Thesis Title

Cheuk Chuen Siow

School of Computer Science

McGill University, Montréal

January 2018

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

Abstract

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].

Excellent intro up to here. I'd introduce CORE here more as a paradigm rather than a modeling technique. For example:

JK replaced: Concern-Oriented Reuse (CORE) introduces a modeling technique that focuses on concerns as units of reuse [5]. CORE enables large-scale model reuse by utilizing the ideas of MDE, Separation of Concerns (SoC), and Software Product Lines (SPL). This is possible because CORE uses aspect-oriented modeling techniques to enable developers to build complex models by incrementally composing smaller, simpler models. by 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 Model-Driven Engineering (MDE), 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 behaviour, 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 JK replaced: by in order to fully integrate CORE with MDE, models JK replaced: of by typically used in other development phases JK replaced: can by should also be supported JK replaced: by integrating with the CORE metamodel by to allow them also to benefit from advanced modularization and reuse support.

This thesis focuses on JK replaced: by adding support for CORE to models at the requirements phase, i.e., models that are typically built earlier than design models. JK replaced: , and by 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], we JK replaced: are interested in examining the possibility of having by concentrated in this thesis on integrating scenario models JK replaced: as part of the CORE toolkit. The goal of this thesis is to determine how UCMs can be integrated by with the concepts of CORE.

This last sentence I would move to future work at the end of the conclusion chapter

JK replaced: 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. by

1.1 Contributions

This thesis advances the state-of-the-art in modelling by proposing a complete solution for augmenting the use case maps modelling notation with concern-oriented reuse capabilities. Specifically, the thesis makes the following contributions:

I just filled in some items, but there might be more, and each of them needs a little bit more explanation about what the contribution is and why it is important

- UCM metamodel integration with CORE
- Definition of UCM weaving algorithm compatible with CORE extension and reuse composition
- Validation of feasibility of proposed solution by implementing metamodel and algorithm in TouchCORE

- Validation of expressiveness of corified UCMs and demonstration of reuse potential by modelling the xxx. case studies

1.2 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 offers background information on CORE and UCM. Chapter 3 presents the integration of CORE with UCM. Chapter 4 validates the resulting integration process. Finally, Chapter 5 concludes the thesis and discusses future work.

Where are you talking about related work? It must be mentioned in the outline as well.

Chapter 2

Background

CORE builds on three key components—MDE, SoC, and SPL—to support large-scale model reuse. We are also interested in studying the early phases of software development, where the requirements of the software to be built are elaborated. In this chapter, we provide an overview of CORE as a reuse technique with the current state of development in Section 2.1. Then we describe UCM as part of the requirements engineering tool and its use in specifying scenarios in Section 2.2.

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

You can reuse some of our text from previous papers here. Focus on what is really interesting for us in this thesis, i.e., interfaces, features + realization models, customization mappings, weaving algorithm. You should also show the CORE metamodel here (or at least a simplified version of it that is used as a basis for the integration in chapter 3. You can use some of the text from our SoSyM paper.

.

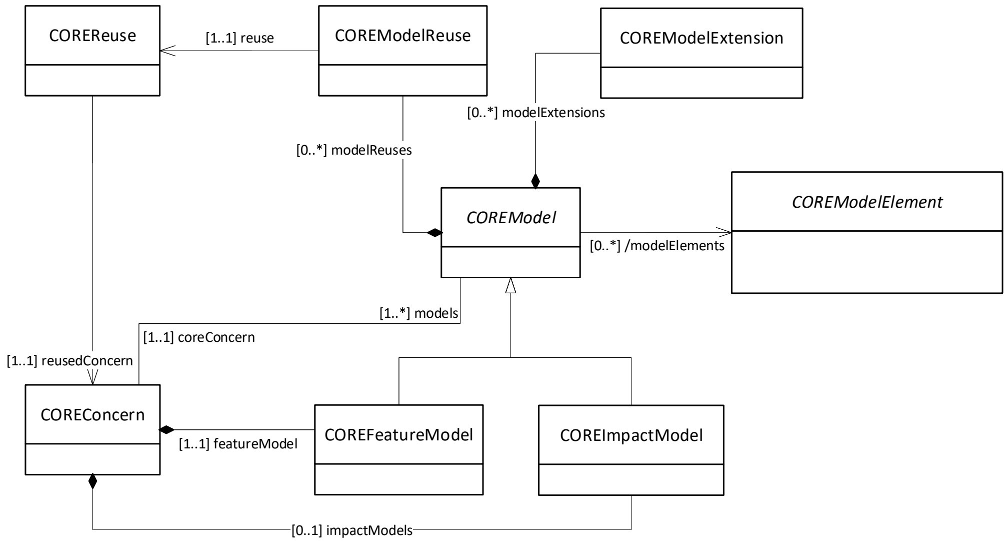


Figure 2.1: CORE metamodel: basic structure of a concern

2.1.1 Concern Interfaces

A concern groups related models serving the same purpose, and provides 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.

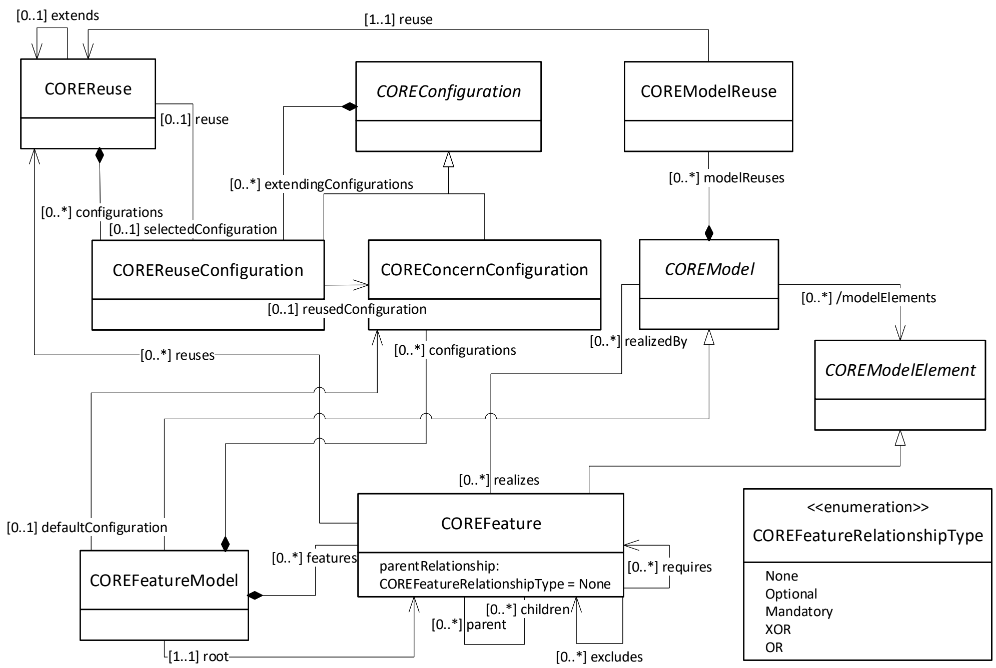


Figure 2.2: CORE metamodel: variation interface - features

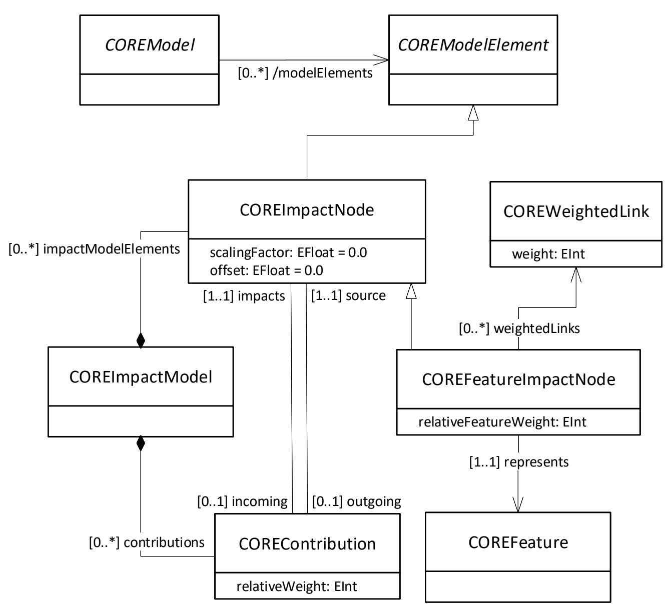


Figure 2.3: CORE metamodel: variation interface - impacts

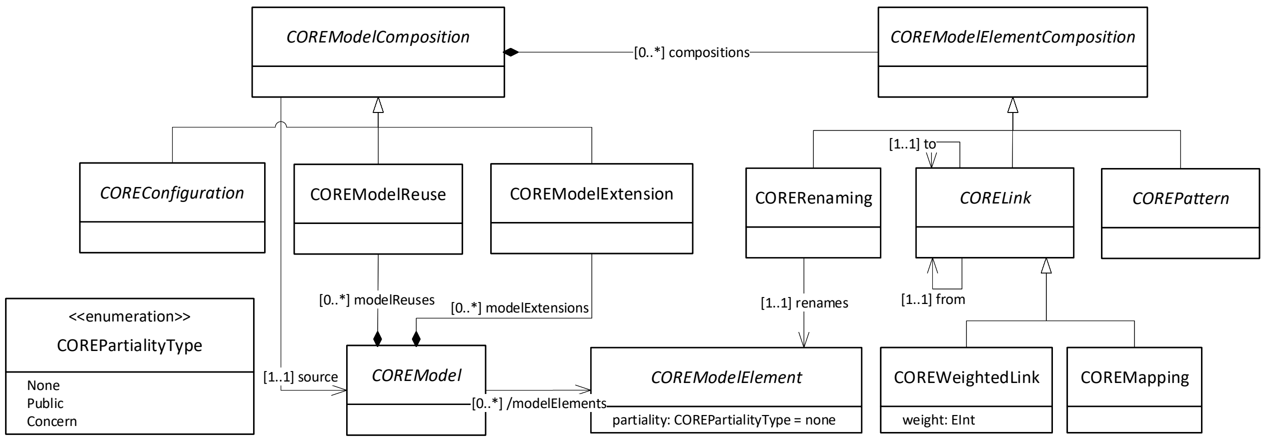


Figure 2.4: CORE metamodel: customization interface

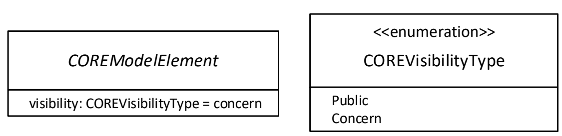


Figure 2.5: CORE metamodel: usage interface

Variation Interface

Customization Interface

Usage Interface

2.1.2 Reusable Aspect Models (RAM)

No need to explain too much here, since we don't use RAM really in the rest of the thesis, no? Explain mostly that RAM is the only language currently integrated with CORE, and that the models are about design. You could use class diagrams to give an example of weaving, if you think it is necessary

.

2.1.3 Customization Mappings

2.1.4 CORE Weaver

2.2 Use Case Map (UCM)

Brief overview and what UCMs are used for. Maybe one example UCM (that is used later on)

.

UCM metamodel (or a simplified version of it)

.

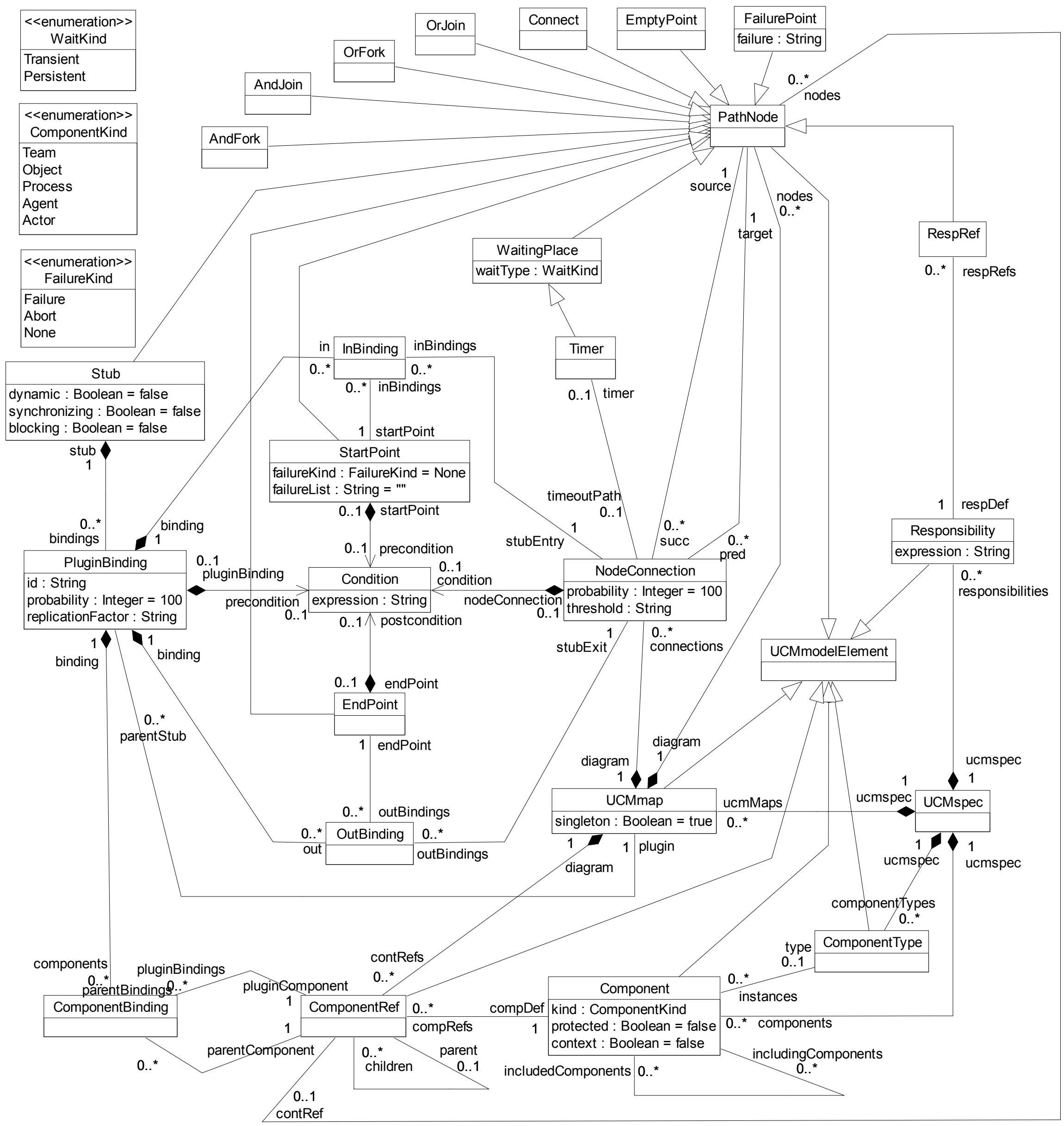


Figure 2.6: Abstract grammar: UCM metamodel. Image courtesy of ITU [10]

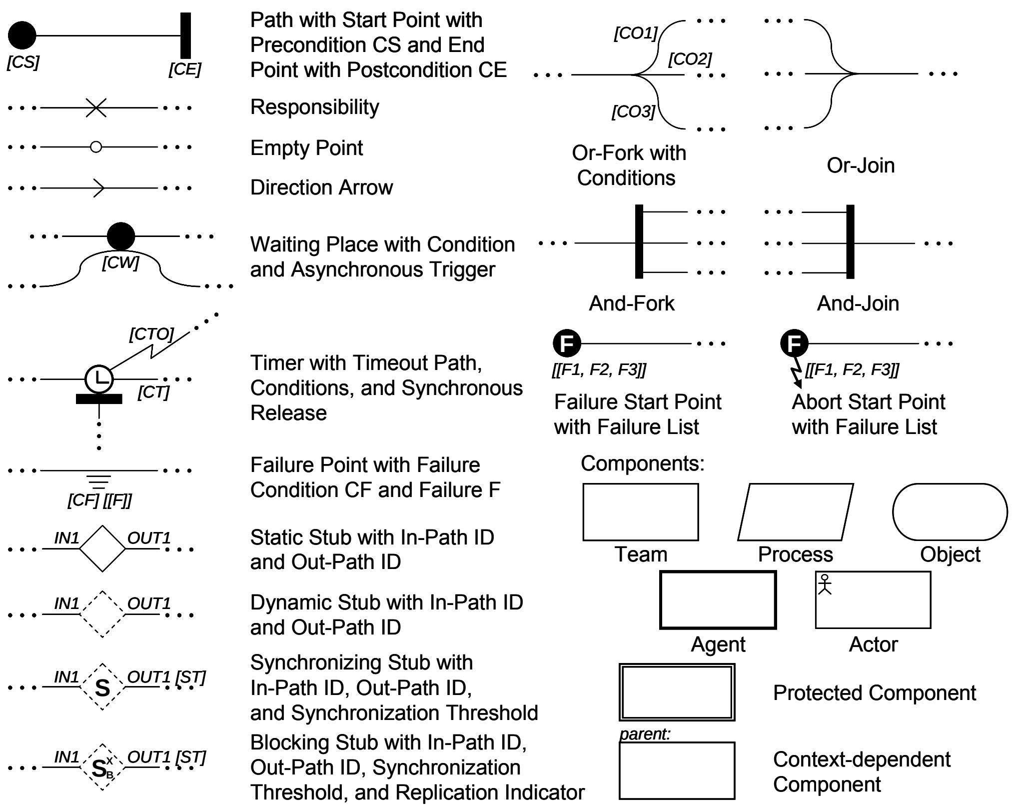


Figure 2.7: UCM notation. Image courtesy of ITU [10]

2.2.1 Aspect-oriented Use Case Map (AoUCM)

Most of your related work will be Gunter's AoUCM stuff, which you probably mention above already. jUCMNav as well, but this you could also mention above when you talk about UCMs.

.

Chapter 3

Adding Support for UCM to CORE

This chapter presents the corification of a modeling language for the requirements phase using the CORE metamodel. We describe the steps taken to corify UCM in Section 3.1. We also present the weaving algorithm specific for UCM in the context of CORE in Section 3.2.

3.1 Corification of UCM

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 [10] closely in corifying the UCM metamodel. Figure 3.1 shows a partial view of the corified UCM metamodel, focusing on the elements that extend

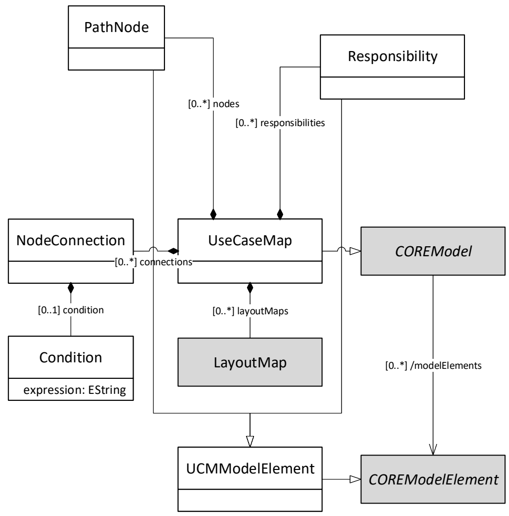


Figure 3.1: Extension of the CORE metamodel by UCM

the CORE metamodel through subclassing (from an existing metaclass in the modeling language to an abstract CORE metaclass [1]). By subclassing the necessary abstract classes of the CORE metamodel, UCM is able to provide all the properties of CORE:

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

Reusing a concern from a UCM model prompts the feature selection process, by asking the feature that the UCM model realizes to reuse the other concern with the selection of feature(s) it wants to reuse. 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

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

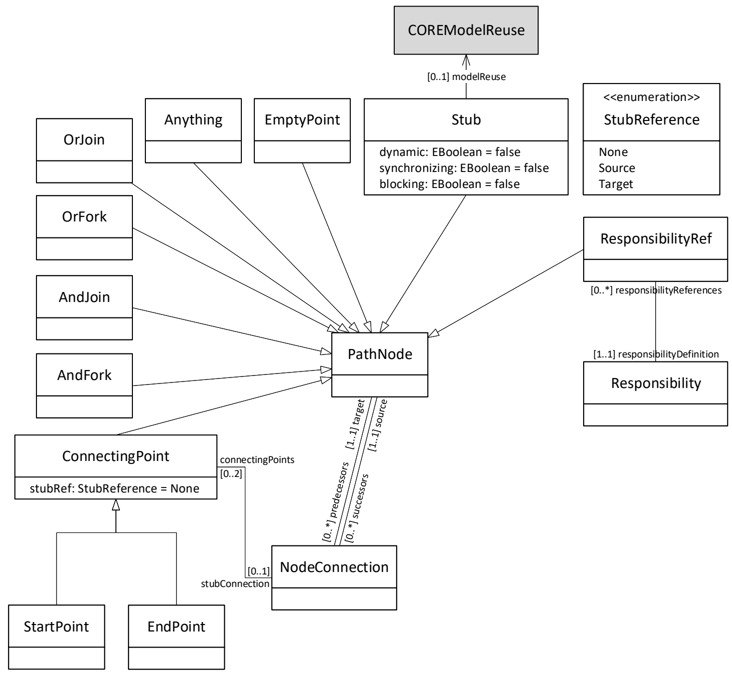


Figure 3.2: Path nodes for corified UCM

(see Figure 2.1, association between COREConcern and COREModel). This allows a UCM to realize a feature (see Figure 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 [11]. Anything acts as a wild card and can represent a subset of nodes in a path. This is useful for facilitating complex model weaving, as it allows any sequence of UCM model elements, including an empty sequence, to be matched.

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 source and target 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 customization mappings.

Since we are using COREMapping to specify customizations, it is necessary for UCMModelElement to subclass *COREModelElement*. That way, all subclasses of UCMModelElement

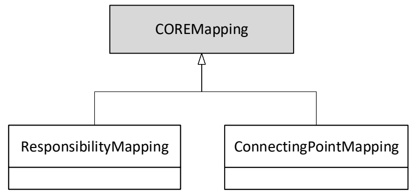


Figure 3.3: Customization mappings for corified UCM

(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

As explained in Section 2.1, 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 [12]. 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

The algorithms presented here are specific for single weaving, meaning that a composition is performed from one model (UCM_1) to another model (UCM_2). This action can be chained together with other compositions, even with the hierarchical structure of the concern features. Here, we specify the subscript $_1$ for the model elements of a UCM the weaver composes from, and the subscript $_2$ for the model elements of a UCM the weaver composes to. UCM_1 and UCM_2 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.

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. Algorithm 1 illustrates the procedure of weaving for responsibility mappings. The function *WeaveResponsibilityMapping* initiates the process by identifying the mapped responsibilities (*from* UCM_1 *to* UCM_2), and traversal begins from the point of $responsibility_2$ in both directions: (i) toward predecessors until start point encountered; and (ii) toward successors until end point encountered. 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. The

Algorithm 1 Weaving Algorithm: Responsibility Mapping

```
function WeaveResponsibilityMapping(ucm, composition)
    node_1 ← get first node of composition mapping (from)
    node_2 ← get second node of composition mapping (to)
    mark node_2 as visited
    indicate start point has not been encountered
    indicate end point has not been encountered
    call TraverseToSource(ucm, node_2, node_1)
    call TraverseToTarget(ucm, node_2, node_1)
    remove node_1 from ucm
end function
function TraverseToSource(ucm, node_2, node_1)
    for each predecessor of node_2 do
        sourceNode ← get source node of predecessor
        if linkage exists from previous mapping then
            set the source of node_2's connection to node_1's predecessor
            disable linkage
            skip this loop
        end if
        if sourceNode is visited then
            skip this loop
        else if sourceNode is not Anything then
            mark sourceNode as visited
        end if
        if sourceNode is StartPoint and start point is not encountered then
            if visibility of sourceNode is Concern then
                set the source of node_2's connection to node_1's predecessor
                remove sourceNode from ucm
                indicate start point has been encountered
            end if
        else if sourceNode is Anything then
            set the source of node_2's connection to node_1's predecessor
            if sourceNode does not have any predecessor then
                remove sourceNode from ucm
            end if
        else
            recursively call TraverseToSource(ucm, sourceNode, node_1)
        end if
    end for
end function
```

```
function TraverseToTarget(ucm, node_2, node_1)
    for each successor of node_2 do
        targetNode ← get target node of successor
        if targetNode is visited then
            skip this loop
        else if targetNode is not Anything then
            mark targetNode as visited
        end if
        if targetNode is Endpoint and end point is not encountered then
            if visibility of targetNode is Concern then
                set the target of node_2's connection to node_1's successor
                remove targetNode from ucm
                indicate end point has been encountered
            end if
        else if targetNode is Anything then
            set the target of node_2's connection to node_1's successor
            if targetNode does not have any successor then
                remove targetNode from ucm
            end if
        else if targetNode exists in compositions mapping (to) then
            copy node connection of successor
            set the target of copied connection to node_1's successor
            add the copied connection to ucm
            enable linkage to next mapping
        else
            recursively call TraverseToTarget(ucm, targetNode, node_1)
        end if
    end for
end function
```

path of a UCM may consist of mapped responsibilities interspersed with other path nodes. While traversing forward, lines 59-63 handle the next mapped responsibility. If exist, forward traversal stops for this specific composition and appropriate nodes are connected between UCM_1 and UCM_2. 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_1 and UCM_2. This pattern continues until the weaver reaches end point, whereby the predecessor of UCM_2's end point connects to the successor of mapped responsibility ($from$) UCM_1 and this end point gets deleted (lines 48-53). Same goes for backward traversal until the weaver reaches start point (lines 24-29). Lastly, mapped responsibility (to) UCM_2 retains while mapped responsibility ($from$) UCM_1 gets deleted for the final woven UCM model.

UCM may have multiple start points merging to a path, or a path may branch to multiple end points. In this case, we allow a start or end point to set its visibility level. By default, 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_2 to capture the actual nodes that are specified in UCM_1. If an anything node is encountered during traversal, lines 30-34 and 54-58 signals 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 (loop cycle). 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_1 *to* UCM_2), and determine the type of composition to be performed based on whether the connecting points mapped from UCM_1 are attached to a stub or not. If the mapped start and end points from UCM_1 are attached to a stub (lines 5-6 and 11-12), it means that the connecting points are hidden and belong to a stub in UCM_1 and are mapped to actual end and start points of UCM_2, respectively (cross mapping). This type of composition is model extension. Vice versa for model reuse (lines 7-8 and 13-14).

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 end point. The weaver first obtain the pair of mappings for the stub. The initial mapping usually maps the end point of a stub [2] to the start point of a UCM, and the weaver executes lines 18-24. The second mapping usually maps the start point of a stub [3] to the end point of a UCM, and the weaver executes lines 32-38.

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

Algorithm 2 Weaving Algorithm: Connecting Point Mapping

```
function WeaveConnectingPointMapping(ucm, composition)
    node_1 ← get first node of composition mapping (from)
    node_2 ← get second node of composition mapping (to)
    if node_1 is StartPoint then
        if node_1 is connected to a stub then
            call ExtendingStub_End(ucm, node_2, node_1)
        else if node_2 is connected to a stub then
            call ReusingStub_Start(ucm, node_2, node_1)
        end if
    else if node_1 is EndPoint then
        if node_1 is connected to a stub then
            call ExtendingStub_Start(ucm, node_2, node_1)
        else if node_2 is connected to a stub then
            call ReusingStub_End(ucm, node_2, node_1)
        end if
    end if
end function
function ExtendingStub_End(ucm, node_2, node_1)
    source_2 ← get source node of node_2
    source_1 ← get source node of node_1 via stub connection
    target_1 ← get target node of node_1 via stub connection
    call MergePaths(source_2, source_1, target_1, node_1, node_1)
    remove node_2 from ucm
end function
function ReusingStub_Start(ucm, node_2, node_1)
    target_1 ← get target node of node_1
    target_2 ← get target node of node_2 via stub connection
    source_2 ← get source node of node_2 via stub connection
    call SplitPaths(target_1, target_2, source_2, node_2, node_1)
    remove node_1 from ucm
end function
function ExtendingStub_Start(ucm, node_2, node_1)
    target_2 ← get target node of node_2
    target_1 ← get target node of node_1 via stub connection
    source_1 ← get source node of node_1 via stub connection
    call SplitPaths(target_2, target_1, source_1, node_1, node_1)
    remove node_2 from ucm
end function
```

```
function ReusingStub_End(ucm, node_2, node_1)
    source_1 ← get source node of node_1
    source_2 ← get source node of node_2 via stub connection
    target_2 ← get target node of node_2 via stub connection
    call MergePaths(source_1, source_2, target_2, node_2, node_1)
    remove node_1 from ucm
end function
function SplitPaths(target, target′, source′, node, node′)
    if target′ is Stub then
        mark target′ as removable stub
        set target node of node’s stub connection to target
    else if target′ is AndFork or OrFork then
        create node connection between target′ and target
    else
        referenceStub ← get target node of node′ via stub connection
        forkNode ← if referenceStub is dynamic then create AndFork else OrFork
        place forkNode in between source′ and target′
        create node connection between forkNode and target′
        create node connection between forkNode and target
        set target node of node’s stub connection to forkNode
    end if
end function
function MergePaths(source, source′, target′, node, node′)
    if source′ is Stub then
        mark source′ as removable stub
        set source node of node’s stub connection to source
    else if source′ is AndJoin or OrJoin then
        create node connection between source and source′
    else
        referenceStub ← get source node of node′ via stub connection
        joinNode ← if referenceStub is synchronizing then create AndJoin else OrJoin
        place joinNode in between source′ and target′
        create node connection between source′ and joinNode
        create node connection between source and joinNode
        set source node of node’s stub connection to joinNode
    end if
end function
```

Model reuse, on the other hand, operates in reversed orientation—obtaining the pairs of mappings that mapped the start and end points of UCM_1 to the connecting points of a stub that is automatically generated in UCM_2 when reusing UCM_1. 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 and 39-45, respectively.

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 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 [10]. 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 concern-oriented UCMs by means of case studies in section 4.2. Finally, we demonstrate that concern-oriented UCMs 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 [13]. Previous project, TouchRAM, successfully implemented concern-oriented software design paradigm, but support is limited to RAM (class, sequence, and state diagrams) [14, 15]. TouchCORE extends TouchRAM with numerous enhancements, most notably the support for arbitrary modeling languages in addition to RAM. Since we have a well-defined corified UCM metamodel, we attempted to add support for UCMs in TouchCORE as proof of concept, enabling TouchCORE the capability to build scalable and reusable scenario models.

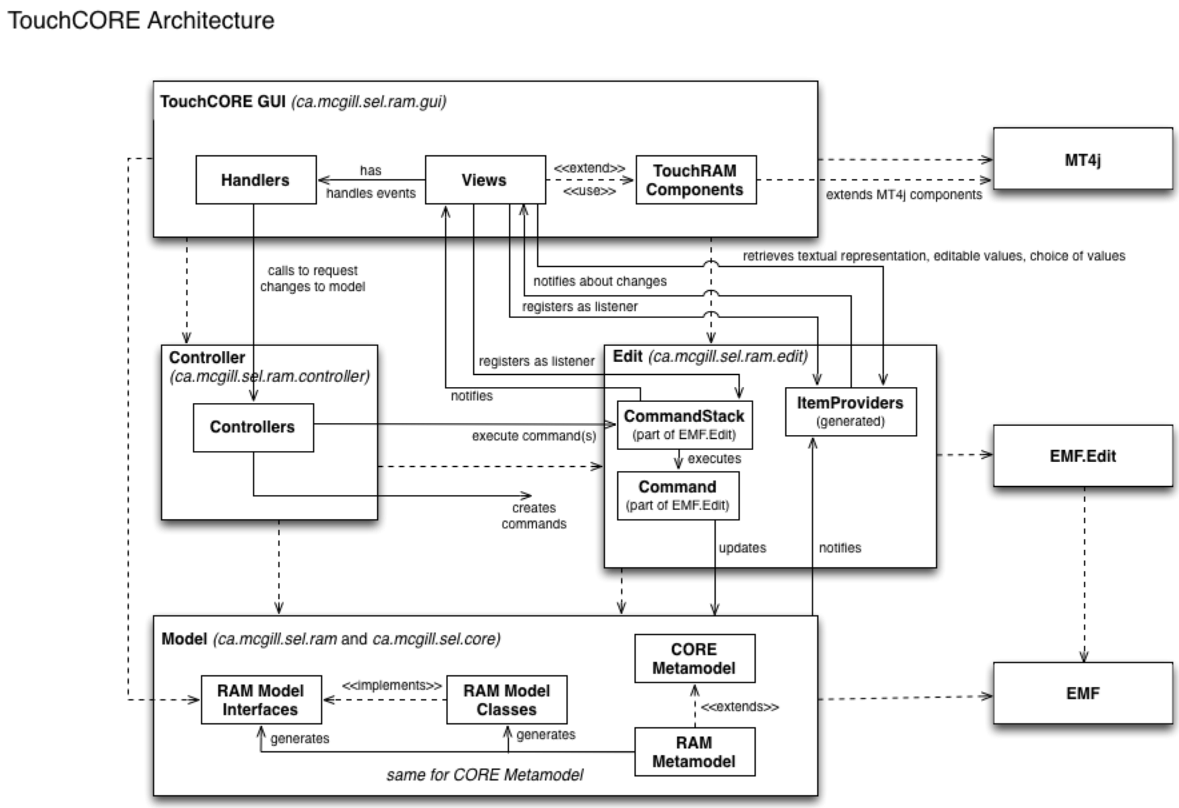


Figure 4.1: TouchCORE architecture. Image courtesy of Software Engineering Lab, McGill University

The project uses Java SE Development Kit 8 as the implementation language and Eclipse Modeling Framework (EMF) [16] 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). 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 [17]. 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.

4.1.1 Supported Concrete Syntax

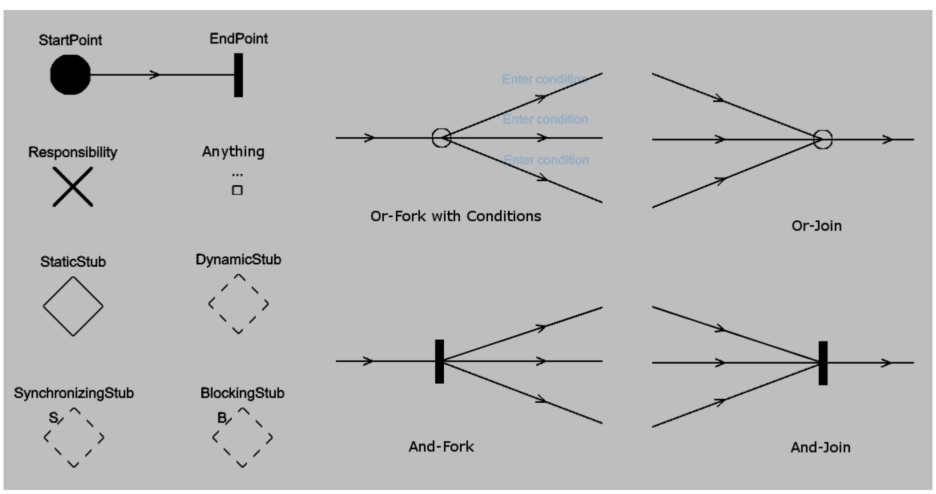


Figure 4.2: UCM notation in TouchCORE

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 [10], with the exception of Anything that is taken from the extended AoUCM metamodel [11]. 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 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 start and end points to set its 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 its 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

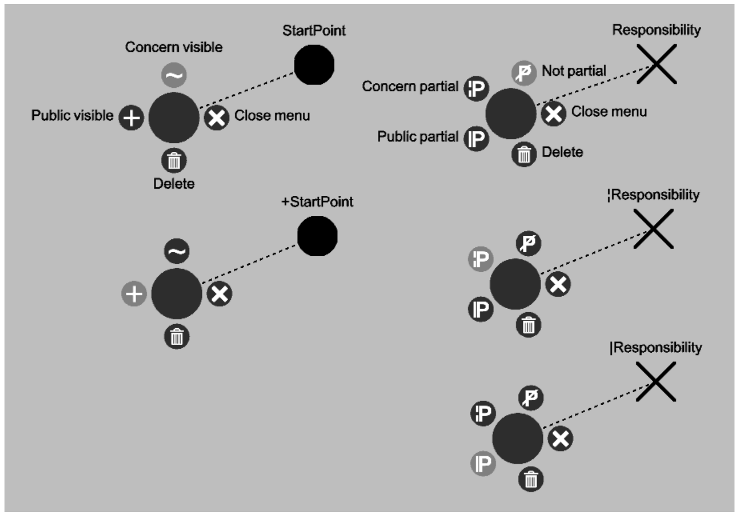


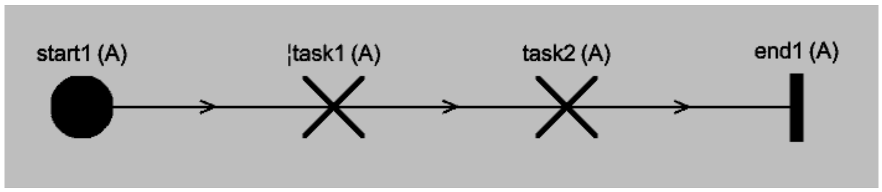
Figure 4.3: Visibility and partiality

is partially defined and require appropriate composition to be semantically complete. A responsibility that is concern partial should fulfill its significance through model extension, whereas a responsibility that is public partial should fulfill its significance through model reuse.

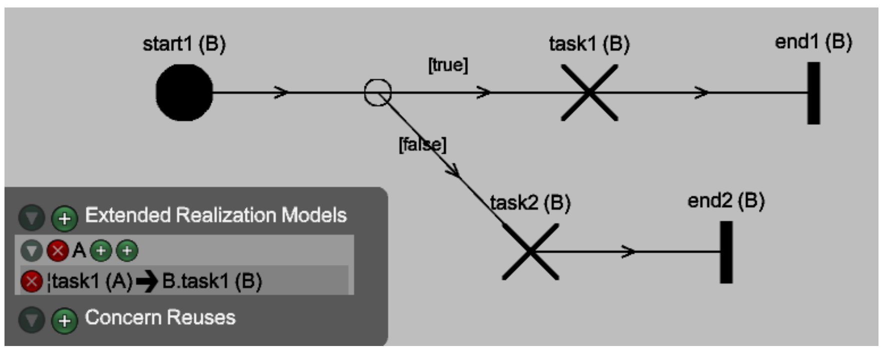
4.1.2 Scenario Model Composition

Model Extension

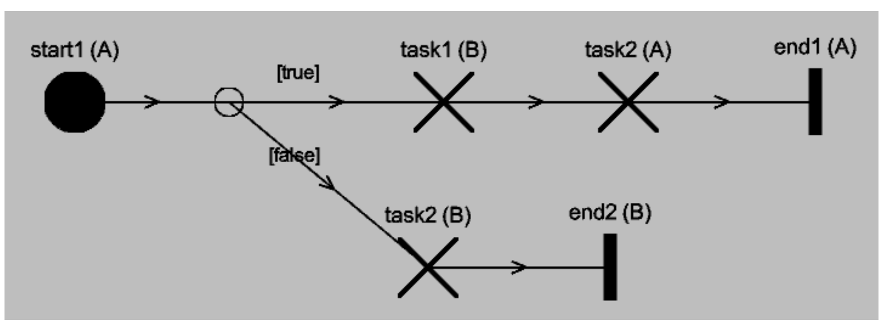
Figure 4.4 illustrates the usage of UCM model extension within a concern. Given a concern with two features in a hierarchy, the model of a child feature (Model B) extends the model of a parent feature (Model A). Composition specifications are specified in Model B, where an element of Model A is mapped 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



(a) Model A - parent UCM



(b) Model B - child UCM



(c) Woven Model B_A

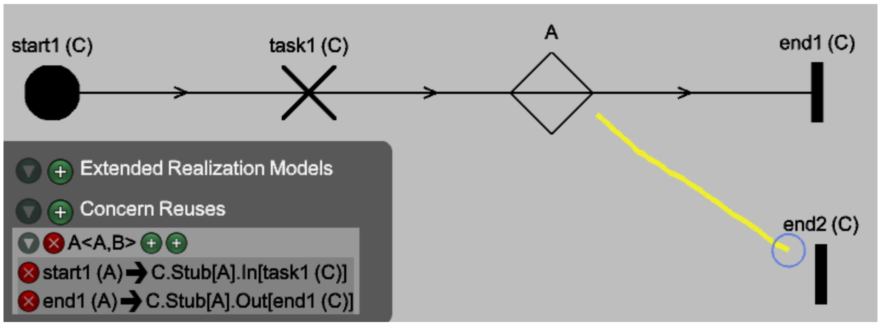
Figure 4.4: Schematic representation of model extension

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.

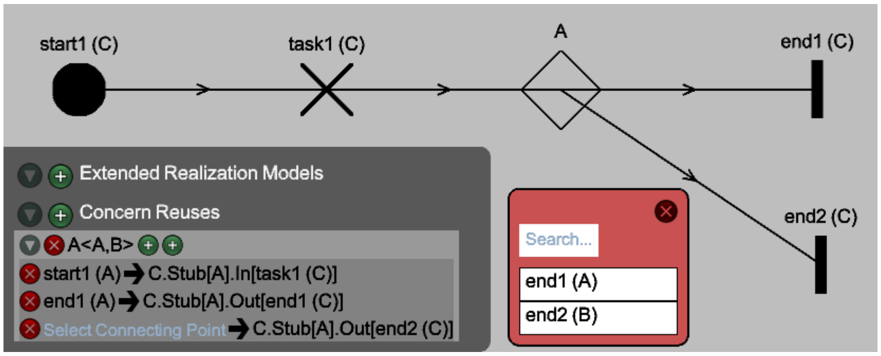
The idea of isolating features as individual models supports the use of advanced separation of concerns—each feature encapsulates its realization model. Features of a concern are nested in a hierarchical order, and the connection between features can be seen as parent-child relationship. Extension of 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.

Model Reuse

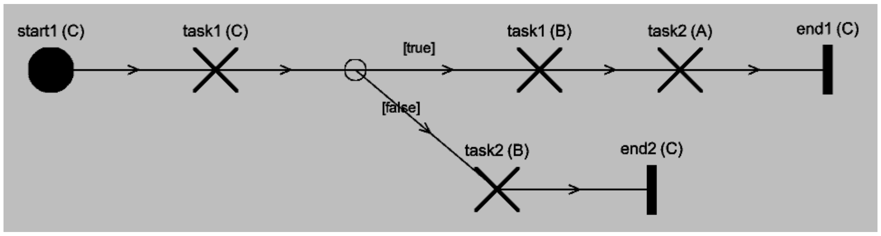
Figure 4.5 illustrates the usage of UCM model reuse across concerns. Given that Model C of a concern reuses Concern A, the configuration for the set of features of Concern A will be displayed. Here, we chose to use features A and B, thus woven Model B_A (see Figure 4.4c) is generated and represented 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 C to Model B_A 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 concern-oriented UCMs 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, as well as the ability to reuse the Authentication concern within the Online Payment concern.

4.2.1 Authentication

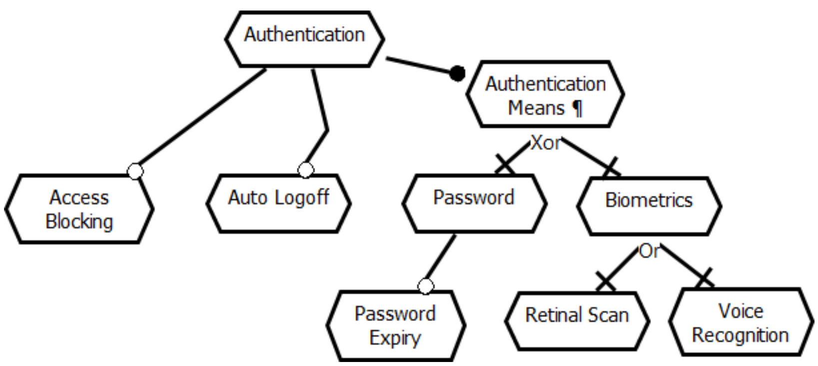


Figure 4.6: Feature model for Authentication (jUCMNav). Image courtesy of Nishanth Thimmegowda et al. [18]

We design the Authentication concern based on a reference model that we have previously described in jUCMNav format [18]. 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*.

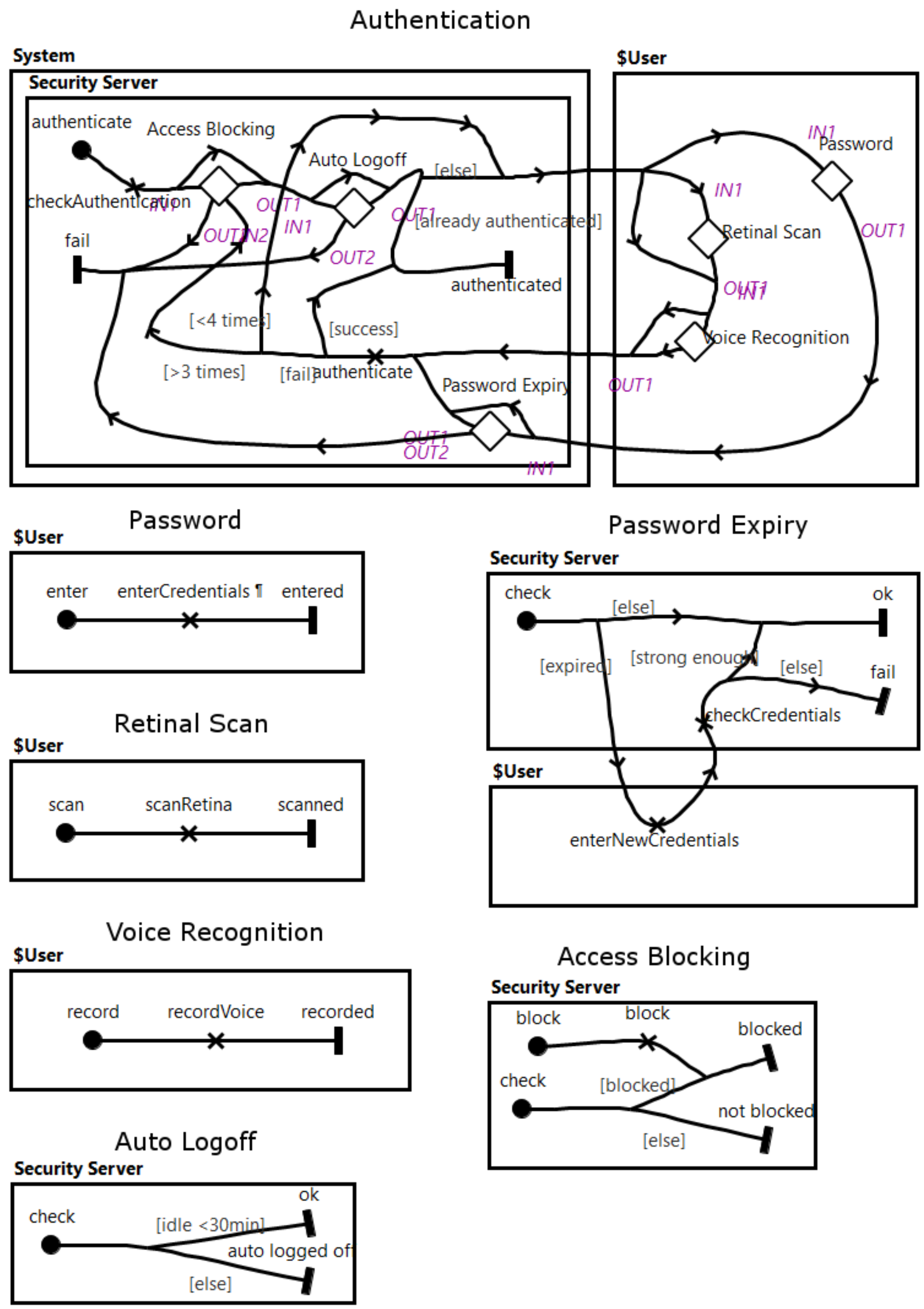


Figure 4.7: Scenario models for Authentication (jUCMNav). Image courtesy of Nishanth Thimmegowda et al. [18]

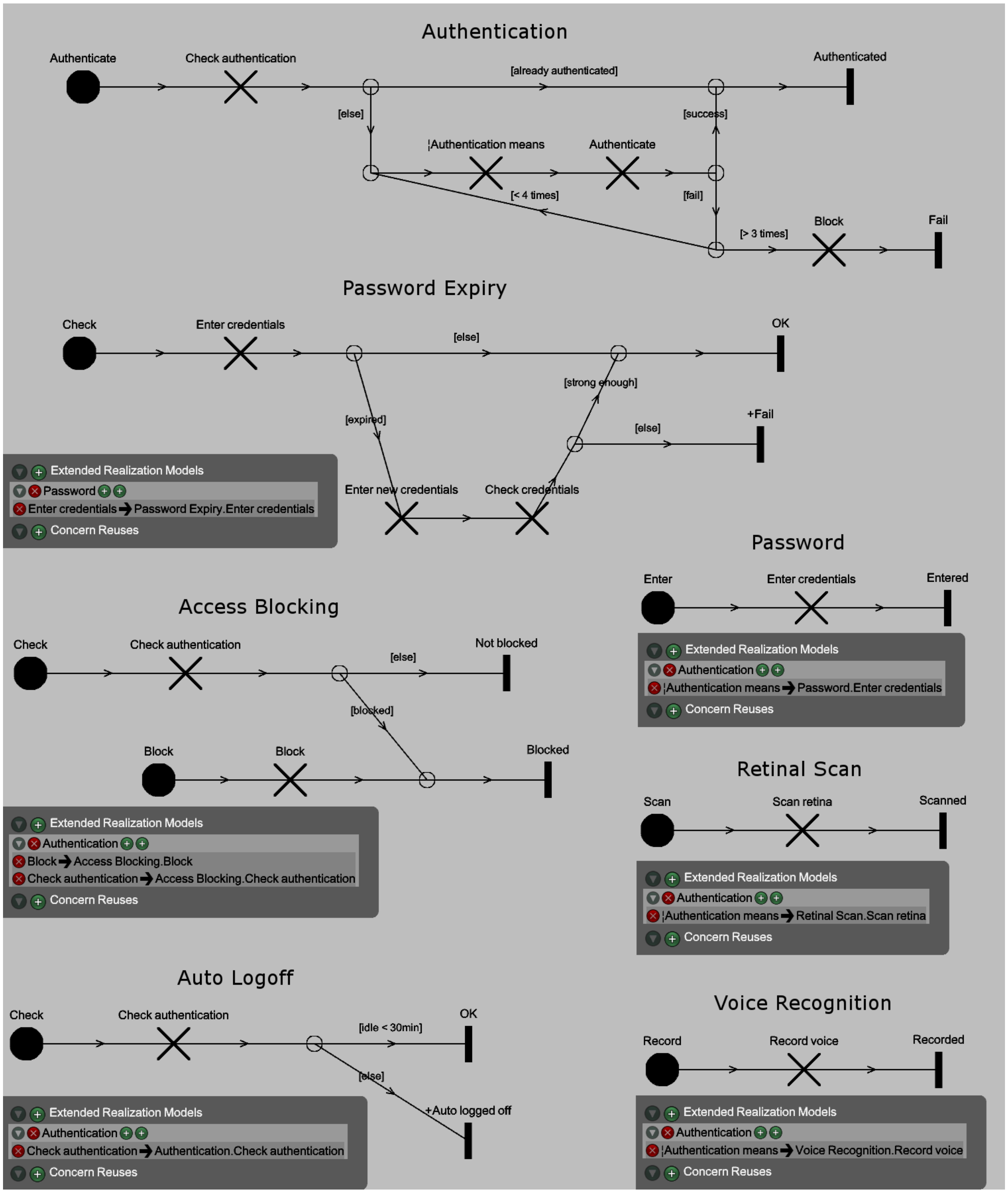


Figure 4.8: Scenario models for Authentication (TouchCORE)

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 of crosscutting the core 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 e-commerce platform. Use cases for Online Payment are adapted from W3C Web Payments Interest Group [19], focusing on the payment schemes in use today. Figure 4.9 shows all the available features that are supported for the concern. Online Payment provides numerous payment methods for the customers to pay by credit card (e.g., Visa, MasterCard, China UnionPay),

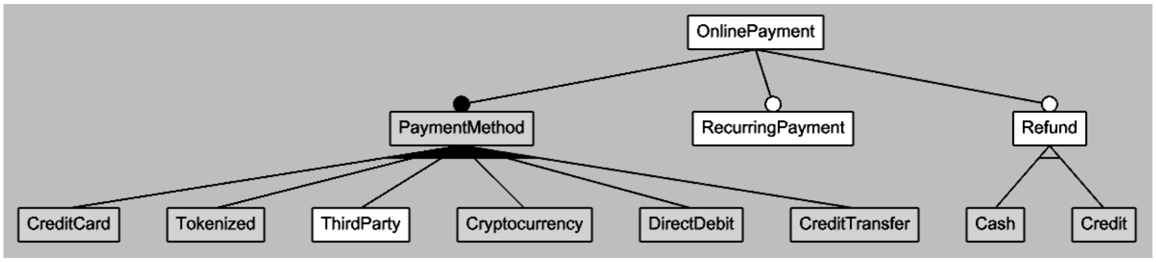


Figure 4.9: Feature model for Online Payment

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 recurring payment option to allow for subscription plan and refund 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 and most methods share 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 is depicted as a static stub, and the selected features are *Password* and *Access Blocking*. The one incoming connection and two outgoing connections of the *Authentication* stub are associated with the single start point (*Authenticate*) and two end points (*Authenticated*; *Fail*) of Authentication UCM (see Figure 4.8).

Optional features of *OnlinePayment* are *RecurringPayment* and *Refund*. Both of the

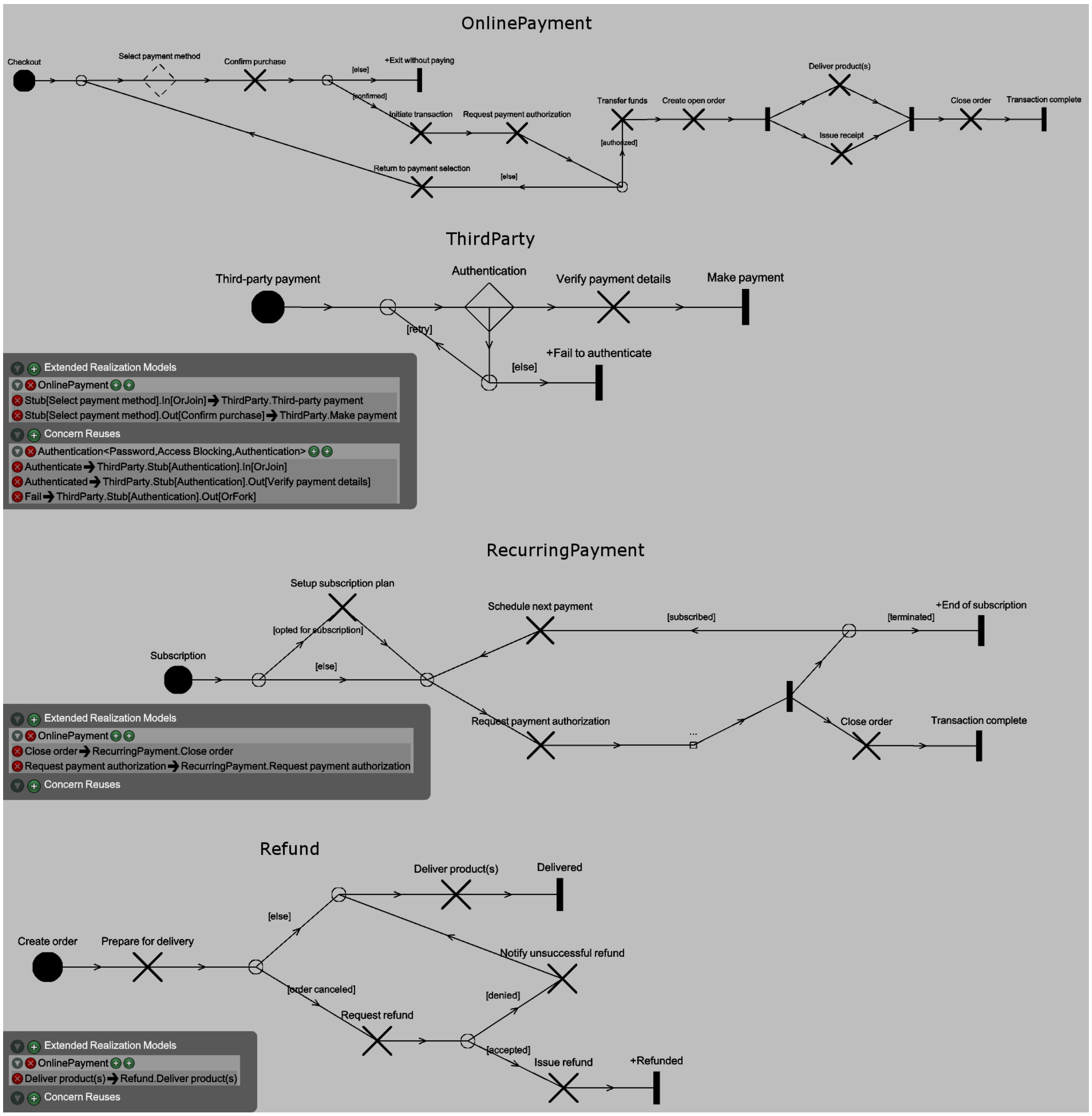


Figure 4.10: Scenario models for Online Payment

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 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 reuses 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 concern-oriented UCMs to implement some of the workflow patterns described by van der Aalst et al. [20]. 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 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.

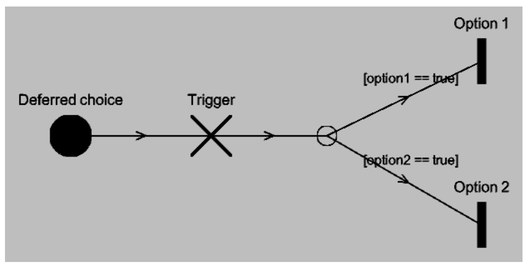


Figure 4.11: Deferred choice pattern

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.

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

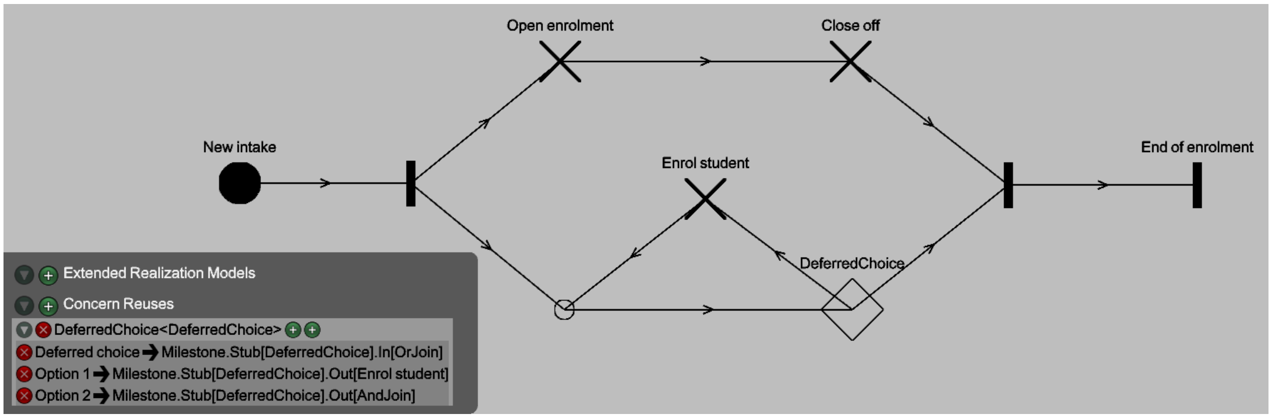


Figure 4.12: Milestone pattern (enrolment example)

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 different. 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* ($option_2$).

Chapter 5

Conclusion

5.1 Summary

Recap of thesis: what I did so far and what can the tool achieve.

5.2 Future Work

Remaining tasks such as components, path drawing, validation, semantics, etc.

Appendix A

Complete Metamodels

A.1 CORE Metamodel

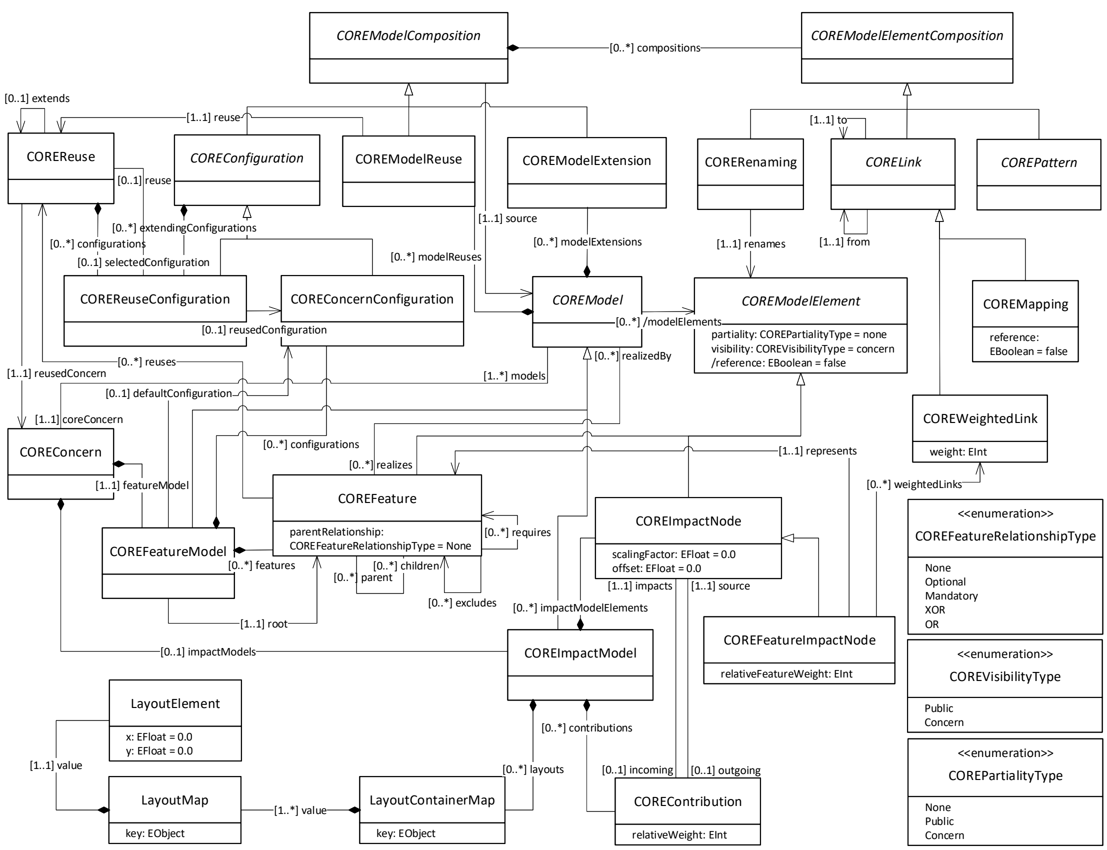


Figure A.1: Abstract grammar: CORE metamodel overview

A.2 UCM Metamodel

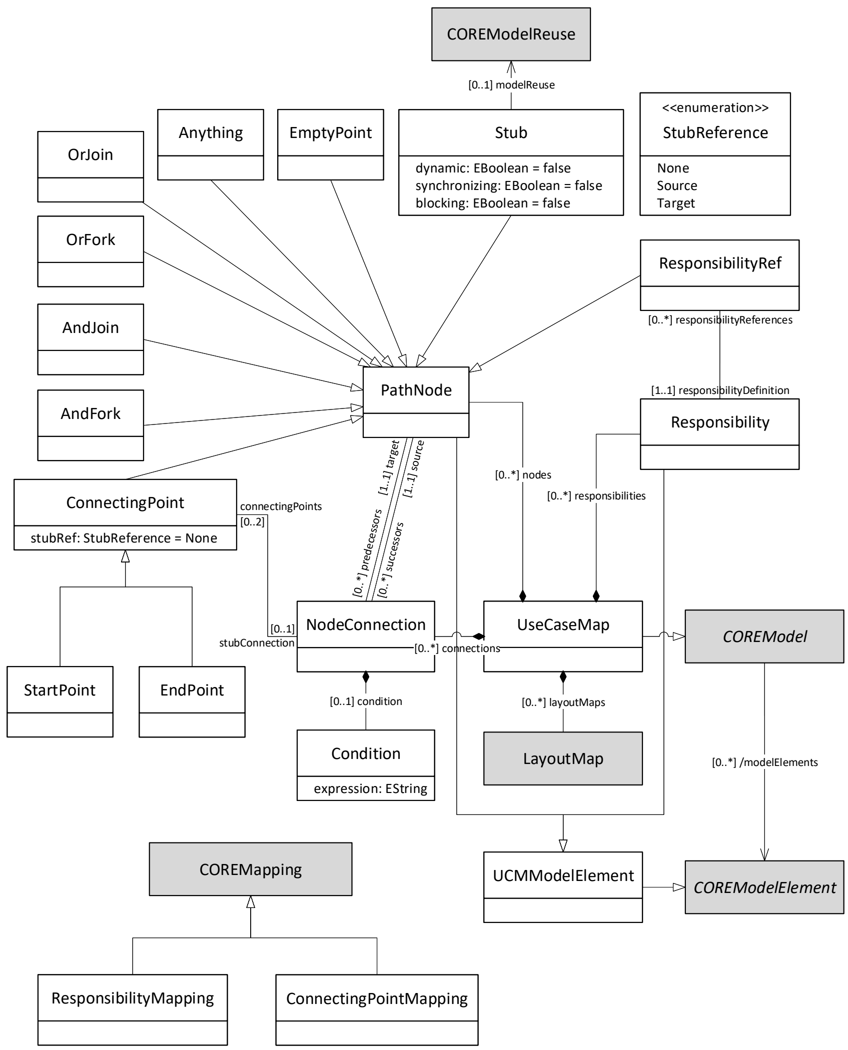


Figure A.2: Abstract grammar: corified UCM metamodel overview

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

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

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

[13] *TouchCORE*. URL: http://touchcore.cs.mcgill.ca/. Jan. 2018.

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

[15] 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.

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

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

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

[19] 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/.

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