Concern-Oriented Use Case Maps

Cheuk Chuen Siow

School of Computer Science

McGill University, Montréal

March 2018

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

Abstract

Concern-Oriented Reuse (CORE) is a reuse paradigm that extends model-driven engineering with advanced modularization, goal modeling, and software product lines. Previous work enables modeling with CORE at the design level using Reusable Aspect Models (RAM). Requirements elicitation is also a crucial aspect of software development process, and one of the visual notation that expresses use cases as graphical workflows is Use Case Maps (UCM). UCM bridges the gap between requirements and design, and is part of the User Requirements Notation (URN) for scenario modeling. This thesis addresses the need for enabling scenario modeling in CORE. Based on Aspect-Oriented Use Case Maps (AoUCM), we introduce a novel technique that applies advanced separation of concerns for model-driven requirements elicitation with use cases—Concern-Oriented Use Case Maps (CoUCM). We define a metamodel for CoUCM that derives from the CORE metamodel, and formulate the weaving algorithm for CoUCM model composition. We then implement CoUCM in the TouchCORE tool as proof of concept. We present a working application of scenario modeling with TouchCORE, in which we further validate our implementation through case studies and workflow patterns. Validation shows that CoUCM is able to model requirements concerns that are reusable and scalable, and that further work is required to provide a more comprehensive implementation of CoUCM.

Abrégé

Acknowledgements

Preface

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Since 1960s, software development has been evolving rapidly to address the increasing demands of complex software. The complexity of modern software brings about difficulties in developing and maintaining quality software. Software engineering as a discipline ensures that developers follow a systematic production of software, by applying best practices to maximize quality of deliverables and minimize time-to-market. Various methodologies exist through the efforts of active research by theorists and practitioners, but the core of software development process typically consists of the following six phases—requirements gathering, design, implementation, testing, deployment, and maintenance.

Conceptual models help illustrates complex systems with a simple framework by creating abstractions to alleviate the amount of complexity. Hence, the use of models is progressively recommended in representing a software system. This simplifies the process of design, maximizes compatibility between different platforms, and promotes communication among stakeholders. Model-Driven Engineering (MDE) technologies offer the means to represent domain-specific knowledge within models, allowing modelers to express domain concepts effectively [1]. MDE advocates using the best modeling formalism that expresses relevant design intent declaratively at each level of abstraction. During development, we can use models to describe different aspects of the system vertically, in which the models are refined

from higher to lower levels of abstraction through model transformation. At the lowest level, models use implementation technology concepts, and appropriate tools can be used to generate code from these platform-specific models [2].

Modularity is key in designing computer programs that are extensible and easily maintainable, but concerns that are crosscutting and more scattered in the implementation are more likely to cause defects [3]. This poses obstacles for MDE because modeling such crosscutting concerns in a modular way is difficult from an object-oriented standpoint. Furthermore, reusability is also a main factor in allowing developers to leverage reusable solutions such as libraries and frameworks provided for a given programming language, thereby improving the development speed without having to implement existing software components from first principles. Model reuse is still in its early stage, but modeling libraries are emerging as well [4].

Concern-Oriented Reuse (CORE) is a new software development paradigm or approach that puts reuse at the forefront of software development [5]. In CORE, software development is structured around modules called *concerns* that provide a variety of reusable solutions for recurring software development issues. Techniques from MDE, Software Product Lines (SPL) engineering, and software composition (in particular feature-orientation and aspect-orientation) allow concerns to form modular units of reuse that encapsulate a set of software development artifacts, i.e., models and code, during software development in a versatile, generic way.

The main premise of CORE is that recurring development concerns are made available in a concern library, which eventually should cover most recurring software development needs. Similar to class libraries in modern programming languages, this library should grow as new development concerns emerge, and existing concerns should continuously evolve as

alternative architectural, algorithmic, and technological solutions become available. Applications are built by reusing existing concerns from the library whenever possible, following a well-defined reuse process supported by clear interfaces. To generate an executable in which concerns exhibit intricate crosscutting structure and behavior, CORE relies on additive software composition techniques, feature-oriented technology, and aspect-oriented technology.

Currently, CORE supports models at the design phase [6] only, but in order to fully integrate CORE with MDE, models typically used in other development phases should also be supported to also allow them to benefit from advanced modularization and reuse support. We are interested in adding support for CORE to models at the requirements phase, i.e., models that are typically built earlier than design models. We chose to concentrate on the User Requirements Notation (URN), which sets the standard as a visual notation for modeling and analyzing requirements [7]. URN formalizes and integrates two complementary languages: (i) Goal-oriented Requirements Language (GRL) to describe non-functional requirements as intentional elements, and (ii) Use Case Map (UCM) to describe functional requirements as causal scenarios. GRL and UCM are used to capture goal and scenario models, respectively. Since CORE already supports the use of goal models to analyze the impact of choosing features [5], this thesis focuses on integrating scenario models with the concepts of CORE.

1.1 Contributions

This thesis advances the state-of-the-art in modeling by proposing a complete solution for augmenting the UCM modeling notation with CORE capabilities, resulting in a variant of UCM—Concern-Oriented Use Case Maps (CoUCM). Specifically, the thesis makes the following contributions:

- UCM metamodel integration with CORE as the first necessary step to extend CORE concepts to UCM and formalize the language.
- Definition of weaving algorithm for UCMs that allows a requirements engineer to modularize scenarios according to concern features (i.e., a weaving algorithm compatible with CORE extension) as well as reuse existing scenarios when creating new ones (i.e., compatible with the CORE reuse mechanism).
- Validation of feasibility of proposed solution by implementing the UCM metamodel and weaving algorithm in TouchCORE as proof of concept.
- Validation of expressiveness of CoUCMs and demonstration of reuse potential by modeling the Authentication concern and Online Payment concern as case studies.

1.2 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 offers background information on CORE and UCM, as well as existing modeling techniques closely related to our work. Chapter 3 presents the integration of CORE with UCM. Chapter 4 validates the resulting integration process. Finally, Chapter 5 concludes the thesis and discusses possible future work.

Chapter 2

Background

CORE builds on three key components—MDE, model reuse, and Separation of Concerns (SoC)—to support large-scale model reuse. In this chapter, we provide an overview of CORE as a reuse technique with the current state of development in Section 2.1. Since we are also interested in studying the early phases of software development, where the requirements of the software to be built are established, we describe UCM as part of the requirements modeling tool and its use in specifying scenarios in Section 2.2. We then survey on existing aspect-oriented modeling techniques related to our work in Section 2.3.

2.1 Concern-Oriented Reuse (CORE)

CORE is a reuse technique that extends MDE with best practices from advanced modularization and SoC techniques [8], as well as features and goal modeling to support SPL [9]. Variations exist for any given solution, and creating a collection of similar software systems from a shared set of software artifacts to support SPL is possible through software reuse. MDE, reuse, and SoC form the three fundamental principles of CORE.

The objective of MDE is to develop software through modeling, where models are built using the formalisms that best describe and encapsulate the relevant properties for each level of abstraction. Through model transformations, models at high levels of abstraction

can be integrated with solution-specific models at lower levels of abstraction. This process continues until a final model, which may be executable, is generated. 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 aim of reuse is to develop software by reusing existing software artifacts instead of creating them from scratch. Software reuse results in a hierarchy of reusable artifacts, where smaller reusable artifacts are integrated to form increasingly larger reusable 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 [10, 11]. Benefits of reuse include increase in productivity and maintainability, as well as reduction in cost and time-to-market.

The third foundation of CORE is based on the ideas of SoC and information hiding [8, 12]. When concerns are well-separated, individual sections can be reused, optimized independently, and insulated from potential failures of other sections. Multi-dimensional SoC provides the ability to separate overlapping concerns and further extends SoC by defining a conceptual framework where concerns are treated as first-class entities during software development [13].

In the next subsection, we describe in detail how CORE enables reuse hierarchies through the three interfaces that a CORE concern provides, and formalize the key concepts of CORE for each interface in the CORE metamodel.

2.1.1 Concern Reuse Process

A CORE concern consists of a feature model, an optional impact model, and various realization models. The features in a feature model affect stakeholder goals that are expressed

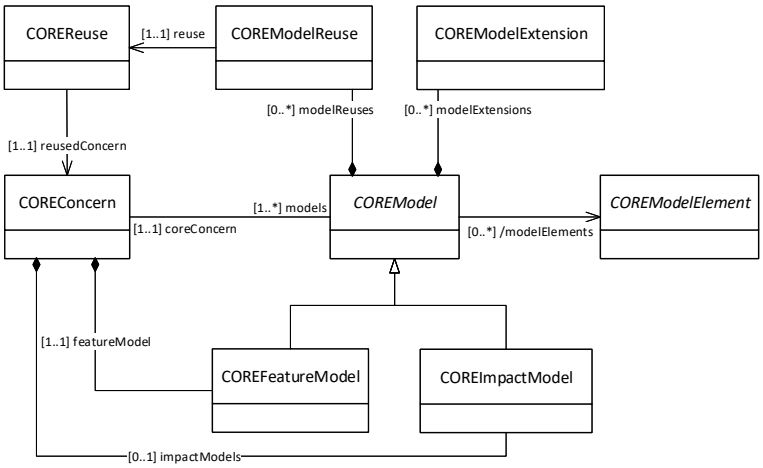


Figure 2.1: CORE metamodel: basic structure of a concern

with the impact model, while realization models characterize the features.

Figure 2.1 formalizes the key concepts for the basic structure of a concern (see Figure A.1 in Appendix A for the complete CORE metamodel). A concern (COREConcern) groups related models (*COREModel*) together and provides at least one model by default—a feature model (COREFeatureModel; subclass of *COREModel*). Additionally, a concern may have an optional impact model (COREImpactModel; subclass of *COREModel*) and several realization models that subclass *COREModel* defined by a corified* modeling language (not shown in Figure 2.1). A *COREModel* groups related model elements (*COREModelElement*).

A concern can reuse other concerns, which results in a simple reuse hierarchy consisting of two concerns, with the reused concern and the reusing concern having the same internal structure. A reusing concern has a COREReuse associated to the reused concern, and reusing a concern entails reusing its models. Consequently, a COREModelReuse is defined for each model of the concern that is reused. In addition, reuse also occurs within a concern, namely

*The term *corify* (verb: **corify**; past participle: **corified**; present participle: **corifying**; noun: **corification**) means to integrate a modeling language with CORE.

the extension of a feature (COREModelExtension), because the realization models of one feature may reuse the realization models of another feature.

The concern reuse process offers three interfaces to facilitate reuse [5]. The variation interface presents the design alternatives and their impact on non-functional requirements. The customization interface of the selected alternative details how to adapt the generic solution to a specific context. Finally, the usage interface specifies the provided behavior. These three interfaces allow a concern user to reuse a concern by following the simple three-step process outlined below.

Step 1: Variation Interface

The variation interface describes the possible variations of the concern and the impact of different variants on high-level stakeholder goals, system qualities, and non-functional requirements. The variations are represented with a feature model [14] that specifies the individual features that a concern offers as well as their dependencies. The impact model represents the impact of choosing a feature and can be specified with goal models using GRL, which is part of the URN standard [15].

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 [16]. Inter-feature relationships allow the specification of (i) mandatory and optional parent-child feature relationships as well as (ii) exclusive (XOR) and alternative (IOR) feature groups. 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 XOR feature group if its parent feature is selected, whereas at least one feature must be selected in an IOR feature group if its parent feature is selected.

In Step 1 of the CORE reuse process, 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 [17]. To maximize the reusability of the concern that is being built, the user should select from the reused concern only the features that are absolutely necessary to achieve the required functionality and goals. Decisions about potential use of alternative or optional features should be delayed by reexposing them [18]. Based on a configuration, the CORE modeling tool composes the generic realization models of each modeling language that realize the selected features to yield new, user-tailored realization models of the concern corresponding to the desired configuration. Depending on the root phase of the concern, this composition may involve requirement models, architecture models, design models and/or code.

The metamodel excerpt in Figure 2.2 describes the part of the variation interface of CORE related to the feature model. A feature (COREFeature) is contained in COREFeatureModel. Exactly one feature is the *root* feature of the feature model. COREFeature has an attribute (*parentRelationship*), which is an enumeration of COREFeatureRelationshipType, that specifies the relationship of a feature with its parent. Possible relationships include 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. Each feature has at most one *parent* and may have many *children*. A feature may be *realizedBy* many *COREModel*s, and similarly,*COREModel* may *realize* many features. This association is used to link features to the models of the corified modeling lan-

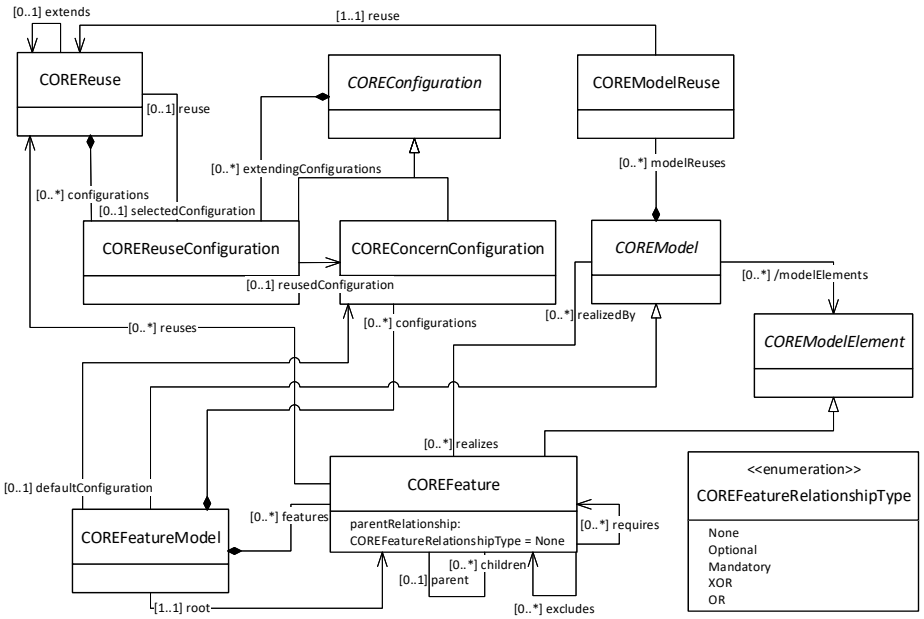


Figure 2.2: CORE metamodel: variation interface - features

guage. Furthermore, a configuration (*COREConfiguration*) defines the *selected* and *reexposed* features. A feature model may predefine several commonly used *configurations*, one of which may be designated as the *defaultConfiguration*. A COREReuse defines its own *configurations*, one of which may be designated as the *selectedConfiguration*. This allows configurations of the reused concern to be modified according to changes in the reuse context. A configuration of a COREReuse may either directly select and reexpose features of the reused concern, or it may choose one of the predefined configurations (*reusedConfiguration*) of the reused concern and optionally select and reexpose additional features.

In addition to the feature model, the impact model is also part of the variation interface that helps the user perform trade-off analysis when selecting the features. Goal models are used for impact analysis because of the ability to allow vague, hard-to-measure system qualities to be evaluated, in addition to more quantifiable qualities. Goal modeling 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 find the best suited alternative for a particular situation. These principles also apply in the CORE 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 concern user. The goal model for the variation interface is called impact model to signify that goal models in CORE are different from traditional goal models—not only because the main focus is on capturing the impact of features on qualities, but their use is more restricted and specialized. Impact models use features in place of tasks, and they use relative quantitative contributions exclusively to express the impact on goals of importance for the concern [19].

Similar to the feature model, the impact model is also reused by reconnecting the impact model of the reusing concern with the impact model of the reused concern. This allows for system-wide trade-off analysis and is accomplished by a feature impact node. The feature impact node expresses (i) that an impact only occurs if the feature is selected, (ii) which reused concerns impact the goal in the context of the feature, and (iii) how much the feature itself and the reused concerns contribute to the goal.

The metamodel excerpt in Figure 2.3 describes the part of the variation interface of CORE related to the impact model. A COREImpactModel contains both nodes (COREImpactNodes) and contributions (COREContributions). A goal is represented by a COREImpactNode (a subclass of *COREModelElement*). Its *scalingFactor* and *offset* are used by the normalization step to ensure that the satisfaction value of a node is always between 0 and 100. On the

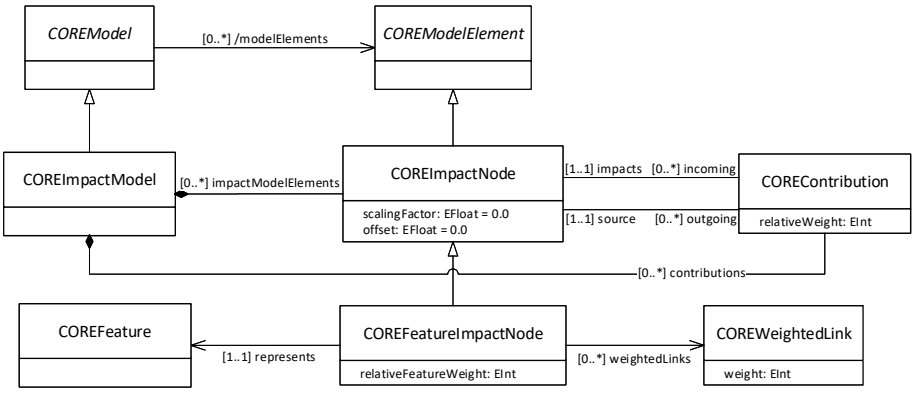


Figure 2.3: CORE metamodel: variation interface - impacts

other hand, any COREFeature is represented by a COREFeatureImpactNode (a subclass of COREImpactNode) in the impact model. Each COREContribution has a *relativeWeight* integer attribute to store its contribution value, describing how one COREImpactNode (*source* role) impacts another COREImpactNode (*impact* role). If a feature makes use of a reused concern, then the impact of a COREFeatureImpactNode relative to the impact of the reused concern is described by its *relativeFeatureWeight* and the *weight* of its COREWeightedLink that represents the impact of the reused concern used by the feature [20, 21]. A COREFeatureImpactNode is created for each goal that is impacted by the feature. A COREWeightedLink exists for each reused concern used by the feature that impacts the goal.

Step 2: Customization Interface

The customization interface describes how a chosen variant can be adapted to the needs of a specific application. 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 modeling elements of the application that intends to reuse the concern. The customization interface is hence used when a specific

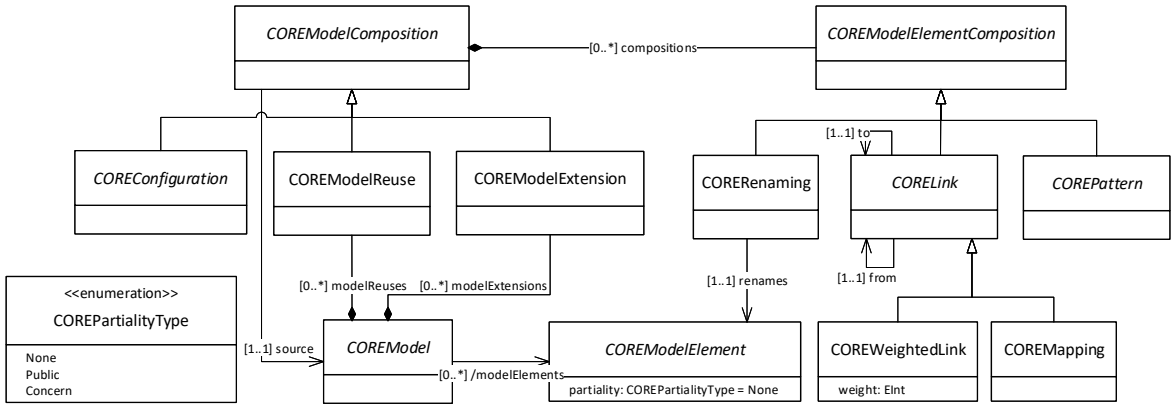


Figure 2.4: CORE metamodel: customization interface

variant of a reusable concern is composed with the application.

In Step 2 of the reuse process, the concern user has to adapt the generated user-tailored, yet still generic, realization models of the concern to the application context by mapping elements from the customization interface to elements from the application context. Depending on the root phase of the concern, this step might require customizing requirement models, architecture models, design models, and/or code. Based on the customization mappings, the CORE tool then composes the user-tailored concern realization models with the application-specific models to yield adapted realization models of the concern.

The customization interface manifests itself in the metamodel excerpt shown in Figure 2.4 by the *partiality* attribute of *COREModelElement*. A model element must be adapted either by a reusing concern (*Public*), by another feature in the same concern (*Concern*), or not at all (by default *None*; it is not a generic element).

The remaining metaclasses provide the means to compose an element in the customization interface with an element from the reusing concern. The type of composition (*COREModelComposition*) depends on the model that needs to be composed. *COREModelCom-*

position captures the fact that all compositions combine a *source* (i.e., reused model) with the reusing model and that several elements of the model may have to be composed (*CORE*ModelElementComposition). *COREConfiguration* is dedicated to the composition of feature models (discussed in Step 1 of the reuse process). *COREModelElementComposition*s are not needed for feature models, but the other two concrete subclasses of *COREModelComposition*—COREModelReuse and COREModelExtension—do require *COREModelElementComposition*s. *COREModelElementComposition* has two abstract subclasses—*CORELink* and *CORE-Pattern*—representing two general ways for specifying compositions for modeling languages other than feature models [22, 23]. *COREPattern* is used when the composition is specified using pattern matching, while *CORELink* is used when a pair of *COREModelElement*s is composed. 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. The subclasses of *CORELink* differentiate weighted (COREWeightedLink; used for impact models and discussed in Step 1 of the reuse process) and non-weighted mappings (COREMapping). In total, there are three different techniques for composing model elements covered by the CORE metamodel.

Step 3: Usage Interface

The usage interface describes how the application can finally access the structure and behavior provided by the concern. While one variation interface consisting of feature and impact models exists for the whole concern, the customization interface and usage interface typically exist for each other type of model/modeling language used to realize the functionality encapsulated within the concern (i.e., for each corified modeling language).

In Step 3 of the reuse process, the concern user can then use the functionality provided by the selected concern features that are exposed in the usage interface of the adapted

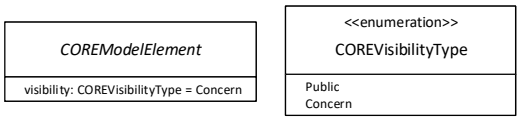


Figure 2.5: CORE metamodel: usage interface

realization models within the user's own application models.

The usage interface manifests itself in the metamodel excerpt shown in Figure 2.5 by the *visibility* attribute of COREModelElement. A model element can either be accessed by a reusing concern (*Public*) or is only visible within a concern (*Concern*).

2.1.2 CORE Weaver

As explained in Step 1 of the reuse process, after the user selects the desired features of the reused concern, the modeling 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. This subsection describes the weaving process that flattens an entire concern hierarchy given a specific configuration to generate a final realization model where all involved concerns are merged. The process of model composition utilizes the weaver that weaves recursively by traversing the concern hierarchy, as illustrated in detail by Kienzle et al. [24], in which the composition algorithms support delayed decision making in CORE [18]. Each corified modeling language has its own weaver to compose a particular model as different models require specific algorithm for composition. Nonetheless, CORE has a generic weaver that administers model composition to the appropriate weaver. The CORE weaver performs either a *single weave* or *complete weave*.

Single Weave: This process weaves two directly dependent models together. This is the fundamental unit of weaving as model composition of a concern begins with this step.

Model elements are copied from the lower-level/mode generic model to the higher-level/more specific model within a concern prior to composition, and references of elements need to be updated post-composition based on the information of elements mapped from lower-level model to higher-level model.

Complete Weave: This process weaves all dependent models of a concern hierarchy, forming an independent model that represents the merged models from the selected features of a concern. The weaver first selects the pair of models that has the highest depth level in the concern (tree), reducing the pair of models into a woven model through *single weave*. This is carried on recursively until the weaver resolves all dependencies, resulting in a final woven model once there are no dependencies left.

2.2 Use Case Maps (UCM)

UCM is a visual notation that aims to link behavior and structure of a system [25, 26]. UCM paths are first-class architectural entities that describe causal relationships between responsibilities that may be bound to underlying organizational structures of abstract components [27]. UCMs are used to describe and integrate use cases representing the requirements, and UCM paths represent scenarios that intend to bridge the gap between requirements and detailed design.

Figure 2.6 illustrates the basic elements of the UCM notation. A map contains any number of paths and optionally components. Paths express causal sequences and may contain several types of path nodes, each linked with *node connections* (wiggly lines). Paths begin at *start points* (filled circles—representing preconditions or triggering causes) and terminate at *end points* (bars—representing postconditions or resulting effects). *Responsibilities* (crosses—representing actions, tasks, or functions to be performed) describe required actions

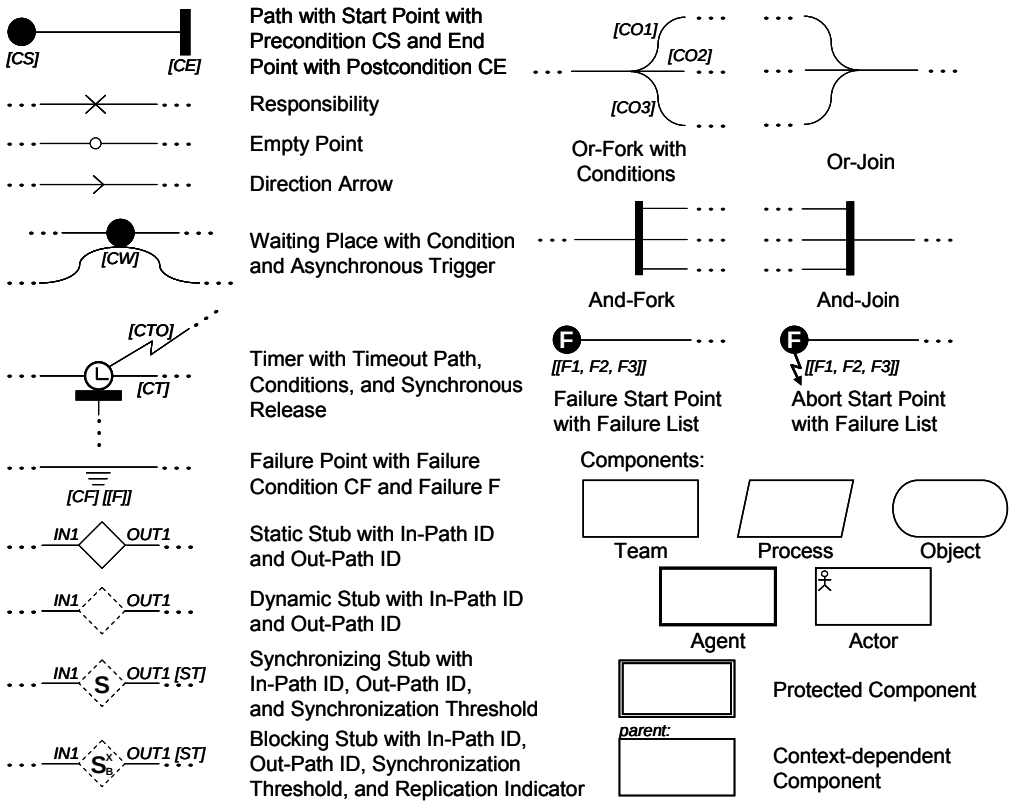


Figure 2.6: UCM notation. Image courtesy of ITU [15]

or steps to fulfill a scenario. Alternatives are represented as overlapping paths. An *OR-fork* splits a path into multiple alternatives, while an *OR-join* merges multiple overlapping paths. Alternatives may be guarded by conditions represented as labels between square brackets. On the other hand, concurrent routes are represented through the use of vertical bars. An *AND-fork* splits a path into multiple concurrent segments, while an *AND-join* synchronizes multiple paths together. Other notational elements include *waiting places* (filled circles—representing synchronous or asynchronous interactions), *timers* (clocks—representing waiting places triggered by timely arrival of specific events), *aborts* (zigzag lines—representing paths that terminates the execution of other causal chain of responsibilities), and *failure points* (ground symbols—representing potential failure points on a path).

A map that is too complex to be represented in a single UCM model can be decomposed into sub-maps. A top-level UCM, referred to as a root map, can include containers (called *stubs*) for sub-maps (called *plug-ins*). *Plug-in bindings* define the continuation of a path between stubs and plug-ins, by connecting the path segments coming in and going out of a stub with start and end points of the plug-ins. A stub of kind *static* (plain diamond) can only contain at most one plug-in, whereas a stub of kind *dynamic* (dashed diamond) may contain several plug-ins. A *selection policy* (often described with preconditions) determines which plug-ins of a dynamic stub to choose from during runtime. A stub of kind *synchronizing* (dashed diamond with symbol S) is a dynamic stub that allows its plug-ins to be executed in parallel and continues only once the plug-ins have synchronized. Finally, a stub of kind *blocking* (dashed diamond with symbol S_B) is a synchronizing stub that does not allow its plug-ins to be visited more than once at the same time.

Components are used to specify the structural aspects of a system. A responsibility is bound to a component when the cross is inside the component. As such, the compo-

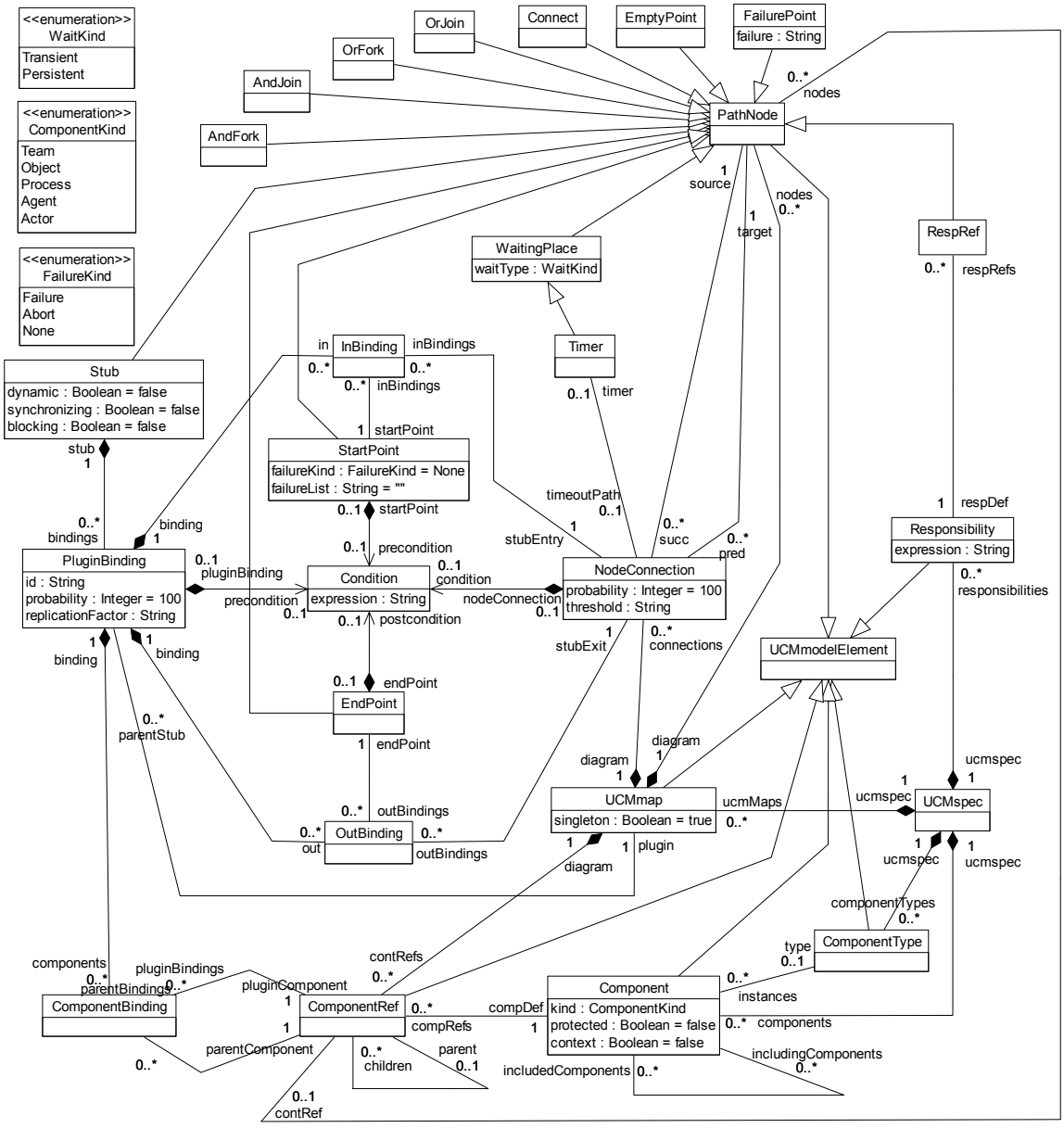


Figure 2.7: UCM metamodel. Image courtesy of ITU [15]

nent is responsible for performing the action, task, or function represented by the responsibility. Components can be of different kinds, have various attributes, and may contain sub-components. For example, a component of kind *object* (rounded rectangle) represents a passive component that does not have its own thread of control, while a component of kind *process* (parallelogram) represents an active component that does. A component of kind *actor* (rectangle with person symbol) represents an entity interacting with the system, whereas a component of kind *team* (rectangle) is allowed to contain components of any kind. A *protected* component (double nested rectangles) does not allow a second path to enter the component if one path is already executing inside the component. Any kind of component can be protected.

The metamodel in Figure 2.7 expresses the formalism of the standard UCM notation. The UCM notation is mainly composed of path elements and components. Path elements are derived from a common class PathNode and, along with node connections and components, constitute a UCMmap. RespRef and ComponentRef act as reference points for Responsibility and Component, respectively, such that multiple references may potentially refer to the same definition. StartPoint, PluginBinding, EndPoint, and NodeConnection may contain Condition as precondition for triggering causes, postcondition for resulting effects, or condition for choosing alternatives. All model elements are derived from UCMmodelElement.

2.3 Related Work

Many languages have been proposed for the design and specification of workflow processes. One of the notation that describes workflow models is UCM and it is part of the URN specification. URN is currently the only standard that graphically addresses both goals (non-functional requirements with GRL) and scenarios (functional requirements with UCM)

in one unified language. Since GRL has been integrated in CORE for impact modeling, throughout this thesis we focus on the corification of UCM to allow scenario modeling in the context of CORE.

One of the closely related work is Aspect-oriented Use Case Map (AoUCM), of which it has been described in detail as part of the extension for URN—Aspect-oriented User Requirements Notation (AoURN)—as an effort to introduce aspect-oriented concepts to better encapsulate crosscutting concerns in requirements models [28]. The significance of AoUCM aspect is the introduction of pointcut stub and aspect marker to define its structure/behavior and pattern for its composition rule. Pointcut stub may contain one or more pointcut maps that visually describes pointcut expressions. Pointcut expression is used to match a pointcut map with a base model to identify the join points (functional UCM path nodes) that are affected by the aspect. The insertion points of the join points are indicated in the base model by aspect markers. Nevertheless, we propose a different approach in handling model composition for CoUCM, which we discuss in detail in Chapter 3.

Reusable Aspect Models (RAM) is currently the only aspect-oriented modeling language that has been successfully integrated with CORE [29]. RAM allows a modeler to express the structure and behavior of a complex system at the design level using class, state and sequence diagrams [6]. Previous work on adding support for RAM to CORE has laid the foundation in which our work on adding support for UCM to CORE is based on. In particular, the proper extension of metaclasses *COREModel*, *COREModelElement*, and COREMapping for the UCM modeling language as well as the weaver.

As for the implementation of modeling languages in editing tools, AoURN has been implemented in jUCMNav, and RAM has been implemented in TouchRAM. jUCMNav is a URN editor and analysis tool that is built as an Eclipse plug-in [30], whereas TouchRAM

is a multitouch-enabled tool for agile software design modeling aimed at developing scalable and reusable software design models [31, 32]. Current development of TouchCORE is the updated version of TouchRAM [33]. We attempt to add support for UCM in TouchCORE as proof of concept, which we discuss in detail in Chapter 4. CoUCM in TouchCORE is closely related to AoUCM in jUCMNav in terms of basic workflow functionalities, albeit support is at the alpha phase.

Chapter 3

Adding Support for UCM to CORE

Although hypothetically CORE offers a base that supports multiple modeling languages, only one notation has been integrated with CORE thus far—the RAM modeling notation that is used for design modeling with class, sequence, and state diagrams [6, 29]. The goal of the thesis is to investigate whether CORE can also support an additional modeling language—the requirements specification language, specifically UCM for functional requirements modeling.

This chapter presents the corification of UCM using the CORE metamodel. We describe the steps taken to corify UCM in Section 3.1, in addition to tailor the customization and usage interfaces for UCMs. We also define the weaving algorithm specific for UCM in the context of CORE, fulfilling the needs of a requirements engineer to build modular UCMs using CORE model extensions and reuses, in Section 3.2.

3.1 Corification of UCM

The idea behind integrating a modeling language with CORE is to allow CORE concerns the ability to contain models of that language. This motivates us to investigate the possibility for UCMs to be part of the concerns, such that scenario models built with UCMs can serve as realization models for features of the concerns. To add support for CORE for a particular modeling language, the initial step is to first define the base metamodel for the language, and

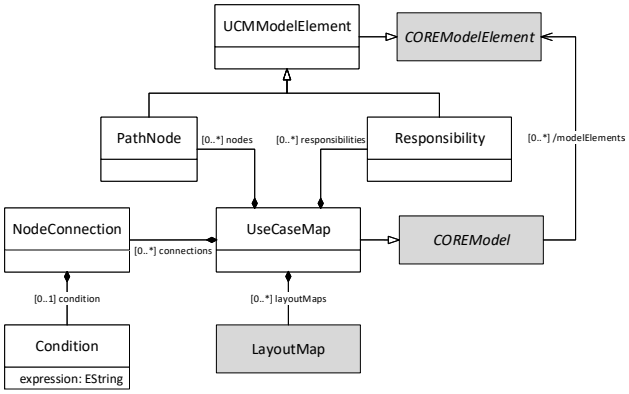


Figure 3.1: Extended UCM metamodel with CORE

then derive the appropriate metaclasses from the CORE metamodel for CORE to recognize the language.

Abstract and concrete classes of the CORE metamodel are utilized differently when corifying a modeling language. The abstract classes *COREModel*, *COREModelElement*, and *COREPattern* serve as extension points and are intended to be subclassed by a modeling language. This enables the addition of arbitrary modeling languages to CORE and also uniform treatment of pattern-based composition. The remaining abstract classes *COREModelComposition*, *COREModelElementComposition*, and *CORELink* are used within the CORE metamodel and seldom subclassed by a modeling language. On the contrary, concrete classes are designed to be used exactly as it is in the corified modeling languages. They provide the necessary mechanisms for model extensions and reuses, feature and impact modeling, as well as a way to implements and visualizes these concepts in its modeling tool.

We follow the URN specification [15] closely in corifying the UCM metamodel. Figure 3.1 shows a partial view of the corified UCM metamodel, focusing on the elements that extend the CORE metamodel through subclassing (from an existing metaclass in the modeling

language to an abstract CORE metaclass).[†] By subclassing the necessary abstract classes of the CORE metamodel, UCM is able to provide all the properties of CORE:

1. A UCM model may now belong to a concern by realizing at least one of its features.
2. A UCM realization model may now have impacts on high-level goals.
3. A UCM model may extend another UCM model that belongs to a different feature.
4. A UCM model may reuse another UCM model that belongs to a different concern.

The first and second properties are achieved once the UCM metamodel is properly extended, since CORE provides feature and goal modeling by default to all supported realization models. The third and forth properties, however, requires some thought on determining the customization interface that best suites UCMs, as customization interface is unique to all corified modeling languages and mappings should be tailored to each of the languages. After much deliberation, we arrived at a decision to provide mappings of different UCM model elements based on CORE model extensions and CORE model reuses.

Since the most common unit of occurrence for UCMs is responsibility that represents much of the actions in use cases, one of the most likely candidates for mapping is the responsibility element and is especially useful for model extensions. Aspect-oriented modeling can be carried out by combining separate UCM paths together, with mapped responsibilities act as join points. To illustrate the concept, we take home security concern as an example, with the base scenario having at least a "lock" responsibility that represents an action to lock the door of the home before leaving. Consider an optional feature "alarm system" that might be added to the home security. A typical UCM would use a stub, along with OR-forks and OR-joins, to model the optional feature in the root UCM. This, however, results in "alarm system" tangled up with the root UCM. Our approach resolves this issue by shifting the

[†]The gray elements in the figures are the classes that derived from the CORE metamodel.

need to specify join points to the optional feature itself, by mapping responsibilities between the parent and child UCMs. In this case, we would also have a "lock" responsibility in the UCM that models the "alarm system" feature, and model the alarm system scenario (e.g., "enter code" and "validate code") prior to "lock", then map the "lock" responsibilities between the root UCM and feature UCM. We simply augment the base scenario through model extension, so whenever the feature "alarm system" is selected as part of the home security, the model would include the scenario for setting up the alarm system as modeled in the "alarm system" feature after composing all the necessary models together. Otherwise if "alarm system" is not selected, the composed model would only have the locking mechanism without any model elements that represent the alarm system tangled up with the root UCM.

The idea of having a stub to represent a container for reusing UCM models is straightforward. Because the purpose of a stub is to bind plug-ins that consists of separate UCMs, a UCM model that is being reused can appear as a plug-in for the stub. Whereas the start and end points of the plug-ins connect the path segments coming in and going out of a stub to form the continuation of paths between the stub and plug-ins, similarly the start and end points of the reused UCM model connect the path segments coming in and going out of a stub. Reusing a UCM model is essentially reusing a concern—the selection of features when reusing a concern determines the final UCM model to be reused.

Reusing a concern from a UCM model prompts the feature selection process. This signals the feature that the UCM model realizes to reuse the other concern with the desired feature(s). The reusing UCM model then establishes the mappings to the reused UCM model that realizes the reused features. This is achieved as follows. The root element UseCaseMap subclasses *COREModel*, which makes it part of a COREConcern (see Figure 2.1, association between COREConcern and COREModel). This allows a UCM to realize a feature (see Fi-

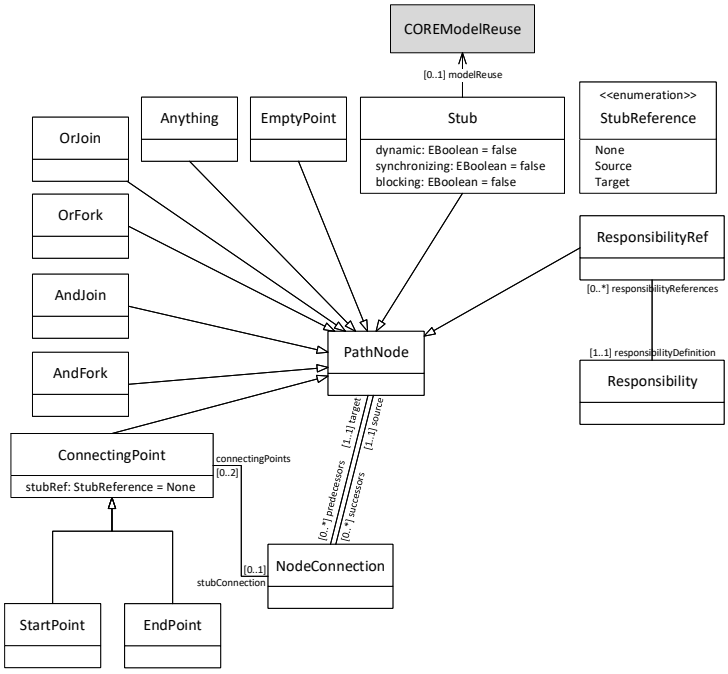


Figure 3.2: Path nodes for corified UCM

gure 2.2, COREModel realizes COREFeature) within a concern. Therefore, the concern can create a COREReuse to reuse another concern. The reusing UCM then creates a COREModelReuse that has a direct association to the created COREReuse and a *COREConfiguration* that selects the desired features from the reused concern. The CORE modeling tool then composes the UCM models of the reused concern that realize the selected features to generate a single woven user-tailored UCM model of the reused concern. Mappings to the model elements of this generated model are established using the class COREMapping, consequently allowing the reusing UCM to customize the generated UCM model of the reused concern.

A standard UCM consists of PathNode, Responsibility, and NodeConnection. LayoutMap is added as part of the composition to allow positioning of the elements for viewing. We omit the inclusion of certain elements such as Component, Timer, and FailurePoint to limit

the scope of this thesis. On the contrary, PluginBinding is excluded on purpose since we utilize COREMapping as our approach to bind separate UCMs to Stub. We incorporate several changes to the path nodes to support aspect-oriented modeling and reuse. Figure 3.2 illustrates the addition of Anything and ConnectingPoint, as well as a directed association from Stub to COREModelReuse, to the UCM metamodel.

Anything: We included the Anything pointcut element from the extended AoUCM metamodel [28]. Anything (denoted by `...`) acts as a wild card and can represent a subset of nodes in a path to allow variations in pattern matching. This is useful for facilitating complex model weaving, as it allows any sequence of UCM model elements, including an empty sequence, to be matched. Similar to the pointcut elements of sequence diagram in RAM [24], the pointcut of an object lifeline represented with a *-box refers to the matched behavior of an advised sequence diagram, whereas the pointcut of a message labeled with a * means that any message between two objects is of interest.

ConnectingPoint: We established a new path element to the metamodel. ConnectingPoint is used to replace PluginBinding and serves as an intermediate node that represents either a StartPoint or an EndPoint. By default, an actual start or end point within a UCM does not have a reference to a stub, hence the default value for StubReference is None. Instead, when we have a NodeConnection that connects an element with a stub, then a hidden connecting point is automatically attached to the node connection (and deleted upon removal of the connection). Each node connection can have at most two connecting points if both the predecessor and successor nodes of the connection are stubs. Incoming connection to a stub generates a hidden end point with the value of stubRef set to Target, whereas outgoing connection from a stub generates a hidden start point with the value of stubRef set to Source. These hidden points allow us to define composition specifications through

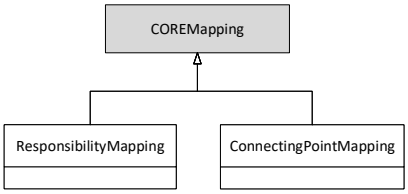


Figure 3.3: Customization mappings for corified UCM

customization mappings.

Since we are using COREMapping to specify customizations, it is necessary for UCMModelElement to subclass *COREModelElement*. That way, all subclasses of UCMModelElement (i.e., PathNode and Responsibility) can be used as source and destination classes for COREMapping. As shown in Figure 3.3, we defined the composition specifications for specific UCM model elements: Responsibility and ConnectingPoint. They were selected so that we can compose UCM models based on the mappings of these elements. This leads us to the next section where we describe in detail how model composition is achieved through weaving.

3.2 UCM Weaving

The role of the weaver is to facilitate model extensions and reuses. We offer two options when mapping elements between UCMs: (i) direct mapping of responsibilities; and (ii) cross mapping of connecting points. Cross mapping is necessary because of the nature of start and end points, where a start point of a UCM maps to an end point of a stub, and vice versa. A stub can be perceived as being superimposed with an end point followed by a start point, and those points collapsed into a point that is the stub [26]. Here, the hidden end point of a stub represents the incoming connection and it signifies the end of the sequence before the stub, and the hidden start point of a stub represents the outgoing connection and

it signifies the start of the sequence after the stub. Both options have different procedures when weaving.

3.2.1 Weaving Algorithm

As explained in Section 2.1.2, CORE models are always composed in pairs with *single weave*, and complex extension and reuse hierarchies are usually composed with multiple levels of single weaves. The algorithms presented here are specific for single weaving, meaning that a composition is performed from one model (UCM_{source}) to another model (UCM_{target}). This action can be chained together with other compositions, even with the hierarchical structure of the concern features. Here, we specify the subscript $_{\text{source}}$ for the model elements of a UCM the weaver composes from, and the subscript $_{\text{target}}$ for the model elements of a UCM the weaver composes to. UCM_{source} and UCM_{target} are merged prior to weaving, retaining all the path nodes and node connections from both models. Then the weaver iterates through the available composition specifications and executes the algorithms based on the specific type of mapping. The output of the woven model results in the amalgamation of UCMs based on the composition specification defined by the designer of the models, as well as the selected features of the concern by the user.

The idea of isolating features as individual models supports the use of advanced SoC—each feature encapsulates its realization model. Features of a concern are nested in a hierarchical order, and the connection between features can be seen as parent-child relationship. Extension of a model depends on this relationship to ensure that models are woven in the correct order. Only the selected features of a concern are woven as a whole, providing only the absolute necessary details to fully describe the different use cases.

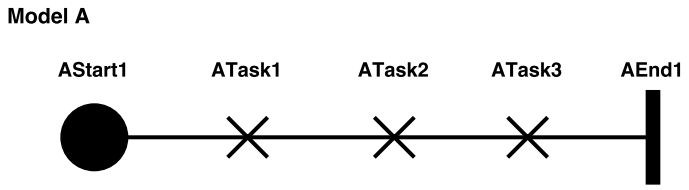


Figure 3.4: Example base scenario to be extended

Responsibility Mapping

Mapping with responsibilities allows for model extensions between parent and child UCMs. Composition specification can be defined by mapping from a parent UCM's responsibility to a child UCM's responsibility. The idea is to insert the paths defined in the child UCM to the appropriate position in the parent UCM based on the mappings. We take a simple UCM in Figure 3.4 as a base scenario for extension examples. Algorithm 1 illustrates the procedure of weaving for responsibility mappings. The function *WeaveResponsibilityMapping* initiates the process by identifying the mapped responsibilities (*from* UCM_{source} *to* UCM_{target}), and traversal begins from the point of $responsibility_{\text{target}}$ in both directions: (i) toward predecessors until start point encountered; and (ii) toward successors until end point encountered (Figure 3.5).[‡] A UCM is represented as a directed graph, with possible cycles via OrForks and OrJoins. As such, we implemented a depth-first search approach for traversing the graph through recursion (lines 36 and 65), and a mechanism to determine whether a node has been explored (lines 19-23 and 43-47).

Furthermore, we allow multiple consecutive mappings between two UCM models (Figure 3.6). The path of a UCM may consist of mapped responsibilities interspersed with other path nodes. While traversing forward, lines 59-63 handle the next mapped respon-

[‡]Red dashed lines indicate path traversal, starting from nodes with the «mapped to» indicator. Red path nodes will be destroyed in the process, and the paths at both ends of the traversal in Model B are fused with the join points surrounding the mapped elements in Model A.

Algorithm 1 Weaving Algorithm: Responsibility Mapping

```
1: function WeaveResponsibilityMapping(ucm, composition)
2:     node_source ← get first node of composition mapping (from)
3:     node_target ← get second node of composition mapping (to)
4:     mark node_target as visited
5:     indicate start point has not been encountered
6:     indicate end point has not been encountered
7:     call TraverseToPredecessor(ucm, node_target, node_source)
8:     call TraverseToSuccessor(ucm, node_target, node_source)
9:     remove node_source from ucm
10: end function
11: function TraverseToPredecessor(ucm, node_target, node_source)
12:     for each predecessor of node_target do
13:         node_target_pred ← get predecessor of node_target
14:         if linkage exists from previous mapping then
15:             set the predecessor of node_target's connection to node_source's predecessor
16:             disable linkage
17:             skip this loop
18:         end if
19:         if node_target_pred is visited then
20:             skip this loop
21:         else if node_target_pred is not Anything then
22:             mark node_target_pred as visited
23:         end if
24:         if node_target_pred is StartPoint and start point is not encountered then
25:             if visibility of node_target_pred is Concern then
26:                 set the predecessor of node_target's connection to node_source's predecessor
27:                 remove node_target_pred from ucm
28:                 indicate start point has been encountered
29:             end if
30:         else if node_target_pred is Anything then
31:             set the predecessor of node_target's connection to node_source's predecessor
32:             if node_target_pred does not have any predecessor then
33:                 remove node_target_pred from ucm
34:             end if
35:         else
36:             recursively call TraverseToPredecessor(ucm, node_target_pred, node_source)
37:         end if
38:     end for
39: end function
```

```
40: function TraverseToSuccessor(ucm, node_target, node_source)
41:     for each successor of node_target do
42:         node_target_succ ← get successor of node_target
43:         if node_target_succ is visited then
44:             skip this loop
45:         else if node_target_succ is not Anything then
46:             mark node_target_succ as visited
47:         end if
48:         if node_target_succ is Endpoint and end point is not encountered then
49:             if visibility of node_target_succ is Concern then
50:                 set the successor of node_target's connection to node_source's successor
51:                 remove node_target_succ from ucm
52:                 indicate end point has been encountered
53:             end if
54:         else if node_target_succ is Anything then
55:             set the successor of node_target's connection to node_source's successor
56:             if node_target_succ does not have any successor then
57:                 remove node_target_succ from ucm
58:             end if
59:         else if node_target_succ exists in composition mapping (to) then
60:             copy node connection of successor
61:             set the successor of copied connection to node_source's successor
62:             add the copied connection to ucm
63:             enable linkage to next mapping
64:         else
65:             recursively call TraverseToSuccessor(ucm, node_target_succ, node_source)
66:         end if
67:     end for
68: end function
```

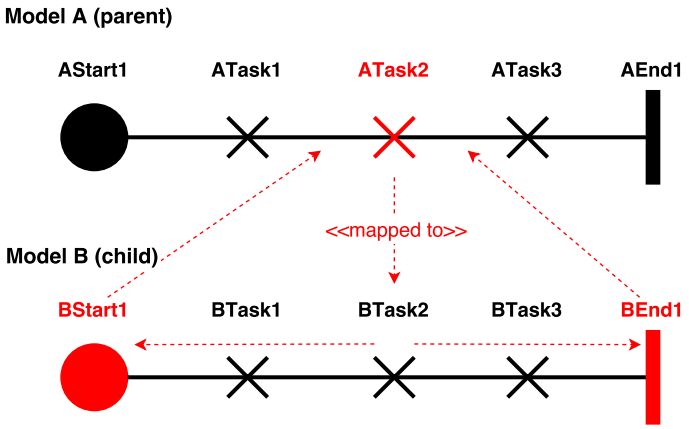


Figure 3.5: Schematic representation of extension with basic mapping

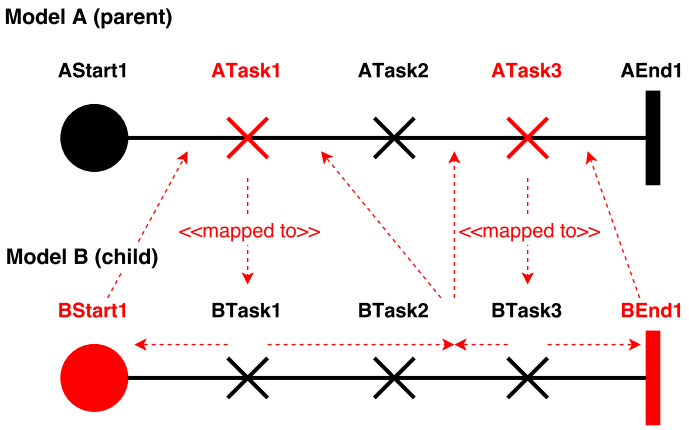


Figure 3.6: Schematic representation of extension with consecutive mappings

sibility. If it exists, forward traversal stops for this specific composition and appropriate nodes are connected between UCM_{source} and UCM_{target}. For subsequent mappings, lines 14-18 handle the linkage from previous mappings, and backward traversal stops at the point of mapped responsibilities and appropriate nodes are connected between UCM_{source} and UCM_{target}. This pattern continues until the weaver reaches an end point, whereby the predecessor of UCM_{target}'s end point connects to the successor of the mapped responsibility (*from*) UCM_{source} and this end point is deleted (lines 48-53). Same goes for backward traversal until the weaver reaches a start point (lines 24-29). Lastly, the responsibility that was

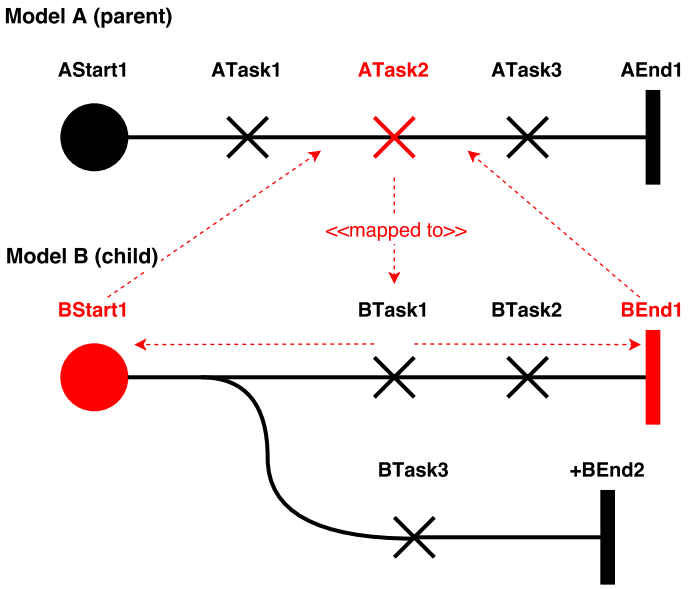


Figure 3.7: Schematic representation of extension with multiple branches

mapped (*to*) in UCM_{target} remains in the model, while the responsibility that was mapped (*from*) in UCM_{source} is deleted in the final woven UCM model.

UCMs may have multiple start points joining into a path, or a path may branch to multiple end points (Figure 3.7). In this case, we allow a start or end point to set its visibility level. By default, a connecting point is given the visibility of Concern that signifies the start or end point is only visible when viewing a UCM model for a specific feature of a concern, but disappears after the composition process. The other option is Public for global visibility and is used to retain the start or end point even after the composition process—the weaver would just ignore Public connecting points and proceed to other branches. This feature is useful in defining multiple entry points, or alternative exit strategies, for a scenario.

Complex scenario model composition is also possible with the help of Anything. An anything node can represent a subset of nodes in a path and is commonly used in UCM_{target} to capture the actual nodes that are specified in UCM_{source}. If an anything node is encountered

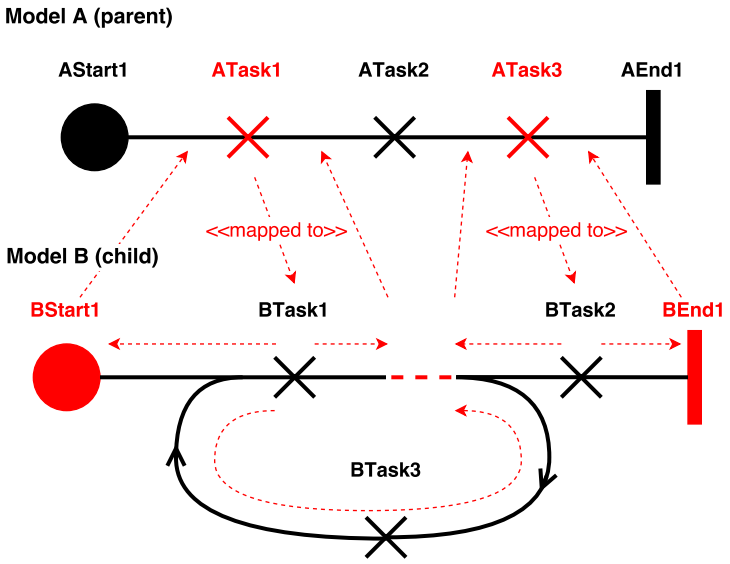


Figure 3.8: Schematic representation of extension with loop using anything node

during traversal, lines 30-34 and 54-58 signal the end of exploration and treat it as an end point. The difference is that the algorithm checks whether the anything node is still connected to other nodes before removal. This is necessary because an anything node has a predecessor node and a successor node, and typically surrounded by forks and joins to allow the insertion of loops in the base scenario (Figure 3.8). Both sides have to be traversed and dealt with before removing the anything node from the woven model.

Connecting Point Mapping

Mapping with connecting points allows for model extensions between parent and child UCMs and also model reuses from UCMs of other concerns. Algorithm 2 illustrates the procedure of weaving for connecting point mappings. The function *WeaveConnectingPointMapping* initiates the process by identifying the mapped connecting points (*from* UCM_{source} *to* UCM_{target}), and determining the type of composition to be performed based on whether the connecting points mapped from UCM_{source} are attached to a stub or not. If the mapped start and

Algorithm 2 Weaving Algorithm: Connecting Point Mapping

```
1: function WeaveConnectingPointMapping(ucm, composition)
2:     node_source ← get first node of composition mapping (from)
3:     node_target ← get second node of composition mapping (to)
4:     if node_source is StartPoint then
5:         if node_source is connected to a stub then
6:             call ExtendingStub_Outgoing(ucm, node_target, node_source)
7:         else if node_target is connected to a stub then
8:             call ReusingStub_Incoming(ucm, node_target, node_source)
9:         end if
10:    else if node_source is EndPoint then
11:        if node_source is connected to a stub then
12:            call ExtendingStub_Incoming(ucm, node_target, node_source)
13:        else if node_target is connected to a stub then
14:            call ReusingStub_Outgoing(ucm, node_target, node_source)
15:        end if
16:    end if
17: end function
18: function ExtendingStub_Outgoing(ucm, node_target, node_source)
19:    node_target_pred ← get predecessor node of node_target
20:    node_source_pred ← get predecessor node of node_source via stub connection
21:    node_source_succ ← get successor node of node_source via stub connection
22:    call MergePaths(node_target_pred, node_source_pred, node_source_succ, node_source, node_source)
23:    remove node_target from ucm
24: end function
25: function ReusingStub_Incoming(ucm, node_target, node_source)
26:    node_source_succ ← get successor node of node_source
27:    node_target_succ ← get successor node of node_target via stub connection
28:    node_target_pred ← get predecessor node of node_target via stub connection
29:    call SplitPaths(node_source_succ, node_target_succ, node_target_pred, node_target, node_source)
30:    remove node_source from ucm
31: end function
32: function ExtendingStub_Incoming(ucm node_target, node_source)
33:    node_target_succ ← get successor node of node_target
34:    node_source_succ ← get successor node of node_source via stub connection
35:    node_source_pred ← get predecessor node of node_source via stub connection
36:    call SplitPaths(node_target_succ, node_source_succ, node_source_pred, node_source, node_source)
37:    remove node_target from ucm
38: end function
```

```
39: function ReusingStub_Outgoing(ucm, node_target, node_source)
40:     node_source_pred ← get predecessor node of node_source
41:     node_target_pred ← get predecessor node of node_target via stub connection
42:     node_target_succ ← get successor node of node_target via stub connection
43:     call MergePaths(node_source_pred, node_target_pred, node_target_succ, node_target, node_source)
44:     remove node_source from ucm
45: end function
46: function SplitPaths(node_succ, node_succ', node_pred', node, node')
47:     if node_succ' is Stub then
48:         mark node_succ' as removable stub
49:         set successor node of node's stub connection to node_succ
50:     else if node_succ' is AndFork or OrFork then
51:         create node connection between node_succ' and node_succ
52:     else
53:         node_stub ← get successor node of node' via stub connection
54:         node_fork ← if node_stub is dynamic then create AndFork else OrFork
55:         place node_fork in between node_pred' and node_succ'
56:         create node connection between node_fork and node_succ'
57:         create node connection between node_fork and node_succ
58:         set successor node of node's stub connection to node_fork
59:     end if
60: end function
61: function MergePaths(node_pred, node_pred', node_succ', node, node')
62:     if node_pred' is Stub then
63:         mark node_pred' as removable stub
64:         set predecessor node of node's stub connection to node_pred
65:     else if node_pred' is AndJoin or OrJoin then
66:         create node connection between node_pred and node_pred'
67:     else
68:         node_stub ← get predecessor node of node' via stub connection
69:         node_join ← if node_stub is synchronizing then create AndJoin else OrJoin
70:         place node_join in between node_pred' and node_succ'
71:         create node connection between node_pred' and node_join
72:         create node connection between node_pred and node_join
73:         set predecessor node of node's stub connection to node_join
74:     end if
75: end function
```

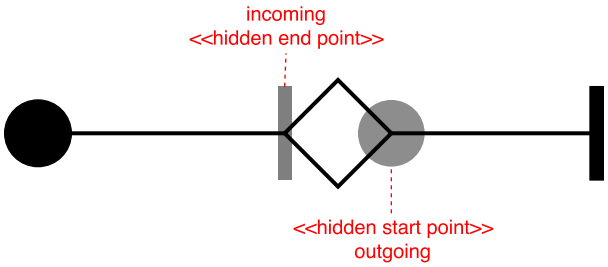


Figure 3.9: Example stub to indicate superimposed connecting points that are hidden

end points from UCM_{source} are attached to a stub (lines 5-6 and 11-12), it means that the connecting points are hidden (Figure 3.9) and belong to a stub in UCM_{source} and are mapped to actual end and start points of UCM_{target}, respectively (cross mapping). This type of composition is model extension. For model reuse, the mapping is the inverse, i.e., the stub belongs to UCM_{target}, and the actual start and end points from UCM_{source} are mapped to it (lines 7-8 and 13-14).

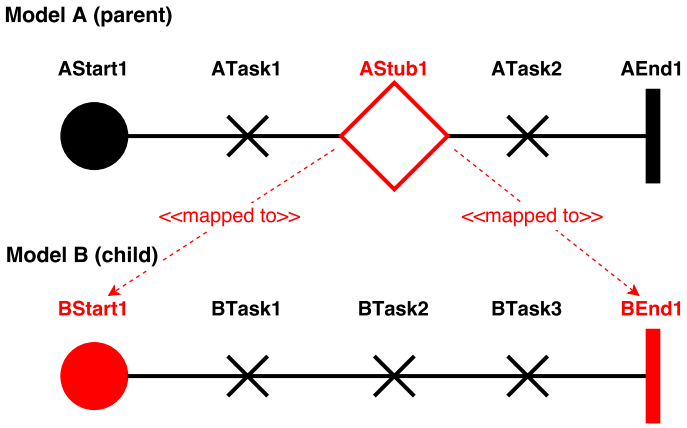


Figure 3.10: Schematic representation of extension through stub

Model extension for stubs work differently compared with responsibilities. No traversal is required since there is no need to explore the whole graph, but the composition specification requires exactly two connecting point mappings for each stub to be complete—one for the start point and the second for the end point (Figure 3.10). The weaver first obtains the

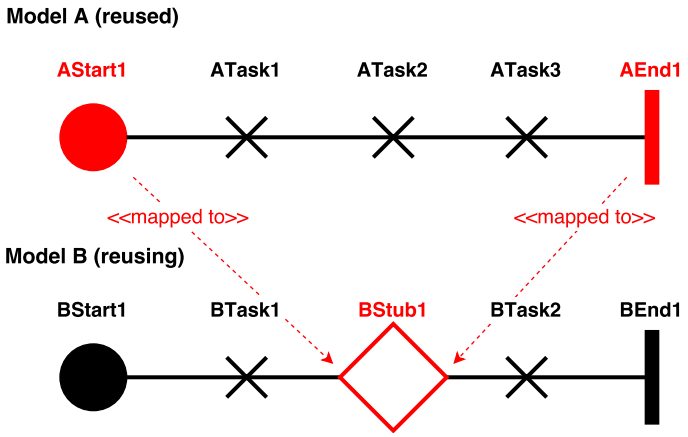


Figure 3.11: Schematic representation of reuse through stub

pair of mappings for the stub. The initial mapping usually maps the end point of a stub[§] to the start point of a UCM, and the weaver executes lines 32-38. The second mapping usually maps the start point of a stub[¶] to the end point of a UCM, and the weaver executes lines 18-24.

As for model reuse, the order of mapping is reversed—start and end points of UCM_{source} are mapped to the connecting points of a stub that is automatically generated in UCM_{target} when reusing UCM_{source} (Figure 3.11). To be precise, the automatically generated stub is always a static stub so that it can only hold a single UCM that originates from the reused concern. The weaver then executes lines 25-31 for mapping that maps from the start point of a UCM to the end point of a stub, and 39-45 for mapping that maps from the end point of a UCM to the start point of a stub.

The execution procedure for both extension and reuse involves replacing a stub with plug-ins (sub-UCMs). Depending on the type of stub, it can bind either a single plug-in or multiple plug-ins. When facing a single plug-in bound to a stub, the weaver simply connects the nodes

[§]The end point of a stub symbolizes incoming node connection to the stub.
[¶]The start point of a stub symbolizes outgoing node connection from the stub.

adjacent to the stub and nodes adjacent to the connecting points of a UCM, followed by the removal of the connecting points and the stub from the woven model (lines 47-49 and 62-64). If there are two plug-ins bound to a stub, the weaver creates branches to link the two UCMs as parallel paths via fork and join nodes (lines 52-59 and 67-74). The type of forks and joins being created is dependent on the type of stub. Synchronizing/blocking stubs produce branches that consist of AndFork and AndJoin, dynamic stubs produce branches that consist of AndFork and OrJoin, and static stubs produce branches that consist of OrFork and OrJoin. This process is also known as semantic flattening [15]. Additional plug-ins bound to a stub are linked via the created forks and joins (lines 50-51 and 65-66).

Chapter 4

Validation

The definition of UCM metamodel and the specification of weaving algorithm described in the previous chapter provide the foundation for the implementation of UCM in TouchCORE, a multitouch-enabled concern-oriented software design modeling tool. In this chapter, we illustrate the realization of scenario models in TouchCORE through the use of UCM notation in section 4.1. Then we attempt to validate our proposed approach of CoUCMs by means of case studies in section 4.2. Finally, we demonstrate that CoUCMs are able to cover the workflow patterns in section 4.3.

4.1 UCM Implementation in TouchCORE

TouchCORE is under active development within the Software Engineering Lab at McGill University [33]. The previous project, TouchRAM, successfully implemented the aspect-oriented software design paradigm, but support is limited to models expressed using the RAM modelling notation, which integrates class, sequence, and state diagrams in one language [31, 32]. TouchCORE extends TouchRAM with numerous enhancements, most notably the support for feature and goal modelling. While CORE also, in theory, supports multiple modelling languages, TouchCORE only supported design modeling using RAM. Since we have a well-defined corified UCM metamodel, we attempt to add support for CoUCM

in TouchCORE as proof of concept, enabling TouchCORE to build scalable and reusable scenario models.

The project uses Java SE Development Kit 8 as the implementation language and Eclipse Modeling Framework (EMF) [34] as the modeling facility for developing TouchCORE. To support a new language, we need to define its metamodel based on Ecore. TouchCORE already has a complete CORE metamodel defined with an Ecore model (see Figure A.1). With RAM as a reference model, we constructed an Ecore model that expresses our complete UCM metamodel, subclassing the appropriate CORE metaclasses, through the use of EMF tooling (see Figure A.2 in Appendix A for complete UCM metamodel). EMF is capable of generating structured Java code from valid Ecore models, allowing us to rapidly program the logic for UCM integration.

The software architecture of TouchCORE follows the model–view–controller (MVC) design pattern to separate the program into three main logical components. Figure 4.1 shows the three interconnected parts for the TouchCORE application: (i) the model layer for managing data, e.g., instances of RAM and UCM models; (ii) the TouchCORE graphical user interface (GUI) that constitutes the view layer for visualizing and manipulating models; and (iii) the controller layer for handling user interactions and act on the data model objects. The GUI for TouchCORE is built on top of MT4j for its multitouch capability [35]. Additional components include weaver, code generator, model validator, and classloader. The integration of UCM in TouchCORE involves modifying its core components with varying degrees, but the program is structured in such a way that we can add subcomponents when implementing a new modeling language, adhering to the open/closed principle.

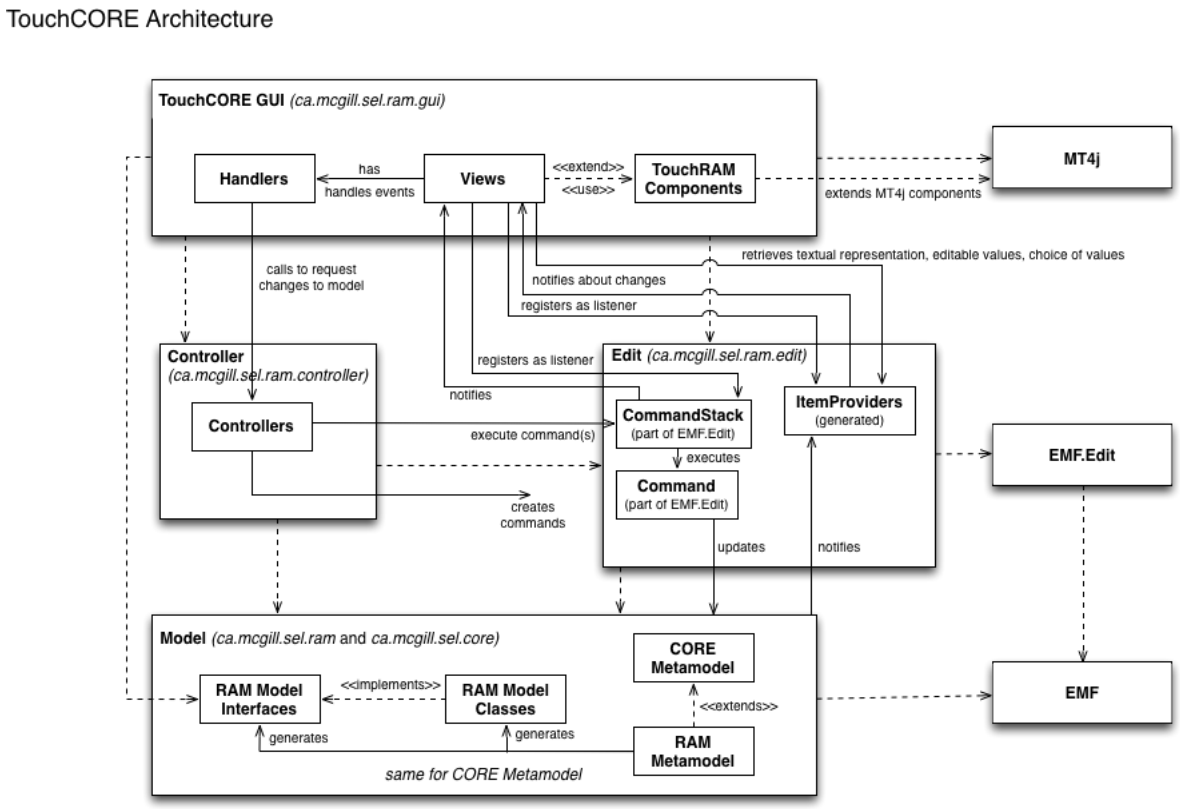


Figure 4.1: TouchCORE architecture. Image courtesy of Software Engineering Lab, McGill University

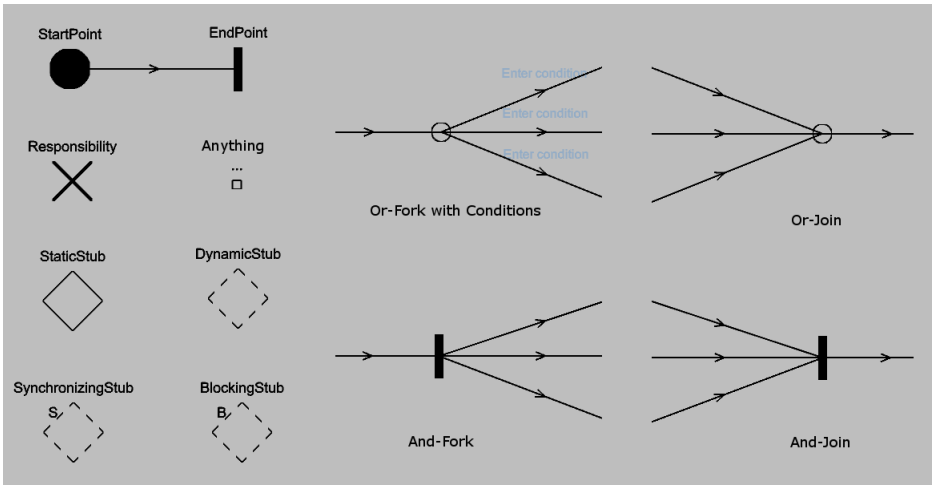


Figure 4.2: UCM notation in TouchCORE

4.1.1 Supported Concrete Syntax

The basic elements of the UCM notation that we implemented in TouchCORE are shown in Figure 4.2. Most of these elements are defined by the standards [15], with the exception of Anything that is taken from the extended AoUCM metamodel [28]. Users can create path nodes by tap-and-hold on the canvas of TouchCORE during runtime and a list of path nodes will be displayed for selection. To create a node connection between two path nodes, simply drag from the area adjacent of one element to the other element.

There are several anomalies with regards to the graphical representation of UCM symbols displayed in TouchCORE as compared with the standards (compare Figure 4.2 with Figure 2.7). For example, the symbol for OR-fork and OR-join is shown as a circle instead of no symbol (just direct branching and merging from the paths); anything is represented as a square with the label ... instead of just ...; and node connection is a straight line path instead of spline. These are some of the limitations that we faced at the moment when implementing the GUI. Our current method of creating nodes is to first create them on the canvas, then build the connections later. OR-fork and OR-join need a space to receive events

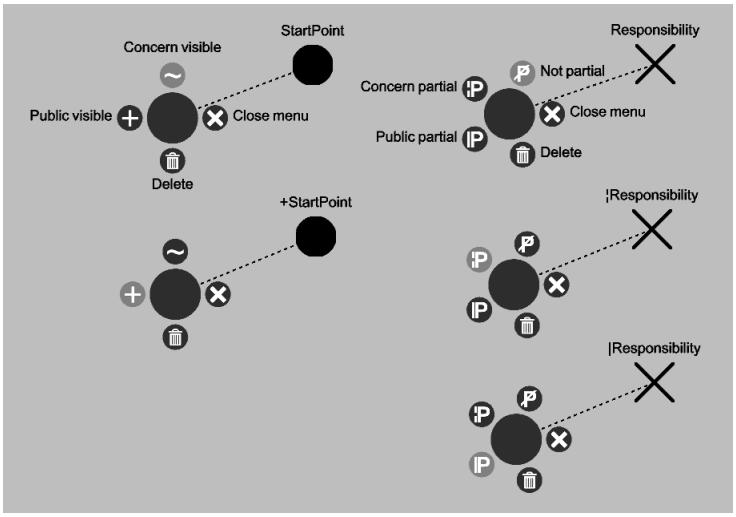


Figure 4.3: Visibility and partiality

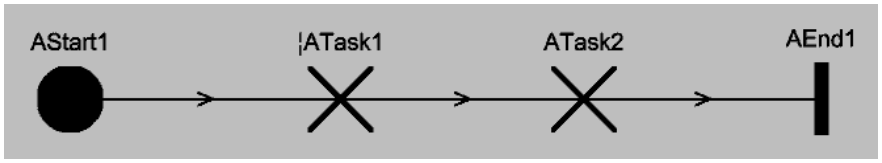
from the user, thus a circle serves as the area of interactivity as well as a statement of presence that an OR-fork or OR-join has been created. The idea of displaying the ... symbol of an anything node is that it should be part of the node connection and move along seamlessly with correct orientation whenever the predecessor or successor node of anything is moved, but since anything is considered a path node, we decided for now to just use a square with the label ... to represent the anything node. Lastly, spline drawing is not yet available in TouchCORE so we use straight lines for the time being.

Elements with extra features can be accessed by tap-and-hold an element (Figure 4.3). We allow both start and end points to set their visibility. By default, all path nodes are *concern visible*, but start and end points can switch to *public visible* (see Section 3.2.1 for visibility discussion). Likewise, we allow responsibilities to set their partiality. By default, all path nodes are not partial, meaning they are well-defined and require no further action. Since we have customization mappings for responsibility, we can specify whether a responsibility is partially defined, i.e., requires a composition to be semantically complete. A responsibility

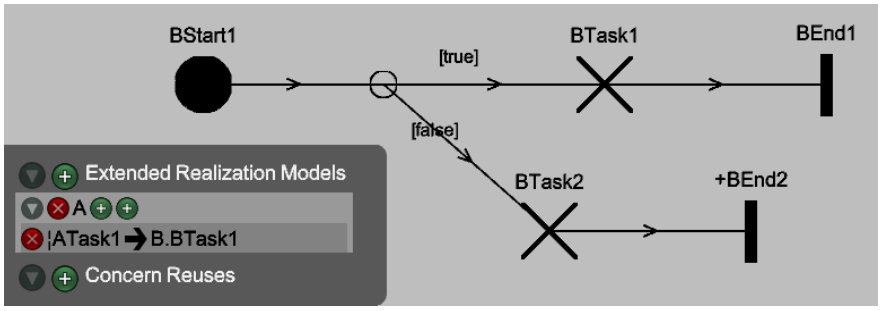
that is *concern partial* should be completed through model extension, whereas a responsibility that is *public partial* should be completed through model reuse.

4.1.2 Scenario Model Composition

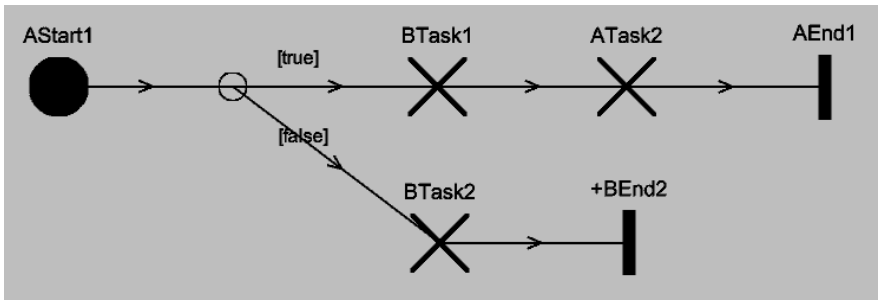
Model Extension



(a) Model A - parent UCM



(b) Model B - child UCM



(c) Woven Model B_A

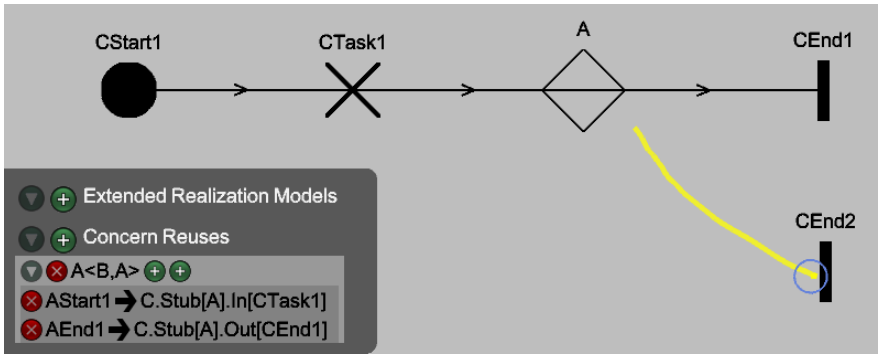
Figure 4.4: Schematic representation of model extension

Figure 4.4 illustrates the usage of UCM model extension within a concern. Given a concern with two features in a hierarchy, the model of a child feature (Model B) extends

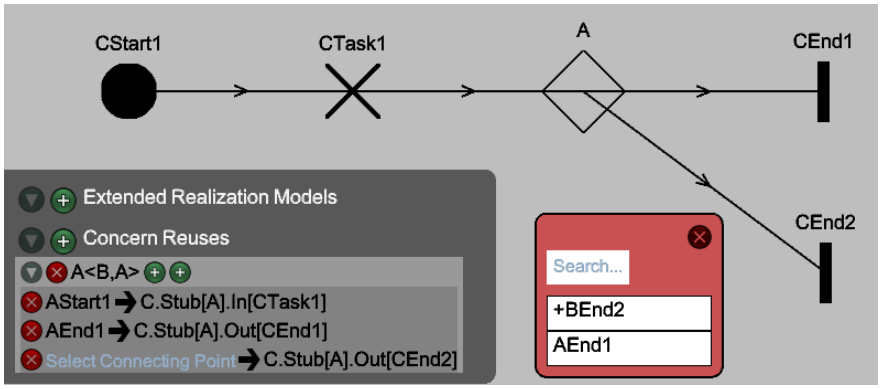
the model of a parent feature (Model A). Composition specifications are specified in Model B, where an element of Model A is mapped to an element of Model B. Multiple mappings can be set per extension as needed and the available types of mapping are defined in the metamodel. The result of weaving Model B to Model A is depicted in Figure 4.4c. Based on the mappings set in Model B, the predecessors and successors of the mapped responsibility from Model B are introduced as adjoined path nodes of the mapped responsibility from Model A, and the mapped responsibility from Model A is being replaced with the mapped responsibility from Model B.

Model Reuse

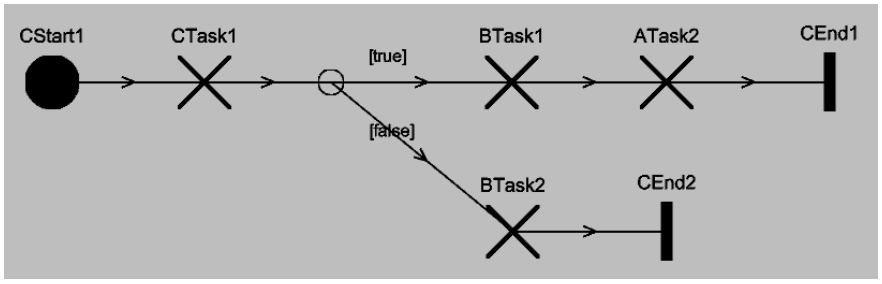
Figure 4.5 illustrates the usage of UCM model reuse across concerns. Given that Model C of a concern reuses Concern A, the configuration for the set of features of Concern A will be displayed. Here, we chose to use features A and B, thus woven Model B_A (see Figure 4.4c) is generated and represented as a static Stub A (appears automatically in canvas after successful reuse). Mappings for connecting points of Stub A can be established by linking a path node to/from Stub A. As shown in Figures 4.5a and 4.5b, a node connection was created from Stub A to an end point, and a list of end points from woven Model B_A will be displayed for the user to set which end point of Model B_A corresponds to which outgoing connection of Stub A in Model C. We label the incoming connection of a stub as `<Model_2>.Stub[<Model_1>].In[<Predecessor>]`, and the outgoing connection as `<Model_2>.Stub[<Model_1>].Out[<Successor>]`. The result of weaving Model B_A into Model C is depicted in Figure 4.5c.



(a) Model C - reuse Concern A with selected features <A,B>



(b) Model C - establish connecting point mapping through node connection



(c) Woven Model C_A<A,B>

Figure 4.5: Schematic representation of model reuse

4.2 Case Studies

In this section, we attempt to validate our proposed technique for CoUCMs with two case studies: Authentication and Online Payment. We chose these two examples as our case studies because they provide different yet appropriate level of complexity to the problem that we are studying, exemplify model extension within the Online Payment concern, as well as showcase the reuse of the Authentication concern within the Online Payment concern.

4.2.1 Authentication

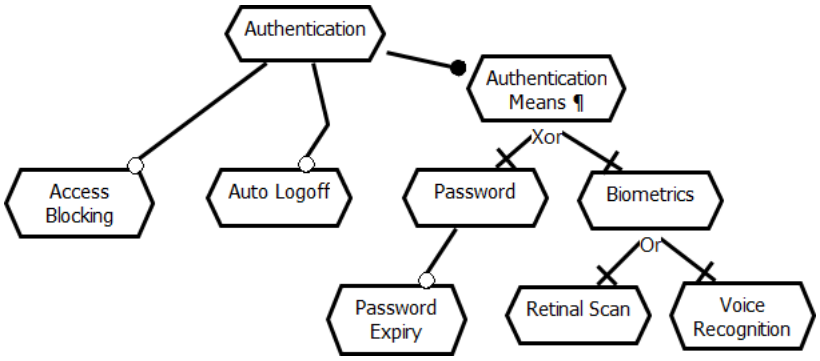


Figure 4.6: Feature model for Authentication (jUCMNav). Image courtesy of Nishanth Thimmegowda et al. [36]

We design the Authentication concern based on a reference model that we have previously described in jUCMNav format [36]. Figure 4.6 shows all the available features that are supported for the concern. *Authentication* has a mandatory *Authentication Means* feature that may either be *Password* that can be extended with the optional *Password Expiry* feature, or *Biometrics* that requires at least *Retinal Scan* or *Voice Recognition*. If necessary, consecutive unsuccessful authentication attempts may result in *Access Blocking* and long idle period may lead to *Auto Logoff*.

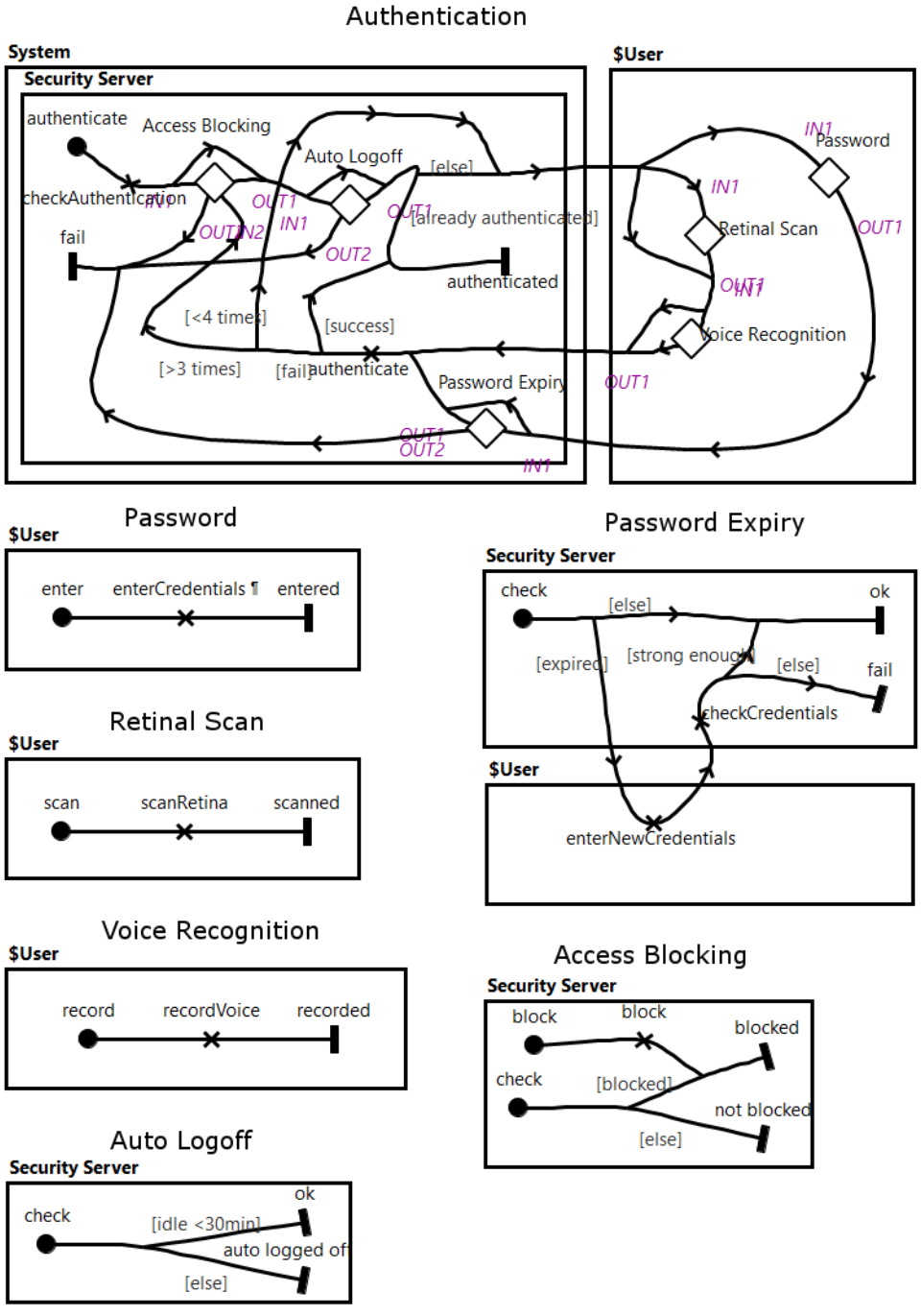


Figure 4.7: Scenario models for Authentication (jUCMNav). Image courtesy of Nishanth Thimmegowda et al. [36]

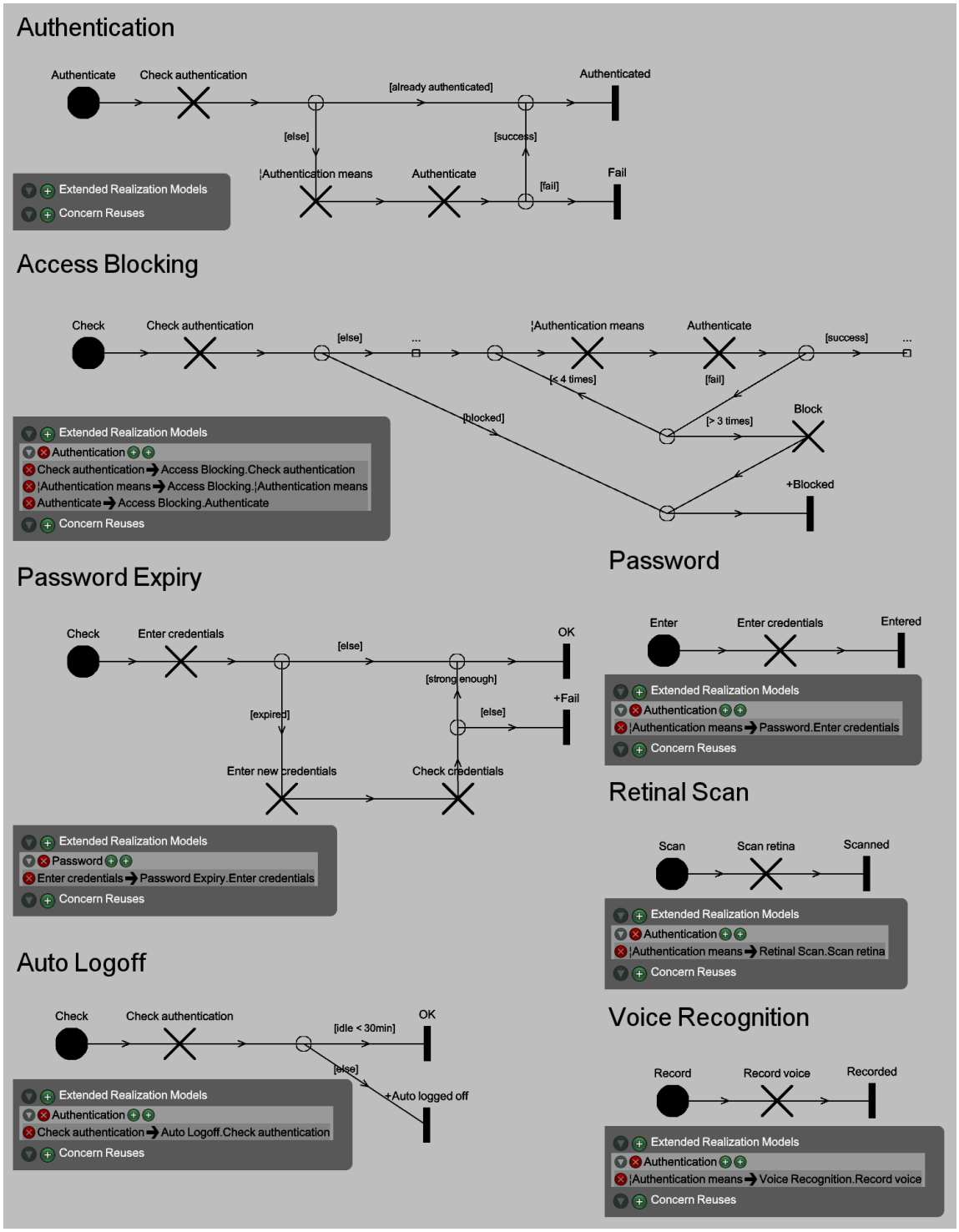


Figure 4.8: Scenario models for Authentication (TouchCORE)

Scenario models can be (optionally) realized for the features to describe how the user would interact with the Authentication concern. Figure 4.7 illustrates the UCM diagrams and plug-ins that are realized for most of the features. Then, with slight modifications to the UCMs specified in jUCMNav, we developed our version of the UCMs using TouchCORE as depicted in Figure 4.8. (Feature model remains unchanged and TouchCORE version of the feature model is omitted.)

Notice that in the root map of Figure 4.7, each feature is represented as a static stub and is bound to a plug-in for the feature. In the root map developed using TouchCORE (see Figure 4.8), we minimize the usage of stubs and instead utilize model extensions, successfully isolating the aspects that crosscut the main concern. Since *Authentication Means* is a mandatory feature, we introduce a responsibility placeholder and set its partiality to *concern partial*. Any UCMs under the *Authentication Means* feature can extend the root UCM via responsibility mapping. One advantage of using CORE approach in modeling UCMs is that by selecting the desired features when reusing this concern, only the UCMs of those selected features will be composed into the root map and a single UCM that consists of only the necessary paths will be generated. The woven UCM can be reused in another concern such as Online Payment.

4.2.2 Online Payment

The Online Payment concern offers a means to build a payment model for an e-commerce platform. Use cases for Online Payment are adapted from the W3C Web Payments Interest Group [37], focusing on the payment schemes in use today. Figure 4.9 shows all the available features that are supported in the concern. Online Payment provides numerous payment methods for the customers to pay by credit card (e.g., Visa, MasterCard, China

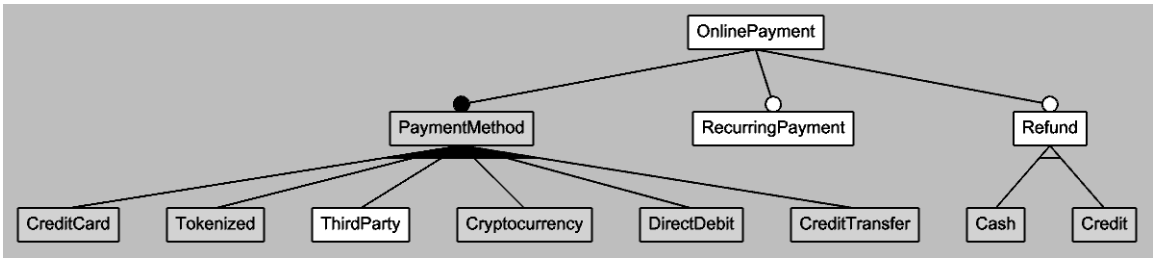


Figure 4.9: Feature model for Online Payment

UnionPay), tokenized payment (e.g., ApplePay, Venmo, CyberSource), third-party payment (e.g., PayPal, Alipay, Google Pay), cryptocurrency (e.g., Bitcoin, Ripple, Ethereum), direct debit, or credit transfer. Optionally, the system supports a recurring payment option to handle subscription plans, and can also process refunds to the payer's payment instrument or store credit.

Figure 4.10 illustrates a typical workflow for Online Payment. In the root map, we have a single entry and exit point—from checkout to transaction complete—and a loop that redirects the user back to the payment selection if payment authorization fails. The payment method selection is modeled as a dynamic stub to receive multiple plug-ins from the subfeatures of *PaymentMethod*. We limit the discussion of *PaymentMethod* to *ThirdParty* since the scenario model for paying through third-party is sufficient to describe the essential features, as most payment methods share a similar scenario.

There are two things worth mentioning in the *ThirdParty* model. First, *ThirdParty* extends *OnlinePayment* through the *Select payment method* dynamic stub, hence mapping is done via connecting points. Second, *ThirdParty* reuses the Authentication concern, which is depicted as a static stub, and the selected features including the root feature are *Authentication*, *Password* and *Auto Logoff*, with *Access Blocking* and *Password Expiry* delayed for future decision (Figure 4.11a). The one incoming connection and three outgoing connecti-

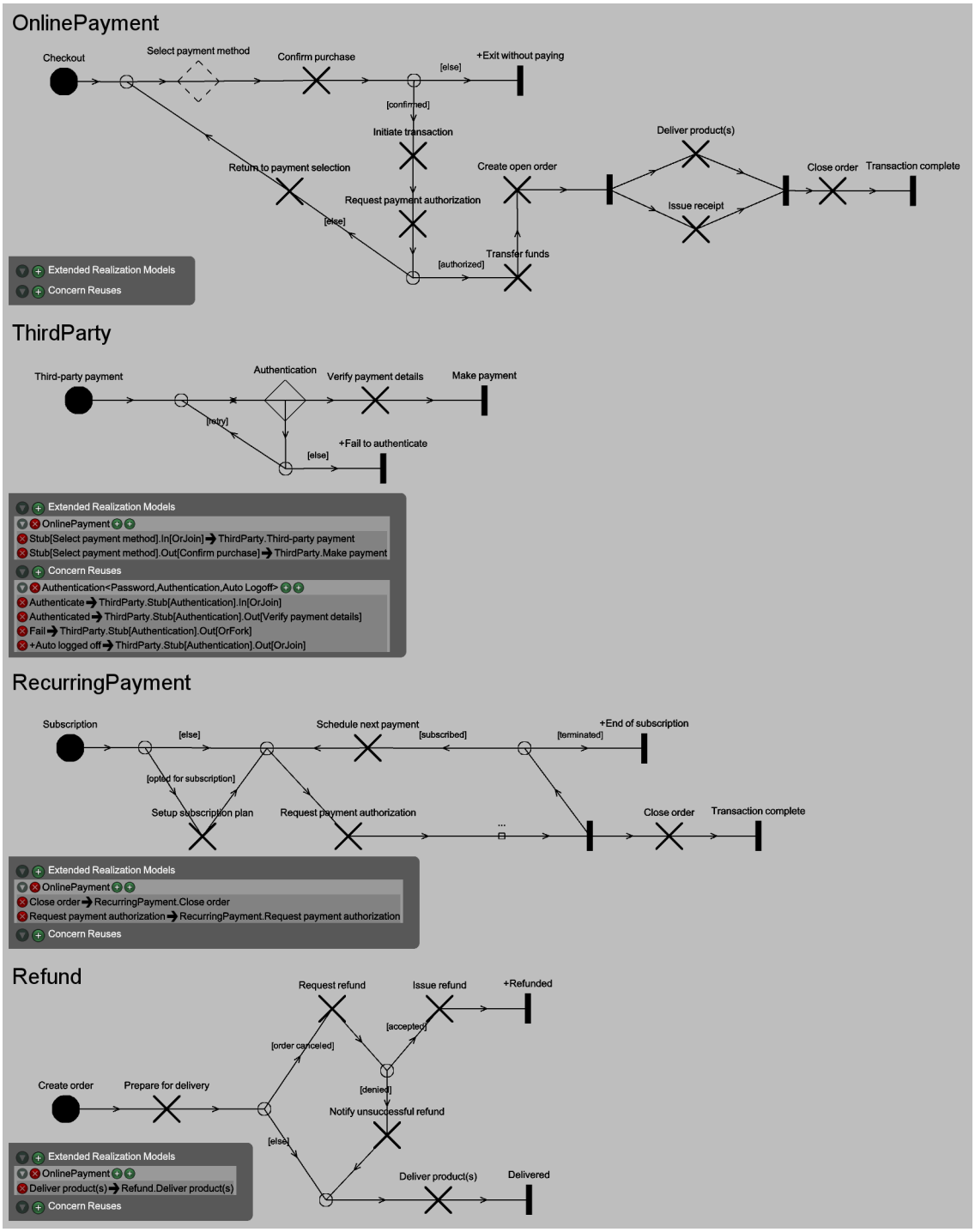
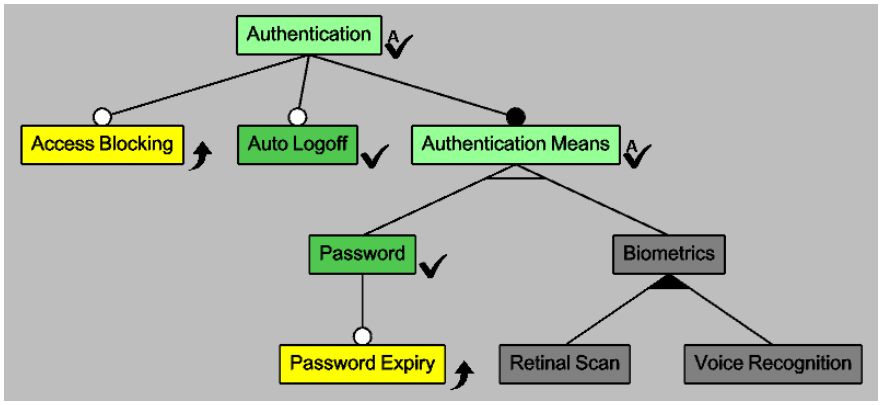
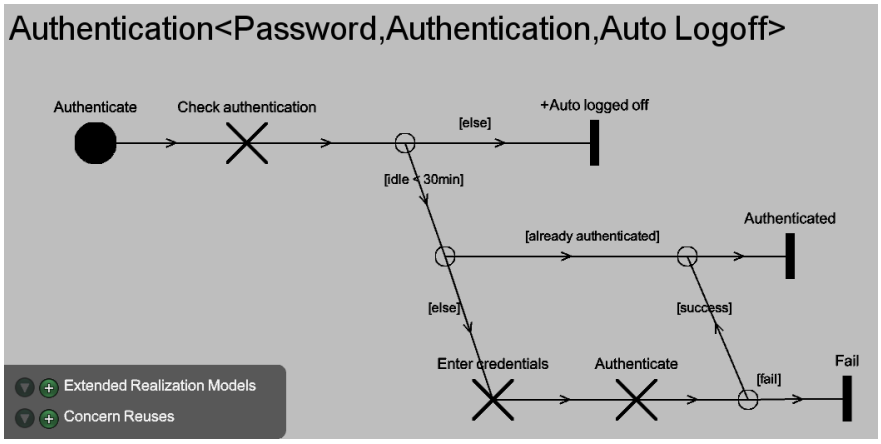


Figure 4.10: Scenario models for Online Payment



(a) Feature selection of Authentication concern



(b) Woven Authentication UCM based on selected features

Figure 4.11: Reused Authentication in ThirdParty model

ons of the *Authentication* stub are associated with the single start point (*Authenticate*) and three end points (*Authenticated*, *Fail*, and *Auto logged off*) of the reused Authentication UCM (Figure 4.11b).

Optional features of *OnlinePayment* are *RecurringPayment* and *Refund*. Both of the features extend *OnlinePayment*. For *RecurringPayment*, the Anything node represents the sequence of nodes on the path of *OnlinePayment*—from *Request payment authorization* to *Close order*—and a loop to enable recurring payment is injected in between the two responsibilities. The *Refund* model we defined here is restricted to the refund policy that allows customers to request for refund after they made the payment, but prior to receiving the goods. Refund after the delivery of product(s) requires a separate UCM and is outside the scope of this case study.

The purpose of these case studies is to demonstrate the application of model reuses, such as the reuse of the Authentication concern in the *ThirdParty* UCM model, as well as model extensions via responsibility mappings and connecting point mappings. Successful application of extensions and reuses allows for the development of scalable and reusable scenario models through TouchCORE. Concerns can be as fine-grained as Authentication, or intermediate concerns that reuse Authentication such as Online Payment, up to a proper application (e.g., electronic commerce websites) that reuses Online Payment.

4.3 Workflow Patterns

This last section demonstrates the use of CoUCMs to implement some of the workflow patterns described by van der Aalst et al. [38]. We chose to cover two of the state-based patterns—*Deferred Choice* and *Milestone*—as they present the appropriate level of complexity, given that some of the workflow patterns are primitive and already supported by the

standard UCM notations, as well as the constraints imposed by our partial implementation of the UCM notations in TouchCORE.

4.3.1 Deferred Choice

The deferred choice pattern allows the moment of choice to be suspended as late as necessary—process can only continue based on external factors. In essence, all branches represent possible future courses of execution. Only once the decision has been made to proceed with a particular branch, execution for the other branches come to a halt.

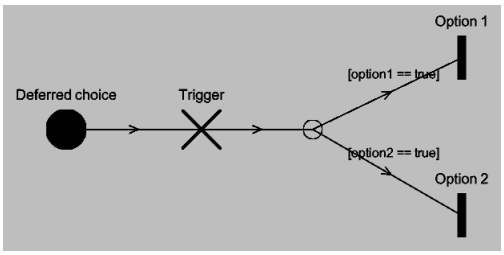


Figure 4.12: Deferred choice pattern

Typical implementation of deferred choice is using an AND-fork to enable all parallel branches. After one of the branches has started processing, all other branches are canceled. Since the UCM notation does not have the ability to signal for cancellation of other branches, an alternative strategy is the use of XOR-split. Figure 4.11 illustrates the OR-fork implementation for the deferred choice pattern; the *Trigger* is responsible for activating the proper branch, by setting $option_1$ to *true* and $option_2$ to *false* or vice versa. The pattern is realized as a feature in a concern and can be reused in other UCMs. One example of reuse is in the milestone pattern.

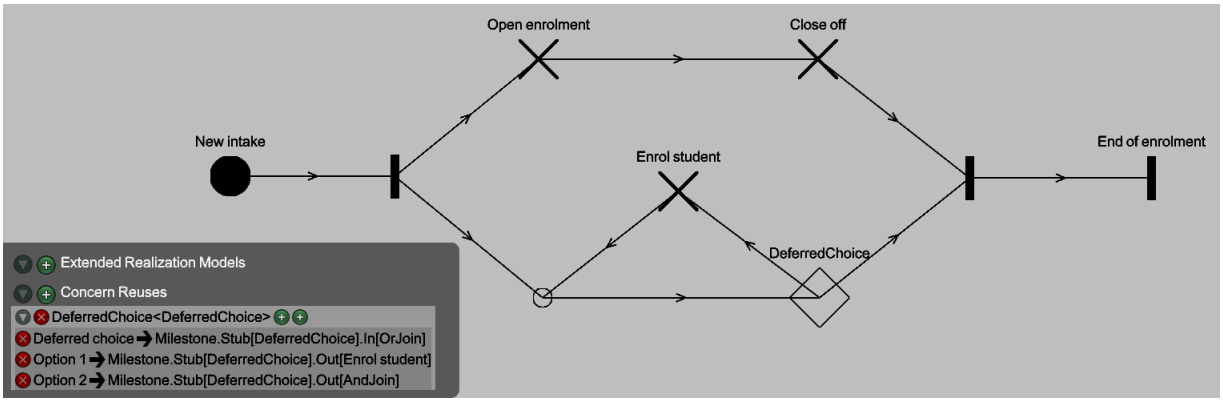


Figure 4.13: Milestone pattern (enrolment example)

4.3.2 Milestone

The milestone pattern supports the conditional execution of a task only if a parallel process is in a given state, i.e. an activity can only be enabled if a certain milestone has been reached and has not expired yet. Different strategies exist for the implementation of the milestone pattern, and one form uses a deferred choice in the workflow. Deferred choice offers two subsequent activities and is modeled with an OR-fork within the reused Deferred Choice concern. The path to one activity is enabled only after reaching a milestone, after which the path merges prior to the deferred choice construct and the same activity can be executed repeatedly, given that the current state is still in the milestone. On the other hand, if the current state leaves the milestone, then the path to the first activity is disabled by the OR-fork, leaving only the path to the second activity.

Whereas the deferred choice pattern is modeled as a reusable concern, the milestone pattern is implemented slightly differently. We took an example of student enrolment, applying the milestone pattern (deferred choice implementation), as shown in Figure 4.12. New enrolments are being accepted when the enrolment period opens (at the point of reaching a milestone) until the enrolment period closes (at the point of deadline) for a given intake.

Ideally, the route to *Enrol student* ($option_1$) can only be activated when the token on the other parallel path reaches *Open enrolment* but before reaching *Close off*. All other instances would result in inaccessible path $option_1$, leading to the only exit path available that is *End of enrolment* ($option_2$).

I don't understand in your example how you specify that the "trigger" activity within deferred choice is going to check whether the enrolment is open or not, and set the correct conditions (option 1 or option 2) for the paths. You don't seem to map "trigger" in your reuse.

Chapter 5

Conclusion

5.1 Summary

Requirements elicitation forms a fundamental piece of software engineering and is typically performed at the initial phase of the software development process. This thesis introduces CoUCM that consolidates scenario modeling with advanced SoC, MDE, and SPL. Along with the already existing GRL support for goal modeling in CORE, the addition of UCM offers a complete URN package for requirements engineering in CORE.

The proof of concept implementation of CoUCM in TouchCORE demonstrates the project's feasibility. TouchCORE is a modeling tool that provides an intuitive interface for concern-oriented software design. Both CoUCM and RAM allow TouchCORE users to model at different level of abstractions, covering the requirements and design phases. In addition, modelers can now populate the TouchCORE library with reusable scenario models encoding essential recurring requirements concerns (e.g., functional units, workflow patterns, etc).

Limitations of the work exists, however, and serve as potential future work for improvements, which we discuss in the next section.

5.2 Future Work

The current UCM metamodel that we support in CORE does not cover all the model elements defined in the UCM standard. In particular, the use of ResponsibilityRefs should refer to a particular Responsibility definition from multiple reference points that belong to other UCMs, as well as the use of Components to model the architectural structure of a system. Implementation of Component to CoUCM poses some difficulties especially when taking model composition into account as responsibilities within a component are bound to the component; we may also have to support mappings of components and this adds complexity to the weaving algorithm. Several other model elements including waiting place, timer, failure point, and abort could be added to the CoUCM metamodel to complete the standard UCM features.

Similarly, the implementation of scenario modeling to TouchCORE is in the alpha phase. TouchCORE features such as traceability and model validation could be implemented to allow for a better scenario modeling experience. Path drawing could be improved as current implementation uses straight lines to connect path nodes; splines would work well if supported by TouchCORE GUI. One of the jUCMNav tool's features is the path traversal mechanism [39]. If implemented in TouchCORE, this mechanism allows for UCM analysis and is particularly useful in evaluating scenario variables when traversing paths.

Supplementary work to the CORE base design is needed to allow a more seamless integration of multiple modeling languages. This leads to the question that begs to be investigated—whether actual MDE, i.e., software development with models at multiple levels of abstraction and model transformations that connect them, is compatible with CORE. Since this is one of the early works (after RAM) that extends a modeling language (UCM) with concern-orientation, future addition of modeling languages to CORE can refer to this work as refe-

rence. We hope that this work would motivate future studies to further improve the CORE paradigm.

Appendix A

Complete Metamodels

A.1 CORE Metamodel

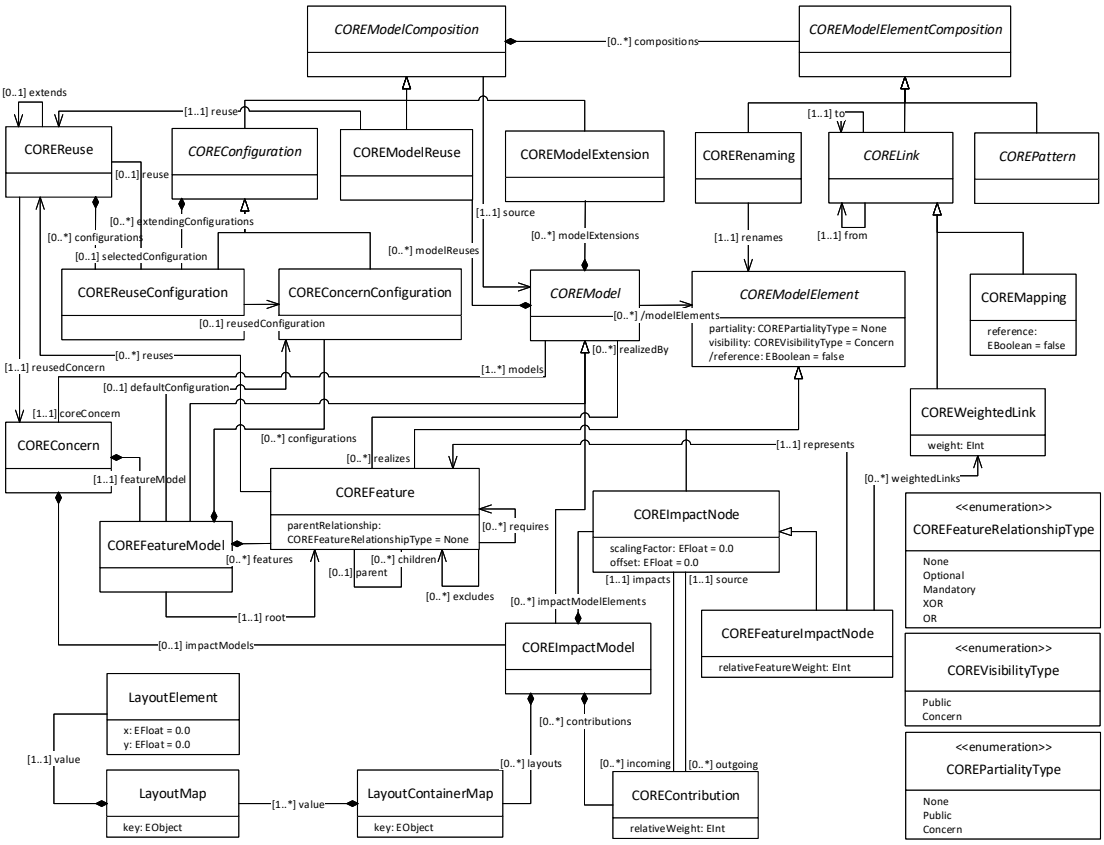


Figure A.1: Abstract grammar: CORE metamodel overview

A.2 CoUCM Metamodel

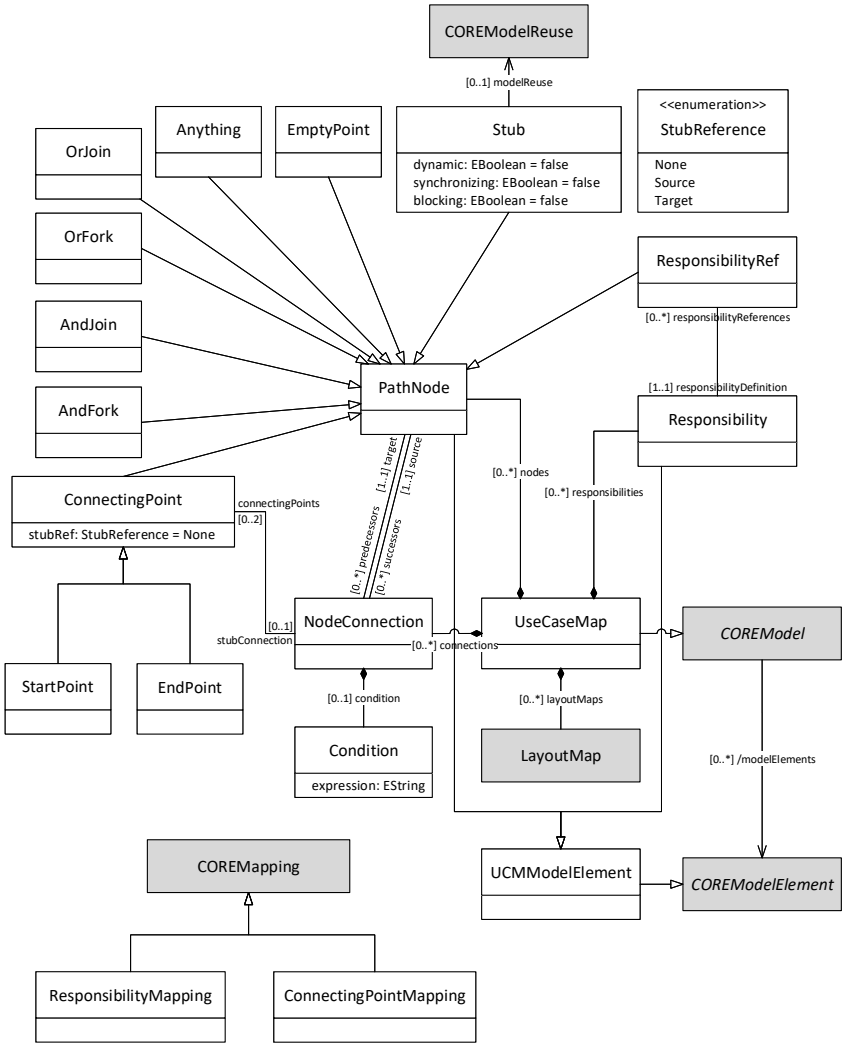


Figure A.2: Abstract grammar: CoUCM metamodel overview

References

[1] Douglas C Schmidt. "Model-driven engineering". In: *COMPUTER-IEEE COMPUTER SOCIETY-* 39.2 (2006), p. 25.

[2] Shane Sendall and Wojtek Kozaczynski. "Model transformation: The heart and soul of model-driven software development". In: *IEEE software* 20.5 (2003), pp. 42–45.

[3] Marc Eaddy et al. "Do crosscutting concerns cause defects?" In: *IEEE transactions on Software Engineering* 34.4 (2008), pp. 497–515.

[4] Robert B France et al. "Repository for model driven development (ReMoDD)". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 1471–1472.

[5] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. "Concern-oriented software design". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2013, pp. 604–621.

[6] Jörg Kienzle et al. "Aspect-oriented design with reusable aspect models". In: *Transactions on aspect-oriented software development VII* (2010), pp. 272–320.

[7] Daniel Amyot and Gunter Mussbacher. "URN: Towards a new standard for the visual description of requirements". In: *International Workshop on System Analysis and Modeling*. Springer. 2002, pp. 21–37.

[8] Edsger Wybe Dijkstra et al. *A discipline of programming*. Vol. 1. prentice-hall Englewood Cliffs, 1976.

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

[10] Bernard Coulange. *Software reuse*. Springer Science & Business Media, 2012.

[11] Charles W Krueger. "Software reuse". In: *ACM Computing Surveys (CSUR)* 24.2 (1992), pp. 131–183.

[12] David Lorge Parnas. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058.

[13] Peri Tarr et al. "N degrees of separation: Multi-dimensional separation of concerns". In: *Proceedings of the 21st international conference on Software engineering*. ACM. 1999, pp. 107–119.

[14] Kyo C Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[15] Z ITU-T. "151 User requirements notation (URN)–Language definition". In: *ITU-T, Oct* (2012).

[16] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. "Staged configuration through specialization and multilevel configuration of feature models". In: *Software Process: Improvement and Practice* 10.2 (2005), pp. 143–169.

[17] Mustafa Berk Duran and Gunter Mussbacher. "Investigation of feature run-time conflicts on goal model-based reuse". In: *Information Systems Frontiers* 18.5 (2016), pp. 855–875.

[18] Jörg Kienzle et al. "Delaying decisions in variable concern hierarchies". In: *ACM SIGPLAN Notices*. Vol. 52. 3. ACM. 2016, pp. 93–103.

[19] Mustafa Berk Duran et al. "On the reuse of goal models". In: *International SDL Forum*. Springer. 2015, pp. 141–158.

[20] Mustafa Berk Duran, Aldo Navea Pina, and Gunter Mussbacher. "Evaluation of reusable concern-oriented goal models". In: *Model-Driven Requirements Engineering Workshop (MoDRE), 2015 IEEE International*. IEEE. 2015, pp. 1–10.

[21] Romain Alexandre et al. "Support for Evaluation of Impact Models in Reuse Hierarchies with jUCMNav and TouchCORE." In: *P&D@ MoDELS*. 2015, pp. 28–31.

[22] Gunter Mussbacher et al. "Assessing composition in modeling approaches". In: *Proceedings of the CMA 2012 Workshop*. ACM. 2012, p. 1.

[23] Omar Alam, Matthias Schöttle, and Jörg Kienzle. "Revising the Comparison Criteria for Composition." In: *CMA@ MoDELS*. 2013.

[24] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. "Aspect-oriented multi-view modeling". In: *Proceedings of the 8th ACM international conference on Aspect-oriented software development*. ACM. 2009, pp. 87–98.

[25] Daniel Amyot. "Introduction to the user requirements notation: learning by example". In: *Computer Networks* 42.3 (2003), pp. 285–301.

[26] Ray JA Buhr and Ron S Casselman. *Use case maps for object-oriented systems*. Prentice-Hall, Inc., 1995.

[27] Raymond JA Buhr. "Use case maps as architectural entities for complex systems". In: *IEEE Transactions on Software Engineering* 24.12 (1998), pp. 1131–1155.

[28] Gunter Mussbacher. "Aspect-oriented user requirements notation". PhD thesis. University of Ottawa (Canada), 2011.

[29] Jacques Klein and Jörg Kienzle. "Reusable aspect models". In: *Abstract book of 11th Workshop on Aspect Oriented Modeling, AOM at Models' 07*. 2007.

[30] Jason Kealey and Daniel Amyot. "Enhanced use case map traversal semantics". In: *International SDL Forum*. Springer. 2007, pp. 133–149.

[31] Wisam Al Abed et al. "TouchRAM: A Multitouch-Enabled Tool for Aspect-Oriented Software Design." In: *SLE* 2012 (2012), pp. 275–285.

[32] Matthias Schöttle et al. "TouchRAM: a multitouch-enabled software design tool supporting concern-oriented reuse". In: *Proceedings of the companion publication of the 13th international conference on Modularity*. ACM. 2014, pp. 25–28.

[33] *TouchCORE*. URL: http://touchcore.cs.mcgill.ca/. Feb. 2018.

[34] Dave Steinberg et al. *EMF: eclipse modeling framework*. Pearson Education, 2008.

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

[36] Nishanth Thimmegowda et al. "Concern-Driven Software Development with jUCMNav and TouchRAM." In: *Demos@ MoDELS*. 2014.

[37] Ian Jacobs et al. "Web payments use cases 1.0". W3C Working Draft 30 July 2015. URL: https://www.w3.org/TR/web-payments-use-cases/.

[38] Wil MP van Der Aalst et al. "Workflow patterns". In: *Distributed and parallel databases* 14.1 (2003), pp. 5–51.

[39] Jason Kealey. "Enhanced use case map analysis and transformation tooling". PhD thesis. University of Ottawa (Canada), 2007.