Concern Oriented Reuse: A Software Reuse Paradigm

Omar Alam

Doctor of Philosophy

School of Computer Science

McGill University

Montreal, Quebec

2015-12-7

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

DEDICATION

This thesis is dedicated to ...

ACKNOWLEDGEMENTS

There are many people I would like to thank for their support during my PhD studies. First and foremost, my sincere gratitude is due to my supervisor Prof. Jörg Kienzle for his academic support. Thanks Jörg, for your mentorship and true sense of humor :-) You made my doctoral studies intellectually stimulating, interesting, and fun. Thanks also for your advice in scholarship applications in the beginning of my studies, and job applications now. Special thanks to Prof. Gunter Mussbacher for feedback, discussions, and collaboration. I also would like to thank my thesis examiners, Prof. Muthucumaru Maheswaran and Prof. Houari Sahraoui for taking the time out of their schedules to read my thesis. Additionally, I would like to acknowledge the examiners of my PhD proposal, Prof. Jörg Kienzle, Prof. Gunter Mussbacher, Prof. Hans Vangheluwe and Prof. Luc Devroye, for their valuable feedback. Thanks to the members of the Software Engineering Lab and the TouchCORE team. Specially, Matthias Schöttle, Wisam Al Abed, Nishanth Thimmegowda, Céline Bensoussan, Julien Gascon-Samson, and Berk Duran. Their team spirit and true sense of camaraderie served to create a productive and dynamic work environment. I also would like to thank my friend Shabir Mamodraza for helping me in translating the thesis abstract to French. In the beginning of my doctoral studies, I worked with Jörg Kienzle and Gunter Mussbacher in designing the Workflow middleware, and later with Benoit Combemale Olivier Barais to further investigate using the Common Variability Language (CVL) with workflow models. It was during the time I spent working with them, and with the TouchRAM team, the ideas of incremental modeling and concern-orientation started to cook in my head. Therefore, I would like to

express my appreciation for their collaborations and discussions. Furthermore, I would like to acknowledge all my collaborators. I would like to specially mention Jörg Kienzle, Gunter Mussbacher, Philippe Collet, Matthias Schöttle, Benoit Combemale, Wisam Al Abed and Nishanth Thimmegowda. I came to Canada seven years ago as a foreign graduate student. Today, I am finishing my PhD as a Canadian citizen. I formed deep bonds and developed valuable friendships with so many people here from different walks of life. I would like to acknowledge anyone who gave me an advice, a word of support, or a nice company. Thanks also to the Natural Sciences and Engineering Research Council of Canada (NSERC) for financially supporting my studies. Finally, this thesis would not be possible without the support from my family and friends. Their love and enouragement are too precious to be recorded in words.

ABSTRACT

Model reuse remains a major challenge in Model Driven Engineering (MDE), despite the success stories in programming languages as exemplified by class libraries, services, and components. Modellers usually create models from scratch because modelling languages offer limited support to reuse existing models and modelling tools in general are not shipped with a library of reusable models. In addition, the crosscutting nature of software development concerns complicates the application of software engineering techniques such as information hiding, decomposition, interfaces, and abstraction in the context of MDE. This thesis mitigates the aforementioned challenges that reuse faces in the context of MDE by proposing Concern-Oriented Reuse (CORE), a novel reuse paradigm that extends MDE with best practices and techniques from advanced modularization and separation of concerns (SoC), goal modelling, and Software Product Lines (SPL). CORE advocates the use of a three-part interface to describe a new unit of reuse called concern that spans multiple development phases. The variation interface describes provided choices and their impact on system qualities. The customization interface allows adapting a chosen variation to a specific reuse context, while the usage interface defines how a customized concern may eventually be used. The thesis lays the foundation of CORE by defining its concepts, a simple three-step reuse process, a metamodel, and composition algorithms to generate realization models based on feature selections and customization mappings. The CORE approach is validated in multiple ways. An extensive literature review compares the concern as a unit of reuse to other reuse units, highlighting its strengths. The effectiveness and practicality of the proposed CORE metamodel is

validated by extending an existing modelling language, Reusable Aspect Models (RAM), to support concern-orientation. The feasibility of the proposed composition algorithms is demonstrated by implementing them within the TouchCORE tool. The effectiveness of the CORE design and reuse process is shown by means of several case studies: the design of a reusable workflow concern as well as a family of Crisis Management Systems. We envision that if CORE is adopted on a large scale, it has the potential to transform the software engineering discipline as a whole. Unlike the current practices that often require software engineers to deal with and be an expert in many concerns simultaneously within each software development phase, CORE would enable software engineers to specialize, i.e., to become concern specialists. Companies could focus on creating long-lived concern libraries, and provide consulting services to customize concerns to specific application context, if necessary. Ultimately, concern reuse, concern libraries, CORE-based tools, and specialization will bring software engineering practices closer to what is done in other engineering disciplines.

RÉSUMÉ

La réutilisation de modèle reste encore un défi majeur dans le MDE, malgréles différents succès dans les langages de programmation, illustré par des exemples tels que les bibliothèques de classe, les services, et les composants. Les modélisateurs créent généralement des modèles à partir de zéro, car les langages de modélisation offrent un support limité pour réutiliser des modèles existants; de plus, les outils de modélisation en général ne sont pas livrés avec une bibliothéque de modèles réutilisables. Mais aussi, la nature transversale des questions sur le développement des logiciels complique l'application des techniques de génie logiciel tels que cacher l'information, la décomposition, les interfaces et l'abstraction dans le contexte de la MDE. Cette thèse atténue les défis mentionnés auparavant, qui réutilisent les faces dans le contexte de la MDE en proposant le CORE, un nouveau paradigme de réutilisation qui étend la MDE avec des meilleures pratiques, des techniques de modularisation de pointe ainsi que la séparation des préoccupations (SoC), le but de la modélisation, et lignes de produits logiciels (SPL). Le CORE préconise l'utilisation d'une interface en trois parties pour décrire une nouvelle unité de réutilisation appelée « concern » qui couvre plusieurs phases de développement. L'interface de variation décrit les choix fournis ainsi que leurs impacts sur les qualités du systéme. L'interface de personnalisation permet d'adapter une variation choisie á un contexte de réutilisation spécifique, tandis que l'interface d'utilisation définit comment une concern personnalisée peut éventuellement être utilisé. La thèse établie les bases du CORE en définissant ses concepts, un processus de réutilisation simple en trois étapes, un méta modèle, et des algorithmes de composition pour générer des modèles de reálisation basésur la sélection

des fonctions et des interfaces de personnalisation. L'approche du CORE est validée de multiples façons. Un examen approfondi de la littérature compare la concern comme une unité de réutilisation à d'autres unités de réutilisation, en soulignant ses points forts. L'efficacité et l'utilité du méta modèle proposé (CORE) est validé par l'extension d'un langage de modélisation existant, le (modèles aspect réutilisables) RAM, pour supporter la concern-orientation. La faisabilité des algorithmes de composition présentés est démontrée par leur application au sein de l'outil TouchCORE. L'efficacité du processus de conception et de réutilisation du CORE est affichée au moyen de plusieurs études de cas : l'élaboration d'une concern d'un flux de travail réutilisable ainsi que d'une famille de systèmes de gestion de crise. Nous prévoyons que si CORE est adopté sur une grande échelle, il a le potentiel de transformer la discipline du génie logiciel dans son ensemble. Contrairement aux pratiques actuelles qui nécessitent souvent des ingénieurs logiciels ou des expertises concernant moult préoccupations simultanément dans chaque phase de développement, le CORE permettrait aux ingénieurs logiciels de se spécialiser, par exemple, dans le domaine des préoccupations. Les entreprises pourraient se concentrer sur la création de bibliothèques de concern à long terme, et, si nécessaire, de fournir des services de consultation pour personnaliser les préoccupations au contexte spécifique de l'application. En fin de compte, la réutilisation de la concern, les bibliothèques de concern, les outils basés sur CORE, ainsi que la spécialisation rapprocheront la pratique actuelle du génie logiciel aux pratiques dans les autres branches de l'ingénierie.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Co-Authorship and Related Publications

This thesis is based on several publications co-authored with my supervisor Jörg Kienzle. Some publications are co-authored with other researchers. Here is the list of papers included in this thesis (in all these papers, I was the primary author):

- Incremental software design modelling. **O Alam**, J Kienzle, CASCON, 325-339 (2013). (Parts of Chapter 8 are based on this paper).

In this paper, I proposed the idea of incremental modeling in software design and incrementally modeled the workflow middleware. Prof. Kienzle participated in the discussions and manuscript preparations. I presented the paper in the conference.

- Designing with inheritance and composition. **O Alam**, J Kienzle. Proceedings of the 3rd international workshop on Variability & Composition. 19-24. ACM. 2012. (Parts of Chapter 8 are based on this paper).

In this paper, I investigated the differences between inheritance and composition in software design. Prof. Kienzle participated with me in the discussions and writing the findings.

- Concern-oriented Software Design. **O Alam**, J Kienzle, G Mussbacher. Proceedings of Model-Driven Engineering Languages and Systems (MODELS'13), 604-621. 2013. (Parts of Chapter 1, Chapter 2, Chapter 3, and Chapter 4 are based on the MODELS'13 paper).

In this paper, I worked with Prof. Kienzle and Prof. Mussbacher on developing the concepts of concern and its interfaces. I worked on validating the concepts using examples. The co-authors participated in the discussions and manuscript preparations. I presented the paper in the conference.

- Composition of Software Concern Lines. **O Alam**, J Kienzle, G Mussbacher, P Collet. Submitted on September 21, 2015 to ACM Symposium of Applied Computing, Pisa, Italy. 2016. (Parts of Chapter 3 and Chapter 5 are based on this paper).

I developed the algorithms presented in this paper and implemented them in the TouchCORE tool. I applied the algorithms on the bCMS case study and produced the feature model presented in the paper. The co-authors participated in the discussions and manuscript preparations.

- Concern-Oriented Reuse. **O Alam**, J Kienzle, G Mussbacher. Prepared for International Journal on Software and Systems Modelling (SoSyM). (Parts of Chapter 1, Chapter 2, Chapter 3, Chapter 4, and Chapter 6 are based on this paper).

This paper presents CORE as a reuse paradigm. I performed an extensive literature comparison to compare CORE with other reuse units, and extended the metamodel in the TouchCORE tool as part of the validation of CORE and prepared the examples. The co-authors participated in the discussions and manuscript preparations.

- Modelling a Family of Systems for Crisis Management with Concern-Oriented Model Reuse. **O Alam**, J Kienzle, G Mussbacher. Submitted to Journal of Software Practice and Experience on August 1, 2015. (Parts of Chapter 9 are based on this paper).

In this paper, I analyzed the requirement specifications for the bCMS case study, and created the feature model and design models for the case study. I also collected the reuse metrics presented in the paper and modeled the feature model in the TouchCORE tool. The co-authors participated in the discussions and manuscript preparations.

Here is the list of publications based on this thesis that are not included in the thesis text (the order of authors reflects approximately the amount of work done on the paper by each author):

Refereed International Conference and Workshop Papers:

1. Feature modelling and traceability for concern-driven software development with TouchCORE, M Schöttle, N Thimmegowda, **O Alam**, J Kienzle, G Mussbacher. Companion Proceedings of the 14th International Conference on Modularity, 11-14 (2015). ACM.
2. TouchRAM: a multitouch-enabled software design tool supporting concern-oriented reuse. M Schöttle, **O Alam**, FP Garcia, G Mussbacher, J Kienzle. Proceedings of the companion publication of the 13th international conference on Modularity (2014), 25-28. ACM.
3. Specification of domain-specific languages based on concern interfaces. M Schöttle, **O Alam**, G Mussbacher, J Kienzle. Proceedings of the 13th workshop on Foundations of aspect-oriented languages (2014). 23-28. ACM.
4. Concern-Driven Software Development with jUCMNav and TouchRAM. N Thimmegowda, **O Alam**, M Schöttle, W Al Abed, T Di'Meco, L Martellotto, Proceedings of the Demonstrations Track of the International Conference on Model Driven Engineering Languages and Systems (MODELS). 1-6. 2014.
5. Revising the Comparison Criteria for Composition. **O Alam**, M Schöttle, J Kienzle. Proceedings of CMA workshop in MoDELS 2013. 1-6.
6. Concern-Oriented Software Design with TouchRAM. M Schöttle, **O Alam**, A Ayed, J Kienzle. Tool demonstration paper at MODELS 2013. 51-54.
7. TouchRAM: A multitouch-enabled tool for aspect-oriented software design. W Al Abed, V Bonnet, M Schöttle, E Yildirim, **O Alam**, J Kienzle. Software Language Engineering (SLE 2012), 275-285. 2012.
8. Using CVL to operationalize product line development with reusable aspect models. B Combemale, O Barais, **O Alam**, J Kienzle. Proceedings of the VARiability for You Workshop: Variability Modelling Made Useful for Everyone. 9-14. ACM. 2012.

9. Assessing composition in modelling approaches. G Mussbacher, **O Alam**, M Alhaj, S Ali, N Amálio, B Barn, R Bræk, T Clark, ... Proceedings of the CMA 2012 Workshop, Barbados. 1-26.
10. CORE Model Submission for the Comparing Modelling Approaches Workshop 2012. **O Alam**, M Schöttle, G Mussbacher, J Kienzle. Proceedings of the CMA 2012 Workshop. 2012.
11. Comparing six modelling approaches. G Mussbacher, W Al Abed, **O Alam**, S Ali, A Beugnard, V Bonnet, R Bræk, ... Models in Software Engineering, 217-243. 2012.
12. Concern-Driven Software Development, **O Alam**, J Kienzle and G Mussbacher, School of Computer Science, McGill University, January 2015, CS-TR-2015.1

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 stakeholders' 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 [44].

Model-Driven Engineering (MDE) [39] 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. 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 impacting differently 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, this thesis proposes a novel reuse paradigm called Concern-Oriented Reuse (CORE), that builds on the ideas of MDE, advanced Separation of Concerns (SoC) [37], Software Product Lines (SPL) [97], and goal modelling [36, 124]. CORE is a software reuse paradigm that introduces a new unit of reuse called *concern*, that enables

broad-scale model reuse at multiple phases of software development. CORE comes with a reference implementation, i.e. a metamodel and guidelines for extending the metamodel. Different modelling languages and tools can extend the CORE reference implementation (i.e. to become *corified*) to support broad-based model-reuse.

This thesis lays the foundation of CORE by defining its concepts, reuse process, and metamodel, and compares concerns with other reuse units. The thesis then validates CORE by extending its metamodel in existing modelling languages and tools, building a reusable concern library and conducting a case study of a family of Crisis Management Systems. Although a CORE concern can span multiple phases of software development, the primary focus of this thesis is on the design phase.

We envision that large-scale adoption of CORE would revolutionize software development. Tool vendors can support concern-orientation by extending the CORE metamodel. As a result, different concern-oriented modelling languages and their tools will emerge allowing concerns to be developed at different abstraction levels using different modelling formalisms. Vendors can sell reusable concerns, and libraries of existing reusable concerns can be used in developing new concerns or applications. Modellers can specialize in developing and maintaining certain types of concerns. For example, a security expert can specialize in developing and maintaining security related concerns. Ultimately, tool support, concern libraries, specialization, and reuse process would bring the software development practice closer to other engineering disciplines.

1.1 Problem Summary and Thesis Statement

1.1.1 Problem Summary

This thesis attempts to solve the issues raised in the introduction. Concretely, the research question this thesis attempts to solve can be summarized as follows: "how can we build a new model (can be application/system) through reusing existing models?"

To answer this question, this thesis introduces resents Concern-Oriented Reuse (CORE), a new software development paradigm that introduces the *concern* as a new unit of reuse. CORE integrates the ideas of model-driven engineering, advanced separations of concerns (aspect-orientation) and software product lines (SPL) to address one of the main challenges of MDD: large-scale model-reuse. A concern groups related software artifacts together (models and code, simply called models in this thesis) addressing a domain of interest to the software practitioner.

The lifecycle of a concern starts at its root phase–a certain point during the development process where it becomes relevant to the software practitioner. For instance, the security concern appears during the requirement phase since it becomes relevant to the external stakeholders. Other concerns, such as database integration appear in the design phase. For each phase, models are built using the most appropriate formalism to express the concern behaviour and properties. This thesis lays the foundation for CORE by defining the concepts and the metamodel for concern-oriented reuse (CORE). The CORE metamodel ensures that modelling approaches under CORE adhere to its guidelines (e.g., they provide the concern interfaces and support reuse). The thesis then focuses on the design phase of CORE, by implementing CORE guidelines for software design. We validate the effectiveness of CORE

through tool support, developing library of reusable concerns, and reusing these reusable concerns in an developing a family of crisis management systems.

1.1.2 Thesis Statement

Concern-Oriented Reuse addresses one of the main challenges of MDD: broad-scale model reuse. Instead of building models from scratch, CORE allows a model (can be an application) to be built by reusing existing models. In addition, CORE allows models to be built for reuse from the start, by dedicating a three-part concern interface and a simple reuse process.

1.2 Thesis Contributions

The contributions of thesis are organized into three parts:

Part I Definitions (Chapter 2 and Chapter 6):

- The thesis introduces and defines the essential concepts of CORE: the unit of reuse *concern,* and the three concern interfaces. The *variation interface* of the concern allows the modeller to select the most appropriate variant among those encapsulated within the concern (closed variability), and reason about the impact of the selection on high-level goals and system properties. The *customization interface* allows the modeller to adapt the concern to a specific reuse context (open variability). The *usage interface* allows the user to use the functionality provided by the adapted concern in the application.
- The thesis defines a *metamodel* that captures the concepts of CORE in a *CORE reference implementation* and outlines strategies that allow different modelling languages to be corified (i.e., to be integrated with the CORE metamodel), so they can support broad-based model reuse as advocated by the CORE paradigm.

Part II Process (Chapter 4, Chapter 5 and Chapter 8):

- The thesis introduces and defines a *concern reuse process* that allows a concern to be built by reusing existing concerns and to defer design decisions to a later point when more requirements and desired qualities have been determined. The process is illustrated by means of an example *Authentication* concern.
- The thesis specifies CORE composition rules and algorithms that allow the variation interfaces and the realization models of concerns to be composed.
- The thesis demonstrates how to build a reusable design concern by incrementally adding small model increments to a base model. Concretely, this incremental concern modelling process is illustrated in practice by elaborating the design of an example concern of considerable size called *Workflow*.

Part III Validation (Chapter 3, Chapter 7, Chapter 8, and Chapter 9):

- The thesis performs an extensive literature comparison highlighting the advantages and disadvantages of the CORE paradigm compared to other reuse technologies. The CORE concern is compared with other units of reuse, such as classes, frameworks, and services.
- The thesis validates the CORE reference implementation by successfully corifying two modelling languages (corification of a modelling language means integrating the language with the CORE metamodel) and by implementing a modelling language tool that applies the CORE reuse process.
- The thesis applies the CORE reuse process to design several reusable design concerns, reusing lower-level concerns whenever possible, to create a *library of reusable design concerns*.

- The thesis applies the CORE reuse process on a case study application to *build a family of Crisis Management Systems.*

1.3 Thesis Organization

This thesis is divided into ten chapters. We start by discussing the principles and the approaches that CORE builds on, and by defining concern and its three-part interface in Chapter 2. Then, we provide an overview of popular units of reuse and related work in Chapter 3, followed by a detailed comparison between concern and other units of reuse. Chapter 4 describes the concern reuse process illustrated by an example. Chapter 5 details the composition rules and algorithms that allow variation interfaces of two concerns to be composed, and to generate a customized concern by composing its realization models. Chapter 6 introduces a CORE metamodel that allows different modelling languages to be corified (i.e., to support concern-orientation). Chapter 7 discusses how the CORE metamodel was successfully extended by two modelling languages and discusses in detail how a modelling language tool (TouchCORE) supports the CORE paradigm. Chapter 8 presents our reusable concern library and elaborates how we built a particular reusable concern of considerable size (Workflow). In Chapter 9, we design a family of Crisis Management Systems using the reusable concern library. We then conclude this thesis and discuss directions for future work in Chapter 10.

Chapter 2
Background and Definitions

This chapter introduces Concern-Oriented Reuse (CORE) concepts and definitions. CORE is a software reuse paradigm that introduces a unit of reuse called *concern* which provides three interfaces to facilitate reuse: the variation interface, the customization interface, and the usage interface. Modelling languages can implement the CORE reference implementation to be *corified* (i.e. to support concern-orientation). Therefore, a *concern* in a modelling language, is a unit of reuse that applies the CORE reference implementation in that particular modelling language. A concrete instance of a *concern*, however, may contain models belonging to multiple modelling languages at different software development phases and abstractions. CORE builds on three main pillars: Model-Driven Engineering (MDE), reuse (i.e., Software Product Lines), and Separation of Concerns (SoC). In this chapter, we detail how we leverage the strengths of these three fundamental principles of software engineering to support broad-based model reuse in Section 2.1, stipulate key characteristics of a reusable artifact in Section 2.2, define concern as a unit of reuse, and its three-part interface in Section 2.3, motivate the need for concern hierarchies and so called Software Concern Lines (SCL) in Section 2.3, and finally conclude this chapter in Section 2.6,

2.1 Three Pillars of Concern-Orientation: MDE, Reuse (SPL), and SoC

CORE builds on ideas belonging to three pillars. As discussed in the introduction, the objective of MDE, the first pillar, is to create software through model creation, composition, refinement, and integration. Models are built using the formalisms that best describe

and encapsulate relevant properties for each level of abstraction. Models at higher levels of abstraction can be integrated with solution-specific, detailed models at lower levels of abstraction, until a final model is produced which may be executed. CORE uses these concepts, especially the ability of MDE to embed domain-specific knowledge into models to bridge the gap between domain and system knowledge.

The second pillar focuses on model reuse. Software reuse is a powerful concept originated in the sixties and aims at developing software by reusing existing software artifacts instead of creating them from scratch. To make software reuse applicable, reusing an artifact should be easier than constructing it from scratch. This entails that the reusable artifacts are easy to understand, find, and apply [67]. Benefits of reuse include increase in productivity and maintainability, and reduction in cost and time-to-market. Creating a reusable artifact is not trivial, requires deep domain knowledge, and involves the definition of clear interfaces for the artifact that communicate efficiently to the user in what context the artifact can be reused. Insufficient knowledge of reusable artifacts and their application limits can lead to catastrophic consequences [8]. The software requirement engineering area has identified reuse to be a key research topic [88], which should receive more attention [123]. Limitations of model reuse are discussed in requirements engineering [85] as well as in software design [16].

Several technologies in software development have been successful in supporting reuse. Programming languages such as Java provide class libraries that contain reusable code with documented instructions for reuse. Design patterns [46] provide abstractions of recurrent design problems with informal descriptions on how a pattern impacts high-level goals. Components [116] provide well-defined interfaces on how to reuse them in some popular application domains, e.g., web applications. Frameworks (e.g., [6]) allow reuse by providing limited

extension points to an application, but it is difficult to reuse multiple frameworks in the same application since they usually impose a specific application architecture. Services [41] are designed to be self-contained and loosely coupled, allowing dynamic reuse and composition. Software Product Lines (SPL) [97] specify the commonalities and the variability within a well-defined scope of products. Features in an SPL allow reuse at multiple levels of software development, but within the closed scope of the domain for which the SPL is built. We provide an extensive comparison between the reuse in these approaches and CORE in Chapter 3.

The third pillar of CORE is based on the ideas of Separation of Concerns (SoC)) [37], information hiding, and encapsulation [92]. There are several successful reuse units that support encapsulation and information hiding. Procedures hide the details of algorithms from their callers. Objects encapsulate related functionality and data, while hiding the detailed structure/behaviour behind well-defined interfaces. Components also hide collaborating objects behind provided and required interfaces. Aspects [60] focus on encapsulation of crosscutting concerns, modular reasoning, and provide advanced composition mechanisms.

2.2 Key Characteristics of Reusable Artifacts

Based on the reuse success stories stated in the previous section, we identify key characteristics that a reusable artifact should have:

- The reusable artifact must be as generic as possible, i.e., it must not refer to unnecessarily specific properties. For example, reusable artifacts such as template classes, components, and services focus on being as generic as possible.

- The reusable artifact must be well-packaged, i.e., it must group all the elements that logically belong to the artifact. For example, objects package related data and functionality together and components package related collaborating objects together.
- The reusable artifact must provide clear interfaces and instructions for reuse. Reusable artifacts, such as objects and components, encapsulate related structure/behaviour behind well-defined interfaces.
- The reusable artifact must be composable and logical dependencies due to reuse and separation of concerns must be aligned with each other and with their syntactic manifestations, which will be motivated in the remainder of this section.

Fig. 2–1.a shows a Java code example where an *Account* class reuses *Authentication* to authenticate the user when performing banking operations, such as *withdraw* and *deposit*. The syntactic dependency goes from *Account* to *Authentication*, since *Account* calls the *Authentication* class. *Authentication* is unaware of *Account*. The *logical reuse* dependency (see Fig. 2–1.a on the right) goes from the application-specific artifact to the generic, reusable artifact, which is the way it is supposed to be.

While the code in Fig. 2–1.a nicely separates the reusable artifact from the application-specific artifact, it does not support SoC. The *Account* class, which specifies business logic and functionality, has to deal with *Authentication* and explicitly refer to it/call it from within every operation that needs to be authenticated. Fig. 2–1.b shows a different implementation of *Authentication* using *AspectJ* [2], an aspect-oriented extension of the Java programming language. *AspectJ* introduces concepts that support SoC into Java, most importantly *aspect*, *pointcut*, and *advice*. The *AuthenticationAspect* shown in Fig. 2–1.b defines a pointcut and advice that specify that, whenever *deposit* or *withdraw* are about to be executed on

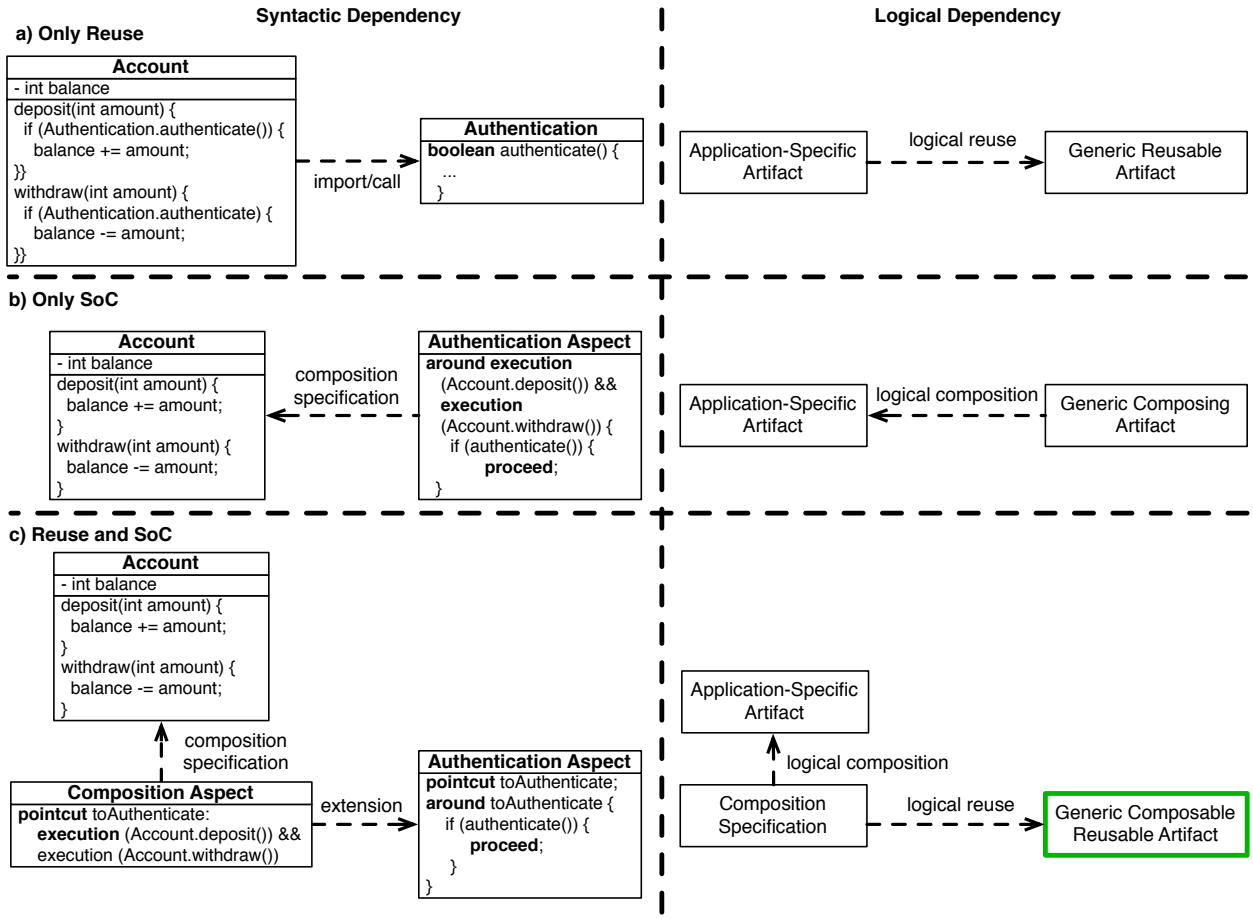

Figure 2–1: Syntactic and Logical Reuse/SoC Dependencies

an instance of the *Account* class, *authenticate* is first executed. Only if authentication is successful, then the flow of control *proceeds* to executing *deposit* or *withdraw*, respectively. To achieve this flow of control, the behaviour of the *AuthenticationAspect* has to be composed with the behaviour of the *Account* class, which is done by the aspect weaver, using the pointcut definition inside the *AuthenticationAspect*. The logical composition dependency is

therefore from the generic composing artifact to the application-specific artifact as illustrated in Fig. 2–1.b on the right.

However, the generic *Authentication Aspect* code now is aware of elements from the application-specific *Account* class, which contradicts reuse principles. Consequently, simple juxtaposition of reuse and SoC techniques results in the two resulting dependencies going in conflicting, opposite directions. To resolve this problem, the generic artifact has to be constructed in such a way that it is composable, but the composition specification that specifies where it is applied in the application must be kept separate, i.e., neither as part of the application-specific nor the reusable artifact. The result is a *generic composable reusable artifact* with non-conflicting logical dependencies as shown in Fig. 2–1.c. In *AspectJ* this can be achieved using abstract pointcuts and aspect inheritance. In line with reuse principles, the *Authentication Aspect* does not contain any application-specific elements, but defines an abstract *pointcut toAuthenticate* that advises potentially any operations, adding the behaviour of authentication. The *Authentication Aspect* is extended by the *Composition Aspect* that links the *AuthenticationAspect's toAuthenticate* to the *Account deposit* and *withdraw* operations. In line with SoC principles, the *Account* class and the *Authentication Aspect* do not contain any elements from other concerns. It is the *Composition Aspect* that depends on both by defining a *pointcut* identifying the location in *Account* where the *Authentication Aspect* is going to be applied. Hence, the *AuthenticationAspect* is now a *generic composable reusable artifact.*

The concerns, the interfaces, and the reuse process defined by CORE described in this thesis are all enabled by this key idea, but applied to all software development artifacts

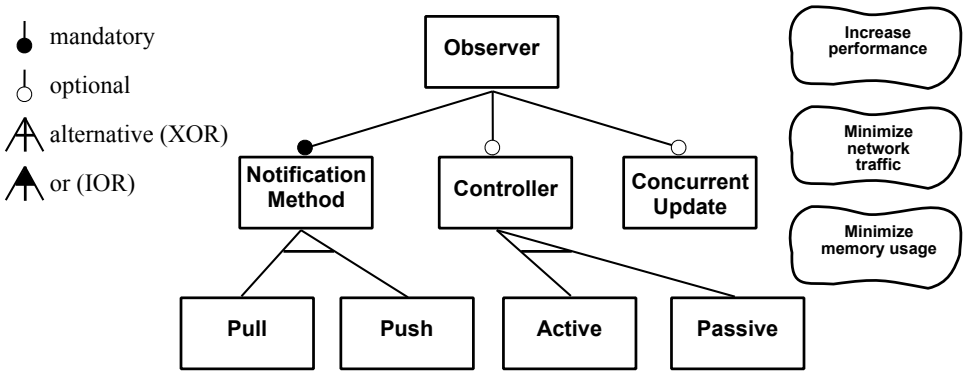

Figure 2–2: The variation interface of the Observer concern.

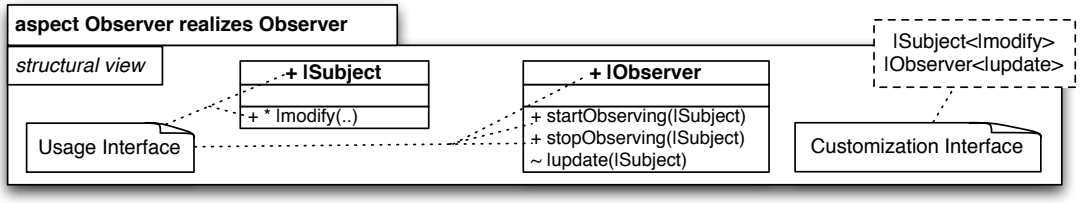

Figure 2–3: Observer RAM Model Interface (Customization and Usage)

(requirement, architecture, design, and implementation). This allows our concerns to be simultaneously generic and composable, and as a result highly reusable.

2.3 Definition of Concern

As mentioned in the introduction, CORE is a software reuse paradigm that advocates the *concern* as its main development artifact, and comes with a metamodel and reference implementation. Here we define the concern as ***a modular unit of reuse that encapsulates a set of models describing all properties of a domain of interest during software development, often spanning multiple phases of software development and levels of abstraction (from requirements and analysis models to design models to***

code). Each concern has a root phase, where the concern manifests itself for the first time. Some concerns appear in early phases of software development, e.g., broadly scoped system properties with functional, non-functional, or even intentional characteristics. Some concerns appear in later phases of software development, e.g., solution-specific concerns such as specific communication protocols, concrete authentication algorithms, and design patterns.

A CORE concern has to provide three interfaces to facilitate modular reuse, the usage, customization and variation interfaces. There are reuse units that support some of these interfaces as we see in the next section and also when we discuss related work in Chapter 3. However, as we show when we compare concern with other reuse units in Chapter 3, there is no reuse unit that support all three interfaces together with guidelines for reuse. The three interfaces of a CORE concern together (usage, customization, and variation interfaces) allow for building applications/concerns by easily reusing other existing concerns as shown in the concern reuse process in Chapter 4, and validated by building a tool and a library of reusable concerns in Chapter 7 and Chapter 8, and by a family of Crisis Management Systems case study in Chapter 9. In the next subsection, we explain concern interfaces in detail.

2.4 Concern Interface

The key concept of CORE promoting reuse and modularity is the *three interfaces* [16] that every concern must provide. In this subsection, we discuss in detail each interface. In addition, we use an example design concern called *Observer* to further illustrate concern interfaces. *Observer* is a popular design pattern [46] in which an object called *Subject* maintains a list of dependent objects called *Observers*. When the state of *Subject* changes, the list of *Observer* objects are updated. There exist many variations of Observer in the literature, of which we choose three to include in our *Observer* concern. The first variation is

related to the notification method used to update the *Observer* list. There are two notification methods: *Push* method in which the *Subject* sends messages to the *Observer* list when its state changes, and *Pull* method in which the *Observer* periodically check whether the state of *Subject* changed. *Push* reduces the network traffic while *Pull* reduces the amount of data exchange. The second variation of *Observer* concern is about concurrency. In a single threaded application of *Observer,* the performance of the system is affected when there is high volume of messages sent over the network or when the update operations of *Observer* objects are computationally intensive. In such cases, a multithreaded application of the *Observer* pattern will help reduce network traffic and/or improve performance. Finally, a design variation of *Observer* introduces a third object called *Controller* which reacts to external events and serves as connector between *Subject* (called *Model)* and *Observer* (called *View*) according to the Model-View-Controller (MVC) principle. The *Controller* receives requests for state changes on the *Model* and then notifies the *View* for update. *MVC* also has two design strategies–*Active* and *Passive*–based on two different ways to pass the control between *Controller, Model,* and *View* objects.

2.4.1 The *Variation Interface*

The Variation Interface definition: *this interface makes reuse simple and straightforward by clearly describing the different variations of the solution it encapsulates, as well as the impact of different variants on high-level stakeholder goals, system qualities, and non-functional requirements.* Typically, there are many ways to solve a specific problem, each solution having advantages and disadvantages. There are, for example, families of algorithms for achieving similar behaviour that have varying run-time resource requirements, and different ways of organizing information into data structures. The choice of data structures

and algorithms has an effect on application performance and memory usage. The existence of a multitude of sorting algorithms, for example, shows clearly that there is no one good way of sorting. A more complex example is *transactions* [51], a design concept for fault tolerance that emerged in the database community. A transaction groups together a set of operations on data objects, ensuring atomicity, consistency, isolation, and durability of data updates. There are many ways of designing support for transactions, including pessimistic/optimistic and strict/semantic-based concurrency control, in-place/deferred update, and logical/physical and forward/backward recovery. Again, each technique has advantages and disadvantages.

Because of the multitude of possible solutions, before a developer can focus on choosing a specific solution, she must carefully consider how each possible solution positively or negatively affects the stakeholders high-level goals and the non-functional properties of the application. Choosing the best solution is arguably the most important activity of the development process, and has a crucial impact on the quality of the entire application. Ultimately it is the capability of choosing the most appropriate solution that distinguishes a good developer from a bad one.

Unfortunately, none of the popular units of reuse makes this important activity easy for the developer. Even if the unit is accompanied with documentation that describes the impact of the solution, the documentation usually does not mention other alternative solutions.

For example, a class typically only provides one solution to a specific problem. At best, the class comes with documentation that describes the impact of the encapsulated design. For example, the `ArrayList` class provided as part of the Java standard class library [50] implements a queue, i.e., a data structure that stores a sequence of elements and provides

operations to insert and remove elements from the sequence, and iterate over the elements in the sequence. However, there is no support in Java to capture the impact of a class on the non-functional properties of an application that uses it. This is not a problem for an experienced Java developer, since she has probably used the class before. If not, other sources of information, i.e., the (textual) Java documentation or Java developer websites, need to be consulted to discover the impact of the class on non-functional application properties. Likewise, there are many ways to store a sequence of elements in Java, i.e., using the `CopyOnWriteArrayList` class, the `Vector` class, the `LinkedList` class, or simply a standard array. Each way has a different impact on performance and memory requirements. Again, there is no direct support in Java to capture this information. The only way to find this information is to assume that all classes that implement a sequence are located in the same Java package (`java.util` for `ArrayList`), and that they all implement the `List<E>` interface.

Similar arguments can be made for other units of reuse, i.e., components [116]. The situation is different, however, for patterns, frameworks (e.g., [6, 5]), and SPLs. A description of a design pattern, for example, is required to contain a *Consequences* section that contains a description of the results, side effects, and trade offs caused by using the pattern. There is also a *Related Patterns* section, which mentions other patterns that have some relationship with the pattern that is being described and discusses the differences between this pattern and similar patterns. Unfortunately, these textual descriptions are informal.

Sophisticated frameworks are often designed in such a way that they provide a variation of similar functionalities to the developer. Typically, the choices are presented to the developer in form of class hierarchies from which the developer can instantiate the class that fits her

requirements best. Whereas the functional impact of the different options is usually explained well, the impact on non-functional application properties is rarely documented rigorously. As a result, using a framework in the most appropriate way for a specific application still requires considerable expertise.

SPLs inherently describe variations, and, consequently, SPL techniques are certainly applicable to some aspects of concern-oriented reuse. The crucial difference is that SPLs are focused on producing a product instead of specifying a possibly crosscutting concern. SPLs typically lack rigorous interfaces that have been designed to support composition of crosscutting concerns, allowing many concerns to be combined for one product and a single concern to be applied to many products.

Based on the aforementioned arguments, we suggest that in order for reuse to be maximally effective, a new, broader unit of reuse that encompasses all solutions targeted at solving a problem is needed. A CORE *concern* accomplishes this objective by providing the *variation interface*. This interface expresses the *closed* variability offered by the concern as in an SPL [97]. Developers have to think about feature dependencies and interactions and express them by means of feature model relationships (mandatory, optional, alternative, requires, excludes). The variations are typically represented with a *feature model* [58] that specifies the individual features that a concern offers, as well as their dependencies such as mandatory, optional, alternative, requires, and excludes. The *impact model*, i.e., the impact of choosing a feature, can be specified with goal models (e.g., with GRL, which is part of the User Requirements Notation (URN) standard [56], or the NFR framework [30], i* [130], and KAOS [36]). For example, a security concern may offer various means of authentication, from *password-based* to *biometrics-based solutions*, each with differing impacts on

the *level of security* as well as *cost* and *end-user convenience*. These qualities have to be weighed, when determining which authentication variant is most appropriate in the current application context.

Fig. 2–2 shows the basic variation interface of the *Observer* concern. We show the feature model of *Observer* and the high level goals for the impact model. A detailed impact model further breaks down the high level goals and shows how features of the feature model contribute to them positively or negatively (we explain the feature model and the impact model in more detail when discussing the CORE reuse process in Chapter 4). Features of the feature model can be realized by one or many realization models. These realization models can belong to different modelling notations at different levels of abstractions as discussed in the previous section. Since *Observer* is a design concern, we model its realization models using a design modelling notation called Reusable Aspect Model (RAM) [62]. RAM is an aspect-oriented multi-view modelling notation based on extended UML. A RAM model is a UML package consisting three views: a structural view that shows the class diagram, a behavioural view that shows sequence diagram and a state view shows the state diagram of the software design. The simple *Observer* RAM model shown to illustrate the usage and customization interfaces in Fig. 2–3 realizes the root feature in the feature model shown in Fig. 2–2. Realization models of features such as *Pull* and *Push* incrementally build on the realization model of the root in a process described in detail in Chapter 8.

2.4.2 The *Customization Interface*

The Customization Interface definition: *the customization interface of a concern describes how a chosen variant of the concern can be adapted to the needs of a specific application.* Typically, a unit of reuse has been purposely created to be as general as possible

so that it can be applied to many different contexts. As a result it is often necessary to tailor the general model to a specific application context. For example, the customization interface of generic or template classes allows a developer to customize the class by instantiating it with application-specific types. For components, the customization interface is comprised of the set of services that the component expects from the rest of the application to function properly (i.e., the *required interface*). The developer can use this information at configuration time to plug in the appropriate application-specific services. The customization interface for frameworks and design patterns is often comprised of interfaces/abstract classes that the developer has to implement/subclass to adapt the framework to perform application-specific behaviour.

Each variant of a concern is described as generally as possible to increase reusability. Therefore, some elements in the concern are only *partially* specified and need to be related or complemented with concrete modelling elements of the application that intends to reuse the concern. The customization interface expresses the *open* variability offered by the concern, similar to what generic classes in programming languages do. This mechanism meets a need to handle flexibility and openness while handling variability in a scoped domain, as it has already been attempted in previous work on components in product populations [125], more flexible SPLs [52, 95], or variability aware modules [59]. The customization interface is used when a specific variant of a reusable concern is *composed* with the application. For example, a security concern may define a generic *User* as a partial class that needs to be merged with the concrete application classes that describe the actual users of the system, e.g., *Administrator* or *Employee*.

Although there are reuse units that implicitly designate a *customization interface* as in the case of components, CORE allows its modelling languages to explicitly define their customization interface. RAM provides an explicit *customization interface* that specifies how a generic design model needs to be adapted to be used within a specific application. To increase reusability of models, a RAM modeller is encouraged to develop models that are as general as possible. As a result, many classes and methods of a RAM model are only partially defined. For classes, for example, it is possible to define them without constructors and to only define attributes relevant to the current design concern. Likewise, methods can be defined with empty or only partial behaviour specifications. The idea of the customization interface is to clearly highlight those model elements of the design that need to be completed/composed with application-specific model elements before a generic design can be used for a specific purpose. These model elements are called *mandatory instantiation parameters*, and are highlighted visually by prefixing the model element name with a |, and by exposing all model elements at the top right of the RAM model similar to UML template parameters. Fig. 2–3 shows that the customization interface for the *Observer* model comprises the class `|Subject` with a `|modify` operation, and the class `|Observer` class with an `|update` operation.

2.4.3 The *Usage Interface*

The Usage Interface definition: *the usage interface specifies the structure and behaviour that the concern provides to the rest of the application.* In other words, the usage interface presents an abstraction of the functionality encapsulated within the unit to the developer. It describes *how* the application can trigger the functionality provided by the unit.

For instance, for classes the usage interface is the set of all *public* class properties, i.e., the attributes and the operations that are visible and accessible from the outside. For components, the usage interface is the set of services that the component provides (i.e., the *provided interface*). For frameworks, design patterns, and SPLs, the usage interface is comprised of the usage interfaces of all the classes that the framework/pattern/SPL offers.

The *usage interface* of a RAM model is comprised of all the *public* model elements (preceded with +), i.e., the structural and behavioural properties that the classes within the design model expose to the outside. To illustrate this, the usage interface of the RAM design of the *Observer* design pattern is shown in Fig. 2–3. The structural view of the *Observer* RAM model specifies that there is a `|Subject` class that provides a public operation that modifies its state (`|modify`) that can be called by the rest of the application. In addition, the `|Observer` class provides two operations, namely `startObserving` and `stopObserving`, that allow the application to register/unregister an observer instance with a subject instance.

While one variation interface consisting of feature and impact models exists for the whole concern, one customization interface and one usage interface typically exist for each type of model in the concern with the exception of the models used for the variation interface. Chapter 4 outlines a three-step reuse process that allows the software practitioners to develop concerns (or applications) by reusing existing concerns. Each step of the reuse process uses one of the concern interfaces.

2.5 Concern Hierarchies and Software Concern Lines (SCL)

An application is built by reusing many concerns. When reusing a concern, the user should be able to select the best solution from the variation interface and perform trade-off analysis. For example, the user of the *Authentication* concern can select among different

ways of performing *Authentication*, e.g., using passwords, facial recognition, or fingerprints. Each solution has impacts on high-level goals such as cost, user-convenience, speed, and CPU/memory consumption. CORE uses ideas from feature modelling [98, 29] and goal modelling (e.g., [36, 130, 18]) to guide the concern user to select the best solution and perform trade-off analysis.

A reusable concern may be used when developing an application, but may also be used to build other reusable concerns, resulting in a complex reuse hierarchy, a *concern hierarchy*. Generally, concerns in CORE are built to be as generic as possible similar to class libraries / generic classes in Java. On top of that, they encapsulate all variations of the functionality and behaviour related to a concern, similar to the way SPLs group all variations of a specific application domain. However, concerns are not product-specific as SPLs typically are. CORE concerns express the variabilities they encapsulate just like SPLs, but are not confined to a specific product family. They are intended for an open, broader reuse scope. To clearly highlight this key difference and at the same time acknowledge one of the pillars of CORE (i.e. reuse), we refer to concerns in concern hierarchies as *Software Concern Lines* (SCL).

When a concern at a higher level (i.e., the *reusing concern*) reuses another concern (i.e., the *reused concern*), the developer should be able to precisely specify how the interfaces of the reusing concern are affected by the interfaces of the reused concern. For the usage interface, the composition needs to determine which elements of the reused concerns should be part of the usage interface of the reusing concern, if any. This is related to information hiding, as it makes functionality of the reused concern accessible at the next-higher level. For the customization interface, mechanisms must be provided so that reused customization elements that have not been concretized by the reusing concern are reexposed in the reusing

concern's customization interface in addition to any new elements it defines. But the most challenging problems come from the variation interface. First, choosing the best variant of a reused concern is only possible when all requirements are known, which is far from being the case when composing some concern with a lower-level one, because it is typically not known how the reusing concern will be used in the future. Second, some features from the reused concern may not be applicable in its specific context of reuse (i.e., when composed at the level of the reusing concern), and third, the qualities of the features being built for the reusing concern are affected by the qualities of reused concerns. We discuss how interfaces of a reusing concern are composed with interfaces of a reused concern in Chapter 5, and detail composition rules and algorithms for the variation interface that allow an application (or a concern) to be built through producing a complex *concern hierarchy.*

2.6 Conclusion

Model reuse has been identified as one of main challenges facing MDE. Models are often created from the scratch with little means to reuse existing models. The reasons why model reuse is challenging include lack of proper model interfaces, support for expressing variability, trade-off analysis, modularity in reuse units, and reusable model libraries. To overcome this challenge, we introduce CORE–a software reuse paradigm that builds on ideas from model-driven engineering, software product lines, separation of concerns (SoC) to support broad-based model reuse. CORE introduces a new unit of reuse called *concern* that groups related models together often spanning multiple development phases and abstraction levels, and designate three interface to facilitate reuse: the variation interface, customization interface and the usage interface. In addition, CORE provides a metamodel and a reference implementation that allows different modelling languages to corified. Through CORE reuse

process, an application or a concern can be built by easily reusing other existing concerns, often producing complex concern hierarchies.

Chapter 3
Relation to Related Research

Software reuse is a powerful concept that originated in the sixties, and is defined as the process of creating new software using existing software artifacts. To make software reuse applicable, reusing an artifact should be easier than constructing it from scratch. This entails that the reusable artifacts are easy to understand, find, and apply [67]. There are characteristics of software artifacts that facilitate reuse, e.g., grouping, encapsulation, information hiding, and well-defined interfaces.

In this chapter we provide an overview of related work to CORE. We start by reviewing some popular units of reuse in the next section. In Section 3.2, we discuss the body of related work including efforts by different approaches to overcome the shortcoming of the popular units or reuse. We conclude this chapter in Section 3.3.

3.1 Units of Reuse

In this section, we review some of the most popular units of reuse. We study if the units have the following characteristics, and we end this section with a summary comparison table.

- Customization: Whether the approach supports adapting its artifacts when reusing them. McKinley et al. [76] classifies adaptation into two types: parameter adaptation, and compositional adaptation. Parameter adaptation seeks to adapt the unit of reuse by introducing changes to it (e.g., through parametrization). Compositional adaptation adapts the unit for a particular purpose through composing it with other units. We here make use of this classification, and elaborate it a bit further. We

classify customization into three groups: Customization, Static Composition, and Dynamic Composition. Customization stands for introducing changes to the unit not only through parameterization, but also through other means (e.g., partially-defined entities).

- Static Composition: Support for static composition of software artifacts that belong to the reuse unit. Static composition stands for composing an artifact with another artifact before running the system.
- Dynamic Composition: Support for dynamic composition. Dynamic composition stands for adapting the system by (re)-composing its artifacts at run time.
- Encapsulating crosscutting concerns: Whether the approach addresses the problem of scattering/tangling by encapsulating crosscutting concerns. This implies that the unit of reuse also supports advanced composition. Crosscutting concerns, e.g. security, require the reuse unit to support advanced composition so it can apply code/model fragments, not only to one location during composition, but to many different locations scattered over multiple entities.
- Variability: Whether the approach provides a variability interface to its provided solutions.
- Non-Functional Properties: Support for specifying non-functional properties and system qualities.
- Impact/Trade-Off Analysis: Support for impact analysis and reasoning about trade-offs using the non-functional properties when choosing between alternative solutions.

The following paragraphs provide an overview of popular reuse units and discuss whether they support the aforementioned characteristics:

Classes: Classes are the most common unit of reuse in the object-oriented programming. They group related state and behaviour (attributes and operations), and allows to reuse these properties in different contexts. Classes can nicely encapsulate a "local" design solution, where design state and behaviour fit into one entity. They fail when the design behaviour crosscuts several application entities. From the aforementioned comparison list, classes provide support for customization through generics and templates only, which allow the user to customize classes by initializing them with types.

Aspects: Aspects [60] address the problem of crosscutting concerns by encapsulating them separately from the application's core concerns. Aspects are *woven/composed* with core concerns to generate a complete application. Some aspect-oriented modelling approaches provide support for customization such as RAM (discussed later in this subsection). Popular aspect-oriented programming languages e.g., AspectJ [2] does not also support dynamic reuse, however, there are aspect-oriented approaches that support dynamic deployment [3] and adaptation [40, 99, 117].

Components: Components [116] are more coarse grained entities that package related functionalities behind well-defined interfaces. They are very popular in some application domains, e.g., web services, where dynamic configuration is desired. Whereas components are designed for reuse, they fail just like classes to encapsulate designs that crosscut the application architectural structure. Components provide some support customization, such as through configuring the properties file in JavaBeans. Adapter Design pattern is also used to customize/adapt JavaBeans components [71]. There also have been efforts to express non-functional properties and support impact analysis for commercial off-the-shelf (COTS) components [47].

Design patterns: Design patterns [46] are abstract design descriptions of solutions to recurring design problems. They capture interactions between classes, and explain trade-offs/impacts of the interaction pattern on high-level goals. However, design patterns are usually described informally in textual format or using incomplete UML diagrams or code examples that cannot be reused as such in an application design without substantial development effort.

Frameworks: Frameworks (e.g., [6, 5]) are software application platforms that are usually big in size and offer many features. Due to their size, they are usually difficult to configure/customize for a specific need. Frameworks often define a limited set of extension points, and dictate the control flow of the application that uses them. They apply the concept of *Inversion of Control (IoC)* [43], which allows frameworks to give their control flow to some custom code written by the application. However, this practice makes it hard to reuse several frameworks in the same application. Encapsulating crosscutting concerns is supported by some frameworks to some extend. For example, the testing framework JUnit [5], uses annotations in pieces of code that need to be tested, allowing testing scattered pieces of code.

Features of Software Product Lines (SPLs): SPLs [97] specify the variabilities and commonalities in a family of products and are an example of large-scale reuse. Reuse is the main focus in the context of software product line (SPL) development [98, 29], and since software design concerns are very close to SPLs we intend to incorporate best practices from this field. However, features in an SPL belong to a single domain of interest, cross-domain reuse of SPL products has not been explored much [75]. Cross-domain reuse requires modelling crosscutting concerns that define partially-developed artifacts, which are yet to be

explored in SPLs. Concern-orientation is about reuse in a broader sense. The customization interface allows the concern to be customized and reused in potentially many application contexts. i.e., within contexts that are not envisioned when the concern is created. Also, concern dependencies – which translates to inter SPL dependencies – are a kind of reuse that has not been explored by SPLs. Non-functional properties and support impact analysis are possible in some SPLs that assign attributes to features. There are also some SPL approaches that allow the features to be dynamically selected during runtime. The possible products in an SPL are limited to possible valid configurations of its features.

Services: Service-Oriented Architecture (SOA) [41] is a software architecture style that views the system as set of services that are self-contained, loosely coupled and can be easily composed. SOA provides set of guidelines that govern how services are represented and used. The key difference between concerns and services is that services, are designed to address business-related behaviour and logic. They "Enable assembly, orchestration, and maintenance of enterprise solutions to quickly react to changing business requirements" [68]. Although there are high-level concerns that address business logic, concerns can also address design solutions such as different design patterns or implementation solutions such as networking. Furthermore, concern allows the user to customize the reused concern according to her application needs. The ability to customize concerns, as mentioned previously, stem from their partially-defined entities. Customization allows concerns to be reused in many different applications and contexts, which is very useful when reusing generic concerns. It is not possible to define partially-defined services, they are usually composed by sending messages to invoke other services. Therefore, as in SPLs, the possible ways a service can

be composed is limited to the number of pre-existing services. There are, however, some approaches that allow customizing services described in the next subsection.

A distinctive characteristic of services, is their ability to be dynamically invoked during run time. There exist SOA approaches that provide some support for variability interface, which is helpful in choosing the most appropriate service during run-time. Service Level Agreements (SLA) specify non-functional properties, however, it is not possible to perform impact and trade-off analysis using SLAs.

Unit of Reuse	Custom-ization	Static Compo-sition	Dynamic Compos-ition	Encapsulating Crosscutting Concerns	Variability	Non-Funct-ional Properties	Impact Analysis
Classes	Yes	No	No	No	No	No	No
Aspects	Limited	Yes	Limited	Yes	No	No	No
Components	Limited	Yes	Yes	Limited	No	Limited	Limited
Design Pattern	No	No	No	No	Limited	Limited	Limited
Frame-works	Limited	No	No	Limited	No	No	No
SPL features	No	Yes	Limited	Limited	Yes	No	No
SPL features with attributes	No	Yes	Limited	Limited	Yes	Yes	Yes
Services	Limited	Yes	Yes	No	Limited	Yes	No
Concerns	Yes	Yes	Possible	Yes	Yes	Yes	Yes

Table 3–1: Comparison table of some popular units of reuse.

Table 3–1 summarizes the characteristics of the above units of reuse. Units that are marked as providing limited support for a characteristic have been explained during the

description of the unit, or there exists related work addressing this characteristic as described in Subsection 3.2.

Concerns as described in this paper provide support for all the characteristics used to classify reuse units in this subsection except for dynamic composition. Exploring dynamic feature selection in concerns could be possible based on models at runtime techniques [80], but is out of the scope of this thesis. To the best of our knowledge, there is no other approach that addresses all above shortcomings together.

3.2 Related Work

The plethora of related work can be organized into four categories. In Subsection 3.2.1, we discuss the work related to model reuse. In Subsection 3.2.2, we discuss the related work in the area of feature model composition. In Subsection 3.2.3, we discuss work related to variable modules. In Subsection 3.2.4, we discuss the work related to goal models and impact analysis.

3.2.1 Model Reuse, Customization and Separation of Concerns

Our approach aims at grouping logically related models together which is related to the research in Separation of Concerns (SoC). Approaches towards developing software systems using concerns (as advocated by aspect-oriented software development) fall into two categories. The first category puts the concern in the centre of the development process and modularizes the target system into concerns at the requirement and/or design phase. There are several approaches that fall into this category, the best known one is aspect-oriented programming. The second category of approaches analyze the existing systems to find the related parts of the system that can be organized into concerns. None of the approaches

detailed here provide support to express variability or to assess the impact on high level goals and properties.

Jacobsen [57] shares his experience in developing the telecommunication system at Ericsson and lessons he learned from the case study on a subsystem of that system in 1978. He notices that a use case is not realized in a single component but rather scattered across the components implementing the system. He concluded from the case study that if the use cases can be kept separate in all the phases of software development including implementation, maintaining and understanding the system will be come much easier. Components form the static structure of a system which usually reflects how the designer organizes the system. This static structure, however, does not express the dynamic behaviour of the system as described by the use cases. Jacobsen proposes a Use Case Driven Development approach where the the system is not only organized statically, but also dynamically through separation of use cases in all phases of development. Jacobsen uses *use case* and *concern* interchangeably to refer to the same concept. He defines use case as sequence of instructions performed by the system that yields a meaning result to the user. In order to achieve modularity in use cases, Jacobsen proposes two mechanisms. First, use cases need to be kept separate in all phases of software development process. He categorizes uses cases into two types: extension use cases and peer use cases. Peer use cases provides the basic functionalities. Extension use cases depend on other use cases and, hence, need to be properly linked with the dependent use cases. Using extensions, a system can be developed starting with a base use case and incrementally adding the extensions to it. The second mechanism that is needed to achieve modularity is use case composition. There needs to be composition mechanism of extension

use cases as well for the peer use cases. Peer use cases may share functionalities with each where composition mechanisms should compose the overlapping functionalities.

Tarr et al. [118] proposes multi-dimensional separation of concerns using *hyperslices* to address the problem of scattering and tangling. A hyperslice slices parts of different units (e.g., classes) that address a particular concern. Different hyperslices that address a particular concern are composed together using composition rules into *hypermodules.* Hypermodules can be nested to contain other hypermodules allowing more modularity and encapsulation. The result system, therefore, is a hypermodule that composes different hyperslices and the composition between hyperslices is based on the common concepts that these hyperslices share. The composition mechanism follows three simple steps: matching units that describe the same concept from different hyperslices, reconciling any differences may occur during the matching process, and finally integrating the hyperslices into a hypermodule. The technique of hyperslices is flexible to be used in requirement, design or the implementation levels.

Theme [28, 32] is an approach for separating concerns at the requirement and design levels. The Theme approach consists of Theme/Doc, which provides the framework for aspect-oriented analysis from the requirement document and allows the aspect to be traced to the design, and Theme/UML, which provides the framework for aspect oriented design. Theme refers to an element of the design that provides the structure and behaviour for a particular feature. There are two types of themes, *base themes*, which may interact with other themes and *crosscutting themes*, which functionalities overlay of the base themes. In Theme/Doc, the requirement analysis process starts with finding the key actions. Actions are verbs that appear in the requirement text and they are extracted by the software practitioner with the help of some lexical analysis implemented by Theme/Doc. For design, Theme/UML

provides a three phase approach: modelling phase, composition phase, and transformation phase. For the modelling phase, Theme/UML provides a marking profile to illustrate the entities in the design with the relevant composition semantics and a graphical UML tool that supports the Theme/UML marking profile and traditional UML features. The composition phase takes the output of the modelling phase and produces a composition model that is an instance of the composition metamodel using the composition specifications. The composition model is then executed to produce an object oriented Product Independent Model (PIM) in an EMF XMI file. Finally, the transformation phase takes the PIM produced in the composition phase and produce a Product Specific Model (PSM) model and eventually the code.

Nistor et al. [87] introduces ArchEvol for managing concerns at a high level of abstraction and tracing the concerns in the code. They develop a tool (an eclipse plugin) that visualizes the system as group of related concerns at the architecture level. Concerns are visualized as a tree to ease the navigation for developers. When the developer selects a concern from the tree, she can view the code fragments that concern is located.

Fuentes et al. [45] uses aspect orientation in developing context-aware pervasive systems. Context aware systems contain crosscutting concerns that are scattered in many components across the system. There is also a need to dynamically change these concerns during run time when the context is changed. The approach developed by Fuentes et al. addresses these issues. Fuentes et al. model the system using components and class diagrams in UML. They model the crosscutting concerns separately as aspects and develop a middleware that weaves aspects at run time. The base aspects in their approach do not refer to the aspects, the aspects are coupled with base classes as they have references to them as discussed earlier

which limits the potential of modularity, the authors are investigating to solve this in the future.

Solberg et al. [113] proposes a framework for model driven development that uses aspect oriented techniques to support separation of concerns. Their basic idea is to separate the concerns in an abstraction level using Aspect-Oriented modelling (AOM) techniques and have similar multiple levels of abstractions and provide transformations between the levels. Their framework aims to separate the concerns horizontally in an abstraction level and vertically across different abstraction levels. Using aspect oriented modelling to separate the concerns at a given level of abstraction and then use transformations to transform the models from one level to another. Their framework supports the transformation from PIM to PSM abstraction levels.

Hovsepyan et al. [55] introduces GReCCo (Generic Reusable Concern Composition), which is an AOM based concern composition framework. The concerns are developed in a best way to be independent from other concerns (i.e., increase their obliviousness) and thus increase their reusability. Concerns allow more variations in their composition by providing template parameters which increase the number of other concerns that can be composed with. GReCCo also uses composition symmetry and treats all concerns in a same way (i.e., it does not differentiate between base and aspect concerns). The composition between concerns in GReCCo is done through the composition model. A generic composition engine takes as input the concerns to be composed, and the composition model specifies the composition between the models and outputs the composed model.

In the previous subsection, we mentioned that encapsulating a crosscutting concern using components is limited. There exist approaches that support that using help from some

aspect-oriented technologies [114, 96]. Suvee et al, [114] introduces JAsCo, a language that allows encapsulating crosscutting concerns, such as access-control, and composing them with components. Their approach is built on AspectJ and aspectual components [73].

Support for customization in services is also limited, as discussed in the previous subsection. There are efforts, however, to customize services through parameterization [1] and using templates [119] . Web services have support for *adaptation* and *personalization*. Abiteboul et al. [10] proposes an approach that allows customized use of web services in XML documents. The approach uses an XML schema that specifies elements/subelements of the XML document that can be specified/replaced dynamically. They provide an example of a schema for news exchange, where the element <item> can be given by a service call that matches the news *service call pattern*, which allows to use any service call that returns an element (news <item>) of the correct type.

3.2.2 Feature Model Composition

Many software systems are developed that serve specific domains, and a system can grow to support different variants of services. Expressing the variability and supporting software reuse within a closed domain has been the main focus of Software Product Line (SPL) development [29, 97]. Concern is a unit of reuse that handles variability in the domain of the concern while ensuring openness for adaptation in arbitrary other domains. The combination of the variation and customization interfaces allows concerns to be malleable for reuse in domains or situations that were not originally envisioned. *Concerns* are close to SPLs, and therefore we incorporate best practices from this field. However, reuse within an SPL is of limited scope, i.e., within the product line. For example, Gomaa [48] expresses Software Produce Lines with extended UML models. He expresses variability using stereotypes, such

as, <<kernel>>, <<optional>>, and <<alternative>> and manually relates the model elements of the system with features in the feature model. The stereotypes used to express variability reside with the model, which negatively affects the understandability of the model. Dependencies between concerns - which can be translated to inter SPL dependencies – are not explored.

A number of different approaches have been proposed to compose feature models in SPLs. In [12], Archer et al. introduce FAMILIAR, a domain-specific language for feature model manipulation. FAMILIAR allows the user to manage feature models, compose them with different operators, and reason about their validity. Boškovic et al. [26] introduce Aspect-oriented Feature Models (AoFM), which combine aspect-oriented techniques with feature modelling. Their approach is capable of minimizing maintenance efforts by modularizing crosscutting concern (e.g. recurring subtrees or feature patterns) within feature models. Acher et al. [11] similarly support separation of concerns by defining merge and insert composition operators for feature models. Bak et al. [21] present Claefer, an approach that supports mixing feature models and metamodels. They support expressing the variability in a class diagram through using references to a feature model. They also allow creating specialized model templates by keeping most of the structure fixed, while allowing some options for variability.

Research in Multi Product Lines (MPL) use composition models and model interfaces to compose different interdependent SPLs together. Schröter et al. [110] provides interfaces for variability modelling, syntactical, behavioural and non-functional levels of the development process. The variability modelling interface is a separate model that restricts and specializes the variability model of the reused SPL. All other interfaces conform to the variability model

interface. The variability model in their approach comprises only a feature model of the SPL. Our variation interface, however, comprises feature and impact models. Rosenmüller et al. [102] automatically generate an initial composition model that connects multiple SPLs in an MPL. Composition models of different SPLs are connected by creating their instances, and the SPL developer can further refine the generated composition model by introducing constraints on the involved SPLs. The MPL approaches are different from our work. Our approach considers reuse from the start. The customization interface helps to identify the artifacts that are partially defined and need to become concrete when they are reused. This allows to define a development process that focuses on reuse, to create small concerns and reuse them to build larger ones. On the other hand, the approaches mentioned in this paragraph are founded on the SPL point of view, as they focus on how to use multiple SPLs to build an MPL. While their work is similar to the variation and usage interfaces of CORE, they do not provide means to develop artifacts that are partially defined, which is supported by our approach and identified by our customization interface.

There has been work to introduce aspect orientation to SPLs [42, 126], however, the research in this line is limited in applying aspect-oriented (AO) techniques within a product line, and not developing partially developed products that can potentially be composed with multiple SPLs. For example, Voelter et al [126], use aspect-oriented modelling techniques to compose optional features with the root feature, they also use Aspect-Oriented Pogromming techniques when generating code. However, as discussed previously, these aspects address the domain of interest of the SPL, and do not provide AO interface for cross-domain reuse by means of partially-developed products.

In Delta modelling [53, 31, 105], the product line is represented by a core model and a set of delta models. A valid configuration of the feature model applies the relevant deltas to the core model incrementally. Their work is related to the our previous work where we implemented the CORE metamodel for the design phase [16]. A valid configuration of a design concern applies its realized model increments to the base model incrementally to generate the customized concern.

The feature model composition operators defined by the aforementioned approaches are quite powerful, and we are investigating if we could exploit them to compose the feature models of our variation interfaces. The main difference between these approaches and our approach is that in our case the composition of feature models of different concerns is driven by concern reuse. Depending on the specific reuse, some features are kept, some reexposed, and some removed. In addition, composition of feature models in CDD takes into account trade-off analysis of stakeholders of the concerns, which is not explored by these approaches.

There is also research to use feature models in services and components. Cubo et al. [34] use feature models to address the variability of each service. A valid feature model configuration corresponds to a particular business process. They have a framework that allows to dynamically change the product configuration when the context is changed, White et al. [127] use feature models to represent the composition model of services, and to correctly reconfigure the system when a failure happens. When a failure happens, their approach uses feature models to derive correct and valid configurations.

Dynamic Software Product Lines (DSPL) [54] allow for dynamic configurations of SPL at runtime. Gomaa and Hashimoto [49] use feature models for dynamically selecting services that are requested at run time. They provide a three-layer architecture for DSPL, that

monitors the running system for any triggers that require reconfigurations, prepares feature requests that meets the new changes in the system, and reconfigures the system to meet the new feature requests. Dinkelaker et al. [38] use AO technology to model dynamic features of a DSPL. They use a domain specific language (called *dynamic feature language, DFL*) that models dynamic features as aspects, which allows them to express constraints on dynamic features and provides them the mechanism for safe composition of dynamic features. The DFL supports modelling late variation points of DSPL via using *domain specific pointcut language* that quantify over *domain specific join point models*. All these approach face the shortcomings of services and SPLs as previously discussed.

3.2.3 SPLs and Variable Modules

Through the composition of variation and customization interfaces, *Software Concern Lines* can be seen as a variable unit of reuse, tackling the problem of handling variability in predefined domains while ensuring a form of openness. This need was previously identified when reusing software components between product families – defining the notion of product populations [125], when proposing variable components [94, 122], when seeking more flexibility in a classic SPL setting [95], or when handling reuse in open-source communities [52].

In [122], Van der Storm defines variable components, but only proposes solving techniques for checking compatibility among them. Plastic partial components [94] also propose to handle variability in model-driven software architectures through components equipped with several variable interfaces and implemented internally with aspect-oriented techniques. Our contribution differs from these two propositions by being dedicated to concerns and by handling definition and composition of open variable parts in the reusable units.

In [59], Kästner et al. propose a core calculus and C-based implementation for variability-aware modules with variability handling capabilities inside modules and on module interfaces. Modular type checking of internal variability is supported and the composition of two compatible modules yields a well-typed module with combined variabilities. Our approach first differs as it aims at supporting hierarchical modularity [25] with concern hierarchies, by the provision of renaming and hiding capabilities and the separate handling of three specific interfaces. Our approach also considers an impact model to document the influence of expressed variability on system qualities and guide configurations.

3.2.4 Goal Models and Impact Analysis

Our approach uses goal models to analyze the impact of choosing features. Research in goal-oriented SPL investigated the mapping between features and goals in different ways.

Benavides et al. [24] extend feature models to include non-functional features (they call them extra-functional features) by expressing relations among feature attributes. They then map the extended feature models onto a Constraint Satisfaction Problem (CSP) that allows some automated reasoning. Users can ask questions to the CSP solvers such as the total number of products of a feature model and define some filters on the model. The extra-functional features in their approach are equivalent to the goals in impact models in our approach. However, their approach does not allow including stakeholders into the analysis. For instance, using their approach, we can not reason about the impact of a particular feature selection on developers versus managers, which is possible using impact models. Our approach also allows expressing dependencies between different impact models as we discussed in impact model composition, while expressing dependencies between extra-functional features of different feature models is not possible. In addition, our approach supports expressing relative

satisfaction values, which is helpful in earlier stages of software development when the requirements are not well defined. The attribute values of extra-functional values come from specific domains and can not be relative. Finally, the performance of their approach deteriorates exponentially when using more than 25 features. Our approach provides linear scalability.

Siegmund et al. [111] present SPL Conqueror, an approach to reason about the non-functional properties in SPLs. They classify non-functional properties into feature-wise quantifiable, variant-wise quantifiable and qualitative properties. Similar to [24], the non-functional properties are encoded in an extended feature model. The variant-wise properties require generating variants, which is costly. They do not discuss assigning properties such as performance to features instead of variants. SPL Conqueror is useful to present the variability of final products, as discussed in their case study section. In our approach, the concerns that we are developing are not customized and are not final products. Hence, a modeller can reason about variability during the development process which is not addressed by SPL Conquerer. In addition, concern dependencies and reasoning about stakeholders are not possible using this approach.

The MPL approach described in the previously [110] provides textual description of non-functional impacts. Our impacts are described using goal-models, which are more formal and provide a concise way to express dependencies among non-functional properties which makes tool-based evaluation possible.

Moreira et al. [78] proposes an approach for multi-dimensional separation of concerns at the requirement level. Their proposed model treats both functional and non-functional concerns uniformly and allows to specify the compositions between concerns. They also

support trade-off analysis to resolve conflicts that arise during compositions. Unlike impact models, their trade-off analysis is done using simple tables and does not allow composing impacts of different concerns.

Asadi et al. [20] also annotate feature models with non-functional properties and use Hierarchical Task Network (HTN) [72] to find the optimal feature configuration given stakeholders' goals and preferences. HTN is a popular technique to generate plans that lead to a goal from initial descriptions and goal conditions. Our approach expresses non-functional (impact models) properties separately from feature models, which allows composition of impact models as previously discussed. However, we can use the techniques provided by Asadi et al. to derive optimal feature configuration that satisfy given stakeholders' goals.

Finally, impact modelling is inspired by work of Mussbacher et al. [84], where they introduced the AoURN-based SPL framework that integrates the concepts of SPL with the Aspect-oriented User Requirements Notation (AoURN) [83]. Their approach allows evaluating an SPL while taking into account trade-off analysis for stakeholders and complex constraints that are not expressed in traditional feature models. Their framework uses the Goal-Oriented Requirement modelling (GRL) notation for goal modelling. Similar to other approaches [131, 19, 112], GRL is influenced by the popular framework for conceptual modelling i* [130]. Impact models of CDD builds on some of the techniques described in [84], for example, when integrating high-level goals during concern composition.

3.3 Conclusion

Over the years, many units of reuse emerged that allow producing new software by reusing existing software artifacts. Reuse units such as classes, components, and services provide ways for reuse through interfaces, libraries, and manual instructions. However, as shown in

the comparison table in this chapter, these popular existing reuse units do not support all key characteristics for reuse. Some reuse units such as classes, aspects and components do not express variability, do not express non-functional properties and do not support impact analysis. Similarly, design patterns and services do not support customization and do not encapsulate crosscutting concerns. CORE concerns build on the strengths of the reuse units discussed in this chapter to allow modular broad-based model reuse. In addition to comparing concerns with other reuse units, we study the efforts in the literature to support some of the reuse characteristics, highlighting the differences between our approach and those efforts. In the next chapter, we delve into the CORE reuse process in detail.

Chapter 4
Concern Reuse Process

Building a concern can be a non-trivial and time consuming task, typically done by or in consultation with a domain expert. Deep understanding of the nature of the concern is required to be able to identify its different features (and capture them in a feature model), to model the common properties and differences of all features of a concern at all relevant levels of abstraction (by building models that (i) realize the features of the concern using the most appropriate modelling notations and (ii) are eventually refined into executable specifications), and to express the impact of the different variants on high-level stakeholder goals and system qualities (using impact models). However, reusing an existing concern is very easy and involves a simple three-step reuse process. This chapter introduces this three-step concern reuse process in the next section, and we illustrate the process through the *Authentication* concern, both using requirement and design modelling notations, in Section 4.2. We discuss delaying decisions and reexposing features in Section 4.3. Section 4.4 concludes this chapter.

4.1 Concern Reuse Process

As we discussed earlier in Chapter 2, a concern has three interfaces that facilitate reuse. Reusing an existing concern is simple, and essentially involves 3 steps. Each step uses one of the concern interfaces:

Step 1: The concern user must first *select the feature(s) with the best impact* on relevant stakeholder goals and system qualities from the *variation interface* of the concern based on provided impact analysis. To maximize the reusability of the

concern that is being built, the user should select from the concern that is being reused only the features that are absolutely necessary to achieve the required functionality and goals. Decisions about potential use of alternative features or optional features should be deferred by reexposing them (see Section 4.3). Based on this configuration, the modelling tool then merges the models that realize the selected features to yield new, user-tailored models of the concern corresponding to the desired configuration. Depending on the root phase of the concern, the merging may involve requirement models and/or design models.

Step 2: The concern user has to adapt the generated user-tailored models of the concern to the application context by mapping customization interface elements to application-specific model elements. Again, depending on the root phase of the concern, this step might require customizing requirement models and/or design models.

Step 3: The concern user can use the functionality provided by the selected concern features which are exposed in the usage interface of the user-tailored, customized models within his own application models. In requirements models, this may mean including workflow segments exposed in the concern's usage interface in the application's workflow models. In design models expressed using sequence diagrams, for instance, using a concern may involve instantiating a class exposed in the concern's usage interface and/or calling one of its public operations.

Typically, a domain-specific or solution-specific concern reuses other general-purpose concerns. Hence, concern hierarchies allow the developer to modularize the application into

different layers of abstraction. However, these layers again have to be flexible. To successfully reduce complexity, layers should allow for separate reasoning, for hiding the complexity of lower levels from upper levels. On the other hand, the layers cannot be completely opaque, since the structure and behaviour of most lower-level concerns crosscuts the structure and behaviour of the upper levels (and the application). At the very least, the quality of the upper level depends heavily on the qualities of the lower levels.

4.2 Example: Authentication

In this section, we show in practice the concern reuse process through reusing an example *Authentication* in a *Bank* application. *Authentication* is a security concern that requires the user to be authenticated before performing a system functionality. It provides different means to authenticate, e.g., through passwords, retinal scans, or facial recognition. Although the focus of this thesis is on the design phase, to illustrate the CORE reuse process at more than one development level, we model the *Authentication* concern both at requirement and design levels. For the requirement level, we use the Aspect-oriented User Requirements Notation (AoURN) [83], and for the design level, we use Reusable Aspect Models (RAM) [66, 62]. However, the focus of this chapter as well as this thesis in general, is not on how to apply any of these individual modelling notations, describing in detail how to model a system with such a modelling notation. For that, the interested reader may consult existing documentation about feature models [58], AoURN models [82], and RAM models [61]. The focus is rather on how to reuse existing concerns that encapsulate such models to build an application.

4.2.1 Variation Interface of Authentication

As described earlier, the variation interface of a concern consists of feature and impact models. Kang *et al.* [58] introduced feature models to capture the problem space of a Software

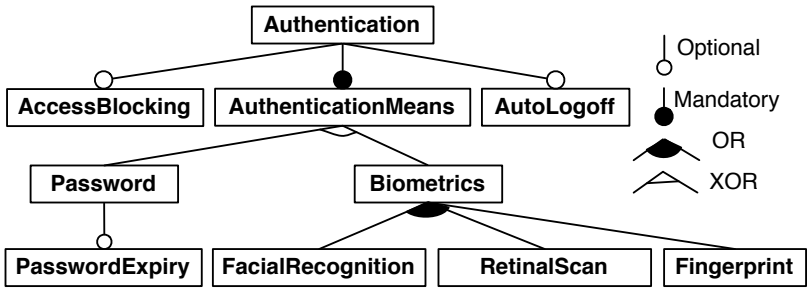

Figure 4–1: Feature Model for Authentication

Product Line (SPL). A feature model captures the potential features of members of an SPL in a tree structure, containing those features that are common to all members and those that vary from one member to the next. A particular member is defined by selecting the desired features from the feature model, resulting in a feature model configuration [35]. A node in a feature model represents a feature of the SPL (e.g., *AccessBlocking* in Fig. 4–1). A set of inter-feature relationships allow to specify (i) mandatory and optional parent-child feature relationships as well as (ii) alternative (XOR) and or (IOR) feature groups (see legend in Fig. 4–1). A mandatory parent-child relationship specifies that the child is included in a feature model configuration if the parent is included. In an optional parent-child relationship, the child does not have to be included if the parent is included. Exactly one feature must be selected in an alternative (XOR) feature group if its parent feature is selected, while at least one feature must be selected in an or (IOR) feature group if its parent feature is selected. Often, *includes* and *excludes* integrity constraints are also specified, which cannot be captured with the tree structure of the feature model alone. An *includes* constraint ensures that one feature is included if another one is. An *excludes* constraint, on the other hand, specifies that one feature must not be selected if another one is. Note that integrity

constraints are not required to express the variation interface of the example *Authentication* concern.

Fig. 4–1 shows the feature model that captures all the distinctive, user-visible characteristics of the *Authentication* concern. The root feature *Authentication* has one mandatory feature (*AuthenticationMeans*) and two optional features (*AccessBlocking* and *AutoLogoff*). *AccessBlocking* enforces that the user has only a limited number of attempts to authenticate, and the system blocks the user when she exceeds the limit. *AutoLogoff* ensures that the user is logged off if she remains idle for longer than a certain specified time. The user must select one of the *AuthenticationMeans*. There is an XOR relationship between the children of *AuthenticationMeans*, which ensures that only one of its children can be selected (either *Password* or *Biometrics*). *Password* has an optional child, *PasswordExpiry*, which forces the user to renew her password when it expires after a certain time. *Biometrics* has three subfeatures with an IOR relation between them (*RetinalScan*, *Fingerprint*, and *FacialRecognition*). This means the user can select any number of these features simultaneously.

In addition to the feature model, our example *Authentication* concern provides an impact model for three system qualities that helps the user perform trade-off analysis when selecting the features. We choose goal models for impact analysis because goal models allow vague, hard-to-measure system qualities to be evaluated, such as user convenience or security, in addition to more quantifiable qualities such as cost, performance, or memory usage. Goal modelling is typically applied in early requirements engineering activities to capture stakeholder and business objectives, alternative ways of satisfying these objectives, and the positive/negative impacts of these alternatives on various high-level goals and quality aspects. The analysis of goal models guides the decision-making process, which seeks to

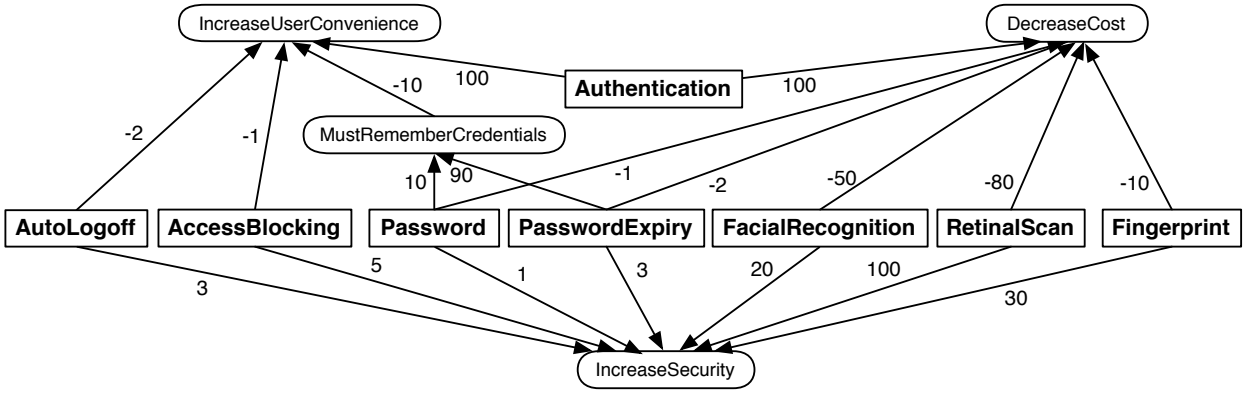

Figure 4–2: Impact Model for *Authentication*, with Impacts *DecreaseCost, IncreaseUserConvenience*, and *IncreaseSecurity*

find the best suited alternative for a particular situation. These principles also apply in our context, where an impact model is a type of goal model that describes the advantages and disadvantages of features offered by a concern and gives an indication of the impact of a selection of features on high-level goals that are important to the user of the concern.

In the context of CORE, the goal model for the variation interface is called impact model, not only because the main focus is on capturing the impact of features on qualities, but also as a reminder that goal models in CORE are different from traditional goal models, i.e., their use is more restricted and specialized. Impact models use features ($\square$) in place of tasks (e.g., *Password* in Fig. 4–2), and they exclusively use quantitative contributions ($\longrightarrow$) to express the impact on goals ($\bigcirc$) of importance for the concern (e.g., *DecreaseCost* in Fig. 4–2). Fig. 4–2 shows the impact model of the Authentication concern with the goals *IncreaseSecurity*, *DecreaseCost*, and *IncreaseUserConvenience*. Features are linked to the goals that they impact, annotated with a contribution weight (i.e., value on a link) that ranges between -100 and 100. When a goal model is evaluated, values called initial satisfaction values are assigned

to nodes in the goal model, and then propagated throughout the goal model taking its links into account until the satisfaction values of all nodes in the goal model have been determined. Therefore given a selection of features (called a *configuration*), an impact model can be evaluated to determine how well each goal is satisfied by the configuration. The satisfaction value of a feature is either 0 when it is not selected or 100 when it is selected. The satisfaction and contribution values are propagated upwards using a weighted sum divided by 100 and restricted to be between 0 and 100 (as described in the goal modelling literature [18, 56]). As a result, the satisfaction value of a goal always ranges between 0 and 100.

For example, Fig. 4–2 shows that the most secure configuration is the one where *RetinalScan* is chosen, as it yields a satisfaction value for *IncreaseSecurity* of 100. Choosing *Password*, *PasswordExpiry*, *AccessBlocking* and *AutoLogoff* would result in $[(100 * 1) + (100 * 3) + (100 * 5) + (100 * 3) + (0 * 20) + (0 * 100) + (0 * 30)]/100 = 12$, which is less than, for instance, choosing only *FacialRecognition*. Fig. 4–2 also shows the impact on the *IncreaseUserConvenience* goal. In this case, all features decrease the contribution of the root feature *Authentication*. Two features, namely *Password* and *PasswordExpiry*, require remembering credentials, hence, they contribute to a subgoal called *MustRememberCredentials* which contributes -10 to *IncreaseUserConvenience*. For example, if *Password* is selected, the satisfaction value of *MustRememberCredentials* subgoal will be $[(100 * 10) + (0 * 90)]/100 = 10$. However, the contribution of the root feature *Authentication* must be considered when calculating the satisfaction value of *IncreaseUserConvenience*, resulting in $[(10 * -10) + (0 * -1) + (0 * -2) + (100 * 100)]/100 = 99$. If both *Password* and *PasswordExpiry* features are selected, then the satisfaction value of *IncreaseUserConvenience* will be $[(([(100 * 10) + (100 * 90)]/100) * -10) + (0 * -1) + (0 * -2) + (100 * 100)]/100 = 90$.

4.2.2 Requirement modelling of Authentication

AoURN is an aspect-oriented extension of the User Requirements Notation (URN), a requirement modelling notation standardized by the International Telecommunication Union (ITU) [56]. We use AoURN to model the *Authentication* workflow, i.e., the flow of use cases and the system-user interactions, with the help of the Aspect-oriented Use Case Map (AoUCM) notation (a subnotation of AoURN).

Fig. 4–3 shows the basic AoUCM model (also called workflow model) for *Authentication*, which realizes the root feature in the feature model in Fig. 4–1. A feature can be realized by more than one model or no model at all according to the CORE metamodel in Chapter 6. The AoUCM model also shows the interactions of the root feature with other features of *Authentication*, i.e., *AccessBlocking*, *AutoLogoff*, *PasswordExpiry*, and *AuthenticationMeans*. A feature is shown as a static stub (◇), which serves as a container for one other AoUCM model that specifies the details of the feature, hence realizing the corresponding feature from the feature model in Fig. 4–1. The dashed outline of the stub for *AuthenticationMeans* indicates that this is a dynamic stub and contains several features, i.e., it contains several detailed AoUCM models for subfeatures of *AuthenticationMeans*. If a feature is not selected, the workflow continues through a stub without visiting its lower-level AoUCM models.

The *Authentication* workflow starts at the *authenticate* start point (●) and ends at the pointcut stub (⊕), if successful, or otherwise at the *fail* end point (▮), i.e., only if the pointcut stub is reached is the application reusing the *Authentication* concern allowed to continue. The start point is followed by a responsibility (×) to check whether the |*User* is already authenticated. A responsibility identifies a step in the workflow. The workflow then continues to the stubs for the *AccessBlocking* and *AutoLogoff* features, which both can either succeed

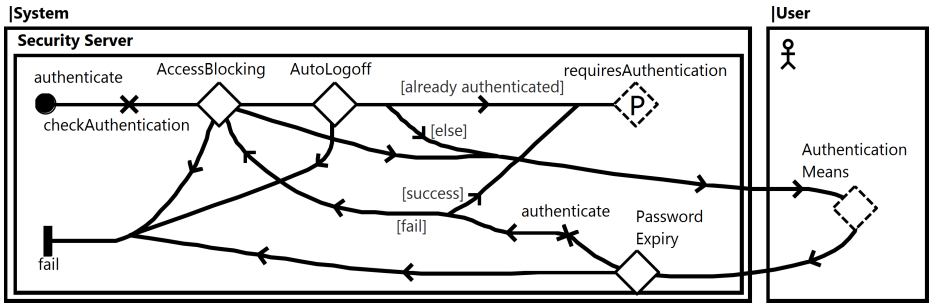

Figure 4–3: AoUCM Model for *Authentication*

or fail. In the latter case, the workflow ends at the *fail* end point (▮). If successful, the workflow continues on to an OR-fork (—≺) with two branches corresponding to the possible results of the earlier *checkAuthentication* responsibility: one where the |*User* is *already authenticated* and one where the |*User* is not. In the *already authenticated* case, the workflow successfully ends at the pointcut stub. In the *else* case, the workflow exits the *Security Server* subcomponent of the |*System* component and enters the |*User* component (▭). In the |*User* component, subfeatures of *AuthenticationMeans* (i.e., *Password*, *RetinalScan*, *Fingerprint*, or *FacialRecognition*) are used to enter the user's credentials. Afterwards, the *PasswordExpiry* feature is executed if selected, which again may fail or succeed. In the success case, the credentials are authenticated and, if that is also successful, the workflow ends at the pointcut stub. If authentication fails, then the *AccessBlocking* feature keeps track of how many invalid authentication attempts the |*User* has made and blocks the |*User* if there were too many or otherwise allows the |*User* to reenter the credentials.

The *requiresAuthentication* pointcut stub (🎯) is a special type of stub that represents the locations where the *Authentication* workflow is to be applied in the application reusing the *Authentication* concern. To reuse the workflow, the reusing application has to specify a pattern identifying those locations. The pattern is then matched against the workflow

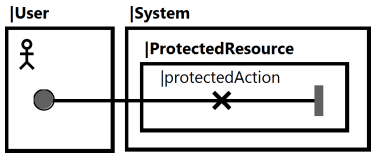

Figure 4–4: Predefined AoUCM Pattern for *Authentication*

model of the application, yielding the actual locations where the *Authentication* workflow is to be inserted. Because the *Authentication* workflow occurs before the pointcut stub in the AoUCM model, the *Authentication* workflow is inserted before the identified locations.

The *Authentication* workflow comes with a predefined pattern shown in Fig. 4–4 that states that the *Authentication* workflow is needed whenever an interaction between the |*User* and the |*System* causes a |*protectedAction* of a |*ProtectedResource* of the |*System* to be performed. The model elements preceded with a vertical bar | have to be customized by the reusing application and hence constitute the *customization interface* for the AoUCM *Authentication* model. The *usage interface*, on the other hand, is defined by the *authenticate* start point and all the start points of lower-level AoUCM models for the various features depicted by stubs.

As mentioned earlier, each feature in the feature model is represented by a stub in the *Authentication* AoUCM model, and each stub is a container for a lower-level AoUCM model. As examples, Fig. 4–5 shows the AoUCM model for *Password*, which simply consists of only one responsibility to enter credentials. The *PasswordExpiry* AoUCM model in Fig. 4–6, on the other hand, checks whether the password is expired and also allows checking the new credentials entered by the |*User*.

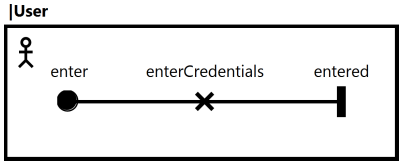

Figure 4–5: AoUCM Model for *Password*

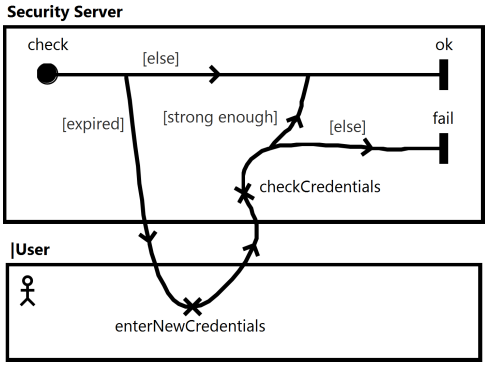

Figure 4–6: AoUCM Model for *PasswordExpiry*

4.2.3 Design modelling of Authentication

We model *Authentication* at the design level using Reusable Aspect Models (RAM). Reusable Aspect Models (RAM) [62] is an aspect-oriented multi-view modelling approach for software design modelling. A RAM model consists of a UML package specifying the structure and the behaviour of a software design using class, sequence, and state diagrams. In this thesis, we only focus on class and sequence diagrams.

The Authentication Feature

Each feature in the feature model in Fig. 4–1 is realized by a RAM model. Fig. 4–7 shows the RAM model that realizes the root feature *Authentication*. The model consists of three compartments that show the structural view, reuses, and message views. The structural view is the design class diagram for *Authentication*. The class *Session* represents the session that is authenticated, and has an association to the |*Authenticatable* class, which is a partial

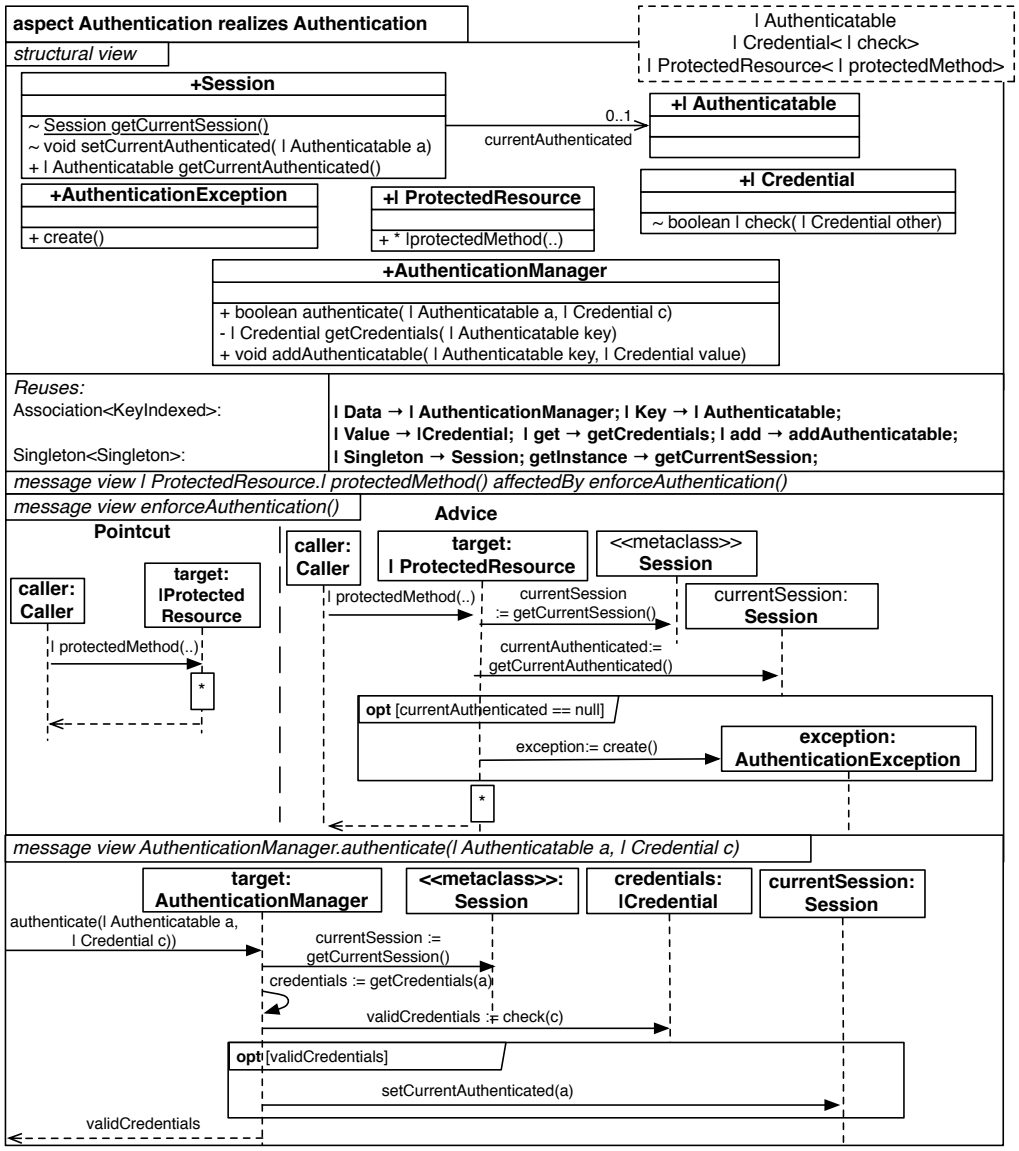

Figure 4–7: *Authentication* RAM Model

class. Partial classes and methods in RAM are preceded with a vertical | and designate the customization interface of the RAM model. Upon reuse, all partial classes/operations need to be completed by mapping them to application-specific classes/operations. Note that classes and operations that precede with a + modifier are designated to have public visibility, whereas the ones that precede with ~ or - are respectively designated to have concern or private visibility. A model element with concern visibility is only visible within the concern, i.e., only other models within the concern can access them. During the reuse process in RAM [16, 14], the class |*Authenticatable* is mapped to a concrete class representing the entity in the system that needs to be authenticated. There are two other partial classes, |*Credential* and |*ProtectedResource*, which along with their partial methods comprise the *customization interface* of the *Authentication* RAM model and are presented in a dotted box at the top right of the model. The class |*Credential* represents the credential that needs to be checked (through the |*check* method) and the class |*ProtectedMethod* contains the method that requires authentication when it is called.

The reuse compartment in Fig. 4–1 shows that the *Authentication* aspect reuses two concerns. The first reuse is the *Association* concern, which allows instances of |*Data* to be associated with instances of |*Associated*. The reuse selects the feature *KeyIndexed*, which provide |*Data* with the functionality of looking up instances of |*Associated* using a key. The customization mappings, shown in the right of the reuse compartment, specify that the partial class |*Data* of the *KeyIndexed* model is mapped to *AuthenticationManager*, |*Key* is mapped to |*Authenticatable*, and |*Value* is mapped to |*Credential*. The second reuse in the reuse compartment is the *Singleton* concern, which implements the *Singleton* design pattern. The customization maps the class |*Singleton* to *Session* to ensure that there is only

one instance of *Session* when the system is running. For both reuses, the operations are mapped similarly to classes.

In the message view compartment we only show two message views for space reasons. The first is an *aspect message view* for |*protectedMethod* called *enforceAuthentication*. Aspect message views consist of two parts: pointcut and advice, and message views can specify by which aspect message views they are affected (e.g., |*protectedMethod* is *affected by enforceAuthentication* as shown in Fig. 4–7). The pointcut part shows the intercepted message in the message view where the advice part will be inserted. In the example here, the intercepted message is the call to |*protectedMethod()* (in the pointcut part), and the advice is inserted before the rest of the |*protectedMethod* body (represented by a box with a star). The advice of *enforceAuthentication* gets the *currentAuthenticated* from the *Session*, and, if it does not exist, throws an *AuthenticationException*. The second message view *authenticate* takes two parameters (|*Credential c* and |*Authenticatable a*). It checks whether its passed |*Credential c* is valid by first calling *getCredentials* on |*Authenticatable a* to find the stored credentials for *a*, and then comparing the returned credentials with *c* by calling *check*.

The Password Feature

Fig. 4–8 shows the RAM model that realizes the *Password* feature. This RAM model *extends* the *Authentication* RAM model. In RAM, when a model extends another model, all model elements of the extended model become visible in the extending model [15], allowing *Password* to access classes or operations from the *Authentication* aspect. We discuss model extension in detail in Chapter 8. The structural view of *Password* adds the class *Password* and the implementation class *String*. Implementation classes are preceded with a stereotype *<<impl>>* and refer to classes that belong to the target implementation language (in this

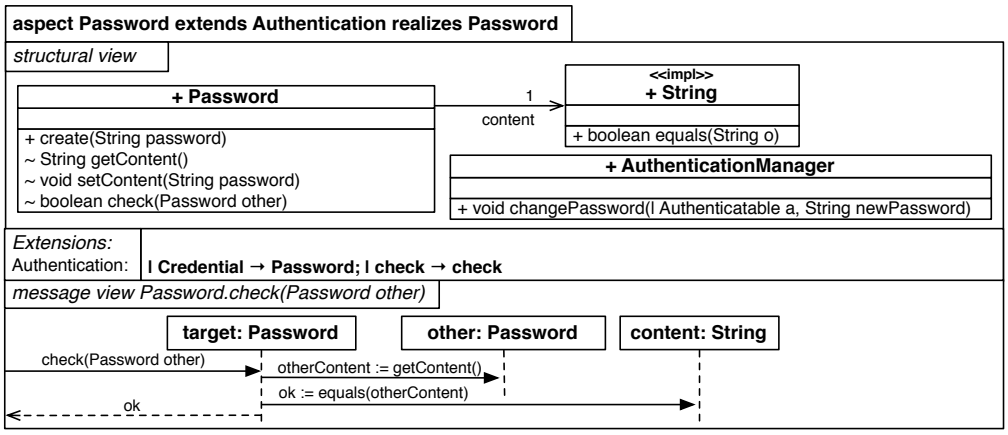

Figure 4–8: *Password* RAM Model

case Java). The class *Password* has an association to a *String* instance called *content* to store the content of the password. Furthermore, a new operation *changePassword* is defined for the *AuthenticationManager* class. Finally, we show one message view *check* of *Password*, which takes as input another password and checks if both passwords have the same content.

In the extensions compartment, we can see that |*Credential* from *Authentication* is mapped to the class *Password*, and the operation |*check* is mapped to a concrete operation *check* in *Password*. Because the *Password* model extends *Authentication*, when the two aspects are woven together, both classes of *AccountManager* (in *Password* and *Authentication* aspects) are going to be merged together, resulting in one class that contains all the methods.

The PasswordExpiry Feature

The model of *Password* is further extended in *PasswordExpiry* in Fig. 4–9 to allow the password to expire after a certain time set by the user. The idea is to force the user to periodically update the password to make it more secure. To do that, the *setPassword*

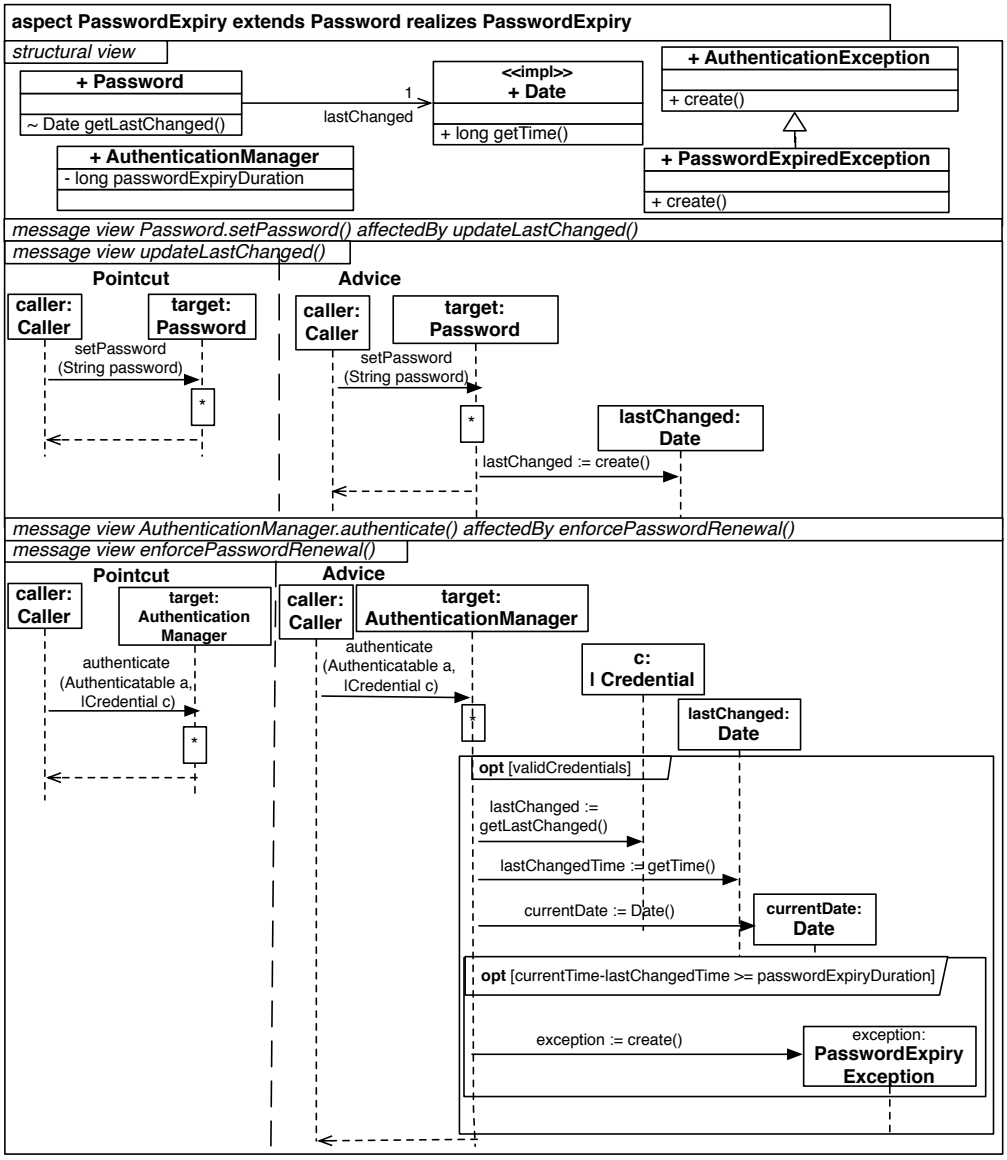

Figure 4–9: *PasswordExpiry* RAM Model

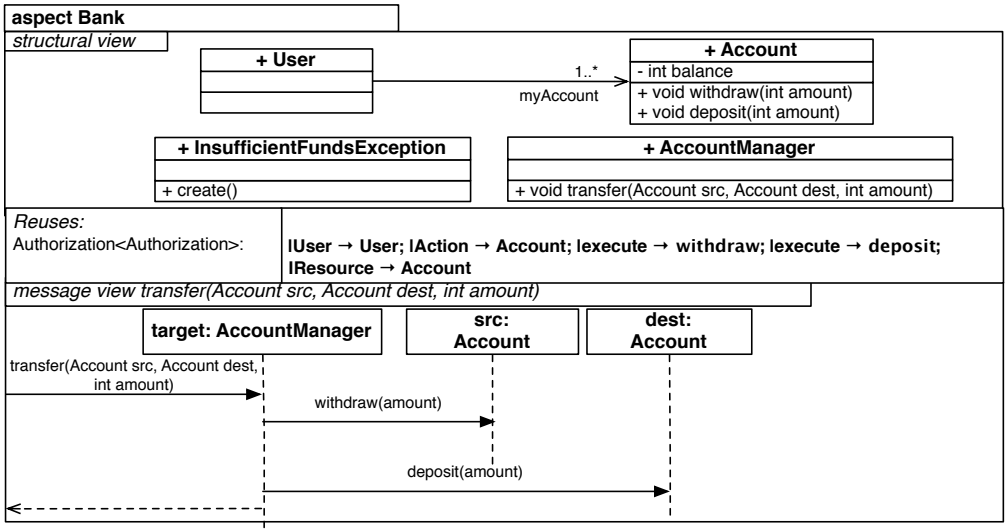

Figure 4–10: RAM Design Model for a Simple *Bank* Application

method is advised by the *updateLastChanged* aspect message view to remember the last time the password was changed. The second aspect message view makes sure that the password is changed periodically. To do so, the advice of *enforcePasswordRenewal* adds after the original message of *authenticate* an optional combined fragment where it checks if the $currentTime - lastChangedTime \geq passwordExpiryDuration$.

4.2.4 Reusing the Authentication Concern

Suppose that we have a simple *Bank* application as illustrated in Fig. 4–10. In our example, *User* has an association with *Account*, which has two operations *withdraw* and *deposit*. The class *AccountManager* defines an operation *transfer* to transfer money from one account to another. The behaviour of *transfer* is shown in the message view compartment, which withdraws a given amount from the source account and deposits it into the destination

account. To ensure that only authenticated users can execute a transfer, we are going to reuse the *Authentication* concern in the *Bank* application.

Step 1: Selecting the Desired Features using the Variation Interface

To select the features of *Authentication* that best fulfill the quality requirements and goals of the bank, the designer of the bank uses the variation interface of the *Authentication* concern to perform a trade-off analysis. As shown in the reuse compartment of Fig. 4–10, the designer chose to select *Authentication* (which is always selected as it is the root feature), *AuthenticationMeans* (which is always selected as it is a mandatory subfeature of the root), the *Password* and *PasswordExpiry* features. Using the impact model of *Authentication* shown in Fig. 4–2 the tool determines that this feature selection is relatively cheap (the *DecreaseCost* goal is evaluated to $[(100 * 100) + (100 * -1) + (100 * -2) + (0 * -50) + (0 * -80) + (0 * -10)]/100 = 97$), but decreases the user convenience $[(([(100 * 10) + (100 * 90)]/100) * -10) + (0 * -1) + (0 * -2) + (100 * 100)]/100 = 90$ as shown previously in Section 4.2.1. *PasswordExpiry* is chosen although it decreases user convenience, because it is a cheap feature that increases security albeit slightly (IncreaseSecurity is evaluated to $[(0 * 3) + (0 * 5) + (100 * 1) + (100 * 3) + (0 * 20) + (0 * 100) + (0 * 30)]/100 = 4$).

Design Models. As explained before, after the user selects the desired features of the concern she is reusing, the modelling tool composes the realization models of the selected features to yield a realization of the concern that only contains the features that the user intends to use. In our bank example, the RAM weaver composes the realization models corresponding to the selected features (*Authentication* shown in Fig. 4–7, *Password* shown in Fig. 4–8, and *PasswordExpiry* shown in Fig. 4–9) to create the *Authentication* concern

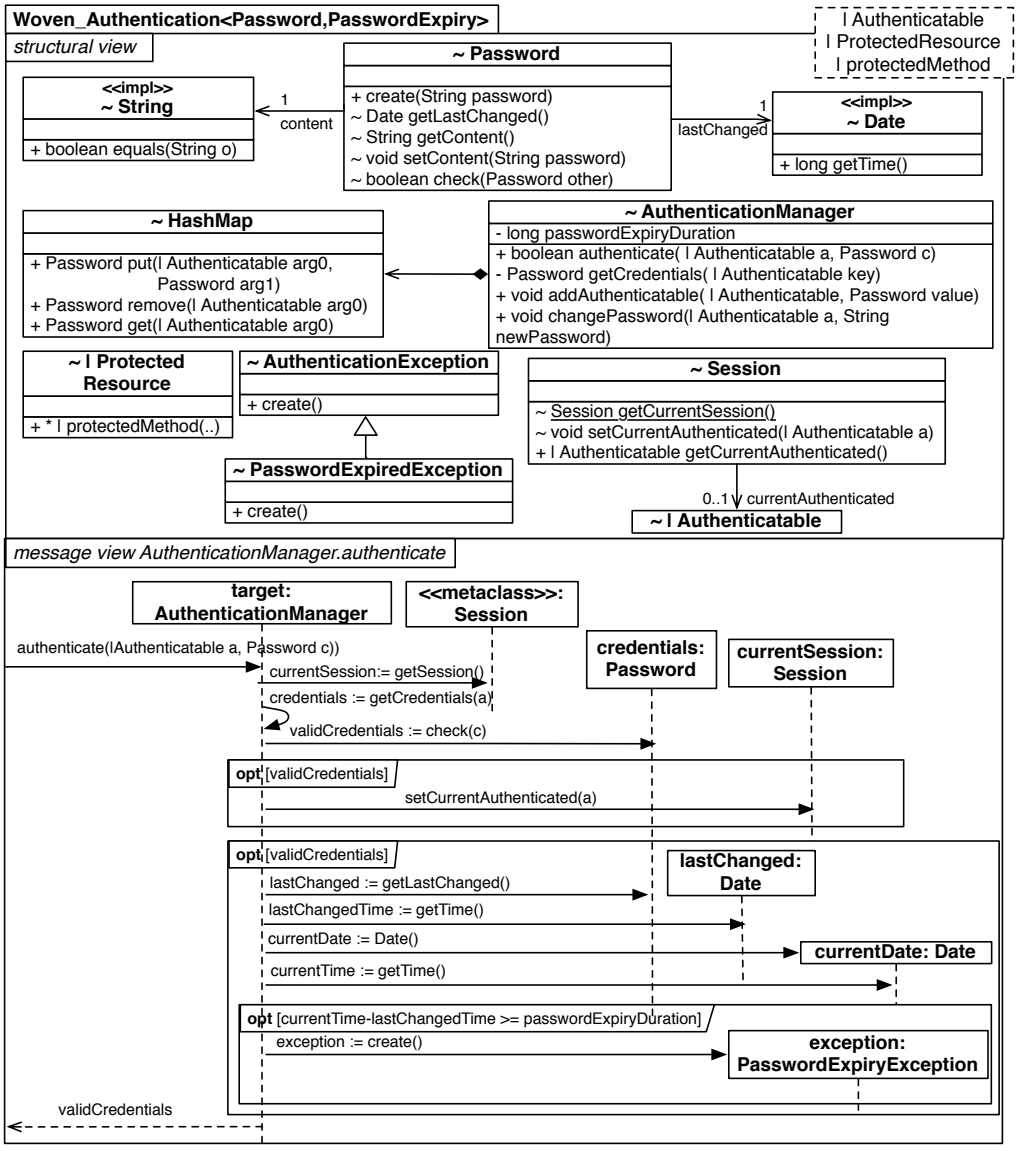

Figure 4–11: Woven Model of Features *PasswordExpiry*, *Password*, and *Authentication*

realization model tailored to this specific selection as shown in Fig. 4–11. The model composition uses the RAM weaver, which performs structural and behavioural weaving recursively by traversing the instantiation directives as illustrated in [62, 64]. The structural weaving is done by performing class merge between the classes of the same name or between the classes that have mappings in the instantiation directives (e.g., between |*Credential* and *Password* in Fig. 4–8). The combined and merged classes for all three aspects (*Authentication*, *Password*, and *PasswordExpiry*) are shown in the structural view of Fig. 4–11. For behavioural weaving, RAM uses the algorithm specified in [64]. Recall that the message view *authenticate* in the *Authentication* aspect is advised in *PasswordExpiry* by an aspect message view which defines behaviour that is added after the body of the *authenticate* method (see Fig. 4–7 and Fig. 4–9). The RAM weaver inserts the advice part of *PasswordExpiry* after the message body of *authenticate* and produces a final woven message view for *authenticate* as in Fig. 4–11.

Requirements Models. Similarly, the AoUCM model (see Fig. 4–3) tailored to the needs of the *Bank* application also only contains the selected features, i.e., the *AccessBlocking* and *AutoLogoff* stubs are now empty and therefore skipped, while the *AuthenticationMeans* stub only contains the *Password* model. Last but not least, the *PasswordExpiry* stub still contains the *PasswordExpiry* model.

Step 2: Adapting the Reused Concern to the Bank using the Customization Interface

The *customization interface* of the generated, user-tailored *Authentication* concern realization consists of the three partial components |*User*, |*System*, and |*ProtectedResource* and the partial responsibility |*protectedAction* in the AoUCM model as well as the two partial classes |*Authenticatable* and

|*ProtectedResource* and the partial operation |*protectedMethod* in the RAM model that need to be mapped to model elements in the *Bank* application.

Requirements Models. When the *Authentication* concern is reused by the *Bank* application, the application developer has to finalize the AoUCM pattern shown in Fig. 4–4 by specifying the concrete model elements of the application that correspond to the model elements in the *customization interface*: |*User* = *Customer*, |*System* = *Bank*, |*ProtectedResource* = *Account* or *AccountManager*, and |*protectedAction* = *deposit* or *withdraw* or *transfer*. Therefore, the customized, user-tailored AoUCM model for the *Authentication* concern is created in this case simply by replacing – according to the reuse directives – |*User* with *Customer* and |*System* with *Bank* in Fig. 4–3.

Design Models. According to the reuse directives in Fig. 4–10, the *Bank* application customizes *Authentication* by mapping |*Authenticatable* to the class *User* and |*ProtectedResource* to the *AccountManager* as well as the *Account* class. The |*protectedMethod* operation is mapped to *transfer*, *withdraw*, and *deposit*, since the bank wants to ensure that only authorized users can perform transactions on accounts. The customization step results in a new model, where all the specified mappings have been applied to the corresponding model elements. Therefore, the customized, user-tailored design model for *Authentication* concern is the same as the tailored model of *Authentication* shown in Fig. 4–11, with the partial elements replaced by those specified in the reuse directives defined in this paragraph.

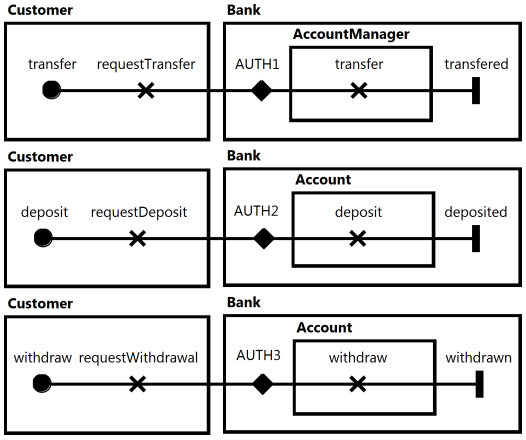

Figure 4–12: Woven AoUCM Model for a Simple *Bank* Application

Step 3: Using the Reused Concern in the Bank through the Usage Interface

The *usage interface* consists of all the start points in the customized, user-tailored AoUCM model of the *Authentication* concern as well as all the public classes and operations defined in the customized, user-tailored *Authentication* design model, which can now be used from within the *Bank* application.

As a result of reusing the *Authentication* concern, the *AUTH* aspect markers (◆) are added automatically by the AoUCM composition mechanism before the matched pattern in the final *Bank* application example in Fig. 4–12, i.e., when an aspect marker is reached, the *Bank* application workflow continues with the *Authentication* workflow and only returns to the *Bank* application workflow if the pointcut stub is reached.

The final design model that combines the design model of the *Bank* application and the design model of the customized, user-tailored *Authentication* concern is shown in Fig. 4–13. The aspect message view |*protectedMethod* is woven with *transfer* as shown in the message view compartment where the body of |*protectedMethod* is inserted before the message view

of *transfer*. The *withdraw* and *deposit* message views are also woven similarly (not shown here). Note that the visibility of elements of the reused concern (*Authentication*) are changed from *public* to *concern* (preceded with ~ in Fig. 4–13), to apply the information hiding principles [92] by concealing internal design details, in this case the elements of the reused concern in the woven model, from the rest of the application.

4.3 Delaying of Decisions

The previous section presented models of the *Authentication* concern at both requirement and design levels. As explained in Chapter 2, a concern encapsulates *sets of models* that describe relevant properties at all levels of abstraction required to sufficiently understand the concern. Typically, a requirement concern, e.g., security, needs to comprise not only models that specify different ways of achieving security (authentication, role-based access control, encryption, etc.), but also different ways of realizing them (password-based authentication vs. biometrics, etc.). Even for a given realization, there are different possible implementation architectures (centralized password server vs. local, distributed databases, etc.). It comes with no surprise that low-level design solutions, such as various design patterns, transaction controls, or resource allocation are quite general solutions that can be reused in many contexts. Complex applications consist of many intertwined, interacting concerns, and concern-orientation advocates developing an application by reusing as many already existing concerns as possible. The same principle applies to the development of a concern itself.

4.3.1 Concern Hierarchies

To fully reap the benefits of reuse, it is therefore important to allow the creation of *concern hierarchies*. To increase scalability and avoid duplication of effort, a high-level concern (or to

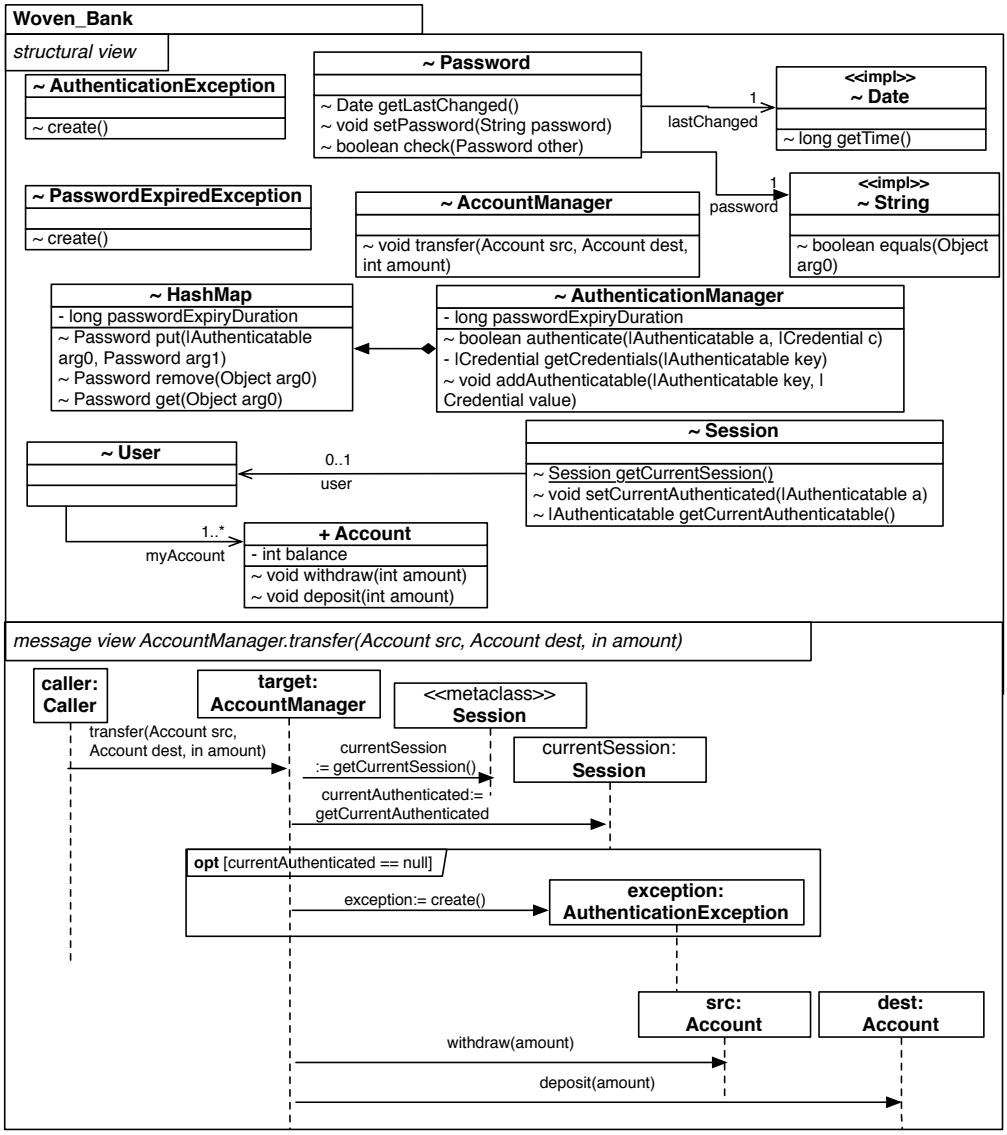

Figure 4–13: Woven RAM Model for a Simple *Bank* Application

be more precise, a feature of a high-level concern) should be able to reuse the functionality (structure / behaviour / properties) of a lower-level concern when appropriate. Doing so creates a concern hierarchy where a concern at a higher level of abstraction reuses other lower-level concerns. Similarly, a more domain-specific or solution-specific concern can reuse other more general concerns.

Concern hierarchies allow the developer to modularize the application into different layers of abstraction. But these layers again have to be flexible. In order to successfully reduce complexity, the layers should allow for separate reasoning, and hide the complexity of the lower levels from the upper levels. On the other hand, the layers must be composable, since the structure and behaviour of most lower-level concerns crosscuts the structure and behaviour of the upper levels (and the application). At the very least, the qualities of the upper level are heavily influenced by the qualities of the reused concerns at the lower levels.

Concern-orientation addresses this problem by allowing the concern designer to precisely specify how the variation, customization, and usage interface of the concern being built are affected by the interfaces of the lower-level concerns that are reused.

4.3.2 Reexposing Features

When building a reusable concern A, the concern designer of A who is designing a feature FA might decide to reuse another, lower-level reusable concern B. In this situation, the concern designer should select from B only the features that are absolutely necessary to achieve the required functionality and goals that FA needs. If there are several features within B that can provide the required functionality but with different impacts, the concern-orientation paradigm advocates to defer such decisions by reexposing those features in the

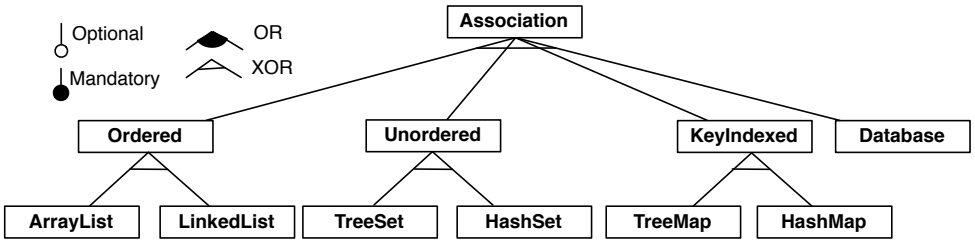

Figure 4–14: Feature Model for the *Association* Concern

variation interface of A. As a result, they become subfeatures of FA in the feature model of A, where they encode different variations of achieving the functionality of FA.

For example, we demonstrated in the previous section how the *Bank* application reuses *Authentication* using both AoURN and RAM models. The RAM models of *Authentication* reuse other low-level concerns such as *Singleton* and *Association*. The design of the *Bank* application, hence, creates a three-level hierarchy of concern reuse. In the design of *Authentication* (see Fig. 4–7), the *AuthenticationManager* needs to lookup the |*Credential* of a given |*Authenticatable*, which can be achieved by reusing the *Association* concern. Fig. 4–14 shows the feature model of the Association concern. The features *KeyIndexed* (with subfeatures *TreeMap* and *HashMap*) as well as *Database* provide the needed key-based lookup functionality. At this point, though, *Authentication* does not know which of these features would be most appropriate. This is only known by the user of *Authentication*, which in our case is the *Bank* application. Only the bank knows that the mapping from |*Authenticatable* to |*Credential* should be persisted by using the *Database* feature, for example, even if this solution is slower and costs more. In other reuse contexts, using a *Database* might have been prohibitively slow.

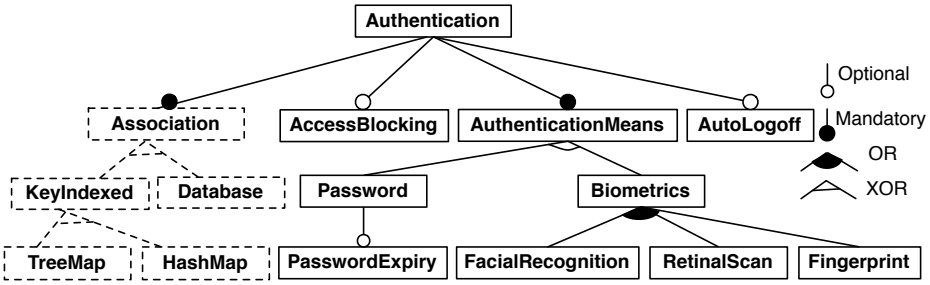

Figure 4–15: *Authentication* Concern with the Features *KeyIndexed* and *Database* Reexposed from the Reused *Association* Concern (dashed boxes show the reexposed features)

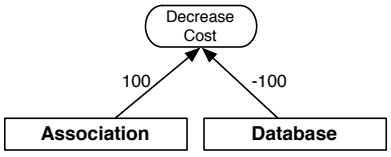

Figure 4–16: Impact Model for *DecreaseCost* for the *Association* Concern

Therefore, instead of directly reusing an arbitrary feature that provides key-based lookup, the designer of Authentication opts for reexposing the parent feature of *HashMap* and *TreeMap (KeyIndexed)* along with *Database* in the variation interface of the *Authentication* concern. This allows the designer to defer feature selection to a later point when more requirements and desired qualities have been determined.

Fig. 4–15 shows how the *Authentication* feature model reuses and reexposes features from the *Association* concern. The reused root feature of *Association* becomes a mandatory child of *Authentication*, and only its reexposed subfeatures are added to the *Authentication* variation interface, according to feature model composition algorithms for concerns [121].

If features of a reused concern are reexposed in the variation interface of the reusing concern, the impact model of the reused concern needs to be connected with the impact

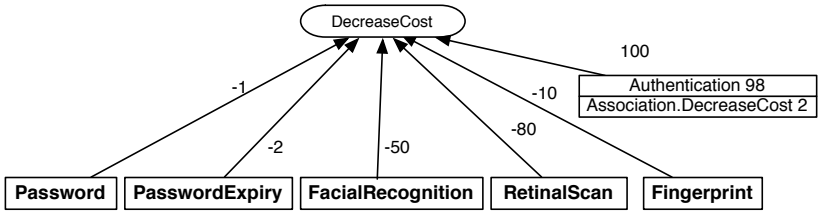

Figure 4–17: Impact Model for *DecreaseCost* with Cost High-Level Goal of *Association* Being Reexposed

model of the reusing concern. This is necessary because the features of the reused concern have an impact on the goals and qualities of the reusing concern, and hence their impacts need to be considered in the trade-off analysis.

Fig. 4–16 shows the impact model for *DecreaseCost* in the *Association* concern. All features of Association contribute 100 to *DecreaseCost* (shown by a contribution link with weight 100 from the parent feature *Association* to the goal), except *Database* which contributes -100. When *Database* is selected, the satisfaction value of *DecreaseCost* becomes $[(100 * 100) + (100 * -100)]/100 = 0$, i.e., the worst choice from a cost point of view.

When *Association* is reused in *Authentication*, the impact model for *DecreaseCost* of *Association* contributes to the impact model for *DecreaseCost* of *Authentication* as shown in Fig. 4–17. Since the reuse is done by the *Authentication* root feature, the model element representing the *Authentication* feature in the impact model lists all the impacts of its reused concerns that contribute to *DecreaseCost*. In our example, the cost coming from *Association.DecreaseCost* contributes 2 to the overall cost of the feature, whereas the contribution of the feature itself is 98.

To evaluate the impact model, the contributions of the reused goals are combined with the contribution of the feature itself. In our example, if the concern user of Authentication

selects *FacialRecognition* and the reexposed *Database* feature, the satisfaction value for the *Authentication* feature will be $[(0 * 2) + (100 * 98)]/100 = 98$, which is then combined with the contributions of all other features *of the Authentication* concern to yield $[(0 * -1) + (0 * -2) + (100 * -50) + (0 * -80) + (0 * -10) + (98 * 100)]/100 = 48$.

4.4 Conclusion

Creating a new concern can be a nontrivial task that requires considerable domain knowledge and expertise. However, once a concern is created, it becomes easy to reuse it to build other concerns or applications. This chapter outlined a simple three-step concern reuse process through reusing an example Authentication concern in a Bank application. The reuse process starts with selecting features from the variation interface and performing a trade-off analysis based on the impacts of the feature selection on the high level system qualities. A user-tailored version of the concern is then generated, the user adapts this concern to be application-specific by mapping the partial elements from its customization interface to concrete elements from the application. Finally, the concern user can use the functionality provided by the selected concern features which are exposed in the usage interface of the user-tailored, customized models within her own application models. In the next chapter, we discuss the composition rules and algorithms that allow the interfaces and the realization models of the reusing concern and the reused concerns to be composed.

Chapter 5
Composition Rules and Algorithms

For the CORE reuse process to work, detailed algorithms are needed to compose concern interfaces and models. This chapter discusses in detail the CORE composition rules and algorithms. It starts by listing the requirements for appropriate composition mechanisms on the three types of interfaces of a *Software Concern Line*, explaining how a concern can reuse other concerns. Then, it describes detailed algorithms that specify how the variation interface that consists of feature models and impact models of the reusing concern and the variation interface of the reused concern are composed based on the concern reuse specification. It notably describes how a reusing concern can internally depend on features of lower-level, reused concerns, or decide to defer decisions and as a result reexpose relevant lower-level features in its own interface. Furthermore, the chapter shows how goal-based impacts of lower-level concerns are integrated with high-level impact models to yield a variation interface that specifies the combined impact of the reused and reusing concerns. The chapter then presents an algorithm that, based on a selection of features from the variation interface of a concern, generates the composition of the associated realization models to yield a user-tailored concern realization, applying dedicated resolution models to deal with feature interactions, if necessary. Finally, it explains how the customization and usage interfaces of the reusing concern are affected by the interfaces of the user-tailored reused concern realization.

5.1 Considerations for Interface Composition.

When designing a concern, the developer decides which variations, customizations, and functionality to expose in the three interfaces. When a concern reuses another concern, the interfaces of the reused concern may have an impact on the interfaces of the reusing concern. Hence, rules need to be defined that describe how interfaces are composed in concern hierarchies. The following considerations have to be taken into account:

C1 Choosing the best variant of a reused concern is only possible once all desired system qualities are known. These may not be known at intermediate levels in a concern hierarchy, but only at the top when the complete application is being built.

C2 A concern encapsulates all possible variants that can be useful in any context. When reused in a specific context, some of these variants may not be applicable.

C3 The qualities of the reusing concern are affected by the qualities of the reused concerns.

C4 In order to obtain an executable application, all customization elements must be concretized.

C5 In alignment with information hiding principles [92], internal details of reused concerns that are irrelevant for the developer should be encapsulated and hidden whenever possible to reduce complexity and minimize unnecessary dependencies.

This chapter's main contribution is to present composition rules and algorithms that address these considerations as detailed in the following sections.

5.2 Feature Model Composition

The rest of this chapter uses a crisis management system (CMS) to illustrate concern hierarchies and explain how variation interfaces are composed. Crisis management involves

identifying, assessing, and handling a crisis situation. A CMS facilitates this process by orchestrating the communication between all parties involved in handling the crisis. The CMS allocates and manages resources, and provides access to relevant crisis-related information to authorized users of the CMS in a timely and reliable manner [63]. Chapter 9 discusses the bCMS case study in more detail.

In a CMS, resources such as vehicles and emergency personnel need to be tracked and assigned to missions. The feature model of the generic, reusable *Resource Management* concern is shown on the left of Fig. 5–1. The main functionality that this concern provides is to keep track of the availability of *resources*. Resources can be allocated to *tasks*, in which case they are marked as unavailable until they are released again. This base functionality is part of the mandatory feature *Allocation*. In more elaborate cases, resources can be heterogenous, i.e., each resource can exhibit a set of possibly different *capabilities*. In that case, *Resource Management* offers the possibility to search for available resources that exhibit *desired* capabilities. This is encoded in the optional feature *Search*. Because finding an optimal solution for assigning resources with capabilities to tasks is *np*-complete, the algorithm that finds the optimal solution is provided in *Optimal*, an optional subfeature of *Search*.

To associate a task with many resources, the mandatory *Allocation* feature reuses a low-level design concern called *Association* (a condensed version of its features is shown on the right of Fig. 5–1). *Association* is useful whenever some design object needs to be related to multiple objects of some other design class. The association might need to be *ordered*, or *indexed by a key*, or even *persisted*. There are many ways of designing such an association, ranging from using a simple array or linked list, to using hash tables or even databases.

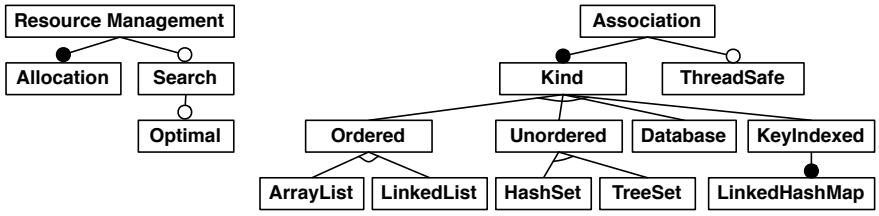

Figure 5–1: *Resource Management* and *Association* FMs

Furthermore, if the association implementation is used in a multi-threaded application, it should also be *thread safe*.

To address C1 and C2 from Section 5.1, the developer should be able to *explicitly reexpose* all features of the reused concern that provide the required functionality in the reusing concern's variation interface (and indirectly exclude those features that do not have the desired properties) to defer the decision about which specific variant to use to the next level in the concern hierarchy. Hence, for each concern reuse, we propose that the reusing concern needs to specify the variation interface composition as follows:

1. *Select the features that the reusing concern needs from the variable features of the reused concern as specified in its variation interface.* The best strategy from a reuse standpoint is to select the *minimally required features* from the reused concern. This selection is used to compose the realization models associated with the features as presented in Section 5.4 to generate a customization and usage interface of the reused concern for this specific selection. These interfaces are then used within the realization models of the reusing concern wherever the functionality provided by the reused concern is needed.
2. *Reexpose features of the reused concern in the variation interface of the reusing concern, if appropriate, so that they may be reused by even higher-level concerns or the final*

application. Sometimes, the reused concern offers features that provide alternatives to the functionality minimally needed by the reusing concern. However, since the desired system qualities of the final application are not known to the developer of the reusing concern, it is impossible to decide which alternative is the best choice. Also, sometimes the reused concern provides additional, variable functionality that could also be useful to the developers of even higher-level concerns in certain contexts. In such cases, the best strategy from a reuse point of view is to propagate any alternative and optional features of the reused concern to the variation interface of the reusing concern by explicitly reexposing them (and optionally renaming them to better reflect their semantics in the context of the reusing concern).

The syntax for specifying a concern reuse is:

```
FEATURE 'reuses' CONCERN'<'[FEATURE_LIST]
  ['reexpose' RENAMING_FEATURE_LIST]'>'
```

where the optional FEATURE_LIST stands for a comma-separated list of selected features, and the optional RENAMING_FEATURE_LIST represents another comma-separated list of reexposed features with optionally specified renamings in the form of FEATURE 'to' NEW_NAME.

In our resource management concern example, certain features that *Association* provides are not useful in the context of *Allocatable*, e.g., *Key-Indexed* and its subfeatures, since we do not need to index resources with a key. Likewise, there is no need for the collection of resources that are associated with a task to be ordered. The functionality required by *Allocation* is offered by any of the subfeatures of *Unordered*, and, depending on the requirements of the application that are going to reuse *Resource Management*, it might also make sense to use a database to persistently keep track of allocations between tasks and resources. Since the

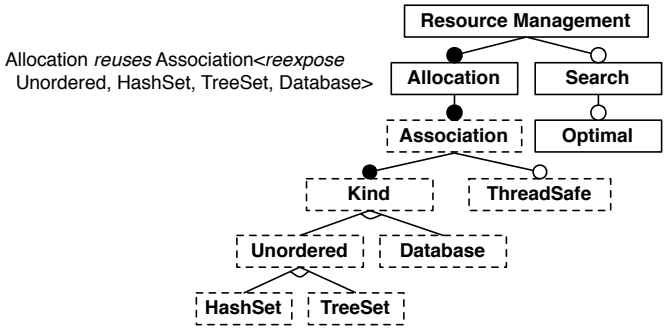

Figure 5–2: *Resource Management* FM with Reexposition

developer of the *Allocation* feature in the *ResourceAllocation* concern does not know which one of these variants is the most appropriate for future users of *ResourceAllocation*, and since it is not known if *Resource Management* is going to be used in a multi-threaded environment, these choices are reexposed in the variation interface of *ResourceAllocation*. Fig. 5–2 shows how parts of the variation interface of *Association*, namely *Unordered*, *TreeSet*, *HashSet*, *Database*, and *ThreadSafe*, are *reexposed* as subfeatures of *Allocation*. The equivalent textual concern reuse specification is shown on the left of Fig. 5–2 (no selected feature, four reexposed features without renaming).

Based on the concern reuse specification defined by the developer, a CORE tool first verifies the correctness of the reuse, and then composes the variation interface of a reusing and reused concern as described by the following rules and algorithms.

Verification of Reuse Specification

Because of the intricate interaction of selection and reexposing, the correctness rules for specifying concern reuse are as follows:

- General Rule: All ancestor features (from the parent of a feature to the root feature) of a selected or reexposed feature must either be mandatory, selected, or reexposed.
- XOR Group Rule: If the parent of an XOR group is mandatory, selected, or reexposed, then either exactly one feature of the XOR group must be selected, or at least two features of the XOR group reexposed.
- OR Group Rule: If the parent of an OR group is mandatory, selected, or reexposed, then either at least one of the features of the OR group must be selected, or at least two reexposed.
- Requires and Excludes Rules: A feature that is required by a selected feature has to also be selected. A feature that is required by a reexposed feature has to be either selected or reexposed. A feature that is excluded by a selected feature is neither allowed to be selected nor reexposed. A feature that is excluded by a reexposed feature is not allowed to be selected.

If these rules are followed, then either the current selection is already a valid selection in the classical sense of feature models, or it is possible to complete the current selection with additional features from the set of reexposed ones to obtain a valid selection.

Composition Algorithm

Based on a valid reuse specification defined in feature F of the reusing concern, Algorithm 1 composes the feature model of the reused concern (REUSED) with the feature model of the reusing concern (REUSING).

There is a special case that requires the feature model of the reusing concern to be restructured after step 5 and before executing step 6. Attaching a mandatory subfeature to

Algorithm 1 Feature Model Composition Algorithm

1. Create REUSED_Copy, a copy of REUSED.
2. Remove from REUSED_Copy all features that are not selected or reexposed. The root feature is always kept. A mandatory feature is kept if all its ancestor features are either mandatory, selected, or reexposed.
3. For each feature F_i in an XOR group in REUSED_Copy: if F_i is selected, replace the XOR link from F_i to the parent feature with a mandatory link (given the verification rules explained above, there can only be at most one such selected feature).
4. For each feature F_i in an OR group in REUSED_Copy: if F_i is selected, replace the OR link from F_i to the parent feature with a mandatory link.
5. For each feature F_i in an OR group in REUSED_Copy: if F_i is reexposed and if at least one feature F_j in the OR group is selected, replace the OR link from F_i to the parent feature with an optional link.
6. The root feature of REUSED_Copy is added to REUSING as a mandatory subfeature of F.

a parent feature is structurally only possible if the parent feature is not the parent of an XOR or OR group. Otherwise the resulting model is not a valid feature model anymore. In this case, an intermediate, mandatory child feature of F is introduced as the new parent of the XOR or OR group. After the restructuring, step 6 can be executed safely.

5.3 Impact Model Composition

To address C1, C2, and C3, specific algorithms are needed to compose impact models of the reusing concern with those of the reused concern.

An impact model is a variant of a GRL goal graph [56] and contains features (which represent specific solutions), goals (for intentions/objectives), and weighted contribution links between features and goals. Contribution weights are expressed relatively by numerical values. Satisfaction values for nodes in the impact model determine the degree with which an element is satisfied, and typically range from 0 (not satisfied at all) to 100 (fully satisfied). During impact model analysis, satisfaction values of leaf elements in the goal model (typically

solutions) are propagated up towards the root(s) of the goal model (typically relevant system qualities). Consequently, trade-off analysis compares the satisfaction values of high-level system qualities given the sets of solutions that are to be considered. Even though feature models may be augmented with feature attributes to capture system qualities [23], CORE uses goal models – called impact models in the context of CORE – to more easily express complex relationships among goals.

Fig. 5–3 shows the impact model part of the variation interface of the *Association* concern for the *Performance* quality. This allows each feature to be compared against the other features of the concern with respect to a single quality. A key point of impact models is that a feature's comparison against other features is relative, e.g., the *HashSet* feature with contribution 100 increases performance twice as much as the *LinkedHashMap* feature with contribution 50. The impact model, however, does not capture whether this is an improvement from 100 milliseconds to 50 milliseconds or 2 minutes to 1 minute – the evaluation remains strictly relative. This allows high level qualities of interest to stakeholders such as user convenience, security, or ease of use to be captured and reasoned about, even though these qualities can typically not be measured precisely. The best solution in a concern for a particular quality results in a satisfaction value of 100 for the top-level quality goal (e.g., this is the case for *HashSet* in the performance example).

The quality properties of a concern depend on the design of the models encapsulated within the concern, but also on the quality of the reused concerns. Given a feature selection for each reused concern, the impact model associated with the reused concern yields satisfaction values for several qualities. All of these impact models contribute to the satisfaction of qualities of the reusing concern. Consequently, there is a need to compose all impact models

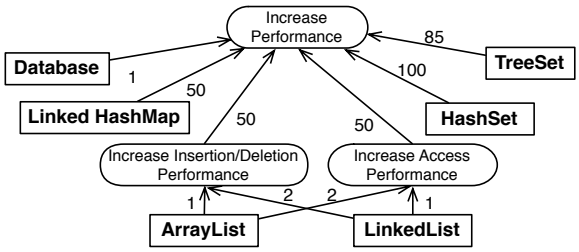

Figure 5–3: *Association* Impact Model

of reused concerns with the impact model of the reusing concern. Since each quality can be addressed individually (i.e., the quality's impact models from the reused concerns only need to be composed with the same quality's impact model of the reusing concern), we will use one quality as an illustrative example: *Performance*. Furthermore, the composition mechanism is the same regardless of whether a high-level concern reuses one or several lower-level concerns. Therefore, we can illustrate without loss of generality the composition of impact models with the performance impact model.

An impact model may contain several goals as shown in the *Performance* example, where the top-level goal is further refined into two lower-level goals. For some features of the *Association* concern it is necessary to differentiate between *Insertion/Deletion Performance* and *Access Performance*. Note that at each level, sibling features are compared in a relative manner (e.g., *ArrayList* contributes twice as much to access performance as *LinkedList*). The variation interface is defined by all goals in the impact model. Hence, the variation interface of the *Association* concern contains the three goals from the impact model shown in Fig. 5–3.

There are two key prerequisites for successful composition and subsequent impact analysis of concern hierarchies:

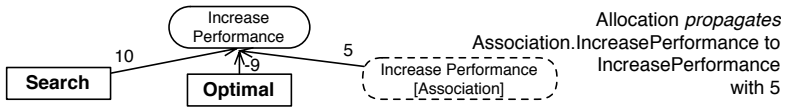

Figure 5–4: *Resource Management* Impact Model

1. *Comparability.* It must be possible to compare the results of impact models. In particular, this requires that the scale used within an impact model does not conflict with the scales of all other impact models. In other words, if the satisfaction results for a quality in the impact models of two reused concerns are the same value n, then the relative degree of satisfaction for each of the two reused concerns must be the same.
2. *Determinability.* It must be possible to determine for a particular quality of a reused concern, if the set of features selected from the reused concern is the optimal or worst solution for this quality of the reused concern.

These two prerequisites are satisfied by following the relative comparison scheme for contributions and by normalizing the result of the contributions to the [0,100] range for each node within each impact model. In other words, the unifying feature across all impact models is the fact that the best possible solution results in a satisfaction value of 100 and the worst in 0. If this is the case, then the developer again only needs to think about the relative comparison of features of the reusing concern and the respective quality goals of the reused concerns as shown in Fig. 5–4. In this figure, it is determined that the *Search* feature of the *Resource Management* concern contributes twice as much to increase performance as the performance goal from the reused *Association* concern (10 vs. 5)[1] . In other words, *Search*

[1] Of course, when the impact model is evaluated during impact analysis, a goal from a reused concern only contributes to the goals of the reusing concern if the feature that reuses

contributes two thirds and the reused concern potentially one third. If the reused concern contributes the best possible performance result (i.e., its satisfaction value is 100 because the *HashSet* feature is selected in the reuse specification), it will contribute the full third to the performance goal of the reusing concern (i.e., 33 after normalization to the [0,100] range). If the satisfaction value of the reused concern is less than 100 (e.g., because the *Database* feature is selected instead), then the contribution is proportionally less. Hence, continuity from the reused impact models to the impact model of the reusing concern is ensured.

The composition of a reused impact model with the impact model of the reusing concern is hence the set of contribution links including their weights that have to be added from the reused impact model to the reusing one. In the example in Fig. 5–4, this set contains only one link for the reuse of the *Association* concern (i.e., the link between *Increase Performance [Association]* and *Increase Performance*). The composition can either be specified graphically or textually given the following syntax (see Fig. 5–4):

(FEATURE 'propagates'
  CONCERN'.'GOAL 'to' GOAL 'with' WEIGHT)*

where FEATURE is the high-level feature reusing the lower-level CONCERN, the first GOAL is the source quality from the reused concern defined in the variation interface, the second GOAL is the target quality from the reusing concern, and WEIGHT is a relative number. Any number of such statements including zero may be specified for a feature that is reusing a lower-level concern.

---

the concern is actually selected. In the case of *Resource Management*, the reuse is done in *Allocation*, which is a mandatory feature, and hence always selected.

Recall that the performance impact model in Fig. 5–3 contains three goals. Because all of them are part of the variation interface, any of them could be used in the composition specification depending on the needs of the reusing concern. In the case of the *Resource Management* impact model, it was decided that the top level goal *Increase Performance* should be used instead of the other goals for insertion/deletion as well as access performance.

From the composition mechanism described in the previous paragraph, it follows that a goal from a reused impact model is only important to the reusing concern, if it contributes to one of the goals defined explicitly for the reusing concern. All other goals of the reused concern are irrelevant (i.e., those goals that are not connected to a goal explicitly defined for the reusing concern). Only goals from the reusing concern are part of the reusing concern's variation interface. For example in Fig. 5–4, all goals with solid outlines are part of the variation interface (only *Increase Performance* in this case) and all goals with dashed outlines (i.e., they come from reused concerns – only *Increase Performance [Association]* in this case) are not part of the variation interface of the *Resource Management* concern. Intuitively, it is not necessary to include goals with dashed outlines in the variation interface, because these goal must have a contribution to a goal with a solid outline, which is part of the variation interface. Therefore, the contribution of the reused concern still propagates upwards the impact model hierarchy.

Up until now, the contribution of the *Optimal* feature in Fig. 5–4 has not been discussed. This feature highlights the need to take into account feature relationships (see Fig. 5–2) when determining contribution weights. Consider the following feature selections for the *Resource Management* concern and the resulting satisfaction value of *Increase Performance*:

- HashSet → Increase Performance [Association] = 100 → Increase Performance = 33 (normalized, because we selected 100*5 out of a possible maximum of 100*15)
- Database → Increase Performance [Association] = 1 → Increase Performance = 0.33 (normalized, because we selected 1*5 out of a possible maximum of 100*15)
- HashSet, Search → Increase Performance [Association] = 100, Search = 100 → Increase Performance = 100 (normalized, because we selected 100*5 + 100*10 out of a possible maximum of 100*15)

Assume that the search functionality of the *Resource Allocation* concern can be improved through the selection of the *Optimal* feature, but at a performance cost, which has been determined to be at a factor of 10, i.e., the optimal search is ten times slower than the regular search feature. Since the impact model captures relative comparisons between features, one could be tempted to assign the weight of 1 to the contribution of the *Optimal* feature (10% of the weight of the contribution of the *Search* feature, which is 10). However, the impact model is always evaluated based on a valid selection of features. As the selection of the optional *Optimal* feature also requires its parent to be selected (i.e., the *Search* feature), the satisfaction value would be determined as:

- HashSet, Search, Optimal → Increase Performance [Association] = 100, Search = 100, Optimal = 100 → Increase Performance = 100 (normalized, because we selected 100*10 + 100*1 + 100*5 out of a maximum of 100*16)

Obviously, this is not the desired result. The *accumulative impact* of the *Search* and *Optimal* features on performance is supposed to be 10 times less than *Search* on its own (or in other words, *Optimal* reduces the performance of *Search* by 90%). Consequently, the correct weight assignment for the *Optimal* feature is -9 as shown in Fig. 5–4, because the resulting

satisfaction value of the *Increase Performance* goal is then 40 (normalized, because we selected 100*10 + 100*-9 + 100*5 out of a maximum of 100*15), as expected. This clearly shows that the topology of the feature model must be taken into account when determining contribution weights.

In general, the following guidelines help with the definition of impact models:

- Each variable feature in a feature model has its own contribution link to a quality. The weight of the contribution link of a variable feature F considers other variable features in the feature model that have to be selected because F is selected.
- Because there is no choice in the selection of mandatory features, the impact of all mandatory features on a quality can be summarized into a single contribution link from the root feature to the quality. Even this single contribution is often omitted, especially when the topology of the feature model guarantees that at least one variable feature must be selected (e.g., the feature model contains an OR or XOR group with a mandatory parent), thus guaranteeing an impact to the quality in any case. Note that the contribution of a variable feature with mandatory children takes the contributions of its mandatory children into account, i.e., there are no separate contributions from the mandatory children as the contribution from the variable parent features also covers the mandatory children.
- Features in an OR group have to take into account that there are potentially many simultaneous impacts from siblings.
- Features in an XOR group can rely on the fact that there will only be one impact from siblings at a time.

Algorithm 2 Impact Model Composition Algorithm

1. Make a copy of the impact model of the reused concern.
2. Create the specified contribution links between the copy and the reusing impact model, resulting in a composed impact model.

The composition algorithm for impact models is quite straightforward, as shown in Algorithm 2 :

5.4 Composing the User-Tailored Concern Realization

5.4.1 Dealing with Feature Interactions

Feature interaction refers to a situation where the functionality provided by a feature is influenced by the presence of other features that are not among its ancestors. Just like in SPLs, feature interactions can occur in *Software Concern Lines*. The concern designer, who is a domain expert, is the one who must identify all feature interactions within a concern and determine whether each interaction can be resolved or not.

Unresolvable feature interactions, sometimes also called feature conflicts, are situations in which it does not make sense to simultaneously use the conflicting features in combination. For example, the *Association* concern shown in Fig. 5–1 encapsulates different ways of implementing an ordered association: an array (feature *ArrayList*) or a linked list (feature *LinkedList*). Both achieve the same goal, and hence it does not make sense to use them in combination. Similarly, in transaction processing systems, there are different ways of providing support for recovery, including the creation of physical copies of the data (*SnapShot*), deferring the update of the original data (*Deferred*), and keeping intention lists (*IntentionLists*). Also, there are different ways of performing concurrency control, e.g., *LockBased* and *Optimistic*. However, it makes no sense to use *Optimistic* concurrency control with *SnapShot* recovery, since optimistic concurrency control allows the concurrent execution of

potentially conflicting transactions. In case a transaction is aborted, the recovery support has to undo the changes of the aborting transaction only, which is not always possible with snapshot-based recovery. Unresolvable feature interactions have to be expressed by the concern designer in the concern's feature model using *XOR* or *excludes* relationships. As a result, the concern user is prevented from choosing conflicting features when using the concern's variation interface during step 1 of the reuse process.

In many cases, though, feature interactions can be resolved. This is the case in situations where it makes sense to use the features in combination, but the structure and behaviour of the realization of one or several of the interacting features need to be changed to provide the desired, combined functionality. Resolvable feature interactions therefore do not need to be exposed to the concern user, as they can be dealt with by the concern designer in the realization. For example, the *Association* concern shown in Fig. 5–1 has a *ThreadSafe* feature that should be selected by the user of *Association* in situations where the reusing concern is accessing the association from multiple threads. The realization of a thread-safe association differs significantly from a non-concurrent realization depending on the data structures that are used. It typically requires acquiring and releasing semaphores whenever the underlying data structures are accessed. In Java, for example, a standard *ArrayList* should be replaced with a *CopyOnWriteArrayList* to correctly deal with multi-threading. Similarly, in transaction processing systems, the implementation of *LockBased* or *Optimistic* concurrency control has to adapt when transactions can be nested. Therefore, for each resolvable feature interaction, the concern designer needs to create realization models that describe the realization for the desired, combined functionality. In case these realization

models affect the goals of the concern, the concern designer must also specify their impact in the impact model.

5.4.2 Generating the User-Tailored Realization

During the concern reuse process, the concern user selects the desired features of the concern that is being reused from the variation interface (see step 1 in the concern reuse process discussed in Chapter 4). In CORE, this selection of features is called a *configuration*. Before the concern user can proceed to step 2 and customize the concern, a CORE tool has to create the user-tailored realization of the concern according to the desired configuration by composing all relevant realization models and dealing with resolvable feature interactions, if any.

This composition involves several algorithms that are detailed in the following subsections. First, the set of realization models that need to be composed has to be determined by taking into account possible feature interactions, if any, as described in Algorithm 3. Then, all selected realization models belonging to the same corified modelling notation have to be composed in a certain order. That order is determined using the algorithm presented in Algorithm 4. Finally, the realization models are composed in pairs by invoking the composition algorithm defined for each modelling notation.

Realization Model Selection Algorithm

In CORE, realization models are associated with the features whose functionality they realize. Most realization models realize a single feature, except for the ones that address a resolvable feature interaction. Those models are associated with the set of features involved in the interaction. As a result, features that have no interactions with other features are associated with a single realization model. Features that have resolvable interactions with

Algorithm 3 The Realization Model Selection Algorithm that Determines All Realization Models that Need to be Composed for a Given Configuration.

1. Initialize the set of features that need to be realized to contain all selected features, and initialize the set of chosen realization models for language *lang*: $ToRealize = Conf$, $ChosenRM_{lang} = \emptyset$
2. Repeat until *ToRealize* is empty:
   (a) Determine the set of realization models expressed using *lang* and involved in realizing only the features that still need to be realized: $Possible = \{rm_{lang} \mid \exists f\,(f \in ToRealize \wedge Realizes(f, rm_{lang})\}$. If *Possible* is empty, then none of the features in *ToRealize* has associated realization models expressed in *lang* and the algorithm terminates.
   (b) Add all the *rm*s in *Possible* that realize *the highest number of features* to $ChosenRM_{lang}$, and eliminate from *ToRealize* the features that these *rm*s realize.

other features have multiple realization models, one for each combination of features that requires resolution. Features that have no associated realization model are usually features that do not provide functionality and were introduced by the concern designer simply to group or classify related features in the variation interface to simplify feature selection during reuse.

Algorithm 3 determines for each modelling language *lang* all the realization models of *lang* that need to be composed to realize a given configuration of a concern. The algorithm resolves any conflicts arising from feature interactions. In the description of the algorithm, a feature is denoted f, a realization model rm, the input configuration containing the set of selected features $Conf = \{f_i\}$, and the realization relationship defined by the concern designer $Realizes = \{f_i, rm_i\}$.

The algorithm ensures that the realization models realizing n features take precedence over the ones realizing m features, where $n > m$. This is necessary in the rare cases where there are feature interactions involving more than 2 features. For example, for a concern

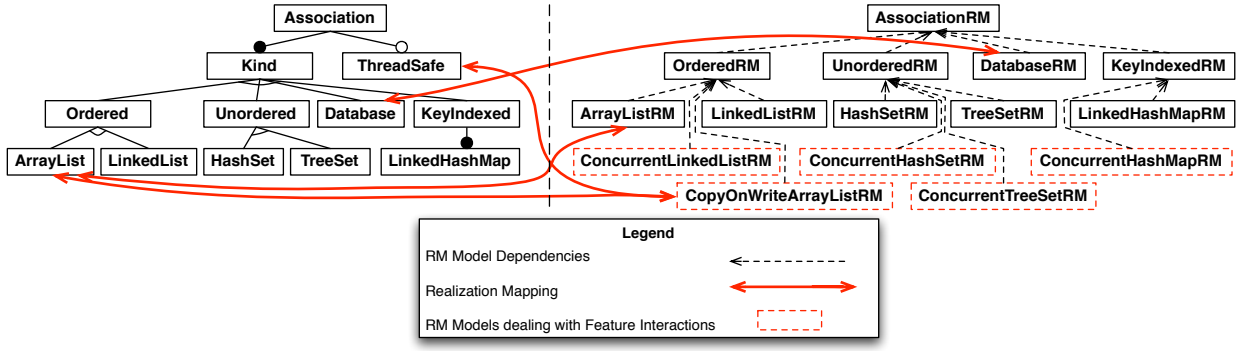

Figure 5–5: *Association* Concern Feature Model and Associated Realization Models including Feature Interaction Resolution Models

with 3 features A, B, and C, it is possible that selecting {A,B}, {B,C}, and even {A,B,C} requires different resolution models. If {A,B,C} is selected, a CORE tool should simply use rm_{abc}. In the case where {A,B} is selected, it should use rm_{ab}. If {A,C} is selected, it should however compose rm_a and rm_c. With Algorithm 3, this can be achieved by specifying that A is realized by rm_a and rm_{ab} and rm_{abc}, B is realized by rm_b and rm_{ab}, and C is realized by rm_c.

Fig. 5–5 shows the features of the *Association* concern on the left, and the realization models (ending in RM) and their dependencies on the right. The realization models in red with dashed outlines are the ones that address feature interactions involving the *ThreadSafe* feature. Our algorithm, if only *ArrayList* is selected, would choose the realization model *ArrayListRM*. However, if *ThreadSafe* is selected as well, it would use *CopyOnWriteArrayList* instead, since it realizes both *ThreadSafe* and *ArrayList*. On the other hand, if *Database* and *ThreadSafe* are selected, only *DatabaseRM* is chosen. This makes sense, since the concern designer knows that databases are inherently thread safe, and hence no feature interaction occurs.

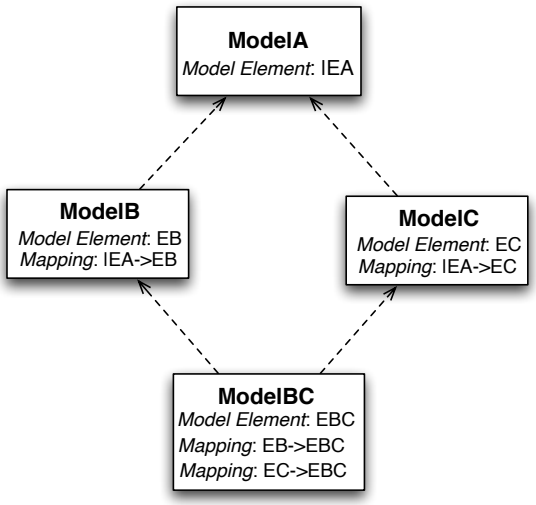

Figure 5–6: A Realization Model *ConfBC* Extends Two Models B and C, Both Extend a Common Model A.

Composition Scheduling Algorithm for Realization Models

After applying the realization model selection algorithm (Algorithm 3), the CORE-based modelling tool then generates user-tailored realization models of the concern. To this aim, for each corified modelling language *lang*, the tool creates an empty model $UserTailored-ConcernRealizationModel_{lang}$ that extends all the selected realization models specified using *lang*. The resulting model hierarchy can be represented as a graph, where models are nodes, and extension links are directed edges from the extension model to the model that is extended. In the graph representing the extension hierarchy of a user-tailored concern realization models there is *one source node* (i.e., the aforementioned empty model created by the tool) and *one or several sink nodes* (i.e., the base realization models of the concern that do not extend other models). In order to flatten the hierarchy, i.e., to create the a woven user-tailored *lang* realization model, the tool successively invokes the model composition

algorithm for *lang*, which takes two *lang* models as input and produces a *lang* model as an output, on the *UserTailoredConcernRealizationModel* and one of the models that it extends. For efficiency reasons, in particular to avoid duplicate weaving of models that have multiple incoming extension links, the order of composition is determined using Algorithm 4.

For illustration purpose, let us consider the realization models written in RAM for the *Association* concern. The right-hand side of Fig. 5–5 shows the extends hierarchy for the realization models of the *Association* concern. In the context of RAM, these models are either *base realization models* or *extension realization model*s. Models are *base realization models* if they are self-contained, i.e. they can fulfill their purpose as is, whereas so-called *extension realization models* have to be applied to some *base realization model* to fulfill their purpose. The *AssociationRM* model on the right hand-side of Fig. 5–5 is a *base realization model*, whereas other models such as *OrderedRM* are considered *extension realization model*s. We discuss these two types of realization models in detail in Chapter 8. The composition algorithm ranks a model in the extension hierarchy based on the longest line of ancestor models. For example, the rank of *HashSetRM* in Fig. 5–5 is 2, since it has two ancestor models (*UnorderedRM* and *AssociationRM*).

Fig. 5–6 shows the directed graph representing the extension hierarchy for the models of an example concern that has three features: feature *A* which is realized by *ModelA*, feature *B* which is realized by *ModelB* and feature *C* which is realized by *ModelC*. Both *ModelB* and *ModelC* are considered *extension realization models* that extend a *base realization model* *ModelA*. *ModelA* has a partial model element */EA*, *ModelB* has a model element *EB*, and *ModelC* has a model element *EC*. Composition specification mappings are also shown in Fig. 5–6: */EA→EB* in *ModelB* and */EA→EC* in *ModelC*. Because of a feature interaction

Algorithm 4 Composition Scheduling Algorithm for Generating the User-Tailored Realization Models

1. Initialize $WovenUserTailoredConcernRealizationModel_{lang}$ to $UserTailoredConcernRealizationModel_{lang}$. Initialize $ToBeComposed_{lang} = ChosenRM_{lang}$ (as obtained by running the algorithm defined in Subsection 5.4.2)
2. Rank each model m in $ToBeComposed_{lang}$ based on the longest path of ancestor models it extends. More formally:
   Let B represent the set of base models in an extends dependency graph, i.e., the models b_i that do not extend any other models. For these models, $rank(b_i)= 0$.
   Let M represent the set of extension models, i.e., the models m_i that extend base models or other extension models.
   In this case, $rank(m_i) = \max_{b_i \in B}(pathlength(m_i, b_i))$.
3. Repeat until $ToBeComposed_{lang} = \phi$:
   (a) Pick m such that $rank(m) = \max_{m_i \in ToBeComposed}(rank(m_i))$ (in case there are several models with the maximum rank, pick any of these models).
   Compose $WovenUserTailoredConcernRealizationModel_{lang}$ with m by calling the composition algorithm provided by lang, passing as an input $WovenUserTailoredConcernRealizationModel_{lang}$, m and the composition specifications from models elements of $WovenUserTailoredConcernRealizationModel_{lang}$ to model elements of m. Store the resulting model back into $WovenUserTailoredConcernRealizationModel_{lang}$.
   (b) For all models n_i such that m extends n_i:
   if $WovenUserTailoredConcernRealizationModel_{lang}$ already extends n_i, merge the corresponding composition specifications.
   else calculate $rank(n_i)$, and add n_i to $ToBeComposed_{lang}$.
   (c) Remove m from $ToBeComposed_{lang}$.

between *B* and *C*, both features are additionally realized by *ModelBC,* which extends both *ModelB* and *ModelC. ModelBC* has a model element *EBC,* and specifies two mappings: *EB→EBC* and *EC→EBC*. When we feed Algorithm 3 with a configuration of selected features *B* and *C*, the output *ChosenRM = {ModelBC},* because it realizes both features. The CORE-based modelling tool creates an empty model *UserTailoredConcernRealizationModel* that extends the set of models in *ChosenRM*, in this case only *ModelBC.* The output set of models are then ranked by Algorithm 4, presented above, and the model with the highest rank is chosen to be composed with *WovenUserTailoredConcernRealizationModel,* which is initialized to *UserTailoredConcernRealizationModel.* The rank of *ModelBC* is 2 because it has two ancestor models through two paths (*ModelB* and *ModelA* through one path, and *ModelC* and *ModelA* through another path). Algorithm 4 then asks the RAM composition algorithm to compose *WovenUserTailoredConcernRealizationModel* with *ModelBC*. As a result of the RAM composition, *WovenUserTailoredConcernRealizationModel* now extends both *ModelB* and *ModelC*, and contains the model elements and the mappings originally contained in *ModelBC*. Then, Algorithm 4 calculates the ranks of the new extended models of *WovenUserTailoredConcernRealizationModel*, in this case, *ModelB* and *ModelC,* which both have rank 1. One of these extended models is chosen next to be composed with *WovenUserTailoredConcernRealizationModel.* Suppose that *ModelB* is chosen, and composed using the RAM composition algorithm. The composition specification mapping */EA→EB* originally in *ModelB* is updated to */EA→ECB* by the RAM composition algorithm. Now, *WovenUserTailoredConcernRealizationModel* extends *ModelA* and *ModelC,* and Algorithm 4 ranks *ModelA* with 0 and *ModelC* with 1. Therefore, *ModelC* is chosen next to be composed

with *WovenUserTailoredConcernRealizationModel,* and the composition specification mapping */EA→EC* that was updated to */EA→ECB* by the RAM composition algorithm is now merged with the previous mapping */EA→ECB*. Finally, *ModelA* is chosen to be composed with *WovenUserTailoredConcernRealizationModel.*

5.5 Customization Interface Composition

To satisfy C4, the rule is that the customization interface of the reusing concern is a union of the new customization elements introduced by the reusing concern and the customization elements of the reused concerns that have not been customized, i.e., that were not mapped to specific elements in the reusing concern. This makes it possible to grow or shrink the customization interface within concern hierarchies, depending on the intent of the developer. A "more specific" concern, for instance, would abstain from introducing new customization elements, and map some of the lower-level customization elements to specific elements of the "more specific" concern. The same rule also ensures that customization elements that are part of reexposed features of the reused concern are incorporated into the customization interface of the reusing concern.

5.6 Usage Interface Composition

In order to satisfy C5, information hiding principles are applied by default. In other words, the accessible structure and behaviour exposed in the usage interface of the reused concern are not included in the usage interface of the reusing concern, unless explicitly indicated by the developer. In these situations, which occur when the reused concern provides functionality that the reusing concern wants to offer, it often makes sense to rename the reexposed functionality to reflect the change in level of abstraction. For example, a concern encapsulating many variants of accessing doors in a building may internally reuse

the *Password* feature of the *Authentication* concern described earlier in Chapter 4. In this case, the `getPassword` functionality of the *Authentication* concern might be reexposed as `getAccessCode` in the usage interface of the concern.

5.7 Conclusion

In this chapter, we discussed the composition of the variation interface, i.e., feature models and impact models, and the composition of realization models.

Before generating the composed feature model, the developer must decide which reusable features should be selected and which reusable features should be reexposed in the variation interface of the reusing concern. As for the composed impact model, the developer must decide on the relative impact of the reused concern on the reusing concern for a number of qualities – one quality at a time – by specifying relative contribution weights. All goals of the reusing concern in the impact model are exposed in the variation interface of the reusing concern, while all goals of reused concerns are hidden.

In addition, we introduced two algorithms that help in generating the user-tailored realization models. The realization model selection algorithm determines which realization models of a concern need to be composed given a desired configuration, taking into account feature interactions, if any. The composition scheduling algorithm determines the most efficient ordering for invoking the pairwise composition algorithms defined by the involved modelling languages.

Chapter 6
CORE Metamodel

The growing number of modelling tools that are used in developing ever-evolving diversified systems require advanced techniques for supporting software reuse. The ideas of concern-orientation can be adopted when creating new modelling languages, but also integrated into existing modelling languages. To this aim, the concepts presented in this thesis are captured in a metamodel that simplifies the corification (i.e., support for CORE) of different modelling languages. A summary of the key concepts in the metamodel follows:

- *Concern*: A concern is a named element that groups related models together. A model itself groups a set of model elements that belong to its modelling language. By default, a concern consists of at least a feature model, which along with an optional impact model belongs to the variation interface of the concern. In addition, a concern provides two other interfaces to facilitate reuse: the customization interface and the usage interface.
- *Feature*: The feature model groups a set of features and defines relationships between them.
- *Impact*: The impact model shows the contributions of features and goals to other goals.
- *Reuse*: A concern can reuse other concerns by following the reuse steps described earlier in Chapter 4. In addition to selecting the features of the reused concern, the reusing concern may choose to reexpose in its interface some features of the reused concern. This allows the concern designer to delay the decision of selecting features of

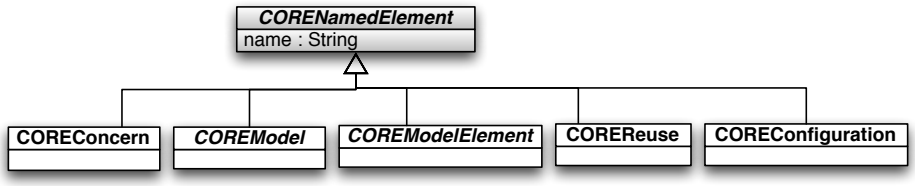

Figure 6–1: Named Elements of the CORE Metamodel

the reused concern that provide alternative or optional functionality to a later point in time, which is particularly helpful when trade-offs and high-level goals are not yet fully known.

- *Composition*: Last but not least, the reusing concern creates compositions between its model elements and the model elements of the reused concern. Three different techniques for composing model elements are covered by the metamodel.

A modelling language that wants to be corified (i.e., wants to add support for CORE) extends 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. We have successfully used the CORE metamodel to corify two modelling languages: AoURN and RAM, which will be discussed in the next chapter. This chapter includes seven sections. In the next section we discuss the named elements of CORE. Then, we discuss each key concept of CORE in a separate section. The concern part of the metamodel is discussed in Section 6.2, the feature part in Section 6.3, the impact part in Section 6.4, the reuse part in Section 6.5, and the composition part in Section 6.6. Finally, we conclude this chapter in Section 6.7.

6.1 General CORE Metamodel

As mentioned above, the CORE metamodel is divided into five different parts that describe the *Concern*, *Feature*, *Impact*, *Reuse*, and *Composition* concepts. Each concrete class in the metamodel represents a concept that must be added to the corified modelling language. Abstract classes represent concepts that may exist in the modelling language, but have associations to the concrete CORE classes. The *CORENamedElement* abstract class provides naming facilities to *COREConcern*, *COREModel*, *COREReuseConfiguration*, *COREModelElement* and *COREReuse* as shown in Fig. 6–1. The grey elements in the figures of this chapter highlight the concepts discussed in detail by the section for each of the five parts of the CORE metamodel, while the white elements are discussed in more detail by a different section. The complete metamodel is shown in Fig. 10–1 in Appendix I.

6.2 Concern

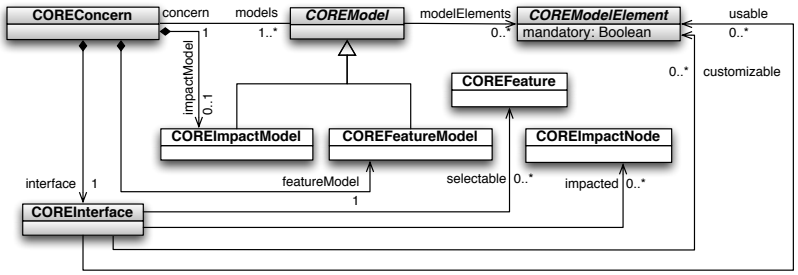

Figure 6–2: The Concern Part of the CORE Metamodel

A concern (*COREConcern*) groups related models (*COREModel*) together (see Fig. 6–2). The abstract class *COREModel* must be subclassed by a corified modelling language to represent its model types. A concern provides at least one model by default – a *COREFeatureModel* (subclass of *COREModel*), representing the feature model. The feature model and the optional impact model (*COREImpactModel*) form the variation interface of the concern. A

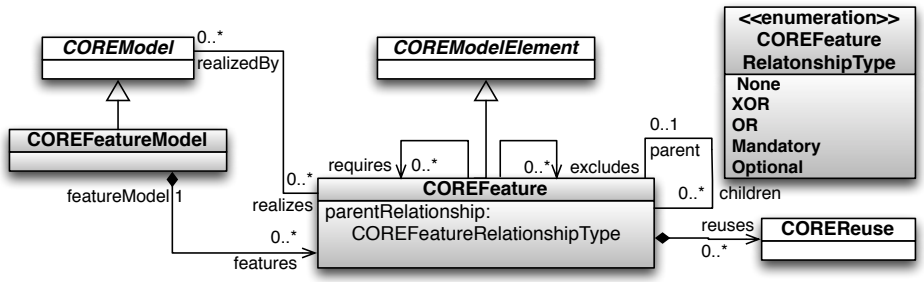

Figure 6–3: The Feature Part of the CORE Metamodel

COREModel groups related model elements (*COREModelElement*), which is also an abstract class that needs to be subclassed by the corified modelling language to represent its model elements. The interface (*COREInterface*) is a main characteristic of a concern. The *variation interface* references *COREFeature*s (*selectable* role) and *COREImpactNode*s (*impacted* role). The *usage interface* and *customization interface* are represented by the *usable* and *customizable* roles, respectively. Any *COREModelElement* that has to be customized when reused (i.e., it designates a partial model element) is indicated by its *mandatory* attribute being set to true.

6.3 Feature

A feature (*COREFeature*) is contained in *COREFeatureModel* as shown in Fig. 6–3. *COREFeature* has an attribute (*parentRelationship*) which is an enumeration type that specifies a feature's relationship with its parent, i.e., whether the feature is part of an *XOR* or *OR* group, whether it is *Mandatory* or *Optional*, or whether it is the root (*None*). A feature selection may *require* or *exclude* the selection of other features (*requires* and *excludes* roles). Each feature has at most one parent and may have many children (*parent* and *children* roles). A feature may be realized (*realizedBy* role) by many *COREModels*, and similarly, a *COREModel* may realize (*realizes* role) many features. This association is used to link features to

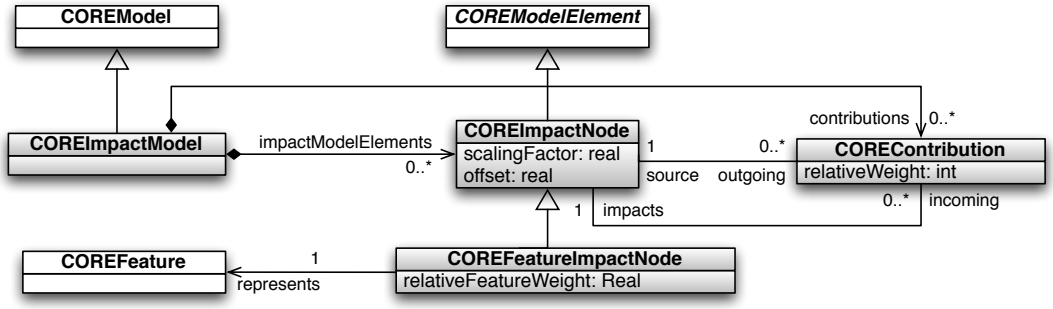

Figure 6–4: The Impacts Part of the CORE Metamodel

the models of the corified modelling language. Finally, the *reuses* composition indicates that a *COREFeature* may reuse (*COREReuse*) many other concerns.

6.4 Impact

A *COREImpactModel* contains both *COREImpactNode*s and *COREContribution*s (see Fig. 6–4). A goal is represented by a *COREImpactNode* (a subclass of *COREModelElement*). A *COREFeature*, on the other hand, is represented by a *COREFeatureImpactNode* (a subclass of *COREImpactNode*) in the impact model. Each *COREContribution* has an integer attribute to store its contribution value, which is a relative weight, describing how one *COREImpactNode* (*source* role) impacts another *COREImpactNode* (*impacted* role).

6.5 Reuse

The class *COREReuse* is the centre of this concept and has a reference to one reused *COREConcern* (*reusedConcern* role) (see Fig. 6–5). Whenever a feature reuses a concern, an instance of *COREReuse* is created and associated with it using the *reuses* association. *COREReuse* contains a set of configurations (*COREReuseConfiguration*) of which at most one can be selected. Each *COREReuseConfiguration* has a set of selected and reexposed features of the reused concern (*selected* and *reexposed* roles, respectively). When a feature reuses a

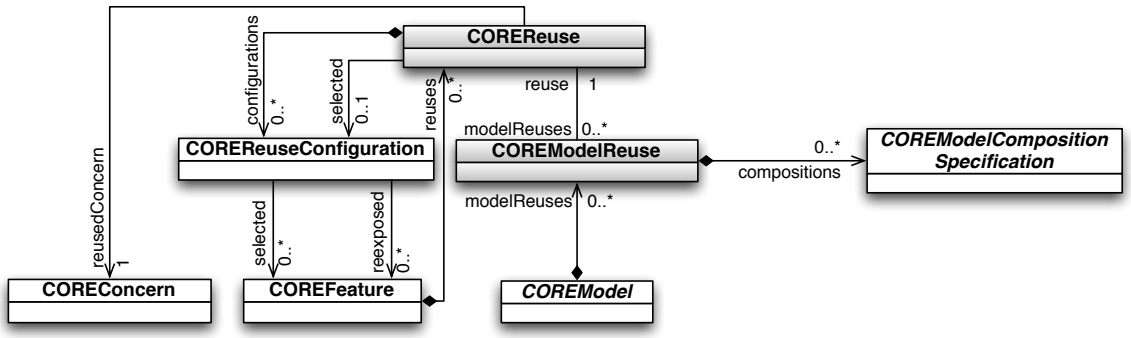

Figure 6–5: The Reuse Part of the CORE Metamodel

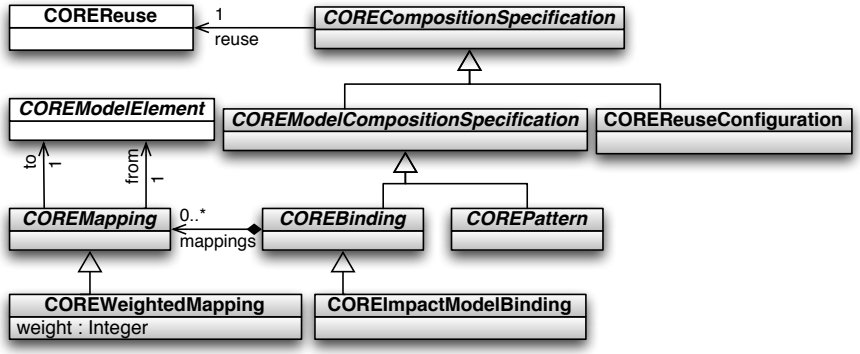

Figure 6–6: The Composition Part of the CORE Metamodel

concern, a model of the corified modelling language (*COREModel*) that realizes the feature can reuse models of a reused concern through *COREModelReuse*, which has an association to the *COREReuse* it belongs to. *COREModelReuse* contains *COREModelCompositionSpecification*s to allow specifying customization compositions of the reused models with the reusing model.

6.6 Composition

Two abstract classes, *CORECompositionSpecification* and *COREModelCompositionSpecification*, define different kinds of composition specifications (see Fig. 6–6); the latter is a subclass of the former. *CORECompositionSpecification* captures the fact that all compositions are related to one specific *COREReuse* (*reuse* role). *COREReuseConfiguration* (a concrete subclass of *CORECompositionSpecification*) is dedicated to the composition of feature models (see Fig. 6–5). *COREModelCompositionSpecification* has two abstract subclasses (*COREBinding* and *COREPattern*), representing two general ways for specifying compositions for modelling languages other than feature models [81, 17]. *COREPattern* is used when the composition is specified using pattern matching, while *COREBinding* is used when a set of *COREModelElement* pairs are composed. Therefore, *COREBinding* contains a set of *COREMapping* (*mappings* role), another abstract class to specify the mapping from (*from* role) and to (*to* role) *COREModelElement*s (the two inputs to the composition). The *from* element always refers to a model element from the reused concern while the *to* element refers to a model element from the reusing concern. A specialized *COREBinding* called *COREImpactModelBinding* establishes the binding between impact model elements with the help of a specialized *COREMapping* called *COREWeightedMapping* to establish mappings that contain weights. To calculate the total impact of a feature F that reuses one or more lower-level concerns on a goal of the reusing concern as described at the end of Section 4.3, the following three inputs are combined with each other: (i) the *weight* of all *COREWeightedMapping*s where the *to* role is the *COREFeatureImpactNode* representing feature F, (ii) the weight of the *COREFeatureImpactNode* (see *relativeFeatureWeight* in Fig. 6–4) representing F, and (iii) the *relativeWeight* of the *COREContribution* that has the *COREFeatureImpactNode* representing F as its *source*.

6.7 Conclusion

This chapter presented the CORE concepts discussed in this thesis in a common metamodel. The metamodel is divided into five parts. The first part presents concern which is a reuse unit that groups related models together. The second part presented the feature model which groups a set of features and defines relationships between them. The third part presented the impact model which shows the contributions of features and goals to other goals. The fourth part specified how a concern can reuse other concerns by following the reuse steps. Finally, the fifth part specified how the reusing concern creates compositions between its model elements and the model elements of the reused concern. In the next chapter, we show how we corified two modelling languages, AoURN and RAM by extending the CORE metamodel in their respective tools.

Chapter 7
Tool Support

For validation purposes, the metamodel discussed in the previous chapter was extended by two modelling languages: the Aspect-Oriented User Requirements Notation (AoURN) and Reusable Aspect Models (RAM), which are at opposite ends of the spectrum of modelling languages with respect to the means of corification required. AoURN, a modelling notation specialized in requirement modelling, already supports feature modelling [74], goal and workflow modelling [56], with aspect-oriented extensions [83], but does not offer any dedicated support for reuse. On the other hand, RAM, which is an aspect-oriented multi-view modelling language for software design, does not support feature and goal modelling, but supports aspect-oriented class, sequence, and state modelling with some support for reuse. Both corified languages are supported by modelling tools that allow for creating, visualizing, saving, and updating models. AoURN is supported by an eclipse-based requirement modelling tool called *jUCMNav* [90, 74] and RAM is supported by a stand-alone, multitouch-enabled tool called *TouchCORE* [109]. Initially, both tools did not support CORE. We extended the tools to support the corified versions of their modelling languages, therefore, they provide CORE properties such as reuse, feature modelling and impact modelling.

As mentioned in the beginning of this thesis, our contribution is implementing and validating CORE in the design phase of software development. Therefore, we discuss the corification of RAM and some key features of the TouchCORE tool in more details as one

of contributions of this thesis. Additionally, we discuss the corification of AoURN to further demonstrate the effectiveness of our approach.

7.1 Corification Strategies of Existing Modelling Languages

Abstract and concrete classes of the CORE metamodel shown in Chapter 6 are extended differently when corifying a modelling language. The abstract classes *CORENamedElement*, *COREModel*, *COREModelElement*, *COREPattern*, *COREBinding*, and *COREMapping* serve as extension points and are intended to be subclassed by a modelling language. This allows for consistent naming, the addition of arbitrary modelling languages to CORE, and uniform treatment of a canonical set of compositions. The remaining abstract classes *COREComposition-Specification* and *COREModelCompositionSpecification* are typically not extended. An actual need to extend them indicates a new form of composition, which should after thorough consideration be added to the CORE metamodel as a new abstract class, so that it is available in the canonical set of compositions to all corified modelling languages through subsequent subclassing.

Concrete classes, on the other hand, are intended to be used as is in the corified modelling language, which ideally directly implements and visualizes these concepts in its modelling tool. However, if a CORE concept already exists in the modelling language, then a directed association from the class representing the CORE concept in the metamodel of the modelling language to the concrete class in the CORE metamodel needs to be established, and the instances of these two classes must be kept in sync. The association approach is necessary, because all concrete CORE metaclasses (with the exception of *COREModelReuse*) are always contained either directly or indirectly in the root of the CORE metamodel, i.e., the *COREConcern*. Similarly, the concepts of a modelling language naturally exist in the

containment hierarchy of the modelling language. Consequently, subclassing (i.e., introducing a generalization relationship from the metaclass in the modelling language to the concrete CORE metaclass) is not an option, because a subclass cannot be contained in two containment hierarchies at the same time.

Therefore, four distinct means of corification exist:

(A) subclassing (introduce a generalization relationship from an existing metaclass in the modelling language to an abstract CORE metaclass),

(B) add and subclass (first add a new metaclass to the modelling language and then introduce a generalization relationship from the new metaclass to an abstract CORE metaclass),

(C) associate (introduce a directed association from an existing metaclass in the modelling notation to a concrete CORE metaclass), and

(D) use a concrete CORE metaclass as is.

Which of the four means is used for a specific modelling language depends on the type of modelling language. The CORE metaclasses cover concepts from feature modelling, goal modelling, aspect-oriented modelling, and reuse. If the modelling language that is to be corified offers feature or goal modelling, then association is most likely the best choice for the classes related to feature and goal modelling. If the modelling language is aspect-oriented, then it is likely that advanced composition techniques are already supported and subclassing the CORE metaclasses related to composition is most likely the best choice. If the modelling language does not support aspect-orientation, then adding a class for the desired type of composition to the modelling notation and subclassing it from the relevant CORE metaclass related to composition is most likely the best option.

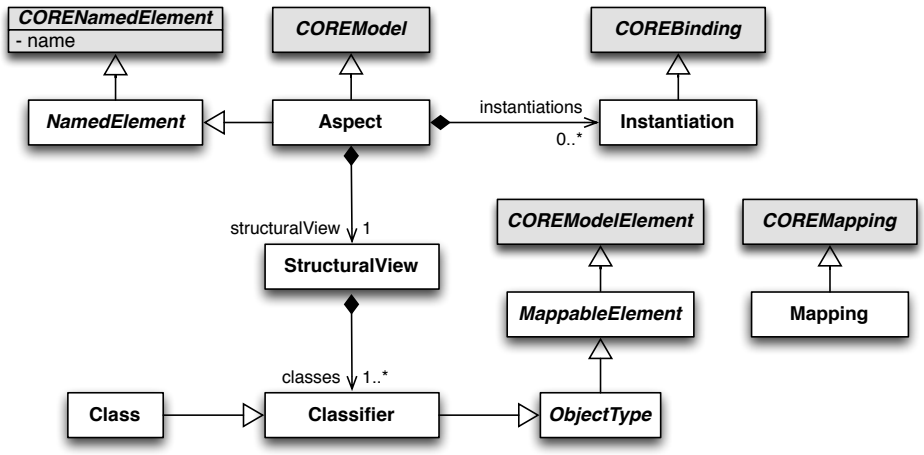

Figure 7–1: Corification of the RAM Metamodel by Subclassing the CORE Metamodel.

7.2 Corification of RAM

Because RAM has no support for feature or goal modelling and does not cover the concepts of concern, all concrete CORE metaclasses can be used as is (corification means D).

Fig. 7–1 shows a partial view of the RAM metamodel, focusing on the parts that extend the CORE metamodel (i.e., corification means A). The grey elements in the figures of Section 7.1 are the abstract classes that come from the CORE metamodel. RAM's *NamedElement* subclasses *CORENamedElement* which allows providing names to different model elements. Initially, *NamedElement* provided an attribute name which we removed to avoid clashing with the name attribute provided by *CORENamedElement*. Aspects in RAM are named elements (*Aspect* subclasses *NamedElement*) and also subclasses *COREModel*. Since RAM already provides advanced support for composition through *Instantiation* and *Mapping*, corification means A is again the best choice. Aspect groups a set of instantiations

that allows reusing other aspects, and the *Instantiation* class subclasses *COREBinding*. Because of that, *Instantiation* contains a set of mappings (*Mapping* class), which is a subclass of *COREMapping,* and the latter has references to *COREModelElement* subclassed by *MappableElement* (Fig. 6–6 showed two references from *COREMapping* to *COREModelElement* one reference ends with a role name *from* to record the source of the mapping, and the other ends with *to* to record the destination of the mapping). An Aspect contains several views, including the structural view that is shown in the figure. Among the model elements that the structural view contains is *Classifier*. The class *Class* is a subclass of *Classifier*, and can also be mapped to other classes (because *Classifier* is a subclass of *ObjectType,* which is a subclass of *MappableElement*. *StructuralView* contains other model elements such as operations and attributes that are not shown in Fig. 7–1.

By subclassing the abstract classes of the CORE metamodel, RAM successfully provides all the properties of CORE. A RAM model may now belong to a concern by realizing at least one of its features, which may have impacts on high-level goals. A RAM model may now also reuse another RAM model that belongs to a different concern, by asking the feature it realizes to reuse the other concern with the selection of feature(s) it wants to reuse. The reusing RAM model then establishes the mappings to the reused RAM model that realizes the reused feature(s). This is achieved as follows. *Aspect* subclasses *COREModel*, which makes it part of a *COREConcern* and allows an *Aspect* to realize a feature (see Fig. 6–3, a *COREModel* realizes *COREFeature*). Therefore, the feature realized by the aspect can create a *COREReuse* to reuse another concern, and since *COREReuse* contains *COREReuseConfiguration*, it can create a configuration and select the feature from the reused concern (by setting

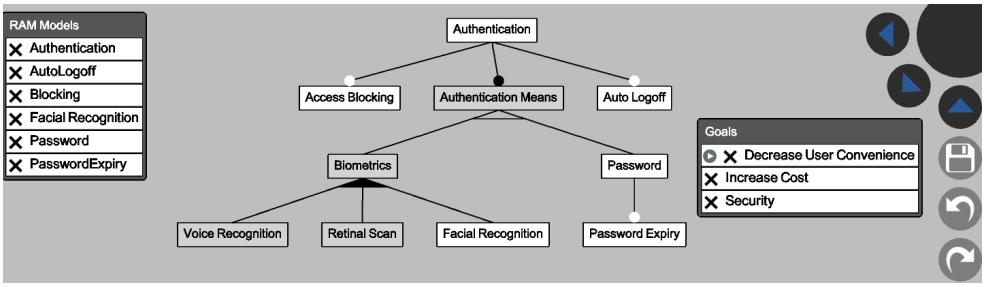

Figure 7–2: Feature Model Design Mode in TouchCORE.

the *selected* role). The reusing aspect also creates a *COREModelReuse* which has an association to the *COREReuse*, and an association to *COREModelCompositionSpecification*. Now, the reusing aspect creates *Instantiation,* which is a subclass of *COREBinding* (*COREBinding* is a subclass of *COREModelCompositionSpecification*) as shown in Fig. 6–6. Mappings to the model elements of the reused aspect are now established, because the class *Mapping* is a subclass of *COREMapping* as shown in Fig. 7–1, consequently allowing a RAM aspect in a concern to successfully reuse another aspect belonging to a different concern.

7.2.1 TouchCORE

The corification of RAM described above was implemented by updating the RAM tool, *TouchRAM* [13], to support CORE. *TouchRAM*, a multitouch-enabled, aspect-oriented tool for incremental software design modelling, underwent many updates over the years to add support for new features and to incorporate metamodel changes [109, 107, 120, 106]. We recently changed the tool's name to *TouchCORE* [109] to reflect its additional support for concern-orientation.

TouchCORE User Interface

The extensions that had to be made to the user interface of *TouchCORE* to support concern-orientation are outside the focus of this thesis, since they were implemented with

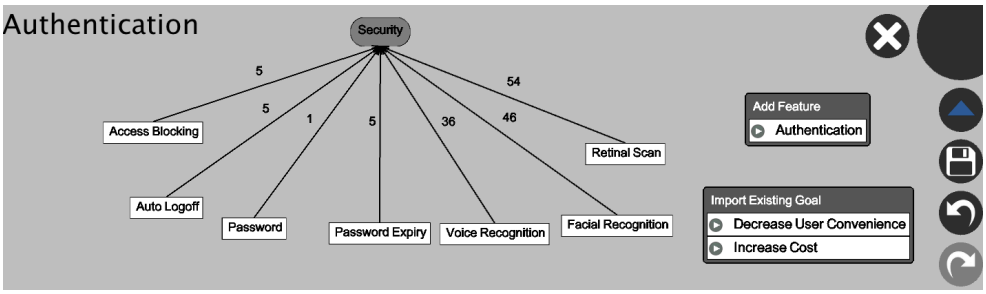

Figure 7–3: Impact Model Design Mode in TouchCORE.

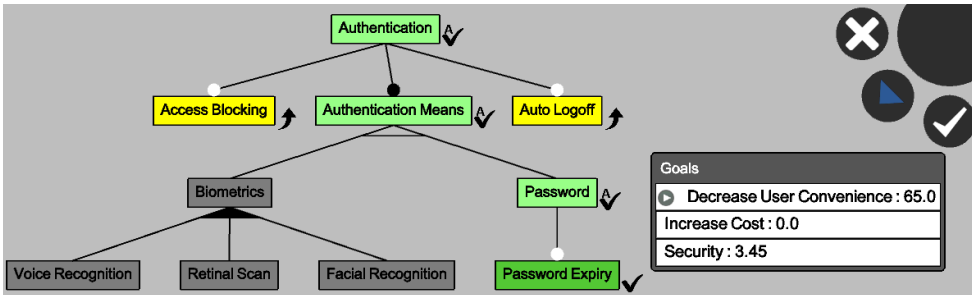

Figure 7–4: Step 1 of the Reuse Process: Feature Selection Mode in TouchCORE.

the help of master students and internship students over the last two years. Nevertheless, we provide in this subsection an overview of some key features of the tool to give the reader a feel of how a modeller experiences concern-orientation during the modelling activity. The interested reader should consult the related publications [109, 107, 120, 106, 13] for more details on the technical aspects of the tool.

TouchCORE is specialized in modelling design concerns. Particularly, modellers can design, edit and save RAM realization models, feature and impact models, and assign realization models to features. Each RAM realization model, as shown throughout this thesis, includes a structural view consisting of a class diagram, and message views consisting of sequence diagrams. For feature and impact models, TouchCORE provides two modes of visualization to the modeller [121]: a design mode and a reuse mode. In the design mode,

the concern designer builds the feature and impact models, assigns realization models to features in the feature model, and specifies impacts of features on high-level goals in the impact model. The reuse mode is the one that the concern user sees during step 1 of the concern reuse process shown in Chapter 4. It presents the modeller that is reusing a concern with the concern's variation interface, and allows her to select features from feature model, while continuously evaluating the underlying impact model based on the current selection and displaying the results to allow the modeller to perform trade-off analysis.

Fig. 7–2 illustrates the design mode for feature models, where the feature model for the *Authentication* concern is shown in the middle, the list of RAM realization models are shown on the left, and the list of high level goals are shown on the right. A realization model is assigned by selecting it from the list and clicking on the feature(s) that it realizes. The design mode for impact models is illustrated in Fig. 7–3, where the impacts of the *Authentication* features on the *Security* goal are shown. The modeller can add features and subgoals, and edit feature contributions to goals in this mode. The RAM realization models are visualized in TouchCORE similarly to how the *Authentication* realization models are shown in Chapter 4. The tool provides sophisticated editing facilities for RAM realization models, which are not presented here since they are out of the scope of this thesis.

The reuse mode is illustrated in Fig. 7–4. The green coloured features are the ones that are selected, and the high level goals on the right show the impact evaluations based on the current selection accordingly.

7.2.2 Implementation of CORE Composition Rules and Algorithms

In addition to extending the CORE metamodel, this thesis contributes to TouchCORE by implementing the CORE composition algorithms discussed in Chapter 5 that are related

to generating the woven RAM model. The realization model algorithms are implemented to generate the user-tailored RAM realization models. The realization model selection algorithm is implemented to determine which RAM realization models of a concern need to be composed given a desired configuration, taking into account any feature interactions. The composition scheduling algorithm is implemented to determine the most efficient ordering for invoking the pairwise RAM composition algorithm. The implementation of these algorithms were straightforward, we replaced *lang* in Algorithm 3 and Algorithm 4 with the RAM composition algorithm (i.e., the RAM weaver [13]).

Beside supporting the essential MDE features such as model hierarchies, interfaces and abstraction, TouchCORE supports some additional features, we highlight two important ones here:

Library of Reusable Concerns

TouchCORE comes with a growing library of reusable design concerns, ranging from high-level concerns such as *Authentication* to low level concerns providing detailed design solutions such as *Networking*. Many of the concerns in the library are discussed in more detail in Chapter 8.

Traceability

As discussed previously in Chapter 4, the CORE reuse process allows for building applications/concerns by reusing other existing concerns, often producing complex *concern hierarchies*. The involved concerns in the hierarchy apply the principles of information hiding as discussed in Chapter 2, and by preventing the model elements of the reused concerns from being publicly visible in the interface of the reusing concern (the visibility of public elements of the reused concern are changed from *public* to *concern* during the composition

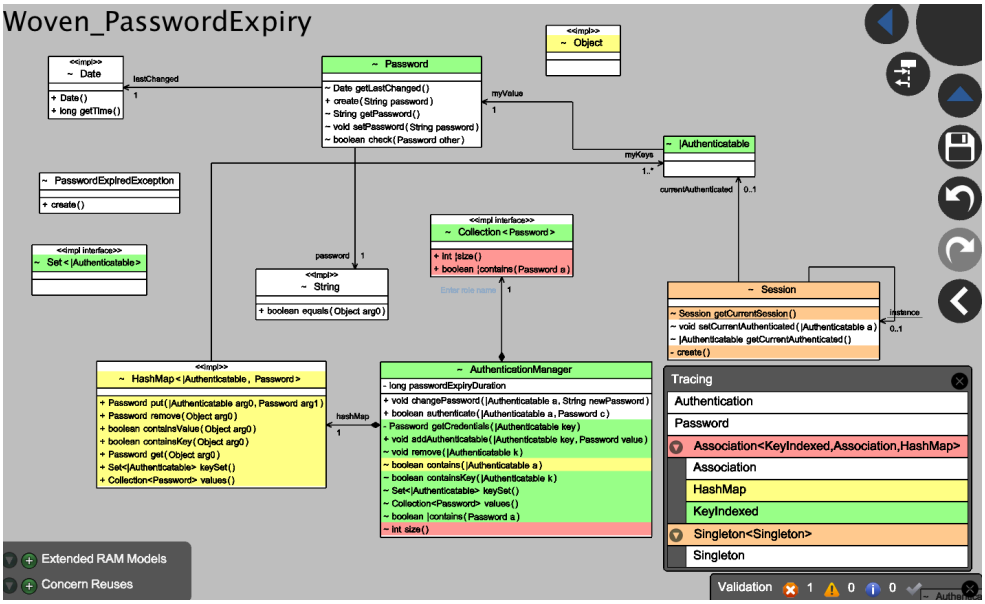

Figure 7–5: Tracing Realization Model Elements of Features of Reused Concerns in TouchCORE

as discussed in as discussed in Chapter 4). Adhering to the principles of information hiding during concern reuse produces different levels of abstractions in concern hierarchies as discussed in detail in Chapter 4 and Chapter 8. However, the modeller may want to find and trace elements of the reused concerns to better understand how the are internally used and how they interact with each other, despite the fact that they are hidden from the outside world of the reusing concern. TouchCORE keeps tracing information for all model elements of the reused concerns (and as well for elements of the extended RAM models) and allows the modeller to highlight the model elements in the reusing concern that are composed with model elements from realization models of features of the reused concerns using colours as illustrated in Fig. 7–5.

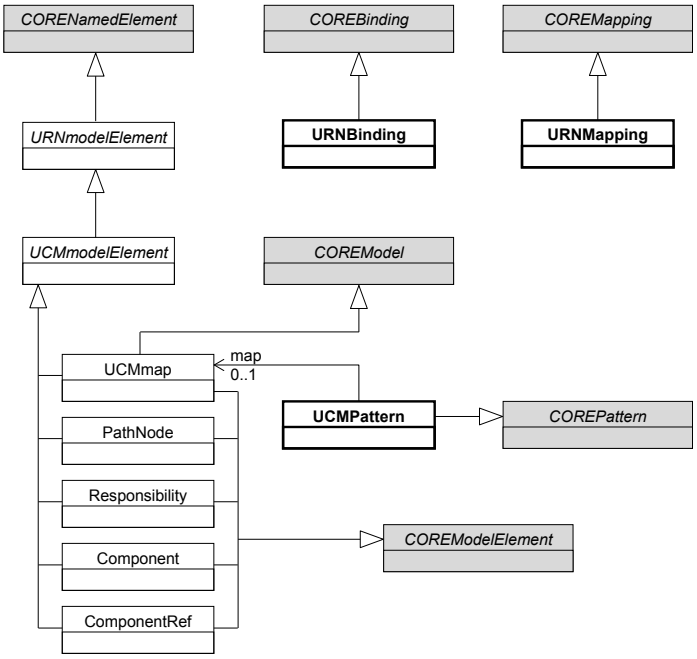

Figure 7–6: Corification of the AoURN Metamodel - Abstract Metaclasses

To summarize, this section of the thesis presented an overview of TouchCORE, a concern-oriented multitouch-enabled tool for modular and incremental modelling of design concerns. The fact that TouchRAM was successfully corified with support for concern-orientation (and hence transformed into TouchCORE) validates several of the theoretical contributions of this thesis: (a) it shows that it is possible to use the CORE metamodel to add the CORE concepts into an existing modelling language, in this case into RAM , (b) it implements the CORE reuse process discussed in Chapter 4, and therefore allows modellers to experience the simplicity of model reuse based on concern-orientation, (c) and it implements the CORE composition rules and algorithms discussed in Chapter 5, which allowed us to actually execute the algorithms and verify the correctness of the resulting woven models.

7.3 Corification of AoURN

This section outlines how another modelling language, AoURN, was corified. As mentioned before, corification of AoURN is discussed here for illustration purpose only, and is not the focus of this thesis. Similar to RAM, the corified AoURN metamodel also subclasses the abstract CORE metaclasses using corification means A (i.e., subclassing) as shown in Fig. 7–6. Therefore, *URNmodelElement* subclasses *CORENamedElement*, *UCMmap* subclasses *COREModel*, and all workflow elements (*PathNode*, *Responsibility*, *Component*, *ComponentRef*) subclass *COREModelElement*. The remaining abstract CORE metaclasses relate to composition specifications: *COREPattern*, *COREBinding*, and *COREMapping*. However, support for composition is provided only implicitly by AoURN, i.e., it is not reified as a first-class modelling element in the AoURN metamodel, but rather existing modelling elements are identified as composition specifications depending on context. A *UCMmap* may be a *COREPattern*, and *COREBinding*s and *COREMapping*s are typically captured in the names of model elements. The integration with the CORE metamodel forces these concepts to now be clearly identified in the AoURN metamodel with the help of corification means B. Therefore, the new AoURN metaclass *UCMPattern* is introduced as a subclass of *COREPattern* and a directed association from *UCMPattern* to *UCMmap* is created, i.e., the metamodel now expresses when a *UCMmap* functions as a *COREPattern*. Note that the directed association goes from *UCMPattern* to *UCMmap* to allow all composition specifications to be accessible from *COREModelReuse*. For *COREBinding* and *COREMapping* new subclasses *URNBinding* and *URNMapping*, respectively, are introduced, which now explicitly encode what was formerly expressed in the name of model elements.

To the contrary of RAM, AoURN cannot use all concrete CORE metaclasses as is (i.e., through corification means D), because the AoURN metamodel already contains concepts related to concerns, feature models, and impact models. Consequently, all already existing concepts use corification means C as shown in Fig. 7–7, while corification means D is used for the remaining concepts. A directed association is introduced from *Concern* to *COREConcern*, *FeatureModel* to *COREFeatureModel*, *Feature* to *COREFeature*, *ImpactModel* to *COREImpactModel*, *IntentionalElement* to *COREImpactNode*, *Contribution* to *COREContribution*, and *EvaluationStrategy* to *COREReuseConfiguration*. The *COREFeatureImpactNode* is treated slightly differently in that a new AoURN metaclass *FeatureImpactElement* is introduced before associating it with the *COREFeatureImpactNode*. This is done to ensure that all nodes in an impact model are modelled with native AoURN metaclasses, which simplifies the management of AoURN models. All of these model elements have to be kept in sync, i.e., when an AoURN model is loaded, changes in the CORE model elements are propagated to the AoURN model and when an AoURN model is saved, changes in the AoURN model are propagated to the CORE model elements. The remaining concrete CORE metaclasses, which are all related to concepts from the reuse and composition parts of the CORE metamodel, which are missing in AoURN, are used as is.

These modifications to the AoURN metamodel allow all concepts of CORE to be provided for AoURN, analogously to RAM as already explained earlier in the previous section.

7.4 Conclusion

This chapter outlined strategies to integrate the CORE metamodel into the metamodel of any modelling language that wants to use the concepts of CORE using subclassing and/or

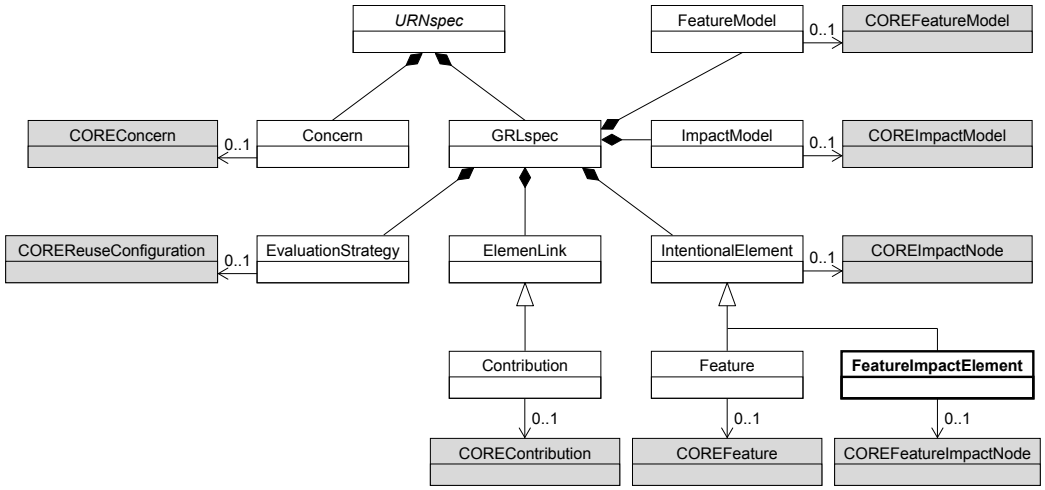

Figure 7–7: Corification of the AoURN Metamodel - Concrete Metaclasses

creating directed associations. The chapter validates the CORE reference implementation/metamodel presented in Chapter 6 and the aforementioned strategies by corifying two modelling languages: AoURN and RAM. Both languages have dedicated modelling tools, *jUCMNav* and TouchCORE, which were extended with support for concern-orientation. We provided details about the corification of the RAM modelling tool *TouchCORE*. In the next chapter, we discuss how we developed some reusable design concerns in RAM, most of which are part of the reusable concern library of TouchCORE.

Chapter 8
Reusable Concern Library

As discussed earlier, building a reusable concern can be a non-trivial and time consuming task, typically requiring the modeller to have extensive domain knowledge. Since all the knowledge needed for building a reusable concern may not be available at the time of its construction, the concern should be extensible for future development, allowing it to be continuously updated by adding new features (and their realization models) when the specifications of these features are known. However, evolving models generally tend to grow in size, to the extent that it becomes difficult to understand and maintain them, posing a scalability problem for developing evolving reusable design concerns. We address this problem by *incrementally modelling* concerns, in which a reusable concern is built by adding small *model increments* to a *base model*. We apply this methodology in building our growing number of reusable design concerns. In this chapter we demonstrate how we build a particularly large *Workflow* concern incrementally.

Incremental modelling proposes to build large software design models specifying complex structure and/or behaviour by incrementally putting together models of smaller size. In the context of incremental modelling, models are named *base models* if they are self-contained, i.e. they can fulfill their purpose as is, whereas so-called *model increments* have to be applied to some base model to fulfill their purpose. Therefore, by its nature, model composition in the context of incremental modelling is asymmetric, i.e., we obtain an *incremented model* by

composing a *base model (also called base realization model or base application)* with a *model increment*.

Breaking down the concern into small *model increments* not only allows the modeller to add an increment when it is needed by the reusing application/concern, but also to add the increment when the knowledge required to model it becomes available. Additionally, incremental modelling contributes towards addressing the scalability problem provided that a *model increment* is small in size and that specifies a logical step towards the final model.

This chapter discusses how we incrementally modelled concerns for the reusable concern library. We start this chapter by discussing how incremental modelling integrates with modern software design processes and practices including Software Product Lines in the next section. We then discuss the properties of model increments in Section 8.2. We then present how incremental modelling can be used for incrementally developing a concern in Section 8.3. The incremental design of the *Workflow* concern is shown in Section 8.4. Section 8.5 provides an overview of other concerns we designed for our reusable concern library and Section 8.6 concludes this chapter.

8.1 Incremental Modelling and the Software Design Process

Incremental modelling integrates very well with modern software design processes such as prototyping, iterative methodologies or the Unified Process [70]. These methodologies design and implement an application in phases. First, a simple version of the application is developed that only provides core functionality and services. Detailed and additional functionalities are added in subsequent iterations.

Vertical Decomposition of Design: Within each iteration, software design typically follows a top-down and/or bottom-up strategy, depending on whether the focus is to first

elaborate high-level abstractions and functionality, or rather to initially flesh out certain important low-level details of parts of the design. For instance, if detailed functional requirements for the software under development have been elaborated, the initial design phase might begin with deciding on a high-level architecture for the system, and how the required functionality is to be decomposed into subfunctionalities and allocated to different components. On the other hand, if a certain subfunctionality is crucial to the functioning of the software under development, or if reusing an existing software artifact such as a middleware is mandatory or highly cost-effective, then low-level details of a specific required functionality might be designed first in order to determine if the design is actually feasible.

To enable such top-down or bottom-up design, *abstraction* and *information hiding* are key to tame the inherent complexity of a system [93]. Information hiding is the activity of consciously deciding what parts of a software module should be exposed to the outside, i.e., the "rest" of the software under development, and what parts should be hidden from external use. To allow a modeller to state what is internal and what is external to a model, the modelling language must provide constructs to define a model's *interface*. The interface of a software design model describes an abstraction of the actual model. Only the structural and behavioural properties that are relevant to use the model and its provided functionality are shown in the interface. The design details pertaining to *how* this functionality is provided are not relevant to the user.

Horizontal Decomposition of Design: When transitioning from one iteration of the software design to the next, it is typical to consider additional functionality. As a result, the core parts of the existing design are complemented with additional functionality, or new components are introduced that take care of providing the additional functionality and the

existing design is adapted to integrate the new components. To make incremental modelling useful in this context, the modelling language and composition techniques used to combine models needs to support *adaptation of* existing *model interfaces*.

Feature-Oriented Decomposition of Design: Finally, incremental modelling also integrates well with software product line (SPL) engineering, a software development methodology in which a family of applications is developed that share common functionality. In SPLs it is common practice to specify the variability in terms of *features* that an application might have separately from the associated artifacts that provide implementations of the features. If incremental modelling is applied in an SPL context, mandatory features would be designed in base models, whereas optional features would be designed in model increments. To obtain a software design model for a specific application (commonly called *product* in SPL terms), the base models would be composed with all the model increments that correspond to the selected features for the product.

8.2 Properties of Design Model Increments

In this section we comment on properties that, according to our experience, a good design model increment should have.

Size: Each individual model increment should be small, as it has been shown in psychological studies that the active *working memory* of a human is limited [77]. When an individual undertakes a mental task (e.g. attempting to analyze a model or answer questions about a model) that exceeds their working memory capacity, errors are likely to occur [115]. Examining or building a model of a system induces a certain mental effort on the modeller, which corresponds to the amount of working memory a certain task utilizes [91]. This value depends on model-specific factors, one of which is model size.

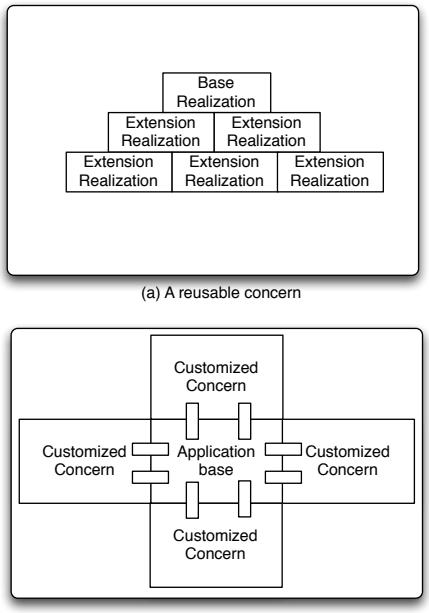

Figure 8–1: Extension versus customization increments in CORE. Figure (a) shows that a reusable concern is built by incrementally adding extension realization models to a base realization model. Figure (b) illustrates that an application is built by composing user-tailored version of the concerns with the base application.

Completeness: What is even more important than size though is coherent modularization. Each model increment specifies a logical step towards the final model, and therefore needs to contain *all* the structural and/or behavioural elements pertaining to that logical step. This is important for two reasons:

- *Internal Consistency*: Having all model elements pertaining to a logical increment in one place is useful for reasoning about the increment itself. It also simplifies making coherent changes to the structure and/or behaviour modelled by the increment.
- *Consistent Use*: Since the model contains all relevant model elements of the logical increment step, the result that is obtained when applying the increment to the self-contained base model is also self-contained.

Kind/Type of Model Increment: Because of the way incremental modelling is used within the software design process, there are essentially two kind of model increments: *extension increments (also called extension realization models in the context of CORE)* and *customization increments (reused concerns in the context of CORE)*.

When using *extension increments*, the modeller's intent is to add additional structural and/or behavioural model elements to the base model that provide *additional*, *alternative* or *complementary* properties to what already exists in the base model. *The extension increment augments the interface of the base model* with additional structure and behaviour. In a sense, an extension increment specifies a horizontal model transformation that maintains the level of abstraction of the base model. *The incremented model can still fulfill the same purpose* as the base model did, and can be also used for additional purposes that are introduced as a result of the extension. Consequently, the model elements exposed in the interface of the incremented model can stem from both the base and the incremented model. Regardless of

where they stem from, all the model elements in the interface are equally relevant to the outside. A extension realization model in the context of CORE is an extension increment, as it provides additional or alternative properties to what other realization models within the same concern already provide.

Fig. *8–1* (a) illustrates extension increments in the context of CORE. Extension realization models of a concern are built incrementally by adding them to some base realization model(s). Therefore, some features of the concern are realized by base realization models, while others are realized by extension realization models. The relationship between features and realization models is shown in the metamodel discussed in Chapter 6, which states that a feature can be realized by many realization models, or by no realization at all (for example, to resolve a conflict with another feature, a feature can be realized by two realization models, one to resolve the conflict). However, usually a feature is realized by one realization model in design concerns. The incremented model resulting from composing a base realization model with an extension realization model is still at the same level of abstraction (i.e., the abstraction level of the concern containing these models).

When reusing a concern, a user-tailored reused concern in the context of its reuser – regardless of whether the reuser is another concern or an application – is considered a customization increment. The structure and the behaviour of a *customization increment* is adapted to serve the purpose of its reuser. In other words, the modeller *alters* or *augments* the properties of the reused concern to render them useful for a *new purpose*. In a sense, a customization increment specifies a vertical model transformation that produces an incremented model (i.e., the reused concern + the reusing concern/application) that is at a different level of abstraction (or domain). *The incremented model fulfills a new purpose*,

where part of the structure and/or behaviour is provided by model elements of the reused concern. The interface of the incremented model exposes only elements produced as a result of customization. These model elements and properties can stem from the reusing concern/application or the altered/augmented reused concern. The fact that the incremented model is partially based on a reused concern is hidden.

Fig. *8–1* (b) illustrates customization increments in the context of CORE. An application is typically built by reusing many concerns, by applying the three-step reuse process discussed in Chapter 4. During the reuse process, a user-tailored version of the reused concern is produced, and the partial elements in its customization interface are concretized in the application. The incremented model resulting from this reuse process is at a different level of abstraction or domain. For example, the incremented model resulting from reusing the *Association* concern in the *Authentication* concern is no longer at the low abstraction level of *Association*. Similarly, the incremented model resulting from reusing the *Authentication* concern in the *Bank* application no longer serves only the security domain.

In summary, vertical transformation in the context of CORE occurs when reusing a concern in another concern or an application, moving the reused concern to a different level of abstraction or domain. Whereas horizontal transformation happens as a result of extending a realization model by another realization model within the same concern, adding additional functionality to what already existed, while still serving the same domain of the concern.

8.3 Supporting Incremental Modelling in CORE

Incremental modelling is integral to CORE. In fact, all three types of decompositions discussed in Section *8.1* are present in the CORE metamodel. The variation interface of

a concern supports the feature model decomposition, and the metamodel allows modelling languages to specify the two types of model increments described in Section *8.2* . In CORE, horizontal decomposition is achieved by breaking down the structure and the behaviour of a concern into a set of realization models, while vertical decomposition is achieved through the CORE reuse process. Concretely, incremental modelling is supported in the CORE metamodel by the following:

- CORE allows modelling languages to define their own composition specifications for their realization models through subclassing from *COREBinding, COREPattern,* and *COREMapping*. A realization model can specify composition specifications to other realization models within a concern, allowing for horizontal decomposition through extension increments.
- The CORE metamodel specifies how to reuse a concern by defining two metaclasses, *COREReuse* and *COREModelReuse* (discussed in detail in Chapter *6*). As mentioned earlier, building an application/concern by reusing existing concerns results in vertical decomposition.
- In addition, CORE allows modelling languages to explicitly specify the visibility of model elements through the *COREVisibilityType* enumeration in the metamodel. *COREVisibilityType* provides two options: *public* and *concern,* specifying whether the model element is visible from outside the concern or only from within the concern, respectively, making it part of the usage interface in the former case. When a concern is reused, the model transformation implementing the composition (i.e., the weaver) changes the visibility of its *public* model elements to *concern.* Additionally, the metamodel requires languages to specify whether the partial elements need to be concretized

within the concern, or outside the concern through the *COREPartialityType* enumeration. *COREPartialityType* provides three options: (i) *none:* means that the model element is complete (i.e. it is not partial), (ii) *concern*: the model element is partial and needs to be concretized within the concern, as shown in the example of the *Authentication* concern in Chapter *4*. Elements of a realization model of a feature, such as |*Credential* in the realization model of the root feature of *Authentication*, can have their partiality type set to *concern* so that they must be concretized later on for every possible feature configuration of the concern (|*Credential* became concretized as *Password* later on), (iii) *public:* the model element is partial and must be concretized outside the concern, i.e. the model element is part of the customization interface. Together, *COREVisibilityType* and *COREPartialityType* allow for scoping of model elements in the context of concern reuse, supporting the principles of information hiding [93].

- Finally, the CORE metamodel requires the concern to specify a feature model (*COREFeatureModel* and *COREFeature*) allowing the concern to be decomposed into features.

8.4 Incremental Design of a Workflow Middleware

To illustrate incremental modelling in the context of CORE, this section describes how we incrementally modelled a design concern for workflow execution engines. A workflow is a depiction of a set of operations that need to be completed in a certain order to fulfill a goal or task. For example, workflows have been used in software development to describe how a system under development is supposed to interact with its environment. A well-known modelling formalism that can be used to describe general workflows is UML Activity Diagrams [89]. Another example is the User Requirements Notation (URN) [56], a visual

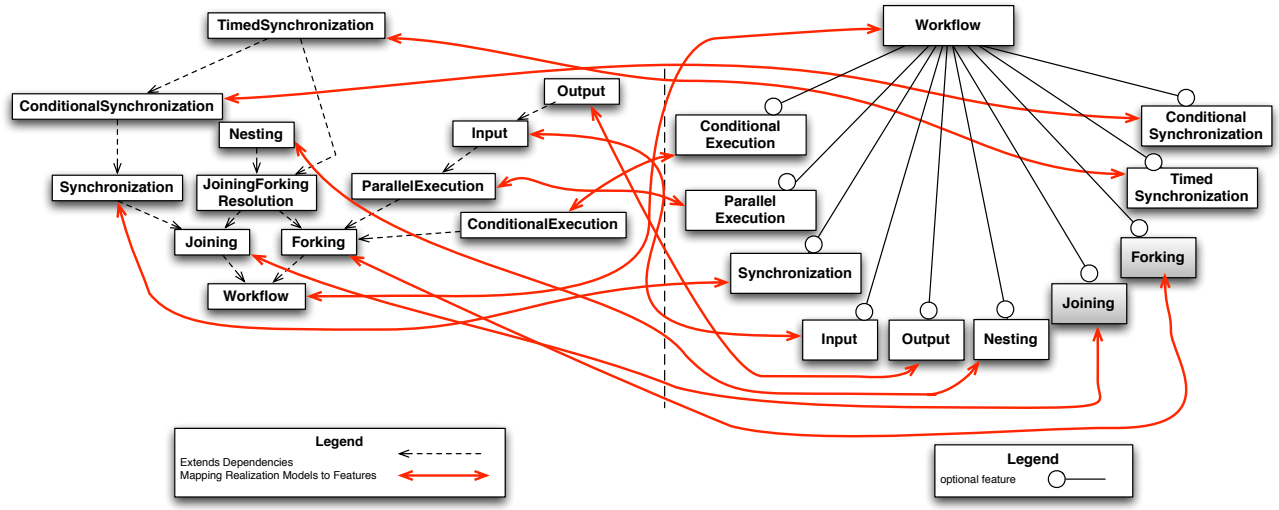

Figure 8–2: The Variation Interface (right) and Realization Models (left) of the Workflow Concern

language standardized by the International Telecommunications Union (ITU) intended for modelling interaction scenarios between a system under development and its environment.

In [33, 15, 14] we have presented an initial aspect-oriented design of a workflow execution engine that provides the user with the functionality to define URN workflows and execute them. In this section, we show how we redesigned that workflow middleware following the guidelines of incremental modelling presented in this chapter.

8.4.1 Identifying Features for the Workflow Middleware

This subsection discusses the features of the *Workflow* concern and its reuses. In the next subsection, we will discuss the realization models of *Workflow* in more detail. The requirements for our workflow middleware were clear: all the workflow constructs defined in the URN standard must be supported. That way, our workflow engine would be able to execute any workflow defined within a URN scenario model, also called a use case map.

An initial decomposition of the design was suggested by the URN standard, as the language defines many different workflow nodes, such as start and end nodes, conditional nodes, timers, synchronization nodes, etc. The right-hand side of Figure 8–2 shows the feature model part of the variation interface for the workflow concern, which specifies that all workflow middlewares are based on the feature *Workflow*, and optionally can be configured with any of its subfeatures. The left-hand side of Figure 8–2 shows the realization models for the *Workflow* concern, and illustrates how each model extends from other models, with *Workflow* being the base model for all extension realization models. We discovered that nesting, synchronization, conditional synchronization nodes, share a common property: both nodes need to be able to handle multiple incoming paths. We therefore encapsulated the structure and behaviour needed to support multiple incoming paths in an extension increment named *Joining*, which *Synchronization* and *ConditionalSynchronization* extend (*Synchronization* directly extends *Joining,* while *ConditionalSynchronization* extends *Synchronization*). Likewise, or-forks, and-forks, timers and stubs are all URN workflow nodes that have more than one outgoing path. We designed a extension increment *Forking* to provide this functionality, which *ConditionalExecution*, *ParallelExecution*, *Input,* and *Output* extend. *TimedSynchronization* and *Nesting* (*Nesting* nodes are the workflow nodes that are used in URN to represent entire subworkflows within a workflow) use functionalities provided by both *Joining* and *Forking,* however, unwanted interactions need to be resolved when using these two models together. Therefore, *TimedSynchronization* and *Nesting* extend a realization model (*JoiningForkingResolution*) which resolves the interaction between *Joining* and *Forking.*

Note that *Joining* and *Forking* are shaded in the right-hand side of Figure 8–2; they are only important internally for other features within the Workflow concern. These two

features are not selectable, i.e., the selectable role name (see the metamodel in Chapter 6) is set to zero for these two features. In other words, *Joining* and *Forking* are not part of the variation interface, therefore, the user of the *Workflow* concern can not select them, they are designed as internal features because they reuse other concerns as we will discuss shortly.

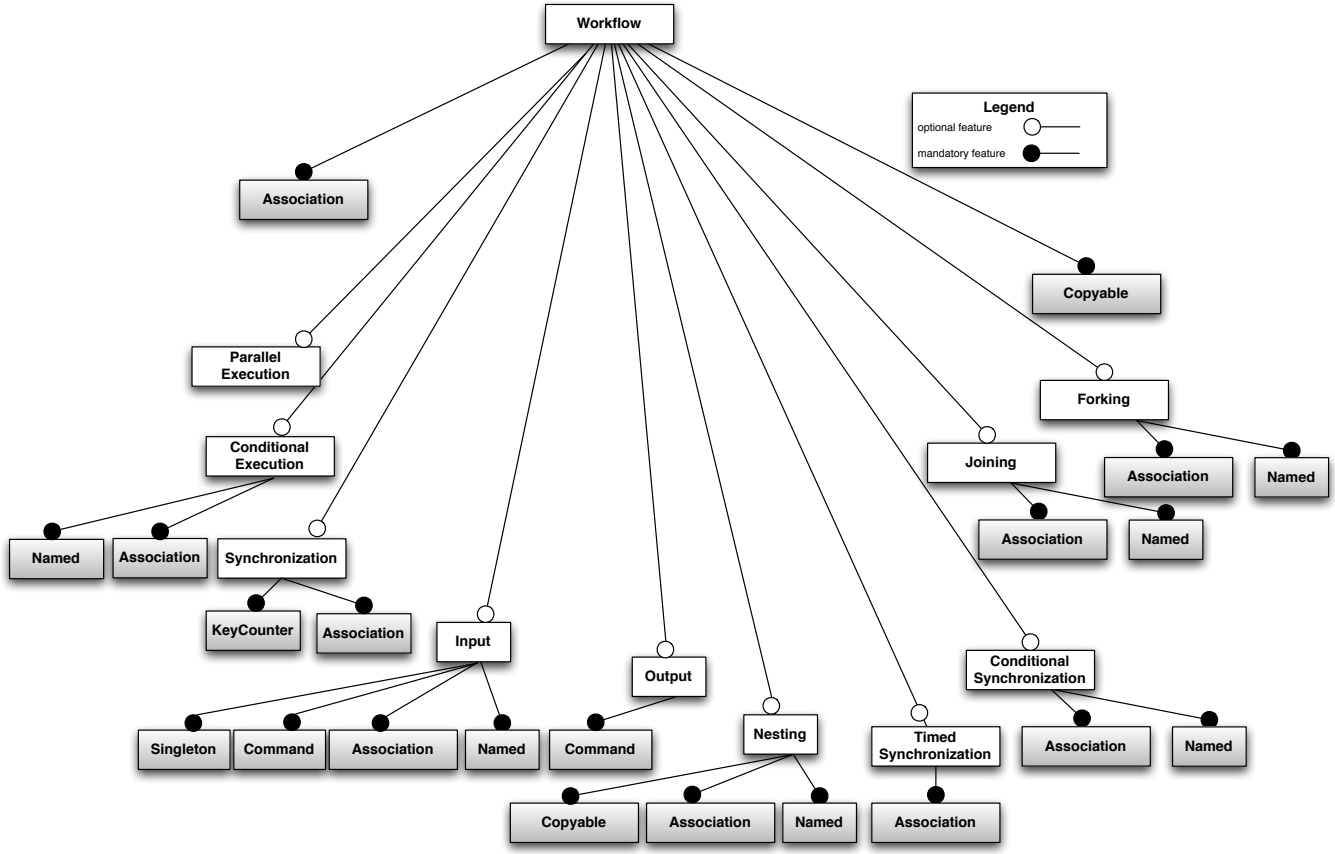

Figure 8–3: Workflow feature model

When designing each feature, we tried to reuse as many existing concerns as possible that we had created for other designs in the past. The reuse dependencies are shown in Figure 8–3, where reuses are shown as mandatory subfeatures in grey rectangles. There are six reused

concerns in total encapsulating: the singleton design pattern (*Singleton*), copying functionality (*Copyable*), naming functionality (*Named*), counting functionality (*KeyCounter*), and a concern that groups common data structures such as arraylist to create associations between two classes (*Association*). In cases where a concern is reused more than once in a feature, we only show one reused concern as a mandatory subfeature. Except for *KeyCounter*, all reused concerns had been developed previously while modelling other concerns or applications. We decided to model *KeyCounter* as a separate concern because of its potential to be reused in the future. Some of these reused concerns themselves reuse other concerns, resulting in a complex reuse hierarchy. This is the case, for example, with *KeyCounter,* which itself reuses *Association*.

When features of the *Workflow* concern are selected (i.e., when the *Workflow* concern is reused), the resulting user-tailored version of the *Workflow* concern (i.e. the incremented model) will be at a different level of abstraction/domain than its reused concerns, demonstrating vertical decomposition. For instance, the resulting user-tailored version of the *Workflow* concern does not provide the naming functionality (which is provided by the *Named* concern) in its interface. The functionality of *Named* is reused in the design of several realization models, but this reuse is hidden from the users of *Workflow*.

8.4.2 Realization Models for Workflow

This section discusses in more detail the realization models of the *Workflow* concern shown in Figure 8–2. We explain how we incrementally modelled the *Workflow* realization models with RAM. Incremental modelling is possible in the context of RAM using model transformation and composition techniques such as *extends dependency* between aspects, *inheritance* and *class merge* for structural design modelling, and *overriding* and *advising*

for behavioural design modelling. We will outline the resulting composition algorithm for structural and behavioural views. All presented concepts are illustrated with example models from the *Workflow* concern case study.

Initially, RAM did not support *extension* increments. To be compatible with CORE concepts, we added the concepts of extension increments, so an aspect can extend from other aspects inside the same concern. Extension increments should use the *extends* dependency using the syntax: `aspect A extends B`. If a model increment B extends a base model A, no mapping of model elements of A to model elements of B need to be specified by the modeller. All model elements of A are automatically visible in B. It is as if B was part of A, and hence classes in B can access properties of classes in A that are not part of A's usage interface. In particular, B can use *inheritance* and *overriding* to extend the structure and behaviour provided by A. Since B is an extension increment, A is likely to define core design concepts that B wants to provide *alternatives* for. To do this, B can use classes defined in A as a superclasses for new classes introduced in B to extend the structure. To extend base behaviour with additional functionality, overriding of operations naturally complements inheritance. Exploiting polymorphism, the extensions to A introduced through inheritance in B can collaborate with the other core classes defined in A.

Note that in the case where the structure and behaviour provided by A needs to be adapted within the concern in order to work correctly with the behaviour introduced by B, the modeller can also additionally use class merge and advising when specifying an extension increment. Of course this means that instantiation directives that map elements of A to elements of B need to be provided. The workflow design example illustrates such a case.

When the weaver combines an extension increment A with a base model B, it combines the aspect interfaces of A and B by performing a *union* operation: the interface of the incremented model contains all model elements of A *and* B and exposes all public operations defined in A *and* B. As a result, the user of the incremented model is given the alternative to either instantiate core (super) classes provided by A or alternative, extended (sub) classes provided by B. In other words, the user can access core functionality provided by A or rely on alternative functionality modelled in B.

The *Workflow* Base Model

After careful analysis we grouped all essential structure and behaviour necessary for defining and executing the most basic workflows within a base realization model that we named *WorkFlow*. It contains the structure and behaviour necessary for supporting start nodes, activities and end nodes, sequential composition, as well as execution infrastructure implementing token-passing semantics for these base constructs. We then designed the more sophisticated workflow constructs of URN as *extension increments* of *Workflow: ParallelExecution* (and-fork in URN terms), *ConditionalExecution* (or-fork in URN terms), *Synchronization* (and-join in URN terms), *ConditionalSynchronization*, *TimedSynchronization*, and *Nesting* (stub in URN terms). Some URN constructs, for example the or-join, did not exhibit any behaviour not already covered by *Workflow*, and therefore did not need to be modelled at all.

Figure 8–4 depicts the RAM realization model for the root feature *Workflow*, which defines the essential model elements for any kind of workflow execution middleware. The *structural view* shows that a workflow is comprised of `WorkflowNodes`, which can be sequence or control flow nodes. `StartNode` and `|CustomizableNode` are sequence nodes, the `EndNode`

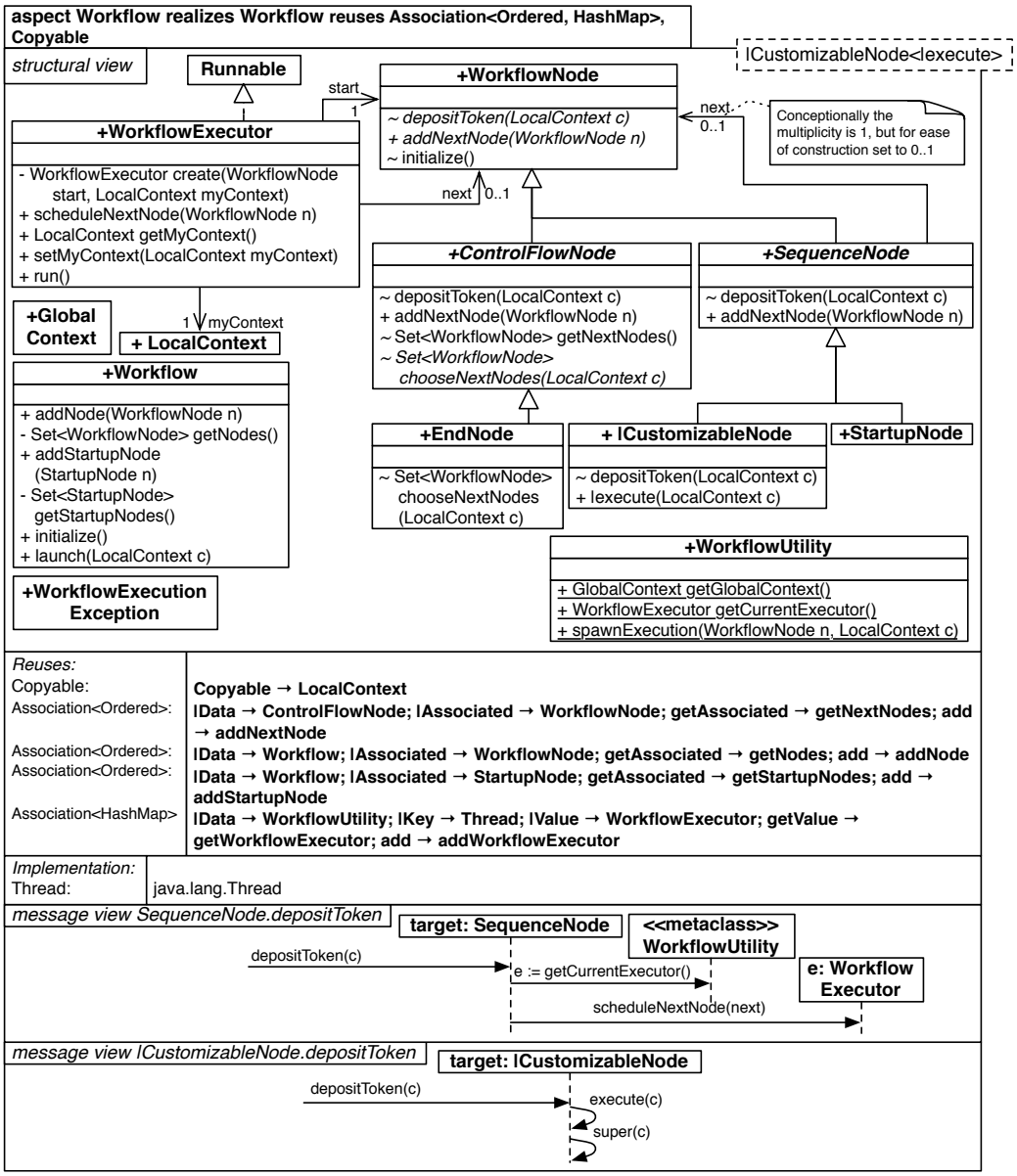

Figure 8–4: The Base Workflow Model

is a control flow node. The class `WorkFlow` has methods to add nodes and startup nodes. `WorkflowExecutor` defines the methods to spawn the execution of a workflow and to schedule the next node in the execution sequence. Figure 8–4 shows the behaviour of two methods using sequence diagrams: the *depositToken* method of `SequenceNode`, and the *depositToken* method of `|CustomizableNode`, the full models of the *Workflow* concern are downloadable from [9].

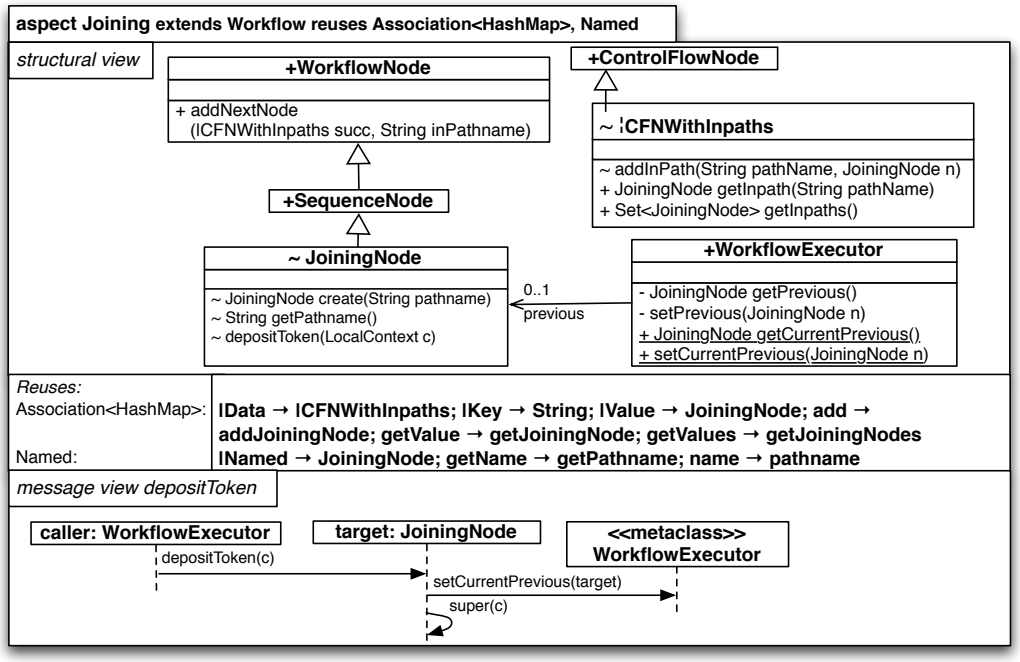

Figure 8–5: The *Joining* Extension Increment

As mentioned earlier, the usage interface is comprised of all the *public* model elements, i.e., the structural and behavioural properties that the classes within the model expose to the outside. These usable elements are highlighted in the model with a + modifier (such as the *addNode* operation). The elements of *COREVisibilityType*: *concern* are prefixed

with a ~, such as the *depositToken* operation. Classes with *COREVisibilityType*: *concern* are also prefixed with ~, and the public classes are prefixed with +. The names of public customizable elements (i.e., *COREPartialityType: public*) are prefixed with a |, and they are grouped together in a template parameter box on the top right corner of the aspect model (which means they are part of the customization interface of the concern). Model elements that must be concretized within the concern when a final model is created are given *COREPartialityType: concern* and are prefixed with a discontinuous vertical bar ¦, as in case of ¦`CFNWithJoinings` in *Joining* shown in Figure 8–5. The customizable elements of *Workflow*, for example, specifies that `|CustomizableNode` and `|execute` are a generic class and a generic method, respectively, that need to be concretized by the reusing concern/application in order for these elements to be useful.

Structural Increments

RAM supports the CORE incremental modelling concepts presented in Section *8.3* by extending the CORE metamodel, which we describe in Chapter 7. RAM's structural and behavioural model composition operators (i.e., the RAM weaver) have been adapted to support incremental modelling (inheritance and class merge for structural composition, and method overriding and advising for behavioural weaving) as outlined in Subsection 8.4.2. Here we illustrate how structural increments are possible in RAM with some *Workflow* realization models.

Class Merge: RAM provides *class merge* as a way to share structural model elements between classes. In RAM, if an aspect B wants to define a class `ClassB` that needs, among others, the properties of a class `ClassA` defined by some other aspect A, then B can **extend** A. In this case, *class merge* (sometimes also called class diagram composition) is used to

merge the model elements in each of the models according to the instantiation directives defined by the modeller. Because of the asymmetry of model increments, the directives are defined in the increment model, and specify how the elements of the base are mapped to elements in the increment. In aspect-oriented terms, the mapping specifies how A is woven into B.

As shown in Figure 8–4, the *Workflow* model customizes *Association<Ordered>,* which is the reused *Association* concern after selecting the feature *Ordered,* three times through class merge. *Association* is a concern at a lower level of abstraction that provides the structure and functionality to associate an instance of `|Data` to multiple instances of `|Associated`. *Workflow* reuses *Association<Ordered>* to associate a `WorkFlow` with multiple, follower `WorkflowNodes`. The mapping, described in the reuses box, also specifies that the operation `getAssociated` should be exposed in the interface of *Workflow* as `getNextNodes`, and that `add` should be used as the behaviour for `addNextNode`. The weaver uses class merge to merge `|Data` with `|Workflow` and `|Associated` with `|WorkflowNode.`

Joining in Figure 8–5 is an extension increment of *Workflow* which provides structure and behaviour to allow control flow nodes to have multiple, named incoming paths. Since *Joining* extends *Workflow*, all model elements in *Joining* that have the same name as model elements in *Workflow* are automatically merged. Using this mechanism, *Joining* adds a new method `addNextNode` to all `WorkFlowNodes`. Similarly, the `WorkflowExecutor` is augmented with an association to a `JoiningNode`, which is used to remember through which path a node was reached during execution. `JoiningNode` also reuses the *Named* concern, and customizes `|Named`, which provides the structure and behaviour to associate a name in form of a *String* with a class. The class `|Named` is merged with `JoiningNode` as specified in the reuses

compartment. In a similar way, *Joining* also reuses the *Association<HashMap>* concern and maps `Strings` to `JoiningNode`.

Inheritance: *Inheritance*, or generalization-specialization as defined by the UML [89], is a well-known concept of object-orientation that makes it possible to share structural properties among objects. Concretely, common structure is defined in what is called a *superclass*. Any *subclasses* that inherit from it also have the same structural properties, and can define additional properties, if needed.

In RAM, incremental modelling with inheritance in class diagrams is supported as illustrated in the *Joining* aspect shown in Figure 8–5. *Joining* extends *Workflow*, and defines a subclass ¦`CFNWithJoinings` that inherits from the class `ControlFlowNode`, which is defined in Workflow. Likewise, a new kind of `SequenceNode` is defined called `JoiningNode`.

Structural Weaving: The model transformation technique that the TouchCORE tool uses in order to combine the structural elements of two aspects A and B, regardless of whether they use inheritance or class merge, is based on the class composition technique published by France et al. [101].

In the case where B *extends* A, the weaver creates explicit mappings for all the implicit mappings (an implicit mapping is between an element in A to an element in B that has the same name and signature for the element in A), and copies all model elements that are not mapped from A to B. In the case where B specifies an instantiation mapping, the mapping is first used to change the names of all model elements in A to the corresponding model elements in B. Then, our tool combines A and B by moving for each pair of model elements (classes, associations, operations, parameters), i.e., $(ma, mb) \mid ma \in A \wedge mb \in B \wedge ma.name =$

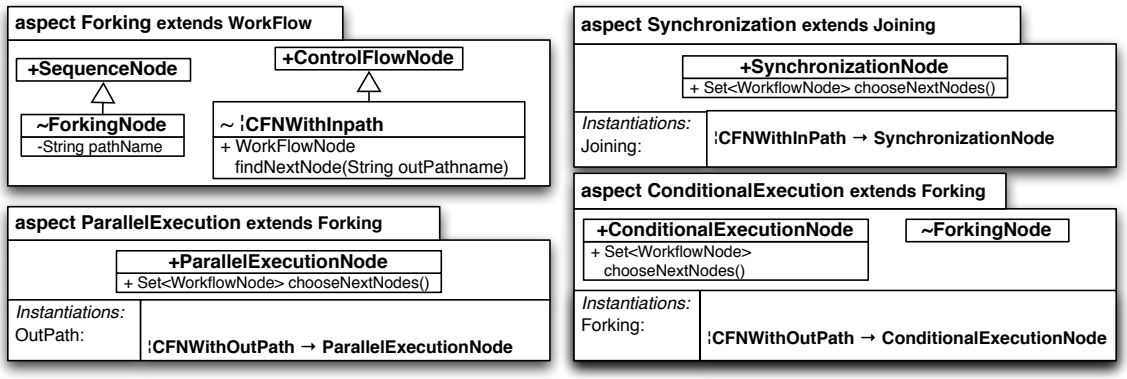

Figure 8–6: Synchronization, Forking, Parallel- and ConditionalExecution

mb.name all properties attached to *ma* to *mb*. The result is a new self-contained model which can therefore be used as a base for further increments.

Figure 8–6 presents very simplified versions of the structural view of additional extension realization models of the *Workflow* concern. A `SynchronizationNode` as defined in *Synchronization* is a node that blocks all incoming executors until it has received a token on each incoming path. The instantiation directives show that `¦CFNWithInpath` from *Joining* is merged with `SynchronizationNode`.

Some nodes in a workflow are not just connected to one following node. Figure 8–6 shows an increment *Forking* that extends the *WorkFlow* base model to provide an alternative control flow node that can have multiple outgoing paths. Again, inheritance is used to provide this alternative. *ParallelExecution* is an increment based on *Forking* that defines a control flow node that executes all following nodes in parallel. The *ConditionalExecution* aspect shows how the designer can increment *Forking* in a different way to define another alternative control flow node that represents conditional execution.

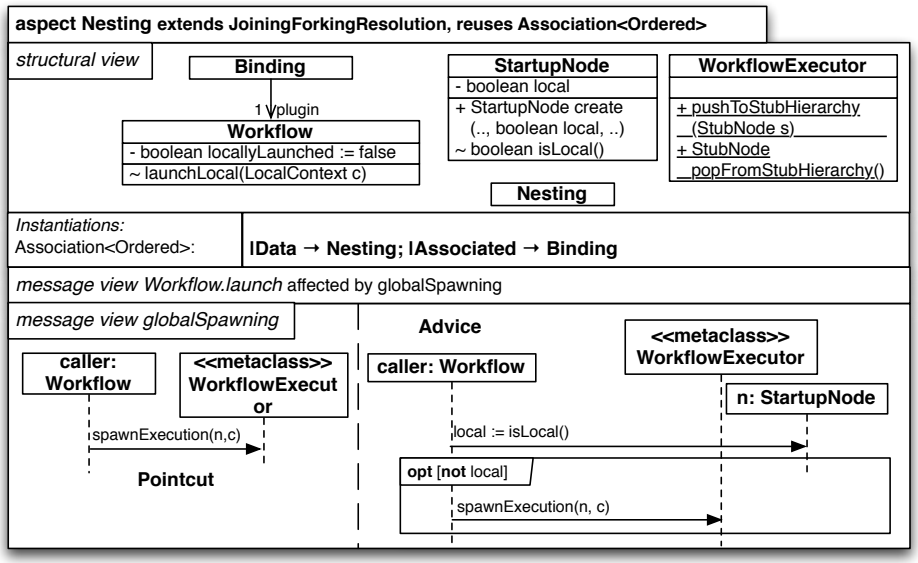

Figure 8–7: The *Nesting* Extension Increment

Behavioural Increments

Overriding: Operation *overriding* is an object-oriented technique that is used in the context of inheritance that allows a subclass `B1` to replace the behaviour of an operation *o* provided by a superclass `A1` by providing a new definition of *o*. Often though, since *B1* shares structural properties with *A1*, the behaviour of *o* for *B1* – the overriding operation – needs to include the behaviour defined for *o* in *A1* – the overridden operation. If that is the case, the overriding operation invokes a special operation called *super*, which results in executing the overridden operation in the superclass. If *super* is invoked, then the overriding operation can be seen as a behavioural increment of the overridden operation.

All workflow nodes in Figure 8–4 provide the operations `addNextNode` and `depositToken` which define shared behaviour. `depositToken` represents the behaviour of all workflow nodes within a workflow, and it is triggered by the workflow executor when it traverses the workflow.

`SequenceNode`, `|CustomizableNode` and `ControlFlowNode` each provide different behaviour for `depositToken`. `SequenceNode` simply schedules the execution of the following node as shown in the `depositToken` message view in Figure 8–4. Figure 8–5 shows how the behaviour of operation `depositToken` in `SequenceNode` is overridden by the model increment *Joining* simply by specifying a new message view for `depositToken` (with the same signature) as the one that is to be overridden. The behaviour shown specifies that the workflow executor first remembers the Joining node it came from before executing the original behaviour provided by `SequenceNode`.

Advising: Besides overriding, RAM also provides a way for a model increment to specify a modification to the behaviour of an operation provided by a base model. This technique is often referred to as *advising*. It allows a model increment to define new (partial) behaviour with a sequence diagram, and specify at which point in the base behaviour this new behaviour should be inserted. To enable advising, sequence diagram weaving technology is used to combine the two behaviours.

The *Nesting* realization model illustrates this in Figure 8–7. *Nesting* provides the structure and behaviour to allow workflows to be nested. According to the URN specification, nesting has an impact on what is supposed to happen when a workflow is launched. Without nesting, execution should begin concurrently at all startup nodes in the workflow. Nested workflows, however, can specify startup nodes to be local or global. When launching a nested workflow, execution should only begin at the startup nodes marked as global. To realize this change in semantics, *Nesting* uses advising. The message view *globalSpawning* defines a pointcut sequence diagram that detects calls to operation *spawnExecution* on instances of *Workflow*. The message view then states that for each detected call, it checks whether the

StartupNode is local, and calls *spawnExecution* only if the startup node is not local. After weaving *Nesting* with *Workflow*, the message view *launch* in *Workflow* is augmented with the messages found in the advice part of *spawnExecution*.

Behavioural Weaving: The model transformation technique that the TouchCORE tool uses in order to combine the behavioural elements of two aspects A and B, regardless of whether they use overriding or advising, is based on the sequence diagram weaving approach published by Klein et al. [65]. In case B *extends* A, the weaver creates explicit mappings for all the implicit mappings, as done in the structural weaving. In the case where B specifies an instantiation mapping, the mapping is first used to change the names of all model elements in A to the corresponding model elements in B. Then all message views of A are moved to aspect B. Next, any advising specified in B is processed. To do that, for each message view *oa* coming from A that is advised by a message view *mb* in B, the match(es) of the pointcut sequence diagram of *mb* in the sequence diagram of *oa* are replaced with the advice sequence diagram of *mb*. Finally, standard and overridden operation invocations are processed as follows: for each message view *mb* in B, any call to operations *oa* originally specified in A or calls to *super* are replaced with the sequence diagram specified in the message view *oa* that came from A, respectively with the message view *mb* coming from A[1] .

[1] It should be noted that this last step is optional and should be executed only if the modeller wants to show the details of the called operation behaviour in the message view where the behaviour is called.

8.4.3 Properties of the Model Increments of the Workflow Concern

- **Size**: The sizes of the model increments in the *Workflow* concern are small. The biggest model increment is the *Workflow* realization base model with 10 classes and 11 public operations, the smallest model is *ParallelExecution* with 1 class and 1 public operation. This remains within the cognitive capacity of a modeller according to [91].
- **Completeness**: All model increments include *all* the structural and/or behavioural elements needed to define workflows with the desired functionality and to execute the workflow with the correct semantics. The reuse process ensures that after selecting features from the feature model in the right-hand side of Figure 8–2, a user-tailored version of the concern is produced in which the extension realization models are woven with the base model following the composition algorithms presented in Chapter 5. An analysis revealed that all possible feature selections allowed by the feature model result in a woven model, i.e., the base model + extension increment(s), that is complete, i.e., it does not contain any concern partial model elements any more.
- **Kind of Increment**: The layout of the dependency graph shown in the left hand side of Figure 8–2 is such that all models above *Workflow* represent extension increments, i.e. they all directly or indirectly extend the interface provided by *Workflow*, directly in the case of *ConditionalExecution*, and indirectly, such as, *TimedSynchronization*. They all are useful for workflow specification and execution, and simply provide different variants of workflow constructs. Reuses in Figure 8–3 illustrate the customization increments, as each reused concern is customized, i.e., after configuring the reuse by selecting the desired features, the model elements from the customization interface are mapped to model elements in *Workflow* to adapted them to the context of *Workflow*.

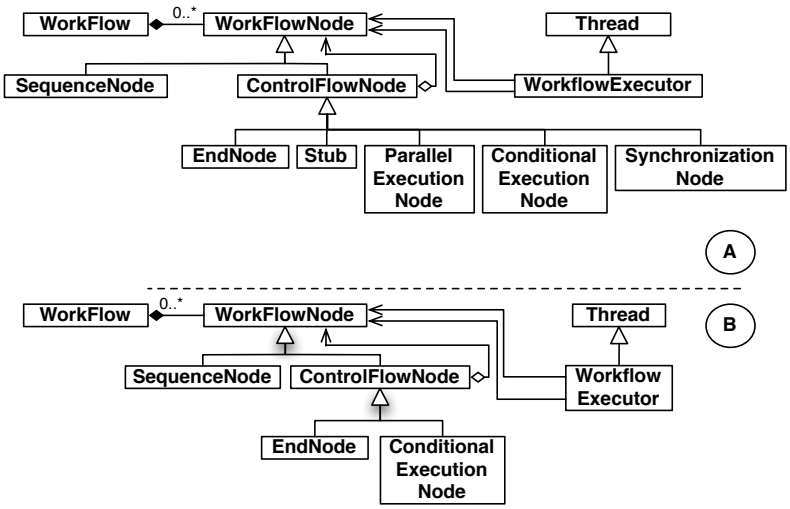

Figure 8–8: Two Possible Final Models

8.4.4 Generating the Complete Design Model

Given all model increments, we are now ready to generate a complete woven model after the user makes a valid feature selection. Figure 8–8 illustrates the power of incremental modelling in CORE. The user of the workflow concern is presented with a feature model (see right-hand side of Figure 8–2), from which he is to select the desired functionality of the workflow design that is to be generated.

The top half of Figure 8–8 (A) shows the generated design model when the features *ConditionalExecution*, *ParallelExecution*, *Synchronization* and *Nesting* are chosen by the user. Because we used alternatives in the increments that designed the individual control flow nodes, a user of the middleware can instantiate the subset of the nodes it needs. For example, one workflow instance can be composed of `SequenceNode`, `EndNode`, and `ConditionalExecutionNode` only, whereas another workflow instance can use a different

subset, e.g. `SequenceNode`, `EndNode`, and `SynchronizationNode`. However, if *all* workflows that the workflow middleware needs to support do not use one kind of node, then a new design model can be generated by selecting only the required features. Figure 8–8 (B) illustrates a generated design model in which only the *ConditionalExecution* feature was selected, and hence only the *ConditionalExecution* extension realization model was composed with *Workflow*.

8.4.5 Reexposed Features in Workflow

As previously discussed, the reexposed features are added to the variation interface of the reusing concern, allowing the developer to defer the decision about which specific variant to use to the next level in the concern hierarchy. The *Workflow* concern reexposes features from two concerns: *Association* and *Networking*. Multiple *Workflow* features reexpose features from the *Association* concern. For example, the root feature Workflow, reexposes the subfeatures of *Ordered*: *Arraylist* and *LinkedList* as shown in Figure 8–9. These reexposed features will provide the user of Workflow to choice of selecting the feature that performs best when there are resource constraints, e.g., limitation on CPU usage. Under such circumstances, the user can weigh the impact of selecting *ArrayList* versus *LinkedList*, as shown previously in Figure 5–3, *LinkedList* performs better than *ArrayList* when inserting/deleting elements, while *ArrayList* performs better when accessing the elements. Under limited CPU capacity, and increased frequency of insertion/deletion of elements, *ArrayList* should be selected. Another example of reexposed features is shown in the *Input* feature, where the network protocol features of the *Networking* concern are reexposed. *Input* reuses the *Command* concern and selects the *NetworkCommand* feature to send commands over

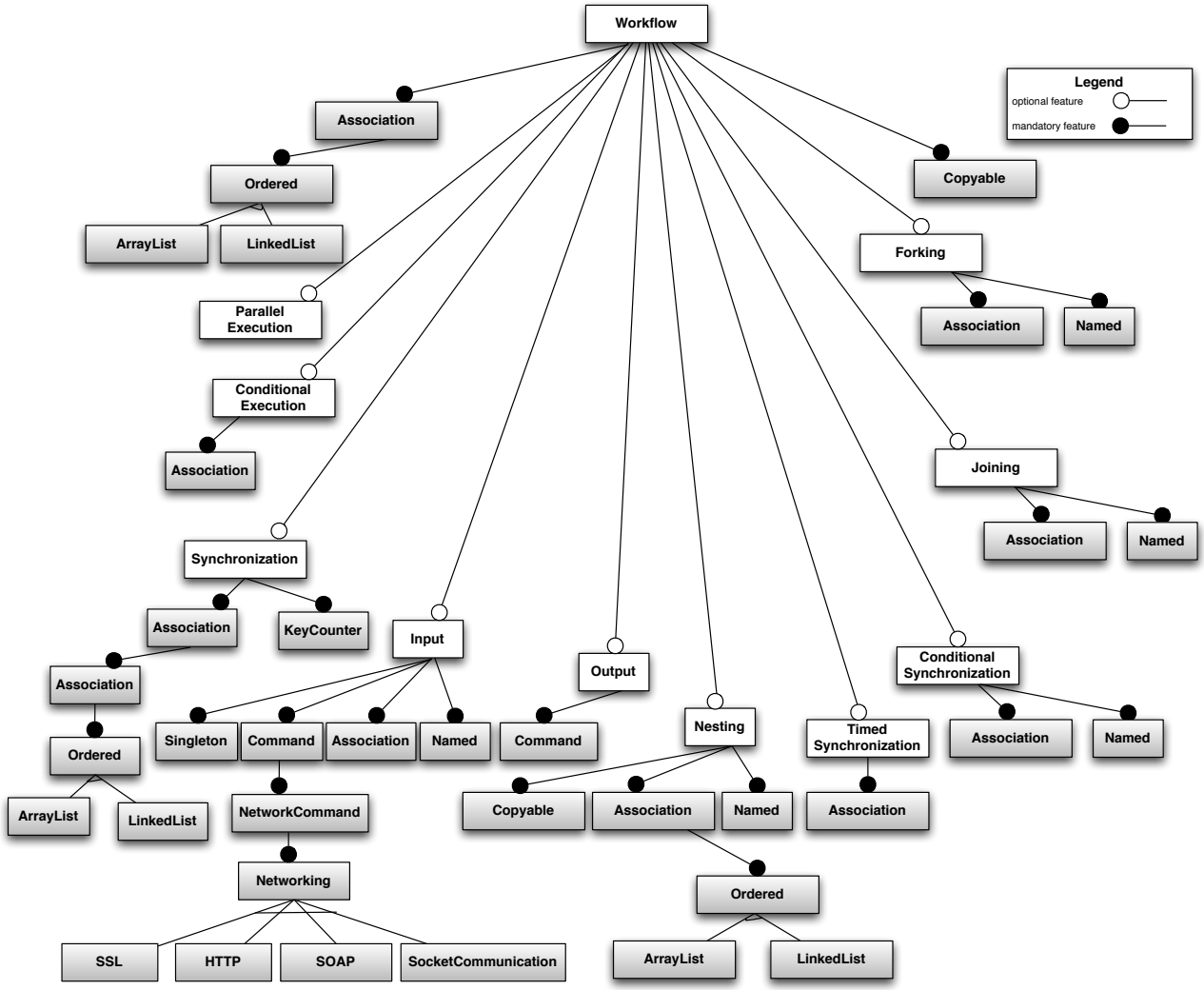

Figure 8–9: The Feature Model of the Association Concern Including the Reexposed Features.

the network. However, selecting the type of network protocol (e.g., HTTP, SOAP, SSL, etc.) is left to the user based on what network type is supported by the *Workflow* user.

8.5 Overview of Other Concerns in the Reusable Concern Library

Using incremental modelling, we additionally designed a number of concerns in our growing reusable concern library. The concerns range from high-level concerns such as *Authentication* that are identified during requirement analysis, to low-level concerns such as *Networking* that address implementation details. Thanks to incremental modelling, we continue to build new concerns by reusing other concerns in the library, and adding new features to existing concerns. Here we provide an overview of some of the concerns in the library beside the *Workflow* concern that we discussed above.

Authentication

This is a requirement-level concern that allows to authenticate the user before providing her access to the system. The variation interface along with some realization models for *Authentication* were shown in Chapter 4. *Authentication* can be done in different ways, including password-based, voice recognition, fingerprint scan, and facial recognition techniques. The password-based authentication feature offers an option of expiring the password after a certain time. In addition, as shown in Figure 4–1, the concern provides features for *Auto Logoff* and *Access Blocking* that allow for automatically logging off the system after a certain idle time and blocking access after a set number of failed trials.

Authorization

The *Authorization* concern provides controlled access to resources. This requirement-level concern is designed to provide functionality of well-known Role-Based Access Control (RBAC) [104] systems. It encapsulates different variants of RBAC, such as using constraints

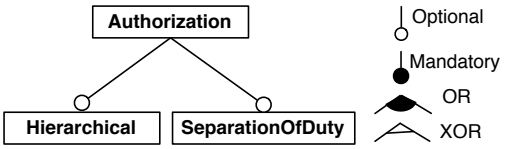

Figure 8–10: Feature Model for Authorization

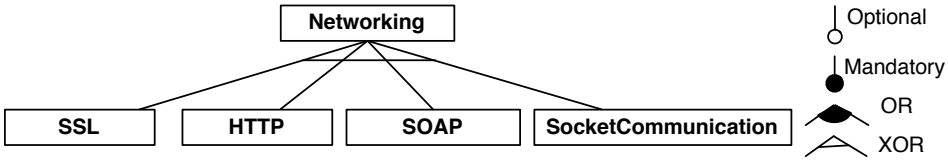

Figure 8–11: Feature Model for the Networking Concern

(*Separation of Duty*) and hierarchies, as shown in the feature model in Figure 8–10. The concern internally reuses the *Authentication* concern.

Resource Management

The *Resource Management* concern provides functionality to manage resources, i.e., find available resources according to a desired set of capabilities and allocate the resources to tasks according. Basic search and optimal search for resources are supported by the concern. The Resource Management concern is discussed in more detail in Chapter 9.

Networking

The *Networking* design concern provides different ways of establishing network communications between processes in distributed systems. So far, the concern provides features for socket-based communication, which we use for sending/receiving messages and objects. However, other means of networking, such as HTTP, SSL, and SOAP shown in Figure 8–11 are planned to be realized in a near future.

Observer

The *Observer* design pattern [46] is a popular software design pattern in which an object, called the subject, maintains a list of dependents, called observers. The functionality provided by the pattern is to make sure that, whenever the subject's state changes, all observers are notified. We discussed the *Observer* concern previously in Chapter 2.

Other low-level design concerns are available in the library such as **Singleton** – models the singleton design pattern that allows objects to restrict its number of instantiated instances to only one instance, **Named** – allows naming and changing the names of objects, **Copyable** – allows creating copies of an object, and **Association** – previously discussed in Chapter 2, provides different ways of establishing associations between classes, particularly covering the case where a class is associated with multiple other classes, and allows ordered, unordered and key-indexed access to the other classes. All of these concerns commonly appear in the design of a system.

8.6 Conclusion

This chapter discusses how we designed concerns for our reusable concern library by applying the incremental modelling methodology. Concepts of incremental modelling such as information hiding, horizontal and vertical decompositions, and assigning models to features are integral to CORE. More specifically, all three types of decomposition necessary for incremental modelling are supported by CORE. Extension realization models allow for horizontal decomposition of a concern, the concern reuse process allows for vertical decomposition of an application or a concern, and the feature model part of the variation interface allows for feature-oriented decomposition. We demonstrated using the *Workflow* concern how a

concern of a considerable size can be modelled incrementally. The same methodology was used to design all other reusable concerns in our reusable concern library.

Chapter 9
bCMS Case Study

So far, the validation of CORE was carried out in this thesis by conducting extensive comparison between concern and other units of reuse, successfully using the CORE reference implementation to corify two modelling languages, and by applying by the CORE reuse process on a growing number of reusable design concerns. This chapter discusses the final contribution of this thesis towards validating CORE: applying the CORE reuse process on a case study of a product family of crisis management system (CMS). A *crisis management system* (CMS) facilitates the handling of a crisis by orchestrating the communication between all involved parties. The CMS allocates and manages resources, and provides access to relevant crisis-related information to authorized users of the CMS in a timely and reliable manner. In [7], a CMS is proposed as a common case study to evaluate the strength and weaknesses of different modelling approaches. The CMS domain was chosen on purpose, because it represents at its core reactive, distributed systems. Furthermore, a CMS needs to be highly dependable, and hence incorporates many crosscutting concerns, such as security and resource management. Additionally, the CMS domain contains much variability due to the different natures of potential crises and ways to address them. The CMS case study has been used extensively by the modelling and software engineering community to demonstrate and evaluate modelling approaches (e.g., to compare aspect-oriented modelling approaches in a special issue of the journal Transactions on Aspect-Oriented Development [7]). Because of the fact that the CMS case study should be applicable to modelling approaches targeting all

software development phases including requirements and variability engineering, the scope of the CMS case study was intentionally very broad. Unfortunately, this broadness prohibits modelling approaches that focus on the detailed design phase to model it in its entirety or at least a significant part of it. Hence, a smaller subset of the case study called bCMS was proposed [27], describing the requirements of a family of car crash CMSs. Because our goal is to demonstrate CORE on a complete system as much as possible, we focus on modelling the bCMS. The bCMS itself has been used to demonstrate concepts of many modelling notations in the workshop series on Comparing Modelling Approaches (CMA) [4].

Fig. 9–1 shows the features of the bCMS as derived from the requirements. Features with a white background are specific to the bCMS. Features with a shaded background represent reused concerns and will be discussed in the next section. The bCMS consists of one main use case (represented by the *MissionExecution* feature), which focuses on the communication between a police station coordinator (PSC) and fire station coordinator (FSC) to handle a crisis. *VehicleManagement* involving police vehicles and fire trucks is also required to accomplish this task. Seven variation points are explicitly mentioned in the requirements document: (i) the bCMS may either handle a *Single* crisis or *Multiple* crises at the same time, (ii) the crisis may be handled by one PSC and one FSC (*SinglePSCFSC*) or multiple coordinators (*MultiplePSCFSC*), (iii) *Authentication*/*Authorization* choices are handled by the respective concerns, (iv) *Encryption* may optionally be provided, (v) the communication among coordinators may vary (handled by the *Networking* concern), (vi) the *VehicleCommunication* may vary, and (vii) the communication between PSC, FSC, and various vehicles may be manual or provided by the system (*PSCToPoliceAndCitizenVehicles*, *FSCToFireTrucks*, *CitizenVehiclesToPSC*).

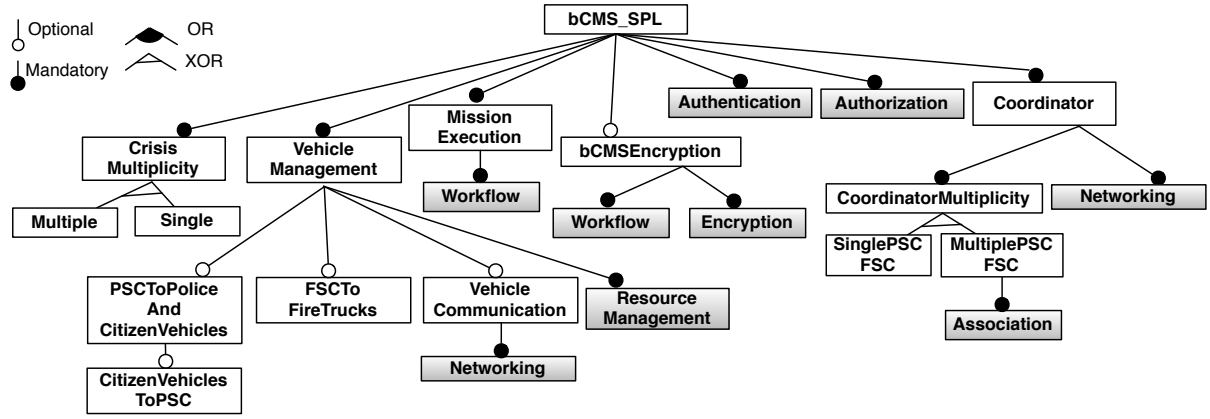

Figure 9–1: The bCMS Feature Model

In the next section, we discuss how we identified the reusing features and the reused concerns from the *bCMS* requirement document. Then, in Section 9.2, we discuss in detail how we reused a particular reusable concern called *ResourceManagement* in *bCMS* following the three-step concern reuse process. We discuss the lessons learnt from this case study in in Section 9.3. Section 9.4 discusses the limitations. Finally, we conclude this chapter in Section 9.5.

9.1 Modelling of the bCMS with CORE

Since a concern uses the most appropriate modelling notations to describe its properties, how a system is modelled follows from the notations used by the concern. The number and types of modelling notations used to describe a concern beyond feature and impact models is rather open-ended. They vary as they are determined by what is best for a particular concern. Without loss of generality, we limit ourselves to workflow and software design notations in this chapter, similar to what we did previously in Chapter 4. Workflows are expressed with the Aspect-oriented User Requirements Notation (AoURN). We use them to capture

a system view of all use cases of a concern and their relationships, i.e., the requirements described as the interaction of the system with its environment. The software design of a concern, on the other hand, is expressed with Reusable Aspect Models (RAM), describing structure and behaviour with class and sequence diagrams, respectively. Consequently, best practices in each of these areas are used to model a system and the bCMS is no exception. However, as mentioned earlier in Chapter 4, we focus in this thesis on how to reuse existing concerns that encapsulate AoURN and RAM models to build an application. We do not focus on how to apply any of the individual modelling notations, describing in detail how to model a system with such a modelling notation. The realistic size of the bCMS allows us to gain valuable experience in concern-driven development, which we discuss further in Section 9.3.

9.1.1 Identification of Concerns and Reusing Features in the bCMS

To fully reap the benefits of reuse, i.e., increasing scalability and avoiding duplication of effort, a feature of a high-level concern should be able to reuse the structure/behaviour/properties of a lower-level concern when appropriate. We have shown examples of concern reuse in several examples in this thesis. Note that if an optional feature reuses another concern and the feature is not selected, then the reuse does not need to be considered. Hence, *concern hierarchies* allow the developer to modularize the application into different layers of abstraction. To develop the bCMS, we loosely follow the strategies of existing aspect-oriented development methodologies to identify concerns within textual requirements or requirements models [79], adapted to the existence of our reusable concern library. Considering that the number of existing concerns in the library is still manageable, a simple match based only on the name of the concern is often sufficient to yield an initial list of reusable concerns.

At the time the bCMS was modelled, ten existing reusable concerns were determined to be applicable to the bCMS. The feature model in Fig. 9–1 shows two levels in the concern hierarchy: the features with white background belong to the bCMS at the top level and the features with shaded background belong to the second level as these are concerns that are reused by features of the bCMS.

As described previously, a concern in CORE may contain models from various abstraction levels, but does not have to contain all of these models. Each concern has a *root phase* in the software development life cycle, where the concern manifests itself for the first time. Depending on what is described by a concern, this phase may be the requirements phase, in which case, e.g., workflow models of the use cases of the concern do exist as well as the corresponding design models. The root phase may also be the design phase, in which case workflow models are not needed, because such a concern does not have interactions with users and other systems in the environment. An example of the former is the *Authentication* concern shown in Chapter 4, which allows a user to gain access to a protected system. An example of the latter is the *Observer* concern which handles how changes in one design artifact are communicated to another design artifact. In the following list of applicable reusable concerns, the root phase of the concern is indicated in parentheses after the concern name. We described each concern in the bCMS in detail in Chapter 8.[1] . Representative of all other reused concerns, the *ResourceManagement* concern is described in more detail throughout this chapter.

[1] See http://www.ece.mcgill.ca/~gmussb1/bCMS/ for all bCMS models.

Authentication (requirements) and *Authorization* (requirements) are applied system-wide to the root feature of the bCMS. *Resource Management* (requirements) allows resources to be managed and allocated to tasks according to the desired capabilities of resources. Basic search and optimal search for resources are supported by the concern. The *ResourceManagement* concern is needed by the *VehicleManagement* feature of the bCMS. *Encryption* (requirements) is an optional feature that applies system-wide. *Workflow* (design) is used to model the *MissingExecution* feature. *Networking* (design) is needed for vehicle and coordinator communication. Other low-level design concerns such as *Observer* (design), *Singleton* (design), *Named* (design), *Copyable* (design), and *Association* (design) commonly appear in the design of a system (they appear at the third or lower levels in the bCMS concern hierarchy).

9.2 Reusing the ResourceManagement Concern

We previously demonstrated the three-step CORE reuse process using an example *Bank* application and the *Authentication* reused concern in Chapter 4. This section describes in detail the three-step process of concern reuse applied to reuse the *ResourceManagement* concern in *bCMS*. The same process is carried out 102 times for 158 feature reuses in *bCMS*.

9.2.1 Step 1: Feature Selection with the Variation Interface

The feature model of *ResourceManagement* at the top of Fig. 9–2 indicates that the concern supports the management of resources, the allocation of tasks, and optionally two ways of searching for resources. In addition, the feature model shows that the *ResourceManagement* concern itself reuses the *Association* concern four times. The developer of the *ResourceManagement* concern has already decided that only six variations out of all variations available in the *Association* concern are applicable in the context of *ResourceManagement*

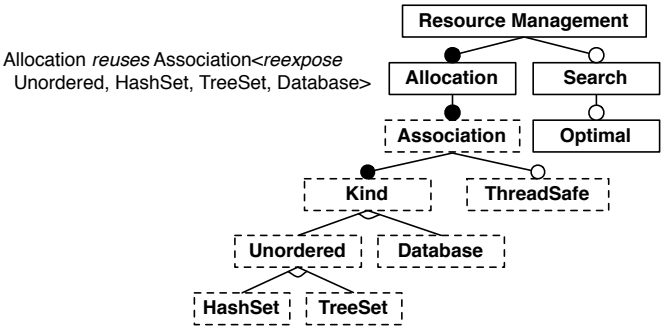

Figure 9–2: Feature Model (top) and Impact Model (bottom) of ResourceManagement Concern

(*Database*, the children of *KeyIndexed* (*HashMap* and *TreeMap*), and the children of *Ordered* (*ArrayList*, *LinkedList*, and *Stack*); the children are not shown in Fig. 9–2). Thus, the developer *reexposes* these features in the variation interface of *ResourceManagement.* The developer of the bCMS may now decide which one of the remaining variations is best for the specific context of the bCMS. In other words, the developer of *ResourceManagement* deferred some decisions about features of the *Association* concern to the developer of the bCMS, who has a better idea of what is suitable in the context of where *ResourceManagement* is actually reused, i.e., the bCMS.

Also part of the variation interface, the *impact model* at the bottom of Fig. 9–2 helps the developer of the bCMS make this decision, as it allows the developer to perform trade-off analyses for different feature selections. The impact model shows that *Database* impacts *Persistence* and each available feature of *Association* impacts *Performance.* In addition, *ResourceSearch* as well as *Optimal* also impact the *Performance* goal with *ResourceSearch* being the better option. Generally, the *Database* feature is better in terms of persistence, but worse in terms of performance, compared to other features of *Association.* The impact model

also shows that *ResourceSearch* and *ResourceAllocation* are more important than *Resource-Management* in terms of performance, because the former are used much more frequently than managing resources. Therefore, the contributions of their reused *Association* concerns to *Performance* are higher. Considering the impact model, the developer of the bCMS selects *ResourceSearch*, because a quicker response time is more important in a crisis management system than optimal resource use, and *Database*, because the status of existing resources and their allocations must be persisted to guard against loss of this vital information.

Each feature of a concern is modelled by one or more realization models (e.g., AoURN workflow models and/or RAM design models depending on the root phase of the concern). Once the developer has selected the desired features, a CORE tool then merges the models that realize the selected features to yield a user-tailored set of realization models of the concern corresponding to the desired feature selection (see Fig. 9–3 for the user-tailored workflow and design models of *ResourceManagement*). Fundamentally, a concern is described as generally as possible to increase reusability. Therefore, as previously pointed out, some elements in the concern are only *partially* specified and need to be related or complemented with concrete modelling elements of the application that intends to reuse the concern. The user-tailored set of realization models for *ResourceManagement* still contains these partially defined structural and behavioural elements, indicated by a vertical bar '|'.

The workflow models in Fig. 9–3 define the management of resources (*Feature: Re-sourceManagement*). Two model elements (|*Administrator* and |*Resource*) are only partially defined, because the developer of ResourceManagement cannot know the application-specific resources that will have to be managed when the concern is actually reused and who will be

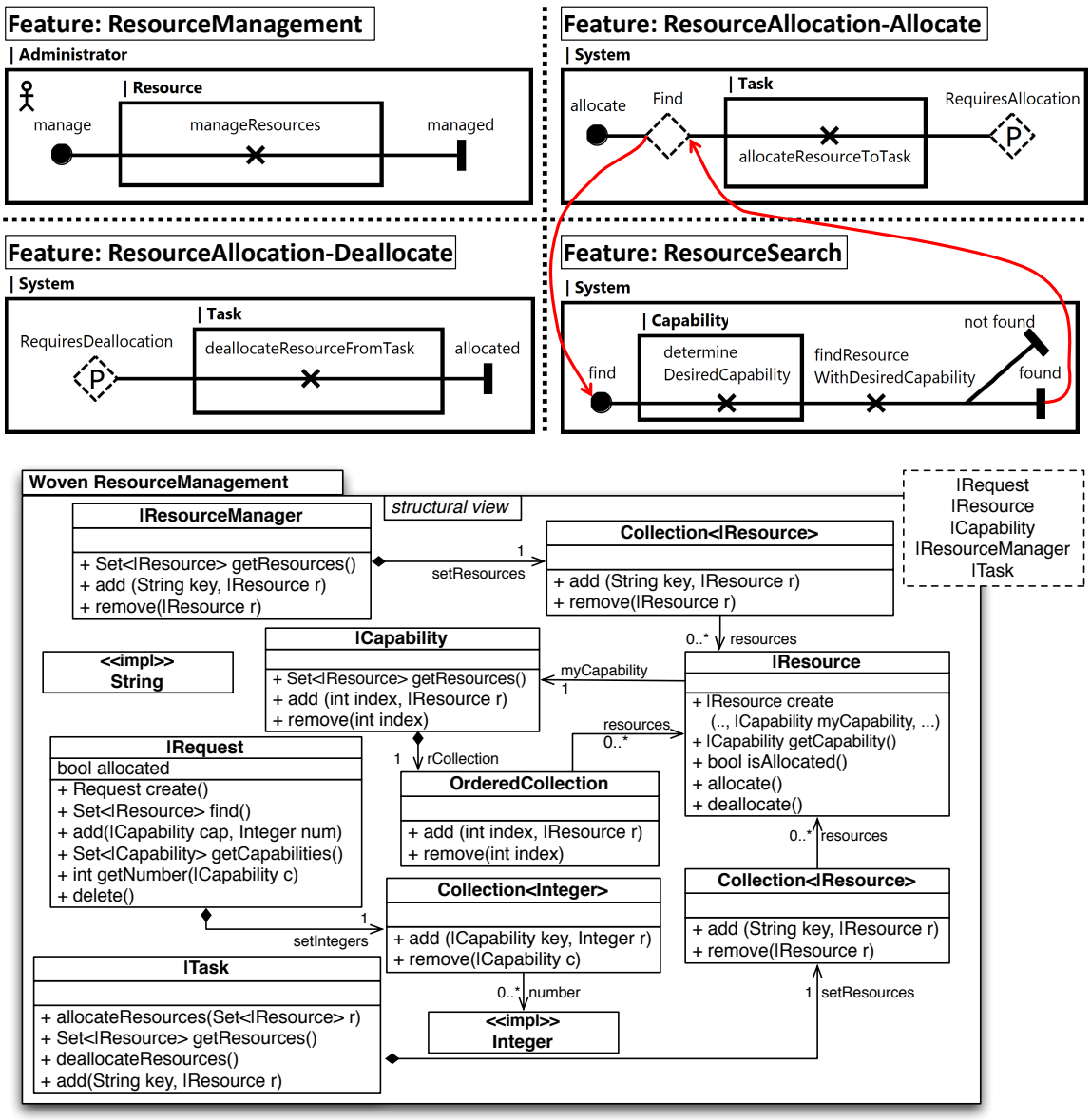

Figure 9–3: Workflow (top) and Design (bottom) Realization Models for User-Tailored ResourceManagement

managing them. In addition, the concern allows resources to be allocated/deallocated (*Feature: ResourceAllocation-Allocate/Deallocate*). These behaviours are to be merged with the application as indicated by the pointcut stub (the dashed, diamond-shaped symbol with a P). The pointcut stubs represent the locations in the application where allocation and deallocation are needed. Once these locations have been specified, the allocation and deallocation behaviour is added to the application before and after the specified location, respectively. This is visually indicated, because the allocation behaviour is shown before the *RequiresAllocation* pointcut stub and the deallocation behaviour is shown after the *RequiresDeallocation* pointcut stub. |*Task* is a partial element, because it is application-specific for what tasks resources are allocated/deallocated. Before resources can be allocated, they need to found. The stub *Find* (dashed, diamond-shaped symbol) in the allocation scenario links to other features in the ResourceManagement concern, i.e., only the *ResourceSearch* feature is connected to the stub in this case, because the *Optimal* feature was not selected and is hence not anymore connected to the stub in the user-tailored realization models of ResourceManagement.

Fig. 9–3 shows the structural diagram of the woven model generated for the user-tailored *ResourceManagement* concern, which contains all design realization models for the selected *ResourceManagement, ResourceAllocation*, and *ResourceSearch* features. |*ResourceManager* allows adding/removing resources (|*Resource),* while |*Task* allows allocating/deallocating them. |*Request* adds capabilities (|*Capability*) to be searched. The *find* operation in |*Request* finds all the resources in its list of capabilities that are currently not allocated. The *Collection* and *OrderedCollection* classes come from the *Association* concern, which was reused four times and for which the developer of the bCMS still needs to make decisions.

The *OrderedCollection* class will become more concrete after selecting one of the subfeatures of the *Ordered* feature (*ArrayList, LinkedList, Stack*) in *Association*. Similarly, *Collection</Resource>* and *Collection<Integer>* will be concretized upon selecting one of the subfeatures of *KeyIndexed (HashMap, TreeMap).* If the *Database* feature of *Association* is selected, the *Collection* classes will be used to cache data retrieved from the database and the methods of the *Collection* classes will be adapted by the *Database* feature to access the database when needed.

9.2.2 Step 2: Adapting the Reused Concern to the Application with the Customization Interface

In the second step of the concern reuse process, the developer of the *bCMS* adapts the generated, user-tailored but still partially defined realization models of the concern to the needs of the *bCMS*. This is done by mapping *customization interface* elements of *ResourceManagement* to model elements of the *bCMS*. A customization interface element is identified by the vertical bar '|', i.e., it is a partially defined element. The customization interface therefore describes how *ResourceManagement* may be adapted to the needs of a *bCMS* and is used when *ResourceManagement* is *composed* with *bCMS*. Consequently, the customization interface allows *ResourceManagement,* which is a generalized concern, to be specialized to the application under development, i.e., *bCMS*. The mapping of partially defined elements is done for each type of model and hence may involve requirements and/or design models.

For example in the workflow models, |*Administrator* is mapped to *Coordinator* in the bCMS and |*Resource* is mapped to *PoliceVehicle* and *FireTruck*. In addition, the locations represented by the pointcut stubs need to be specified for the workflow models. For example, the locations for *RequiresAllocation* are specified as "*proposeRouteForPoliceVehicles*" and "*proposeRouteForFireTrucks*". Consequently, the allocation behaviour is added

before these two locations in the bCMS workflow. Partial elements of design models are also mapped similarly, from Fig. 9–3, |*ResourceManager* is mapped to *VehicleManager* in *bCMS*, |*Capability* to *VehicleKind*, |*Task* to *Crisis*, and |*Request* to the concrete class Request in the *bCMS*.

The result of the second step is a set of realization models that are customized to the specific application context of the *bCMS* and that are ready to be used in the *bCMS*. The resulting models are essentially those shown in Fig. 9–3, except that the partial elements are replaced by concrete elements from the bCMS as defined by the mappings. Note that this additive approach guarantees compositional correctness and consistency of the user-tailored and customized set of design models by construction [62]. The same additive approach applies to the customization of workflow models. The tailoring of workflow models, however, uses the built-in hierarchical structuring mechanism of AoURN [56] to guarantee compositional correctness and consistency by construction.

9.2.3 Step 3: Using the Reused Concern in the Application through the Usage Interface

The composition enabled by the customization interface integrates *ResourceManagement* into *bCMS*. As a result, the *usage interface* describes how *bCMS* can finally access *ResourceManagement*'s structure/behaviour/other properties. Concretely, the usage interface further connects *bCMS* and *ResourceManagement*, e.g., through method calls from an application-specific *bCMS* class to a class provided by the *ResourceManagement* concern or by incorporating a workflow of the *ResourceManagement* concern into the *bCMS*' workflow. While the variation interface is always expressed with feature and impact models and the fundamental concept of the customization interface to identify partial elements is the same for all models in a concern, the exact nature of the usage interface depends on the modelling notation.

For example, the usage interface of the design model of a concern is typically comprised of the concern's *public* classes and methods (e.g. *allocateResources*(...), *allocate*(), and *deallocate()*). For workflow models, on the other hand, the usage interface consists of the *starting points* of the top-level workflows that are made available to the developer reusing the concern (e.g., the start point *manage* in *ResourceManagement*).

9.3 Lessons Learnt

This section first presents metrics to quantify the achieved model reuse, and then describes our experience in applying CORE to a large system.

9.3.1 Substantial and Scalable Model Reuse

This case study is applied to the design of an application of considerable size with real-world requirements. We were able to reuse a substantial number of pre-existing concerns within our bCMS requirements and design models. Some of these concerns themselves reuse other concerns, which creates a concern hierarchy with complex concern dependencies. The maximum depth of the concern hierarchy for our *bCMS* design is 4. Table 9–1 shows reuse metrics collected during the case study. The case study involves 102 reuses of 12 concerns. The reused concerns contain a total number of 61 features that are currently realized by 43 different design models and 16 different requirements models. The number of feature and realization models for a concern may not be identical. Some features may not be realized by any model, and there may exist models that realize more than one feature for resolving feature interactions.

Reuse of previously developed concerns suggests the effectiveness of CORE. All concerns apart from *Encryption* have been developed when building other applications. Some concerns were reused multiple times, possibly with more than one configuration resulting in a total of

158 feature reuses. The 158 feature reuses involve 32 unique reused features (i.e., features that are reused at least once). The most reused concern is the *Association* concern with 38 reuses. The *Named* concern allows to set and modify names and is reused 32 times. The largest concerns in terms of features and realization models are *Association* and *Workflow* (12 features, 11 realization models each).

The concern hierarchy shows that a concern may appear at any level of abstraction. The first number after the concern name in Table 9–1 represents the reuse *depth*, i.e., a concern of depth 1 does not have any reuses, and a concern of depth 2 reuses one or multiple concerns of depth 1. The second number or range explains at which level in the *bCMS* hierarchy the concern is reused. For example, *Association* (1)(1-3) means that *Association* does not reuse any other concerns, and that in the *bCMS* it is reused at *levels* 1, 2, and 3 (the *bCMS* concern at the top of the hierarchy is level 1). Both numbers serve as an indicator of the level of abstraction of a concern. The range additionally serves as an indicator of the concern's genericity. Concerns with a wide reuse range (reused at a wide range of levels) are applicable at many levels of abstraction, which is the case for generic concerns such as *Singleton*, *Association*, *Networking*, and *Named*.

9.3.2 Software Product Line Comes for Free

Reexposing features of lower-level concerns at the *bCMS* level automatically creates a *bCMS* product line, since a developer can then configure *bCMS* according to her needs by selecting among the reexposed features. For example, the developer can choose authentication methods by selecting the reexposed features from *Authentication*. In addition, the composition of impact models allows the impacts of the reexposed features to be included

Reused Concern (Depth)(Reuse Range)	Features	Realizations (AoURN / RAM)	Reuses	Unique Configurations	Reused F. (Unique)	Reexposed F. (Unique)
Association (1)(1-3)	12	0 / 11	38	8	76 (4)	35 (11)
Named (1)(1-3)	2	0 / 2	32	1	32 (1)	32(1)
Singleton (1)(1-3)	1	0 / 1	11	1	9 (1)	–
Copyable (1)(2)	3	0 / 2	3	2	6 (3)	–
Networking (1)(1-3)	6	0 / 1*	7	1	7 (1)	–
Encryption (1)(1)	5	0 / 0*	1	1	1 (1)	4 (4)
KeyCounter (2)(2)	1	0 / 1	2	1	2 (1)	–
Command (2)(2)	2	0 / 2	3	2	4 (2)	–
Authentication (2)(2)	10	8 / 6*	1	1	2 (2)	8 (8)
Res. Mgmt (2)(1)	4	5 / 4	1	1	3 (3)	1 (1)
Authorization (3)(1)	3	3 / 2*	1	2	1(1)	1(1)
Workflow (3)(1)	12	0 / 11	2	2	15 (12)	–
Total	61	16 / 43	102	23	158 (32)	81 (26)

Table 9–1: Reuse Metrics of the *bCMS* Case Study. Realizations with * indicate that some realization models for the concern are still under construction.

in the trade-off analysis when configuring *bCMS*. In total, there are 81 reexposed features in the bCMS (26 of them are unique).

9.3.3 Delaying Decision of Feature Selections

With CORE, it is possible to delay decisions that do not need to be taken until they actually have to be taken. For example, the *bCMS* requirements document describes *Authorization* rather vaguely. CORE allows the *Authorization* concern to be added to the system, but without deciding on its exact features. The features are simply reexposed, adding new variation points to *bCMS* that can be finalized when more information is available. Delaying the decision of selecting features may also be beneficial when the system needs to change feature selections based on evolving environmental conditions. A reevaluation of the impact model may trigger a change in the current feature selection, allowing systems to be more adaptive. Furthermore, modellers have to make many decisions when building complex

systems, and some of them may be forced upon them because the project needs to move forward. With CORE, it is possible to move a project forward, i.e., concerns are chosen and integrated into the system, while still retaining the freedom to choose the most appropriate feature at a later point.

9.3.4 Iterative Decision Support

The features and trade-off analysis provided by a concern expose the modeller and other stakeholders to concern variations that embody important domain knowledge, which may help uncover missing requirements by making the modeller aware of possible solutions. Requirements and associated solutions may be explored more iteratively. The types of models provided by a concern may also influence the software development process. If a particular type of analysis model (e.g., for performance analysis) exists in the concern, it may be incorporated into the development process for further decision support.

9.3.5 Dealing With Crosscutting Concerns in Concern Hierarchies

The *Authentication* concern in *bCMS* is needed in various parts of the system (e.g., vehicle management and mission execution), making this a crosscutting concern. With CORE, it is possible to integrate the *Authentication* concern once into *bCMS* and then selectively apply it to where it is needed in the system with advanced techniques for Separation of Concerns. The *Authentication* concern was reused in the root feature in Fig. 9–1, hence, we are able to apply it anywhere in *bCMS* through extending the realization model of the root feature as explained in Chapter 8. Without such support, the *Authentication* concern would have to be added to each location individually, leading to duplication of effort and evolution issues.

9.3.6 Notations at the Right Abstraction Level

With workflow and design notations describing a concern, CORE supports the composition of concerns at the most appropriate abstraction level. Compositions that are purely behavioural and hence relate to the organization of activities in a workflow are best performed with AoURN at the workflow level, whereas data-centric compositions are best performed using structural diagrams at the RAM design level. Furthermore, in cases where both requirements models and design models of a concern need to be composed with the system, the composition locations identified during the requirements phase often help in locating the composition locations for the design phase.

9.3.7 Tool Support is Essential

We used two CORE-based tools, jUCMNav and TouchCORE. In addition to providing model-editing capabilities, these tools expedite the reuse process by automating impact analysis for feature selections, automated composition of feature models when features are reexposed within concern hierarchies, as well as composition and subsequent customization of realization models based on feature selections.

9.3.8 Incomplete Concerns in the Reusable Concern Library

The requirements of the bCMS mention that several communication protocols should be supported, including HTTP, SSL, and SOAP. The existing reusable *Networking* concern provides inter-process communication, but unfortunately only over TCP/IP sockets. While the reuse of a concern is quite streamlined, building or extending a concern requires significantly more effort than directly adding support for another communication protocol into bCMS. We envision software companies or open-source communities in support of concern-orientation, actively requiring the deep understanding of *all* features within a concern to

extract the common properties, structure, and behaviour, deal with feature interactions, as well as crystallize common customization and usage interfaces.

9.4 Limitations

The variation interface makes it easy to reuse an existing concern by capturing its variations and impacts. However, reexposed features and goals from lower-level concerns that are added to a variation interface pose several challenges to the modeller.

First, the names of the reexposed features and goals may not be relevant in the context of the reusing concern. For example, the names of the reexposed features of *Association* are not intuitive in the context of *ResourceManagement*. The *Ordered* reexposed feature should be renamed to be *OrderedSearch*. Manual renaming can become challenging when the number of reexposed feature is large.

Second, the reexposed features require additional decision-making effort by the modeller. This could be mitigated by allowing the tool to automatically select among the reexposed features using optimization and constraint solving algorithms, or by allowing the concern designers to specify default configurations that can be automatically selected in case the reuser does not want to make explicit selections. We are exploring automatic selections of reexposed features in future work.

Third, if the philosophy of delaying decisions as much as possible is followed by all concern developers, the number of features of high-level concerns may grow significantly. This might cause serious scalability issues for CORE tools, both from an algorithmic (e.g., model composition run-time) and resource consumption (e.g., memory use) point of view.

Furthermore, the impact model in CORE captures the impacts of the reused concern independently of the reuse context. The context of the reusing concern, however, may

influence impacts. For example, when application-specific data is sent over the network, the size of the data that is sent has a huge impact on performance and number of messages transmitted. In future work, we are exploring how to support parametrization of CORE impact models by reuse context.

9.5 Conclusion

This chapter records our experience in modelling a large family of crisis management systems using CORE. Without loss of generality, we model the requirements of this family of systems with the Aspect-Oriented User Requirement Notation (AoURN) and the design with Reusable Aspect Models (RAM). Some concerns have both AoURN and RAM realization models, conforming with the vision of CORE of using the most appropriate modelling notation to express system properties at a given level of abstraction. CORE advocates the creation of reusable concerns that are combined with each other in a reuse hierarchy. Twelve concerns were reused in building this family of systems with a total number of 102 reuses in a reuse hierarchy spanning four levels. This substantial number of reuses is possible because of CORE's streamlined three-step reuse process.

Based on collected metrics and our experience in the design of this system, we observe that there is substantial and scalable reuse of concerns in the crisis management system. A key insight gained is that CORE allows for decisions not to be taken prematurely with the help of CORE's reexposing mechanism. A concern can be added to the system without deciding on the exact features, leaving this decision to a later point when more information is available about which feature's impact on system qualities is most desirable. The study indicates the feasibility of CORE's vision to create large-scale, generic, reusable entities that are expressed with the most appropriate modelling formalisms at the right level of abstraction.

High-level concerns that are reused during requirement modelling such as Authentication can be composed further during the design phase by applying their design realization models in the locations initially identified during requirement modelling. We note that tool support was essential during the design of this family of systems.

Chapter 10
Conclusions and Future Work

10.1 Summary

Model reuse is a major challenge in Model-Driven Engineering (MDE). This thesis proposes Concern-Oriented Reuse (CORE), a novel reuse paradigm that uses ideas from MDE, Software Product Lines, goal modelling and impact analysis, and advanced Separation of Concerns to support broad-based model reuse. CORE introduces the *concern*, a new unit of reuse that groups related models spanning all software development phases and relevant levels of abstraction together. Some concerns appear in early phases of software development, e.g., broadly scoped system properties with functional, non-functional, or even intentional characteristics. Other concerns appear in later phases of software development, e.g., solution-specific concerns such as specific communication protocols, concrete authentication algorithms, and design patterns. The concern provides a three-part interface to facilitate reuse. The variation interface expresses the variability that the concern offers and the impact the different variants have on non-functional requirements and system qualities, thus enabling systematic trade-off analysis. The customization interface exposes the generic and partial elements within the concern that have to be adapted to the reuse context. The usage interface highlights the structure and behaviour that is made available by the concern to the user.

After introducing the CORE concepts and definitions in the beginning of this thesis, we perform an extensive literature comparison to demonstrate the strengths of concern as a unit

of reuse that tackles the challenges facing other reuse units. Then, we show how a concern is constructed by modelling an example *Authentication* concern at both requirement and design levels. We also discuss the concern reuse process, and how a concern modeller can defer decisions by selecting only the minimally required features from the variation interface of the concerns that she reuses and by reexposing any alternative and optional features in her own variation interface, thus creating a *Software Concern Line (SCL)*. In addition, we introduce algorithms that allow concern interfaces and the realization models to be composed, allowing SCL to be operational. To allow modelling languages that want to adapt large-scale model reuse to become concern-oriented, we define the CORE metamodel that specifies all the CORE concepts. We illustrate the corification of modelling languages by showing how two modelling languages, Aspect-oriented Use Case Maps (AoUCM) and Reusable Aspect Models (RAM), successfully integrated the CORE concepts by extending the CORE metamodel. Using the concepts presented in the metamodel, we show how to incrementally model design concerns by adding model increments and reusing existing concerns, illustrated by an example *Workflow* concern. Incremental modelling is further used to develop other concerns in our growing reusable concern library. The library is part of the TouchCORE tool, which along with another tool, jUCMnav, support modelling with the corified versions of the RAM and AoUCM modelling languages. The reusable concerns in the library are used to build a family of crisis management system named *bCMS.*

Our vision is that if CORE is successfully adopted on a large scale, it will transform the software engineering discipline as a whole. While current practises often require software engineers to deal with and be an expert in many concerns simultaneously within each software development phase, CORE would enable software engineers to specialize, i.e., to become

concern specialists. In companies selling concern libraries, security concern specialists, e.g., would solely concentrate on maintaining and evolving the models within the security concern, i.e., adding new security requirements, solutions, techniques, and platforms as they become relevant. Within a company developing applications, a security concern expert would focus on composing the security concern with the other application concerns. Ultimately, concern libraries, concern reuse, and concern specialization would provide a clear structure to software development, and as a result align the practice of software engineering closer to what is done in other engineering disciplines.

To recap, this thesis lays the foundations of CORE, by defining the concepts of concern orientation and introducing a metamodel that allows different modelling languages to be corified (covered by Chapter 2 and Chapter 6). It then introduces the concern reuse process, CORE composition algorithms, and incremental design modelling of concerns (covered by Chapter 4, Chapter 5 and Chapter 8). Finally, it validates CORE by performing extensive literature comparison between concern and other units of reuse, successfully corifying two modelling languages, building a reusable concern library, tool support (both AoUCM and RAM are supported by modelling tools), and conducting a case study of a family of crisis management systems (covered by Chapter 3, Chapter 7, Chapter 8, and Chapter 9).

10.2 Future Work

The foundational work on CORE done in this thesis gives rise to a plethora of possible future work. In this section, we outline some areas that are in our immediate interest, broadly organized into two research directions. The first one groups ideas that aim at defining a concern-driven development methodology and process, and the second one applies CORE to other domains.

10.2.1 Concern-Driven Development Methodology

This thesis has laid the foundation for concern-oriented reuse. We discussed in detail the three-step reuse process that allows building concerns or applications by reusing existing concerns. Our focus, however, throughout this thesis has been on the design phase of software development. The modelling notations supported by our CORE reference implementation are mainly notations for specifying software designs, and we presented an incremental modelling process for the design of the *Workflow* concern. To achieve CORE's vision of broad-scale model reuse, additional modelling notations need to be corified to cover requirements, analysis, architecture and implementation models. On top of that, there is a need to define a concern-driven development (CDD) methodology, that integrates the ideas of model-driven engineering with concern-oriented reuse. In particular, this requires recognizing features during the requirement elicitation phase, identifying concerns that can be reused during all development phases, and defining consistency rules between customizations of models across different levels of abstraction / phases.

To move in this direction, we plan to carry out the following research activities:

Incorporating Model Transformation Techniques into CORE

Similar to what is done within the context of MDE, CDD relies on model transformations to automate software development as much as possible. Within a concern, model transformations would link models across different levels of abstraction. We will endeavour to transform compositions at the requirements level into skeleton compositions at the design level, further streamlining and automating the reuse process across abstraction levels. The modeller may still have to perform some manual tasks, however, our vision is that the majority of design and execution models are automatically generated. We also plan to integrate testing into

CDD. Testing should focus on testing of features, configurations, and entire concerns in isolation, but also to establish the correctness of a reuse, the correctness of concern hierarchies, and finally the correctness of the application..

Enhancing Concern Interfaces

To further assist the concern user to make advanced feature selection and trade-off analysis across development phases, we plan to enhance the variation interface with additional capabilities. In our feature models, we plan to add the strengths of other types of feature models, such as cardinality-based feature models [100], and extended feature models [12, 21, 24].

Cardinality-based feature models allow for features and feature groups to be selected more than once and to specify additional constraints. A child feature can specify how many times it can be contained in its parent feature. This information is encoded with a [n..m] interval, where n denotes the lower bound and m denotes the upper-bound. Adding properties of cardinality-based feature models to CORE will allow the modeller to encode relevant information early early on during requirement modelling. For example, the modeller can specify that an application server must be selected no more than five times.

Extended feature models, e.g., [24] allow adding attributes to features and to express relations among feature attributes. They then map the extended feature models onto a Constraint Satisfaction Problem (CSP) that allows some automated reasoning. Users can ask questions to the CSP solvers such as the total number of products of a feature model and define some filters on the model. Other extended feature models, such as FAMILIAR [12] , allow for advanced cross-tree constraints, which can be used in the context of CORE to support sophisticated feature configurations.

Furthermore, as impact models allow the concern user to perform extensive trade-off analysis of different feature configurations, we will explore a more proactive approach that suggests the most appropriate selection instead of the interactive, explorative approach currently supported. Additionally, we would like to enhance impact models with other concepts from goal modelling such as stakeholders. A concern can have more than one stakeholder, and including them will allow for more sophisticated impact analysis. For example, an *Encryption* concern can be important for the *Border Control Agency* and the *User* stakeholders. *Encryption's* feature *ePassport,* a kind of travel passport that contains electronic data of the passport bearer in a chip, impacts two high level goals*, Security* and *Privacy.* The stakeholders can have different interests; *Security* can be more important to the *Border Control Agency* while *Privacy* can be more important to the *User.* In addition, we are planning to include other types of models to impact models such as Analytic Hierarchy Process (AHP) [103] and performance models [128]. AHP allows for a goal to be decomposed in subgoals and alternative solutions, and integrates both pairwaise comparisons and the importance of subgoals into the decision making. Performance models allow for advanced analysis of the system performance goal.

Finally, we would like to investigate how Domain Specific Languages (DSL) can be used to provide simpler interfaces for concerns than the variation, customization, and the usage interfaces presented in this thesis. In [108], we explored how the three-part concern interface of the *Association* concern can be used to define a DSL that in turn can be used in software design models to establish associations between classes. We further plan to exploit the impact model of a concern when defining a DSL for the concern. For example, in a DSL for

the *Association* concern, a user should be able to specify that she wants a high performant data structure when making feature selection.

Adaptive Systems

In future, we would like to exploit the variability provided in CORE concerns for the development of adaptive systems. Approaches such as Dynamic Software Product Lines (DSPL) [54] allow for dynamic configurations of SPL at runtime. Similarly, concerns can adapt to changing requirements or to changes in the environment at run time by reconfiguring its features. Reconfiguration can benefit from feedback received from re-evaluation of impact models at run time. Dynamic adaptation will benefit a growing number of systems that have to reconfigure at run time, such as cyber-physical systems.

10.2.2 Applying CORE to Other Domains

We plan to apply the concepts developed in CORE to other domains, such as Software Language Engineering and Systems Engineering. We further plan to conduct empirical studies to improve our CORE-based tools. Here are the concrete steps that we plan to undertake in this direction:

Concern-Driven Security

Model-Driven Security (MDS) [22] is a special type of MDE that focuses on modelling secure systems. However, there are challenges for applying MDS in practice. A recent study has shown that MDS catalogs systematically neglect multiple security patterns [86]. Furthermore, empirical studies have shown that using existing MDS catalogs of security concerns does not improve the productivity of developers [129], or the security of systems. Since our experience has shown that concerns are easy to reuse and group models in a modular way, we believe CORE will help overcome the challenges of MDS. Therefore, we

plan to introduce Concern-Driven Security (CDS) for secure systems, that applies the CORE concepts on MDS. We believe that CDS will allow for incremental growth of security patterns in one place, reducing the chances for missing patterns. In addition, we believe that our three-step reuse process will increase the applicability of security concerns and the productivity of the developers. We started to explore CDS in collaboration with a research group from Université du Luxembourg, initial results are being prepared to be published in the Journal of Software and System Modelling (SoSym).

Modular Design of Heterogeneous Modelling Languages

We plan to use the concepts of CORE interfaces to facilitate the communication between heterogeneous modelling languages. In cyber-physical systems, different modelling formalisms need to interact with each other. Certain laws of physics are modelled using continuous-time formalisms, while responding to events is represented using discrete-time formalisms. A hybrid formalism [69] allows for continuous-time and discrete-time modelling to be in one unit by combining discrete model elements with continuous model elements. For example, a hybrid formalism that combines Discrete Event-Scheduling (DES) with Ordinary Differential Equations (ODEs) can be used to model a bouncing ball, where a free-falling ball bounces on the ground. Models of more complex cyber-physical systems such as the ones used in the automobile or aerospace industry can be similarly modeled using hybrid formalisms. For such hybrid formalism to work, dedicated simulators are required that support syntactic and semantic adaptation of the involved model elements belonging to different formalisms.

Currently, the syntactic and semantic adaptations of modelling formalisms involve manually encoded adaptation and synchronization, which is very inefficient. We aim at solving this

problem through using composable units that provide definitions for modelling formalisms called language definition fragments. Each fragment models the specifications of abstract syntax, concrete syntax, semantics, and user interface behaviour. Fragments of different heterogeneous languages are composed by linking their respective specifications. We plan to use CORE-based modelling languages to model these fragments, and exploit the power of customization and usage interfaces to link fragment specifications. The user interface behaviour can be modelled using behavioural modelling formalisms such as state machines. Similarly, modelling of the semantics can be done using other suitable formalisms such as rule-based action model transformation languages, state machines, and action language. After modelling the components, composition of different fragments is specified to produce a hybrid formalism. The same process is then used to experiment with other fragments, and the experience gained will be used to develop a general technique and process for the design through composition of fragments to produce hybrid formalisms. This work is a subject of an ongoing collaboration with a group from University of Antwerp.

Empirical Studies

We will use controlled user experiments to further improve our CORE-based tools. Users can be assigned to model sample case studies using older versions of the TouchCORE or jUCMNav tools before corification, and using the corified versions of the tools. Such user studies can be used to collect metrics such as the number of reused models, time-to-finish modelling the systems, etc. Comparisons between the collected metrics will help assess the productivity of modellers, reusability, and quality of models of the corified tools compared to their older versions before corification. In addition, empirical studies can be used to compare the productivity of Concern-Driven Security (CDS) mentioned in the previous subsection,

compared to traditional Model-Driven Security. This comparison study is a subject of proposed collaboration with research groups from Katholieke Universiteit Leuven and Université du Luxembourg.

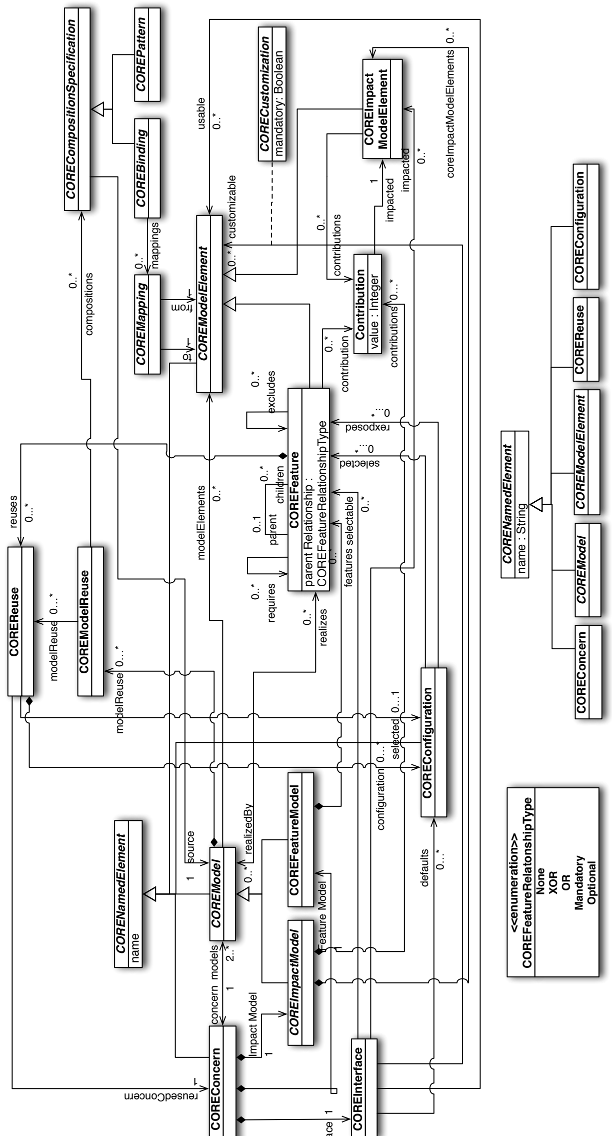

Figure 10–1: The Complete CORE Metamodel.

Bibliography

[1] Amazon: Amazon web services.

[2] AspectJ website: https://eclipse.org/aspectj/.

[3] CaesarJ Homepage. http://caesarj.org/.

[4] CMA series. http://cserg0.site.uottawa.ca/cma2013models/approaches.htm.

[5] JUnit website.

[6] Swing Framework. http://java-source.net/open-source/swing .

[7] *Transactions on Aspect-Oriented Development (TAOSD VII), Special Issue on a Common Case Study for Aspect-Oriented Modeling*, volume 6210 of *LNCS*. Springer, 2010.

[8] *European Space Agency, Ariane 5, Flight 501 Failure. 1996.*, 2013.

[9] Models of bCMS and its reused concerns. URL: http://www.ece.mcgill.ca/ gmussb1/bCMS/ for all bCMS models., 2015.

[10] S. Abiteboul, B. Amann, J. Baumgarten, O. Benjelloun, F. Dang Ngoc, and T. Milo. Schema-driven customization of web services. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 1093–1096. VLDB Endowment, 2003.

[11] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Composing feature models. In Mark Brand, Dragan Gasevic, and Jeff Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 62–81. Springer Berlin Heidelberg, 2010.

[12] 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.

[13] Wisam Al Abed, Valentin Bonnet, Matthias Schöttle, Omar Alam, and Jörg Kienzle. TouchRAM: A multitouch-enabled tool for aspect-oriented software design. In *5th*

International Conference on Software Language Engineering - SLE 2012, number 7745 in LNCS, pages 275 – 285. Springer, October 2012.

[14] Omar Alam and Jörg Kienzle. Designing with inheritance and composition. In *3rd International Workshop on Variability and Composition*, pages 19–24, New York, NY, USA, 2012. ACM.

[15] Omar Alam and Jörg Kienzle. Incremental software design modelling. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 325–339, Riverton, NJ, USA, 2013. IBM Corp.

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

[17] Omar Alam, Matthias Schöttle, and Jörg Kienzle. Revising the comparison criteria for composition. In *Proceedings of the Fourth International Comparing Modeling Approaches Workshop 2013 co-located with the ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, Florida, USA.*, 2013.

[18] Daniel Amyot, Sepideh Ghanavati, Jennifer Horkoff, Gunter Mussbacher, Liam Peyton, and Eric S. K. Yu. Evaluating goal models within the goal-oriented requirement language. *International Journal of Intelligent Systems*, 25(8):841–877, 2010.

[19] Sandra Antonio, Joao Araújo, and Carla Silva. Adapting the i* framework for software product lines. In CarlosAlberto Heuser and GÃŒnther Pernul, editors, *Advances in Conceptual Modeling - Challenging Perspectives*, volume 5833 of *Lecture Notes in Computer Science*, pages 286–295. Springer Berlin Heidelberg, 2009.

[20] Mohsen Asadi, Samaneh Soltani, Dragan Gasevic, Marek Hatala, and Ebrahim Bagheri. Toward automated feature model configuration with optimizing non-functional requirements. *Information and Software Technology*, 56(9):1144 – 1165, 2014. Special Sections from ÒAsia-Pacific Software Engineering Conference (APSEC), 2012Ó and Ò Software Product Line conference (SPLC), 2012Ó.

[21] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. Feature and meta-models in clafer: Mixed, specialized, and coupled. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, pages 102–122, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From uml models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, January 2006.

[23] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, September 2010.

[24] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE'05, pages 491–503, Berlin, Heidelberg, 2005. Springer-Verlag.

[25] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Trans. Program. Lang. Syst.*, 21(4):813–847, 1999.

[26] Marko Boskovic, Gunter Mussbacher, Ebrahim Bagheri, Daniel Amyot, Dragan Gasevic, and Marek Hatala. Aspect-oriented feature models. In Juergen Dingel and Arnor Solberg, editors, *Models in Software Engineering*, volume 6627 of *Lecture Notes in Computer Science*, pages 110–124. Springer Berlin Heidelberg, 2011.

[27] Alfredo Capozucca, Betty H.C. Cheng, Geri Georg, Nicolas Guelfi, Paul Istoan, and Gunter Mussbacher. Requirements definition document for a software product line of car crash management systems. URL: http://cserg0.site.uottawa.ca/cma2011, 2011.

[28] Andrew Carton, Cormac Driver, Andrew Jackson, and Siobhan Clarke. Model-driven theme/uml. In Shmuel Katz, Harold Ossher, Robert France, and Jean-Marc Jezequel, editors, *Transactions on Aspect-Oriented Software Development VI*, volume 5560 of *Lecture Notes in Computer Science*, pages 238–266. Springer Berlin / Heidelberg, 2009.

[29] Lianping Chen and Muhammad Ali Babar. A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology*, 53(4):344–362, April 2011.

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

[31] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 13–22, New York, NY, USA, 2010. ACM.

[32] Siobhán Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison Wesley, 2005.

[33] Benoit Combemale, Olivier Barais, Omar Alam, and Jörg Kienzle. Using cvl to operationalize product line development with reusable aspect models. In *Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone*, pages 9–14. ACM, 2012.

[34] Javier Cubo and Ernesto Pimentel. Damasco: A framework for the automatic composition of component-based and service-oriented architectures. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 388–404. Springer Berlin Heidelberg, 2011.

[35] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

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

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

[38] Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini. A dynamic software product line approach using aspect models at runtime. In *Fifth Domain-Specific Aspect Languages Workshop*, volume 39, page 40, 2010.

[39] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39:41–47, 2006.

[40] Robert Dyer and Hridesh Rajan. Supporting dynamic aspect-oriented features. *ACM Trans. Softw. Eng. Methodol.*, 20(2):7:1–7:34, September 2010.

[41] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[42] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 261–270, New York, NY, USA, 2008. ACM.

[43] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, January 2004.

[44] 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.

[45] Lidia Fuentes, Nadia Gamez, and Pablo Sanchez. Aspect-oriented design and implementation of context-aware pervasive applications. *ISSE*, 5(1):79–93, 2009.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, USA, 1995.

[47] Bart George, Régis Fleurquin, and Salah Sadou. A component selection framework for cots libraries. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, pages 286–301, Berlin, Heidelberg, 2008. Springer-Verlag.

[48] Hassan Gomaa. *Designing software product lines with UML - from use cases to pattern-based software architectures*. ACM, 2005.

[49] Hassan Gomaa and Koji Hashimoto. Dynamic software adaptation for service-oriented product lines. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 35:1–35:8, New York, NY, USA, 2011. ACM.

[50] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, MA, 2005.

[51] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[52] Jilles Van Gurp and Christian Prehofer. From SPLs to open, compositional platforms. In *Combining the Advantages of Product Lines and Open Source*, Dagstuhl Seminar Proceedings. Schloss Dagstuhl, 2008.

[53] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented architectural variability using monticore. In *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*, ECSA '11, pages 6:1–6:10, New York, NY, USA, 2011. ACM.

[54] S. Hallsteinsen, M. Hinchey, Sooyong Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, April 2008.

[55] Aram Hovsepyan, Stefan Van Baelen, Yolande Berbers, and Wouter Joosen. Generic reusable concern compositions. pages 231–245. 2008.

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

[57] Ivar Jacobson. Use cases and aspects-working seamlessly together. *Journal of Object Technology*, 2(4):7–28, 2003.

[58] 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.

[59] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *OOPSLA '12*, pages 773–792. ACM, 2012.

[60] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es), December 1996.

[61] Jörg Kienzle, Wisam Al Abed, Franck Fleurey, Jean-Marc Jézéquel, and Jacques Klein. Aspect-oriented design with reusable aspect models. In Shmuel Katz, Mira Mezini, and Jörg Kienzle, editors, *Transactions on Aspect-Oriented Software Development VII*, volume 6210 of *Lecture Notes in Computer Science*, pages 272–320. Springer Berlin Heidelberg, 2010.

[62] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-Oriented Multi-View Modeling. In *AOSD 2009*, pages 87 – 98. ACM Press, March 2009.

[63] Jörg Kienzle, Nicolas Guelfi, and Sadaf Mustafiz. Crisis Management Systems: A Case Study for Aspect-Oriented Modeling. *Transactions on Aspect-Oriented Software Development*, 7:1 – 22, 2010.

[64] Jacques Klein, Franck Fleurey, and Jean Marc Jézéquel. Weaving multiple aspects in sequence diagrams. *TAOSD*, LNCS 4620:167–199, 2007.

[65] Jacques Klein, Loic Hélouet, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In *AOSD*, pages 27–38. ACM Press, 2006.

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

[67] Krueger. Software reuse. *CSURV: Computing Surveys*, 24, 1992.

[68] R. Krut and S. Cohen. Service-oriented architectures and software product lines - putting both together. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 383–383, Sept 2008.

[69] S. Lacoste-Julien, H. Vangheluwe, J. de Lara, and P.J. Mosterman. Meta-modelling hybrid formalisms. In *Computer Aided Control Systems Design, 2004 IEEE International Symposium on*, pages 65–70, 2004.

[70] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, 3nd edition, 2002.

[71] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709–724, Oct 2007.

[72] S. Liaskos, S.A. McIlraith, S. Sohrabi, and J. Mylopoulos. Integrating preferences into goal models for requirements engineering. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 135–144, Sept 2010.

[73] Karl Lieberherr, David H. Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA 02115, March 1999.

[74] Yanji Liu, Yukun Su, Xinshang Yin, and G. Mussbacher. IEEE 4th international model-driven requirements engineering workshop, modre 2014, 25 august, 2014, karlskrona, sweden. IEEE, 2014.

[75] Silvia Mazzini, John Favaro, and Tullio Vardanega. Cross-domain reuse: Lessons learned in a multi-project trajectory. In John Favaro and Maurizio Morisio, editors, *Safe and Secure Software Reuse*, volume 7925 of *Lecture Notes in Computer Science*, pages 113–126. Springer Berlin Heidelberg, 2013.

[76] P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B.H.C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, July 2004.

[77] G.A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[78] Ana Moreira, João Araújo, and Awais Rashid. A concern-oriented requirements engineering model. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE'05, pages 293–308, Berlin, Heidelberg, 2005. Springer-Verlag.

[79] Ana Moreira, Ruzanna Chitchyan, João Araújo, and Awais Rashid, editors. *Aspect-Oriented Requirements Engineering*. Springer Berlin Heidelberg, 2013.

[80] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, October 2009.

[81] Gunter Mussbacher, Omar Alam, Mohammad Alhaj, Shaukat Ali, Nuno Amálio, Balbir Barn, Rolv Bræk, Tony Clark, Benoit Combemale, Luiz Marcio Cysneiros, Urooj Fatima, Robert France, Geri Georg, Jennifer Horkoff, Jörg Kienzle, Julio Cesar Leite, Timothy C. Lethbridge, Markus Luckey, Ana Moreira, Felix Mutz, A. Padua A. Oliveira, Dorina C. Petriu, Matthias Schöttle, Lucy Troup, and Vera M. B. Werneck. Assessing composition in modeling approaches. In *Proceedings of the CMA 2012 Workshop*, pages 1–26, New York, NY, USA, 2012. ACM.

[82] Gunter Mussbacher, Daniel Amyot, João Araújo, and Ana Moreira. Requirements modeling with the aspect-oriented user requirements notation (aourn): A case study. In Shmuel Katz, Mira Mezini, and Jörg Kienzle, editors, *Transactions on Aspect-Oriented Software Development VII*, volume 6210 of *Lecture Notes in Computer Science*, pages 23–68. Springer Berlin Heidelberg, 2010.

[83] Gunter Mussbacher, Daniel Amyot, and Jon Whittle. Composing goal and scenario models with the aspect-oriented user requirements notation based on syntax and semantics. In *Aspect-Oriented Requirements Engineering*, pages 77–99. Springer Berlin Heidelberg, 2013.

[84] Gunter Mussbacher, JoÃ£o Araújo, Ana Moreira, and Daniel Amyot. Aourn-based modeling and analysis of software product lines. *Software Quality Journal*, 20(3-4):645–687, 2012.

[85] Gunter Mussbacher and Jörg Kienzle. A vision for generic concern-oriented requirements reuse$^{\text{re@21}}$. In *21st IEEE International Requirements Engineering Conference, RE 2013, Rio de Janeiro-RJ, Brazil, July 15-19, 2013*, pages 238–249, 2013.

[86] Phu H. Nguyen, Max Kramer, Jacques Klein, and Yves Le Traon. An extensive systematic review on the model-driven development of secure systems. *Inf. Softw. Technol.*, 68(C):62–81, December 2015.

[87] Eugen C. Nistor and Andre van der Hoek. Explicit concern-driven development with archevol. In *ASE*, pages 185–196. IEEE Computer Society, 2009.

[88] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM.

[89] Object Management Group. *Unified Modeling Language: Superstructure (v2.4.1)*, December 2011.

[90] University of Ottawa. jUCMNav website: http://softwareengineering.ca/jucmnav, 2013.

[91] F. Paas, J.E. Tuovinen, H. Tabbers, and P.W.M. Van Gerven. Cognitive load measurement as a means to advance cognitive load theory. *Educational psychologist*, 38(1):63–71, 2003.

[92] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association of Computing Machinery*, 15(12):1053–1058, December 1972.

[93] D. L. Parnas. A technique for software module specification with examples. *Communications of the Association of Computing Machinery*, 15(5):330–336, May 1972.

[94] Jennifer Pérez, Jessica Díaz, C Costa-Soria, and Juan Garbajosa. Plastic partial components: A solution to support variability in architectural components. In *WICSA/ECSA 2009*, pages 221–230. IEEE, 2009.

[95] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jézéquel. Reconciling automation and flexibility in product derivation. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 339–348, Washington, DC, USA, 2008. IEEE Computer Society.

[96] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A model for developing component-based and aspect-oriented systems. In *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 259–274. Springer Berlin Heidelberg, 2006.

[97] 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.

[98] Klaus Pohl and Andreas Metzger. Variability management in software product line engineering. In *Proceedings of the 28th international conference on Software engineering (ICSE '06)*, pages 1049–1050. ACM, 2006.

[99] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st International Conference on Aspect-oriented Software Development*, AOSD '02, pages 141–147, New York, NY, USA, 2002. ACM.

[100] Clément Quinton, Daniel Romero, and Laurence Duchien. Cardinality-based feature models with constraints: A pragmatic approach. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 162–166, New York, NY, USA, 2013. ACM.

[101] Y. Raghu Reddy, Sudipto Ghosh, Robert B. France, Greg Straw, James M. Bieman, N. McEachen, Eunjee Song, and Geri Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development*, I:75–105, 2006.

[102] Marko Rosenmüller and Norbert Siegmund. Automating the configuration of multi software product lines. In *VaMoS*, pages 123–130, 2010.

[103] Thomas Saaty. How to make a decision: The analytic hierarchy process. *European Journal of Operational Research*, 48(1):9 – 26, 1990. Desicion making by the analytic hierarchy process: Theory and applications.

[104] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[105] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin Heidelberg, 2010.

[106] Matthias Schöttle, Omar Alam, Abir Ayed, and Jörg Kienzle. Concern-oriented software design with touchram. In *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 29 - October 4, 2013.*, pages 51–55, 2013.

[107] Matthias Schöttle, Omar Alam, Franz-Philippe Garcia, Gunter Mussbacher, and Jörg Kienzle. Touchram: A multitouch-enabled software design tool supporting concern-oriented reuse. In *Proceedings of the Companion Publication of the 13th International*

Conference on Modularity, MODULARITY '14, pages 25–28, New York, NY, USA, 2014. ACM.

[108] Matthias Schöttle, Omar Alam, Gunter Mussbacher, and Jörg Kienzle. Specification of domain-specific languages based on concern interfaces. In *Proceedings of the 13th Workshop on Foundations of Aspect-oriented Languages*, FOAL '14, pages 23–28, New York, NY, USA, 2014. ACM.

[109] Matthias Schöttle, Nishanth Thimmegowda, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Feature modelling and traceability for concern-driven software development with touchcore. In *Companion Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16 - 19, 2015*, pages 11–14, 2015.

[110] Reimar Schröter, Norbert Siegmund, and Thomas Thüm. Towards modular analysis of multi product lines. In *SPLC Workshops*, pages 96–99, 2013.

[111] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Control*, 20(3-4):487–517, September 2012.

[112] Carla T. L. L. Silva, Fernanda M. R. Alencar, Joao Araujo , Ana Moreira, and Jaelson Brelaz de Castro. Tailoring an aspectual goal-oriented approach to model features. In *SEKE*, pages 472–477. Knowledge Systems Institute Graduate School.

[113] 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 *Proceedings of the 29th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '05, pages 121–126, Washington, DC, USA, 2005. IEEE Computer Society.

[114] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: An aspect-oriented approach tailored for component based software development. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, AOSD '03, pages 21–29, New York, NY, USA, 2003. ACM.

[115] J. Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2):257–285, 1988.

[116] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[117] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language aop. In *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin Heidelberg, 2005.

[118] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. pages 107–119, 1999.

[119] Annette ten Teije, Frank van Harmelen, and Bob Wielinga. Configuration of web services as parametric design. In Enrico Motta, NigelR Shadbolt, Arthur Stutt, and Nick Gibbins, editors, *Engineering Knowledge in the Age of the Semantic Web*, volume 3257 of *Lecture Notes in Computer Science*, pages 321–336. Springer Berlin Heidelberg, 2004.

[120] Nishanth Thimmegowda, Omar Alam, Matthias Schöttle, Wisam Al Abed, Thomas Di'Meco, Laura Martellotto, Gunter Mussbacher, and Jörg Kienzle. Concern-driven software development with jucmnav and touchram. In *Proceedings of the Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), Valencia, Spain, October 1st and 2nd, 2014.*, 2014.

[121] 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*, MODULARITY Companion 2015, pages 39–42, New York, NY, USA, 2015. ACM.

[122] Tijs van der Storm. Variability and component composition. In *Software Reuse: Methods, Techniques, and Tools*, pages 157–166. Springer, 2004.

[123] Axel van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 5–19, New York, NY, USA, 2000. ACM.

[124] 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–, Washington, DC, USA, 2001. IEEE Computer Society.

[125] Rob van Ommering. Building product populations with software components. In *Proceedings of the 24th International Conference on Software Engineering*, pages 255–265, New York, NY, USA, 2002. ACM.

[126] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 233–242, Sept 2007.

[127] Jules White, Harrison D Strowd, and Douglas C Schmidt. Creating self-healing service compositions with feature models and microrebooting. *International Journal of Business Process Integration and Management*, 4(1):35–46, 2009.

[128] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society.

[129] Koen Yskout, Riccardo Scandariato, and Wouter Joosen. Do security patterns really help designers? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 292–302, Piscataway, NJ, USA, 2015. IEEE Press.

[130] Eric Yu. *Modelling strategic relationships for process reengineering.* PhD thesis, Department of Computer Science, University of Toronto, 1995.

[131] Yijun Yu, Julio Cesar Sampaio do Prado Leite, Alexei Lapouchnian, and John Mylopoulos. Configuring features with stakeholder goals. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 645–649, New York, NY, USA, 2008. ACM.