE approaches which contain units of development which encapsulate cross-cutting concerns. Concerns are separated at design and requirements level with Theme [6]. ArchEvol [32] manages concerns at a higher level of abstraction and can trace concerns back to code level as well.

Several MDE approaches use aspect-oriented techniques to support separation of concerns. [39]proposes a modelling framework which simplifies both the tasks of model development and specifying transformations. Generic Reusable Concern Compositions (GReCCo) [21] proposes a composition method which can be used to compose concern models specified in UML.

In MDSoC, ModelSoC and the aforementioned approaches inspired by aspect-oriented modelling techniques, though, the word *concern* is used in a more narrow sense than in CORE, as the modules in MDSoC typically only dealt with a specific solution for a concern, and did not provide capabilities for impact analysis. In CORE, a concern is a unit of reuse that has a broader perspective, since it additionally modularizes all relevant variants of a development concern and exposes them in a formal variation interface that allows the developer to perform trade-off analysis.

6.2 Variability Modelling

Software Product Line engineering [44] is a research area that focusses on reuse of development assets in the context of a closed family of products. Various SPL approaches have been proposed in the past that use different modelling notations to express the variability within a family of products.

Feature models, first introduced by Kang et al. [25] in Feature Oriented Domain Analysis (FODA), express the variability in a top down tree like format with different parent child relationships between the features. Many variants of feature models exist, and they are discussed in Section 6.3. Decision Modelling [10] uses sets of decisions to separate different products in a software product line and has mainly been applied in an industrial setting. Orthogonal Variability Management [36] documents the variability within a product line with

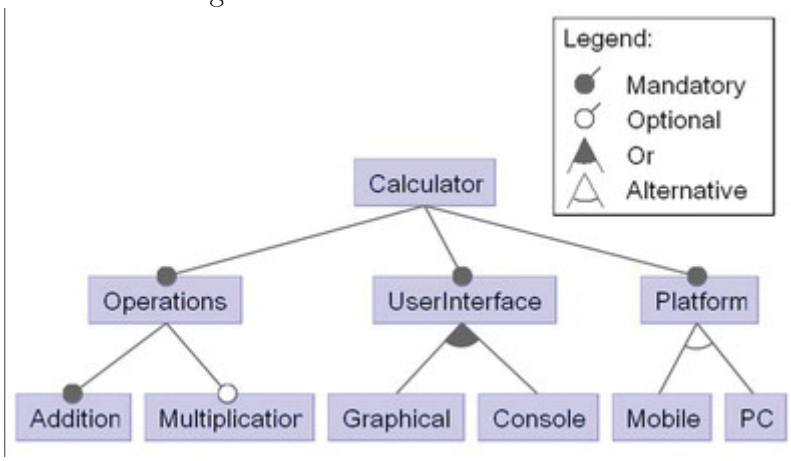
variation points. Each variation point describes the possible configurations and associated constraints.

The Common Variability Language (CVL) [20] is a domain independent language for specifying and resolving variabilities. Just like UML it is an OMG standard. At the heart of CVL variation points are defined over base models, which can be of different types depending on the kind of variation that is to be expressed. Every variation point is bound one variation specification which describe the abstract variability. The variation specification can be used to specify a constraint.

Additionally to specifying variability, CVL also provides mechanisms for operationalizing how a product is derived. This process is called resolving the variability. Each type of variation specification has its own type of resolution. The variation specification contains different information necessary to materialize the product model. Choices represent a yes / no decision, Variables contain the actual values defined and Classifiers imply the creation of instance.

In CORE, the variability is expressed in terms of feature models, since they are the most widely used formalism for expressing variability, and on top of that very intuitive and easy to understand. Every feature specified in the feature model is represented by a realization model which represent the behaviorial and structural aspects of the feature. These features are mapped to their respective realization models. Based on the configuration, all the models that realize the selected features to yield new models corresponding to the desired configuration. In CVL, based on all the variation specification, variability is resolved to formulate the final product. In contrast to CORE, where variations is defined by the

Figure 6–1: Basic Feature Model



configurations which the user can select, CVL defines different variation specification which represent different variations to derive the final product.

6.3 Feature Models

The application of feature models varies across different fields, such as, but not limited to, model driven engineering, feature-oriented programming [31], and reuse applications [29]. All the applications revolve around software product lines. Feature models can also be used in all phases of the software development cycle – from requirements gathering to design.

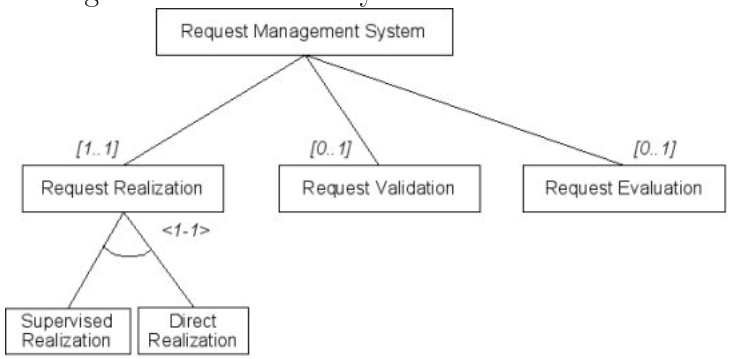
6.3.1 Feature Model Types

Although there exist many different types of feature model languages, we present here the three most common ones.

Basic Feature Models

Basic feature models consists of feature models which consists of the four relationships of Optional, Mandatory, Alternative and Or. Basic feature also allow to specify cross tree constraints in the form of requires and excludes.

Figure 6–2: Cardinality Based Feature Model



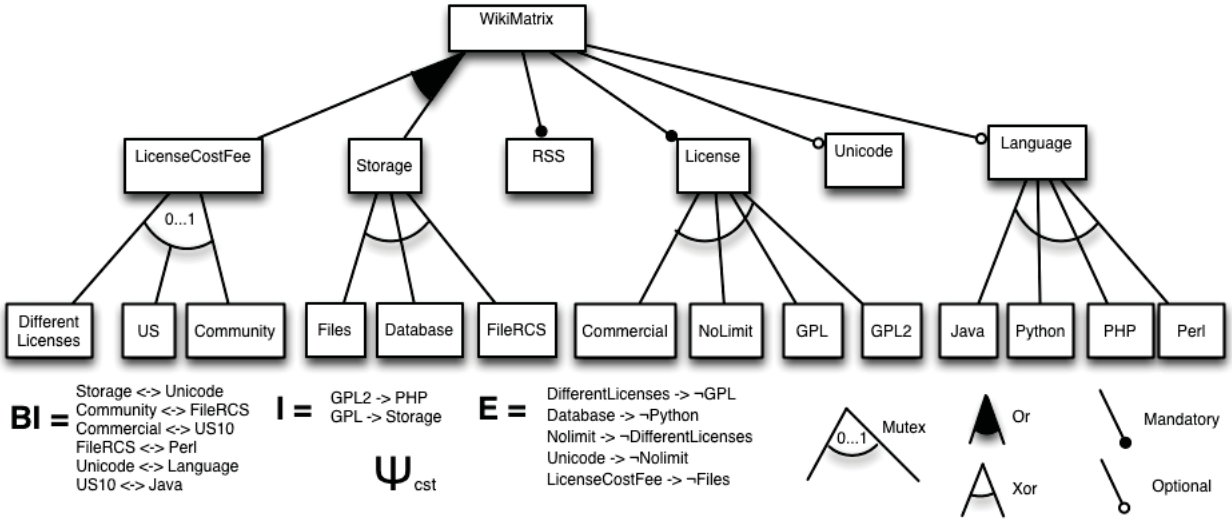
Many researches have worked on expanding basic feature models to improve their capabilities for analysis. For instance, Don Batory [7] introduced the concept of applying proposition logic to feature diagrams to enhance the capability to debug feature models. So far, CORE uses basic feature models only.

Cardinality-Based Feature Models

Cardinality-based feature models refers to integrating a number of extensions to the original FODA specification. It is used by some authors to allow features and feature groups to be selected more than once, and to express additional constraints [14]. This extension was mainly inspired by practical application and conceptual completeness.

In cardinality-based feature models, cardinality refers to the minimum and the maximum number of children a parent feature can contain. It is defined by an interval denoted by [n..m], where n represents the lower bound and m represents the upper bound.

Figure 6–3: Extended Feature Model



Extended Feature Models

Due to the limited amount of information which can be represented and encoded directly in a feature model, several approaches have augmented feature models with additional information. These types of feature models are called *extended feature models*. Typically, the additional information is added in terms of so called *feature attributes*.

Extended feature models are sometimes used to express additional cross-tree constraints or to aggregate the existing relationships [2]. Most often though they are used to perform some form of analysis or reasoning to inform the decision making process to compare different configurations. An example of such an analysis is presented in [8], where the number of potential products a FM contains is estimated so as to determine if the SPL can become more flexible.

6.3.2 Feature Model Implementations / Techniques

FAMILIAR[1] is a Domain Specific Language which is used for management of feature models. The language provides support for separation of concerns. It implements extended feature models and provides support for feature model composition. Constraints can be specified using the language, and it also provides support for evaluating the validity of a selection of features in a given feature model.

FeaturePlugin [4] is an Eclipse [5] based plugin for feature modelling. It supports cardinality-based feature modelling with constraints. Users can group together a set of features and define a configuration to *specialize* a feature diagram. With these specializations, additional analysis can be performed on feature models.

Text Variability Language (TVL) [9] is a proposed text-based variability language with C-like syntax aimed for easier adoption of feature models in an industrial context. The goal of TVL is to be scalable and to support modularity. TVL provides support for all types of feature models, including basic and cardinality-based feature models, which additional support for constraint specification.

[11] proposes a XML-based feature modelling technique which aims at modelling the software assets behind a product line. The technique provides the ability to decompose feature models to extensible and self-contained modules. It uses a XML based technique to express feature models by which any tools can easily derive support.

6.4 Work Related to Impact Models

In addition to expressing the variations offered by a concern, the quality attributes for features are specified in an impact model. An impact model is a type of goal model which specifies the positive / negative impacts on various high level goals and quality aspects.

The analysis of impact model offers provides guidance to the decision making process which identifies the best suited solution. In other words it can be considered as a model which describes the advantages and disadvantages of features of a concern and their impacts on high-level goals.

Impact model i.e., the impact of choosing a feature, can be specified with goal models. Goal models can be described with GRL, which is a part of User Requirements Notation (URN) standard [22], or the NFR framework [13], i* [45] or KAOS [15].

The impact models used in the concern are rarely used in isolation. With the reuses of concern, the impact models are also reused which results in composition of impact models. Feature models have the ability to specify constraints. With the reuse of concerns, the maximum achievable value which a goal might attain can significantly alter.

During my thesis, I also contributed to [17]. It specifies the proposal for the reuse of goal models which have received little attention. The work also presents an evaluation algorithm which calculates the maximum value a goal might attain with the constraints specified on features, when a concern reuse is applied.

Chapter 7
Conclusion

In this thesis we have investigated different ways of operationalizing feature models, with the aim of creating the first CORE-based modelling tool.

Based on the existing CORE metamodel, this thesis first proposed an interface which links an external implementation of the feature model to the CORE tool. The interface defines 11 operations (initializers, getters, setters and complex edit operations), one for the concern and ten that operate on features. The proposed approach and interface were tested by linking the TouchCORE tool with an external implementation of feature models of the Eclipse-based modelling tool jUCMNav. Implementing this approach resulted in the discovery of certain limitations, such as difficulty in navigability, problems with cross-referencing between model elements, difficulties with evolution of the external tool, problems separating the implementations, incompatible library dependencies,etc.

As a result, feature models were subsequently integrated directly into CORE. This was performed by defining the structure of feature models, i.e., features, parent-child relationships and inter-feature constraints, as well as feature configurations and integrate them into the CORE metamodel. This was followed by defining behaviour / operations that implement the basic functionality needed by CORE to create and manipulate feature models. The TouchCORE tool was updated with the new metamodel, the editing operations implemented and a graphical user interface for feature models was devised. Additionally an external

constraint solver was integrated with TouchCORE to determine the validity of partial feature selections.

Finally, in order to maximally support the creation and reuse of concerns, two visualization modes for feature models are proposed. *Concern designers* are users who are responsible for creating generic reusable concerns, and to streamline their interaction with the tool our algorithm shows all the features of the concern being developed. Additionally, any reused concerns and their selected features are visualized as mandatory child features of the feature that made the reuse in the current concern. This gives the concern designer all the information he needs to properly integrate the models he develops with the structure and behaviour of the current concern, in particular the parent features and the reuses they made.

On the other hand, *concern users* use an existing concern by selecting the desired features, performing trade-off analysis and finally by customizing it to his application context. For the concern user our algorithm presents a condensed feature model that includes all the choices that the user has to make, including reexposed choices of internally reused concerns, if any. Any features that have to be reused in any case (i.e., the mandatory and already selected ones) are hidden from the user, since no decision needs to be made. This significantly simplifies the reuse process. Furthermore, we propose two different selection orders to the concern user. Full display involves showing the entire feature model of the concern that is being reused including re-exposed features of lower-level concerns to the user, which provides him with a condensed view of all options and allows him to try out the different selections to compare their impacts. Top-down display presents the reuse choices to the concern user layer-by-layer, starting with high-level decisions. Only when the user has made a decision by selecting a feature, the next layer, i.e., the sub features, are visualized. In this mode, only

the user in confronted with the minimal set of decisions that need to be made, thus reducing cognitive load.

7.1 Future Work

Thanks to the work performed in this thesis, concern-oriented reuse is now operational. It is now possible with the TouchCORE tool to create reusable concerns, and realize the design of each feature with design models using the (previously existing) support for class diagram, sequence diagram and state diagram editing and weaving. This now opens many exciting possibilities for further research, two of which are explaining here in more detail.

7.1.1 Full Integration of Impact Models

During the thesis, due to the shortcomings of the external interface approach, feature models were integrated into the CORE metamodel, but not impact models. In parallel to this work, a Ph.D. student started adding limited impact model support to CORE. Using this initial integration, we worked on investigating how to propagate impacts of a reused concern in the variation interface of the reusing concern, which resulted in a publication at SDL [17]. However, currently only very simple impact models are supported in CORE. Goal modelling, the notation that impact models are based on, supports many more advanced concepts, such as stakeholders, AND and OR goal decomposition, key-performance indicators (KPIs), etc., and it would be interesting to investigate if they would be useful in the context of CORE.

7.1.2 COREification of Additional Modelling Notations

Following the ideas of MDE, CORE has been designed in such a way that concerns can encapsulate models using different modelling notations. In theory, in order to be usable in the CORE framework, a modelling notation has to extend the CORE metamodel to include any CORE concepts that are missing in the modelling language and/or align any

similar concepts that already exist in the modelling language with CORE. Once this is done, realization models can be created in the modelling notation and attached to features of a concern. Thanks to this thesis, this theory can now be put to the test by making an additional modelling notation available to the users of TouchCORE. A project attempting to integrate support for the Aspect-Oriented User Requirements Notation (AoURN) has just begun.

References

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, 78(6):657–681, 2013.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6):657–681, 2013.

[3] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Concern-oriented software design. In *16th International Conference on Model-Driven Engineering Languages and Systems - MODELS 2013*, volume 8107 of *Lecture Notes in Computer Science*, pages 604–621. Springer, 2013.

[4] Michał Antkiewicz and Krzysztof Czarnecki. Featureplugin: Feature modeling plug-in for eclipse. In *The 2004 OOPSLA Workshop on Eclipse Technology eXchange - Eclipse '04*, pages 67 – 72, Vancouver, British Columbia, Canada, 2004. ACM Press, ACM Press.

[5] Michal Antkiewicz and Krzysztof Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72. ACM, 2004.

[6] Elisa Baniassad and Siobhan Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the 26th International Conference on Software Engineering*, pages 158–167. IEEE Computer Society, 2004.

[7] Don Batory. *Feature models, grammars, and propositional formulas*. Springer, 2005.

[8] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Advanced Information Systems Engineering*, pages 491–503. Springer, 2005.

[9] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing tvl, a text-based feature modelling language. In *Proceedings of the Fourth International*

Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January, pages 27–29, 2010.

[10] Neil Burkhard. Reuse-driven software processes guidebook. version 02.00. 03. 1993.

[11] Vaclav Cechticky, Alessandro Pasetti, O Rohlik, and Walter Schaufelberger. Xml-based feature modelling. In *Software Reuse: Methods, Techniques, and Tools*, pages 101–114. Springer, 2004.

[12] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability management in software product lines: a systematic review. In *Proceedings of the 13th International Software Product Line Conference*, pages 81–90. Carnegie Mellon University, 2009.

[13] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer, 2000.

[14] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, 10(1):7–29, 2005.

[15] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[16] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[17] Mustafa Berk Duran, Gunter Mussbacher, Nishanth Thimmegowda, and Jörg Kienzle. On the reuse of goal models. In *SDL 2015: Model-Driven Engineering for Smart Cities*, pages 141–158. Springer, 2015.

[18] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering*, FOSE '07, pages 37–54. IEEE, 2007.

[19] Timothy J Grose. Eclipse modeling framework, 2008.

[20] Øystein Haugen, Andrzej Wąsowski, and Krzysztof Czarnecki. Cvl: common variability language. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*, pages 266–267. ACM, 2012.

[21] Aram Hovsepyan, Stefan Van Baelen, Yolande Berbers, and Wouter Joosen. Generic reusable concern compositions. In *Model Driven Architecture–Foundations and Applications*, pages 231–245. Springer, 2008.

[22] International Telecommunication Union (ITU-T). Recommendation Z.151 (10/12): User Requirements Notation (URN) - Language Definition, approved October 2012.

[23] ITUT ITU-T and Z Recommendation. 151 (11/08), user requirements notation (urn)–language definition. *Geneva, Switzerland, approved November*, 2008.

[24] Jendrik Johannes and Uwe Aßmann. Concern-based (de) composition of model-driven software development processes. In *Model Driven Engineering Languages and Systems*, pages 47–62. Springer, 2010.

[25] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[26] Jacques Klein and Jörg Kienzle. Reusable Aspect Models. In *11th Aspect-Oriented Modeling Workshop, Nashville, TN, USA, Sept. 30th, 2007*, September 2007.

[27] Jacques Klein and Jörg Kienzle. Reusable aspect models. In *11th Aspect-Oriented Modeling Workshop, Nashville, TN, USA*. Citeseer, 2007.

[28] Uwe Laufs, Christopher Ruff, and Jan Zibuschka. Mt4j-a cross-platform multi-touch development framework. *arXiv preprint arXiv:1012.0467*, 2010.

[29] Kwanwoo Lee, Kyo C Kang, Wonsuk Chae, and Byoung Wook Choi. Feature-based approach to object-oriented engineering of applications for reuse. *Software-Practice and Experience*, 30(9):1025–1046, 2000.

[30] Andreas Metzger and Klaus Pohl. Variability management in software product line engineering. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 186–187. IEEE Computer Society, 2007.

[31] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 127–136. ACM, 2004.

[32] Eugen C Nistor, Justin R Erenkrantz, Scott A Hendrickson, and André Van Der Hoek. Archevol: versioning architectural-implementation relationships. In *Proceedings of*

the 12th international workshop on Software configuration management, pages 99–111. ACM, 2005.

[33] University of Ottawa. jucmnav version 6.0. Accessed: 2010-09-30.

[34] Harold Ossher and Peri Tarr. Hyper/j: Multi-dimensional separation of concerns for java. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 734–737, New York, NY, USA, 2000. ACM.

[35] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[36] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[37] Jean-François Roy, Jason Kealey, and Daniel Amyot. Towards integrated tool support for the user requirements notation. In *System Analysis and Modeling: Language Profiles*, pages 198–215. Springer, 2006.

[38] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39:41–47, 2006.

[39] Arnor Solberg, Devon Simmonds, Raghu Reddy, Sudipto Ghosh, and Robert France. Using aspect oriented techniques to support separation of concerns in model driven development. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, volume 1, pages 121–126. IEEE, 2005.

[40] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

[41] Nishanth Thimmegowda and Jörg Kienzle. Visualization algorithms for feature models in concern-driven software development. In *Companion Proceedings of the 14th International Conference on Modularity*, pages 39–42. ACM, 2015.

[42] TouchCORE. Touchcore version 6.0. Accessed: 2010-09-30.

[43] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 249–262, Washington, DC, USA, 2001. IEEE Computer Society.

[44] David M Weiss et al. Software product-line engineering: a family-based software development process. 1999.

[45] Eric Yu. Modelling strategic relationships for process reengineering. *Social Modeling for Requirements Engineering*, 11:2011, 2011.