JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Concern-Oriented Use Cases

Ryan Languay, Nika Prairie, and Jörg Kienzle
School of Computer Science, McGill University, Montreal, Canada

ABSTRACT Modelling languages often lack explicit support for reuse, and there are very few libraries of reusable models available to developers. This is especially true for use cases, one of the most wide-spread modelling languages used to describe systems at a high level of abstraction during requirements elicitation. This paper proposes *Concern-Oriented Use Cases* (CoUC), a use case modelling language designed to support planned and opportunistic reuse. CoUC makes it possible to create libraries of generic recurring interaction scenarios, provides means to modularize crosscutting interaction patterns and supports feature-oriented scenario extensions. We provide a metamodel that defines the hierarchical structure and behavioural scenario descriptions for use cases. We further elaborate a use case composition algorithm capable of combining the reusing and reused use cases. To validate our approach, the CoUC language and composition algorithm have been implemented in the TouchCORE modelling tool, and applied to model three examples which showcase feature-oriented use case extension, reuse of a generic use case, as well as software product line development and evolution.

KEYWORDS Concern-Oriented Reuse, Use Cases, Use Case Composition, Feature Model, Software Product Lines.

1. Introduction

Already during the first NATO conference on software engineering (SE) in 1968 (McIlroy 1969), software reuse (or "mass-produced software components" according to McIlroy) was recognised as crucial for the future of software development. Reuse can increase the quality of software, while simultaneously reducing development time and cost (Krueger 1992).

Reuse in Model-Driven Engineering (MDE) mostly focuses on the reuse of modelling languages and associated model transformations that are part of a MDE tool chain (Burden et al. 2014). Experience has shown that reuse of models is difficult and seen as problematic in industry (Whittle 2012). Typically, models for a system under development are created from scratch, rather than reusing already existing models. Modelling languages often lack explicit support for reuse, e.g., constructs for defining modules, interfaces, and composition (Herrmann et al. 2007; Kienzle et al. 2016), and there are very few libraries of reusable models available to developers.

JOT reference format:
Ryan Languay, Nika Prairie, and Jörg Kienzle. *Concern-Oriented Use Cases*. Journal of Object Technology. Vol. 22, No. 2, 2023. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2023.22.2.a13

This is especially true for one of the most wide-spread modelling languages used to describe systems at a high level of abstraction during requirements elicitation: *use cases* (Jacobson et al. 1992). Use cases describe the interactions between the system under development and its environment to achieve user goals, and hence are by nature application-specific. The benefits of reuse are nevertheless prominent in use cases, because even high-level interaction scenarios exhibit recurring interaction patterns, e.g., to deal with common functional and extra-functional concerns (Araujo & Moreira 2003; Jacobson & Ng 2004). Furthermore, different products that are part of a Software Product Line (Pohl et al. 2005) (SPL) also share common interactions and/or patterns.

While there have been efforts to modularise recurring behaviour descriptions that cross-cut several use cases in the aspect-oriented modelling community (Lu & Song 2008; Yue & Ali 2012), none of them support the many reuse scenarios found in modern software development, such as, hierarchical reuse, genericity, unplanned extension of use cases, and feature-oriented reuse as found in SPL. Furthermore, only very few of the existing aspect-oriented use case approaches have been described to the level of detail that is needed to implement a functioning use case model composition algorithm in practice.

This paper proposes concern-oriented use cases (CoUC),

a use case modelling language designed to maximize reuse. Concretely, the contributions of this paper are:

- The CoUC metamodel for capturing the structure and behaviour of CoUC use cases,
- A composition algorithm, describing how to combine two CoUCmodels into a single resulting model, and
- A validation of the CoUC approach comprising three application examples which cover various important reuse patterns, including feature-oriented extension, concern reuse, modelling and evolution of software product lines, and unplanned use case extension.

The remainder of the paper is structured as follows. Section 2 presents the background on use cases. Section 3 elicits four different reuse situations for use cases, and reviews how the related work on aspect-oriented use cases supports those reuse situations. Section 4 introduces the CoUC language, starting with the metamodel and how it covers use case diagrams and textual scenario descriptions. It further presents the details of the structural and behavioural composition algorithm that combines two CoUC models according to provided composition directives. Section 5 validates CoUC by illustrating how it supports the four reuse scenarios, hierarchical reuse and unplanned evolution. Finally the last section draws some conclusions.

2. Background on Use Cases

Use case models are used to discover and document the requirements and behaviour of a system during early software development phases (Jacobson et al. 1992). Use case models treat the software system under construction as a black box, focusing on the goals that users have with the system, as well as the interactions between the system and its environment to achieve these goals (Rumbaugh et al. 1999). They are typically expressed in plain text, which makes it possible to involve non-technical business resources in the requirements modelling process. Thirty years after their inception they are being used extensively in industry, and have even been adapted to fit agile software development contexts (Jacobson et al. 2016).

Use case models are primarily defined by two concepts: *actors* and *use cases*. *Actors* are idealised representations of entities that the system under development is interacting with. They describe humans, other software systems, hardware devices, or any entity that is external to the system but that must interact with the system in some way. They are displayed as stick figures in a use case diagram (e.g., the *Administrator* actor in Figure 1.

Use cases are the units of functionality provided by the system, as expressed by a sequence of interaction steps, but not including a description of any system internals (Booch et al. 2005). They are displayed as ellipses in a use case diagram. The internals of a use case are defined by use case interactions, and are expressed in units called scenarios (W3 Computing 2020) or flows (Booch et al. 2005).

Use cases typically contain multiple scenarios: a basic scenario describing the ideal execution of the use case, and various scenarios describing alternate executions that either achieve the original goal, result in some alternate outcome, or deal with situations in which the goal can not be successfully achieved.

Use cases and actors are connected via different types of associations, which classify what role each actor plays in the scenario. *Primary actors* typically are the ones that have a goal with the system, and are hence typically the ones that trigger the execution of a use case in the first place. *Secondary actors* designate other actors that the system interacts with to achieve the goal of the use case.

Use cases can also be connected to other use cases via one of three possible relationships: *include*, *extends* and *generalisation-specialisation*. When a use case describes a complex goal, it often is possible to identify sub goals that can be modularized in subfunction level use cases. In that case, the higher-level use case explicitly invokes the subfunction use case at a particular point in its flow, similar to a method call. From a behavioural point of view, the higher-level use case is functionally incomplete on its own, as it depends on the lower level use case to achieve the sub goal. This hierarchical dependency is represented by a dotted arrow with an «include» stereotype (e.g., *Authenticate* includes *Entering Credentials* in Figure 1).

Functionally complete use cases that achieve a specific goal can also define explicit extension points in their flow to indicate that under certain conditions some additional functionality should or could also be achieved at these points. Another use case, called an *extending use case*, can then define additional behaviour to be executed at those extension points. Extension is represented in the use case diagram with a dotted arrow with a «extend» stereotype with the arrow pointing from the extending use case towards the use case that is being extended.

Finally, use cases can also make use of generalisation-specialisation relationships. In a similar way as polimorphism in object-orientation, the key property of use case inheritance is that a more specific child use case can be used in place of any reference to a parent use case (e.g., *Scan Retina* is a specific way of *Entering Credentials* in Figure 1).

2.1. Structured Use Cases

The explicit structure of how to textually specify the details of a use case scenario is intentionally not specified. Many modelling tools therefore offer general plain text editors to enter scenario information, but several templates have also been proposed to introduce additional structure to the textual elaboration of use case details (Sendall & Strohmeier 2000; Jacobson et al. 2011).

The approach which provided the base for our work is the Restricted Use Case Model (RUCM) proposed by (Yue et al. 2009). This approach restricts the structure of the use case detail to containing one basic flow consisting of a sequence of interaction steps, and zero or more alternate flows. The content of a step is also restricted by the template; a step can either represent an action or a keyword (Yue 2010). An action can be one of five possible interactions, either between the system and an actor (inputs from a primary actor, or outputs to either a primary or a secondary actor), or internally to the system (in the case of validation or modification of the internal state). A keyword step typically represents control flow of surrounding steps or a reference to an external use case. RUCM also places certain restrictions on the language that can be used in step definitions, e.g., that steps must be written in an active voice.

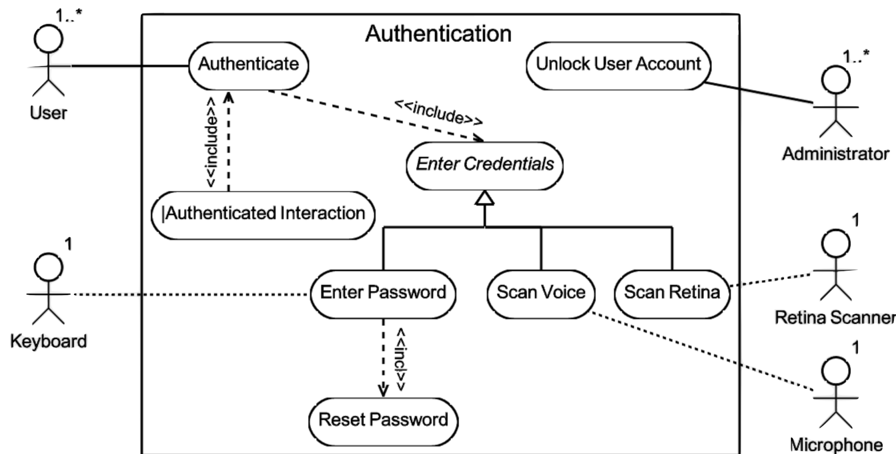

Figure 1 Use Case Diagram of the *Authentication* Concern including all Features

3. Reuse Scenarios for Use Cases and Related Work

Reuse has received relatively little attention in the context of use cases, maybe because use cases typically describe user goals, i.e., very application-specific scenarios. However, based on examples in the literature and our own experience we have identified four prominent reuse scenarios for use cases, depicted in Figure 2. They can occur 1) within an application when multiple usage scenarios share common interaction steps, 2) within a software product line when multiple products share common usage scenarios, and 3) across multiple applications.

Case A, shown at the top left of the figure, depicts the common case where a recurring scenario is encapsulated in a (probably subfunction-level) use case. Multiple user-goal use cases can reuse the scenario by referring to the use case in one or several of their steps. This is supported within a standard use case model with the «include» dependency between use cases. It can also be used when several features of a SPL need the same scenario, or when the same scenario occurs in multiple applications. The reused scenario does not need to know whether and how it is being reused, and hence «include» supports case A for both planned and unplanned reuse.

Case B, depicted on the top right of Figure 2, illustrates the case where a complete user use case is to be reused in a different application or within a feature of a SPL, but some additional application- or feature-specific interactions are needed. Standard use cases support case B for *planned reuse* only, i.e., when the developer specifying the reused use case is aware that additional interactions are going to be added. This is the case, for example, when developing different features of a SPL. If the additional interactions form a coherent unit of subfunctionality, then the reused use case can explicitly include an *abstract subfunction-level use case* in one or several of its steps. The reusing application or feature then specifies a *concrete child use case* with the additional context-specific interactions as shown in Case B3.

If the reuse is planned, but the additional interactions do not form a unit of subfunctionality, then the reused use case can specify one or several explicit *extension points* in its interaction flow. The application or feature can then add new interaction steps at these extension points by means of the «extends» mechanism. This is illustrated in Case B2.

However, *unplanned reuse* of a complete use case, illustrated in Case B1, is not supported by standard use cases. It requires a new use case composition mechanism, depicted in the figure with a blue «augment» dependency. Advanced use case composition techniques are discussed in the following subsection.

Case C, shown at the bottom left of Figure 2, identifies the situation where there is a recurring interaction pattern that a developer wants to make reusable. This is useful when, for example, a certain interaction pattern is required in order to deal with a specific concern consistently. An example of such an interaction pattern would be, for example, to consistently ask for an acknowledgement message when communicating with an unreliable actor. This kind of reuse is therefore always planned, either because the concern is reused within several features of a SPL, or because the concern is useful in several applications.

Standard use cases do not support Case C out-of-the-box. First, a new unit of modularization is potentially required to group the recurring interaction pattern (shown by the dashed blue ellipsis). Furthermore, a new composition mechanism is required to apply the pattern to the interaction scenarios in the reusing use cases (depicted with the blue «apply» dependency).

Case D, depicted at the bottom right, highlights the opposite reuse situation. In this situation, which can occur in planned and unplanned reuse, a recurring pattern needs to be applied to multiple use cases of an application, either to implement a feature of an SPL, or to create a new version of an application augmented with interactions related to a specific concern.

Again, this reuse case is not supported by standard use cases, as they do not provide a module to specify a pattern (shown by the dashed blue ellipses) nor do they support the application of a pattern across multiple use cases (depicted by the «apply to» dependency in the figure).

Finally, for any of the above cases, *hierarchical reuse* must also be supported. In other words, reuse should not be limited to only one level, i.e., it should be possible to build a use case that can be reused on top of other reusable use cases.

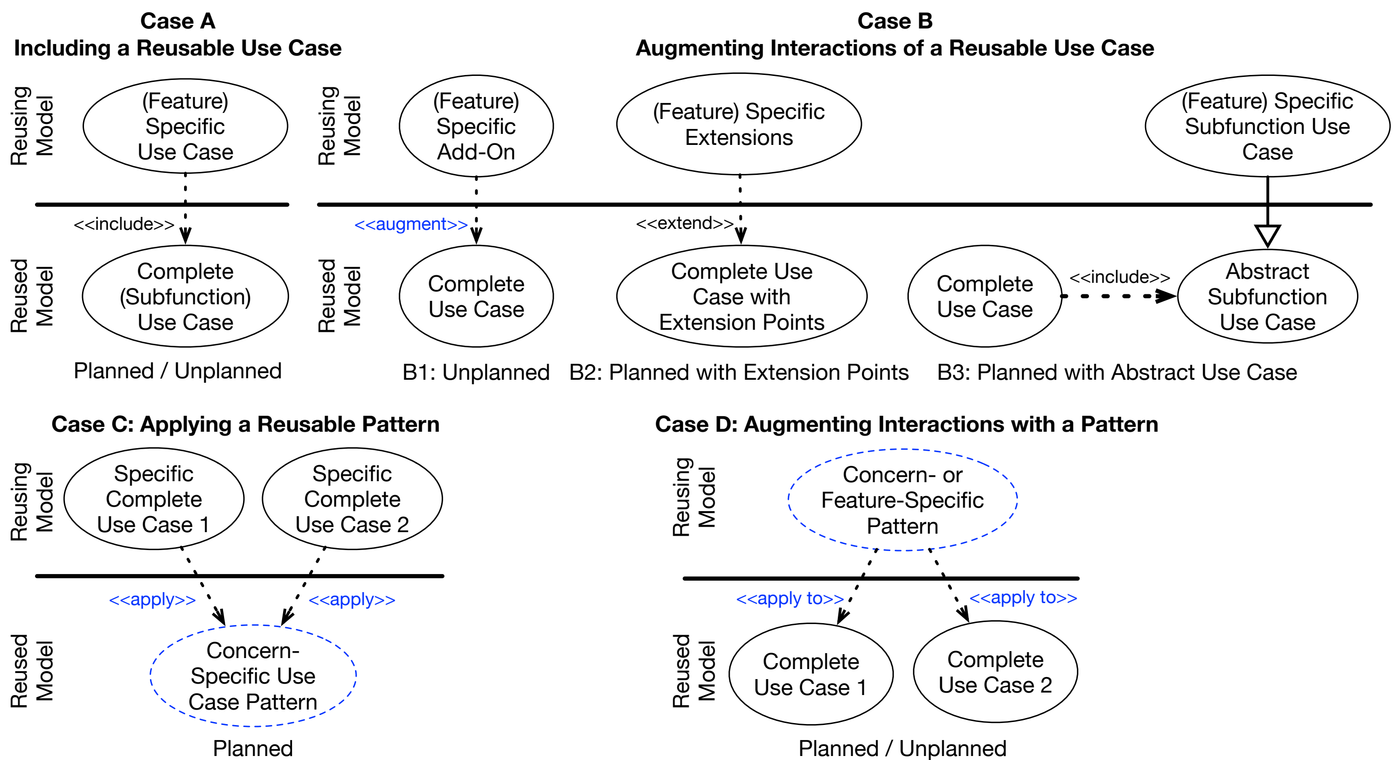

Figure 2 Use Case Reuse Situations

3.1. Aspect-Oriented Use Case Approaches

Use cases as defined by Jacobson (Jacobson et al. 1992) and standardised in UML do not support reuse cases C and D, where interaction patterns crosscut several use case scenarios. However, there have been several proposals aimed at modelling crosscutting interaction patterns in use cases in the aspect-oriented modelling community (*Aspect-Oriented Modeling Workshop Series* 2002–2010). The majority of these approaches focus on non-functional concerns, which often tend to be cross-cutting.

The earliest approaches for aspect-oriented use cases focused on extension and reuse within the context of a single model. Araujo & Moreira (2003) propose a use case modelling process that encouraged the discovery of “candidate aspects”, which are use cases (both functional and non-functional) that contain logic that is used in multiple other use cases. This is modelled using the standard inclusion relationship.

Jacobson & Ng (2004) propose a more elaborate approach which introduces the *use case slice*, which defines a partial use case model. It contains partial use cases or patterns, which can then be merged with other slices (matching by name) using merging algorithms inspired by aspect-oriented programming. They also discuss the use of aspect pointcuts, modelled just like extension points. They allow for the insertion of content from other use case slices. However, they do not elaborate a use case composition algorithm that explains how (aspect) models can be combined to generate a composed use case scenario description. Furthermore, because they require the specification of pointcuts in the base use case, they can not support unplanned reuse in cases B and D.

Lu & Song (2008) propose a new aspect use case template to describe non-functional use cases, while also defining join points in use cases where aspect use cases can add additional content. Because of the distinction between standard use cases and aspect use cases, the approach is asymmetric, which adds significant accidental complexity and prevents hierarchical reuse. Furthermore, the approach is also excessively restrictive in terms of which use cases are permitted to extend or be extended, which requires careful planning and certainly prevents most unplanned reuses.

In the SPL community, Eriksson et al. (2004) propose the "Product Line Use case modelling for Systems and Software Engineering" (PLUSS) approach, which introduces the concept of use case parameters to capture the variability of a use case model within the context of a complete software product line. As such, it supports only limited parameter-based planned reuse for cases A, B, C and D.

Somé & Anthonysamy (2008) introduce new relationships between use cases. In particular, they propose a new one-to-many «aspect» relationship, which links a single advice use case (which need not be a well-formed use case) to potentially many use cases from the base model, via a set of provided conditions and locations. They also define a «variability» relationship to support SPL development. The locations that can be targeted can be any use case description model element (such as a step or an extension point), and are called joinpoints. Unfortunately the composition algorithm that combines use cases is not presented in the paper, but the resulting use case executions are represented as Petri nets.

Approach	Kind	Unplanned	Composition Alg.	Hierarchy
Jacobson/Ng	Asym.	No	Not Presented	No
Lu/Song	Asym.	Partial	No	No
PLUSS	Sym.	No	No	Yes
Somé/Anthonysamy	Asym.	Yes	No	No
AoRUCM	Asym.	Partial	Yes	No
CoUC	Sym.	Yes	Yes	Yes

Table 1 Related Work Comparison Table

Most likely the approach by (Somé & Anthonysamy 2008) can deal with all reuse cases mentioned above. However, because the composition algorithm is not presented we were not able to verify this claim. Furthermore, the approach introduces advice use cases, which makes the approach asymmetric. As a result, hierarchical reuse, i.e., where a use case augments a use case that in turn augments a third use case, is not supported. Additionally, the approach defines new use case associations that were not previously defined in the UML specification (namely the «aspect» and «variability» associations). This introduces additional modelling complexity compared to traditional use cases. Finally, the proposed joinpoint model, while very powerful and expressive, adds an additional level of complexity to composition specifications.

The one approach that describes the proposed aspect-oriented composition algorithm in detail is Yue and Ali's *Aspect-Oriented Restricted Use Case Model* (AoRUCM), an aspect-oriented extension of the RUCM template discussed previously (Yue & Ali 2012). AoRUCM allows modellers of an aspect use case model to define a pointcut, containing either a set of actors (an actor pointcut), or a set of use cases with additional specifications for flows and steps (a use case/flow pointcut). Pointcuts are defined in OCL, and can therefore be validated for correctness. AoRUCM defines a composition algorithm which accepts two models and the relevant pointcuts and combines them into a single combined model. The algorithm supports both composition of inclusion or extension relationships. AoRUCM also defines pre- and postconditions for flows, and allows for composition of these conditions through the use of "and" and "or" operators.

This approach is much more robust than the others mentioned previously, and allows for a great degree of flexibility in defining aspect use case models. Furthermore, aside from the inclusion of new model elements in the use case diagram representing the pointcuts, AoRUCM does not introduce any new notation or base model elements to the standard RUCM template, making it more straightforward to use when coming from a conventional modelling background. However, AoRUCM is asymmetric, and hence does not support hierarchical reuse. Also, the AoRUCM composition algorithm does not allow for the merging of use case flows. Instead, it only allows for the insertion of inclusion or extension steps at various points in the text of the base use case. Similarly, AoRUCM does not allow for an extending use case to augment the existing base use case with new alternative flows, i.e., it does not support Case B1 or some scenarios that can occur in Case D.

The aforementioned related works and how they compare to CoUC are summarised in Table 1.

4. Concern-Oriented Use Cases

None of the existing approaches reviewed in the previous section support all four cases of reuse scenarios, and furthermore the asymmetric aspect-oriented approaches do not support hierarchical reuse. We therefore propose in this section *Concern-Oriented Use Cases* (CoUC), a use case modelling language that follows the Concern-Oriented Reuse (CORE) paradigm to enable planned and unplanned reuse of use case models across applications and across features within a SPL for all four reuse scenarios as well as in hierarchical settings.

This section starts by reviewing the main concepts of CORE, and then introduces the CoUC metamodel. The last subsection then presents the CoUC model composition algorithm.

4.1. Concern-Oriented Reuse

Concern-Oriented Reuse (CORE) (Alam et al. 2013; Schöttle et al. 2016) is a reuse paradigm for general-purpose software development that combines best practices from Model-Driven Engineering (MDE), Component-Based Software Engineering (CBSE), Software Product Lines (SPL), advanced Separation of Concerns (SoC) (including feature-oriented and aspect-oriented software development), and goal modelling.

In CORE, software development is structured around modules called *concerns*. They are units of reuse that encapsulate a variety of reusable solutions (i.e., models and code) for recurring software development issues in a versatile, generic way. Applications are built by reusing existing concerns from a *concern library* whenever possible, following a well-defined reuse process supported by clear interfaces. To generate an executable in which concerns exhibit crosscutting structure and behaviour, CORE relies on additive software composition techniques, as well as feature-oriented and aspect-oriented technology.

Concern Interface A concern provides a well-defined, three-part interface (Kienzle et al. 2016). The *Variation Interface* (VI) of a concern is composed of a feature model (Kang et al. 1990) that expresses the closed variability (Svahnberg et al. 2005) of solutions and techniques encapsulated within the concern by the *concern designer*, similar to what is done in SPLs for a specific application domain. Additionally, the VI specifies the impacts of selecting a feature on non-functional goals and qualities with an impact model that is expressed using a variant of the Goal-oriented Requirements Language (International Telecommunication Union (ITU-T) 2012). This allows a *concern user* to perform trade-off analysis between offered features, e.g., for design-time exploration.

For example, Figure 3 shows the VI for an *Authentication* concern. The feature model on the left specifies that the concern offers the choice of three features for authenticating (*Password*, *Retinal Scan* or *Voice Recognition*). Optionally, passwords can be set to expire (feature *Password Expiry*), access for users can be blocked when there were too many unsuccessful authentication attempts (feature *Access Blocking*), and the system can also automatically log out users that are idle for too long (feature *Auto Logoff*). The impact model on the right shows how each one of the features impacts the desired goals of increasing security, decreasing cost and increasing user convenience.

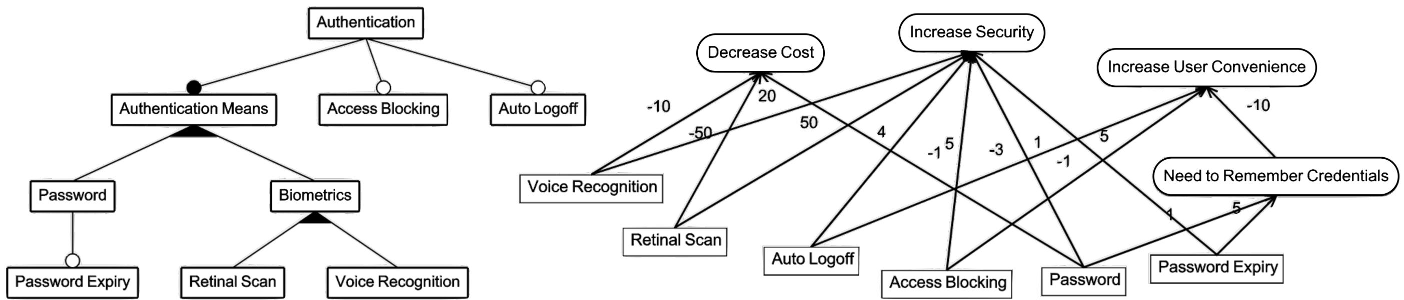

Figure 3 Variation Interface of the *Authentication* Concern

The *Customisation Interface* (CI) of a concern designates the generic, partially defined structural and/or behavioural elements that enable open variability (Svahnberg et al. 2005), and how these elements have to be connected to application-specific elements when the concern is reused. For example, an *Authentication* concern might define a generic *User* actor that is designated as partial and needs to be mapped to one or several application-specific actors, e.g., *Administrator*, *ProjectManager*, and/or *Developer*, when it is being reused.

Finally, the *Usage Interface* (UI), similar to classic APIs, designates the elements of the concern that can be accessed by the reusing context. Any element that has its visibility set to *public* is part of the usage interface and can therefore be used. For example, an *Authentication* concern might offer public services to authenticate or to update the credentials of users.

CORE Reuse Process The reuse process in CORE is based on three steps guided by the VCU interface: 1) Choose the desired solution among the available encapsulated alternatives from the VI, 2) Map any generic or partial elements of the realisation of the chosen solution (generated based on the selection made in step one) to the specific reuse context using the CI, and 3) Use the chosen, customised realisation (generated based on the mappings from step 2) for the targeted purpose via the UI.

4.2. CoUC Metamodel

In this subsection, we elaborate the metamodel definition of the CoUC modelling language.

4.2.1. Use Case Diagram Figure 1 shows an example use case diagram that encapsulates different ways of *Authentication*. It contains actors, use cases and relationships between them.

Figure 4 shows the CoUC metamodel that captures the concepts of use case diagrams. It is inspired by the use case metamodel of UML (Object Management Group 2017). The root of any model in CoUC is a *UseCaseModel*, which contains both *Actor*s and *UseCase*s. *Actors* and *UseCases* can be connected to one another via a *primary*, *secondary*, or *facilitator* role.

We allow *Actors* and *UseCases* to have a *parent*, as well as to be declared *abstract*, which enables inheritance and polymorphism within use case models. The use of inheritance for use cases is purposefully left ambiguous in UML. In practice, in concern-oriented use cases, we use inheritance primarily to model cases where there is a high-level, abstract goal that can be achieved in different concrete ways. This type of situation occurs very frequently in concerns, typically when a feature model contains an OR group of features that represent different ways to accomplish a particular goal. In this case, the base feature defines an abstract use case representing a high-level goal, and then each feature in the OR group defines a concrete use case that inherits from the abstract one and achieves the goal with feature-specific interaction steps. For example, in the *Authentication* concern in Figure 1, the *Authenticate* use case includes an abstract subfunction use case *Enter Credentials*. How credentials are actually entered depends on the features that are activated. In our case, three features are available, each one modelled as a concrete child use case involving different secondary actors. For example, *Voice Recognition* might require using a microphone, whereas *Password* requires entering a password using a keyboard. Each concrete child use case can specify what *Condition* has to be verified so it executes[1].

4.2.2. Use Case Details One of the goals of CoUC is to enable the analysis of use case behaviour, as well as generate new artefacts from the use case models using model transformations, e.g., Use Case Maps (International Telecommunication Union (ITU-T) 2012) or BPMN (Object Management Group 2014). It is therefore important that the flow of input and output interactions is clearly expressed in the text.

We chose to follow the textual templates proposed in (Sendall & Strohmeier 2000) and (Yue et al. 2009). As an example, Figure 5 shows the textual details of the *Authentication* base model. The metamodel that defines the language elements to capture the details for the textual scenarios in CoUC is shown in Figure 6. It is inspired by the RUCM approach as proposed by (Yue & Ali 2012).

The details of a use case are primarily defined by their main success scenario, which is an instance of a *Flow*. Flows contain a sequence of *Step*s, which define the sequence of interactions that describe the use case behaviour. There are four possible kind of steps:

1. *CommunicationStep* is used to model interactions between the system and one or more actors in the environment. Communication step descriptions must contain the name of at least one actor associated with the use case. The direction of the communication (*input* to the system, or *output* from the system) is also stored. For example, in Figure 5, step 2 is an output communication step.

[1] Conditions are explained further in the following subsection.

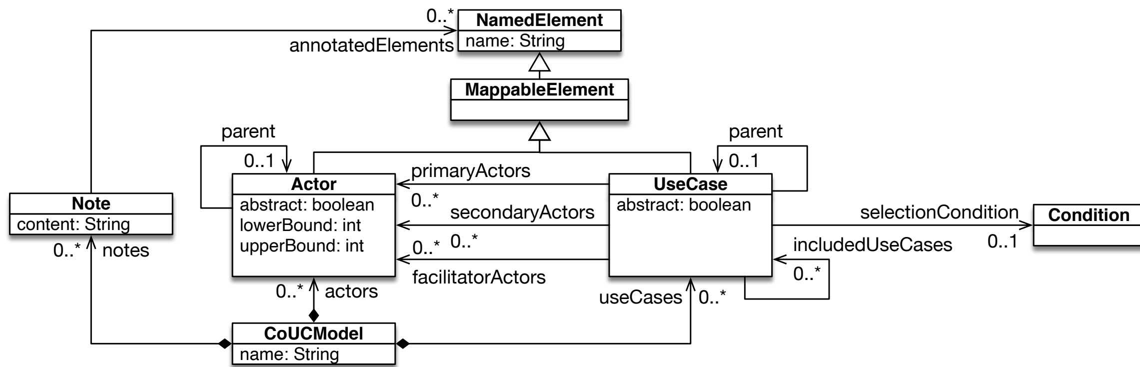

Figure 4 CoUC Metamodel - Use Case Diagram Part

Name: Authenticate

Level: UserGoal

Intention: The intention is for the User to authenticate to a system.

Multiplicity: Multiple Users can authenticate simultaneously.

Primary Actors: User

Main Success Scenario()

1. User provides their credentials to the system. (ref. Enter Credentials)
2. System informs User that they have successfully authenticated. <<out>>

Extensions

2a. System determines that the provided credentials are incorrect.

2a.1. System informs the User that the provided credentials are incorrect.

Use case continues at step 1.

Figure 5 Authenticate Use Case Details

2. *UseCaseReferenceStep* is used when one use case wants to execute the behaviour of another one, thus creating a dependency. Referencing a use case causes an «include» relationship to be drawn in the use case diagram. For example, in Figure 5, step 1 is a use case reference step that references the *Enter Credentials* use case.

3. *ContextStep* is used to describe actions or interactions that do not cross the system boundary. This includes internal system logic (to validate a condition of the system's state, for example), steps executed by actors external to the boundaries of the system, or control flow instructions.

4. *AnythingStep* is the only metamodel element in CoUC that specifically supports generic or partial scenarios. It is used as a placeholder to represent any number of steps of any kind. When composing two flows, the CoUC composition algorithm uses the *AnythingStep* to locate where to insert steps from a source flow (see subsection 4.4). Visually, the *AnythingStep* appears in the text as an ellipsis ("...").

Each *Flow* instance can also contain a set of nested *alternateFlows*. Alternate flows have to specify for which steps in the parent they are alternates for. If an alternate flow does not explicitly list any steps for which it is an alternate for, it is assumed to be global and applies to all steps in the parent flow. All communication, use case reference, and internal context steps are implicitly assigned a unique step number based on their position in the flow hierarchy using a Dewey numbering system. Alternates are highlighted with letters of the alphabet.

Each alternate flow also has a *triggeringCondition*, which is a step that describes the internal condition, input communication or timeout that triggers the alternative. The *Authenticate* example in Figure 5 has one alternate flow, where the internal context step 2a is the triggering condition. Finally, alternate flows can optionally define a *postCondition*.

4.2.3. References to Actors in the Text

The text of the majority of use case steps references one or several actors. It is important to ensure that actor names are kept consistent between the use case diagram and the text in the steps of all use cases. For example, renaming an actor in the use case diagram should propagate the new name to all steps automatically, rather than requiring tedious manual updates. To this aim, the CoUC metamodel defines a metaclass *TextWithActor* that is used instead of primitive strings whenever the model needs to store a string of text. The class internally stores the text, but replaces actor references by tokens. It also stores a list of the actors referenced in the text, which is used to query the name of the actor whenever the text content is accessed.

4.3. Customization and Usage Interface

Following the CORE paradigm, the unit of reuse in CoUC is the *concern*, and hence it needs a *VCU* interface. The VI always takes the form of a feature model and associated goal model, and is provided by the CORE framework. The CI and UI on the other hand are different for each modelling language and need to be carefully designed so that the composition algorithm can support all the reuse scenarios outlined in section 3.

Customisation Interface To define the CI for CoUC, we need to decide which model elements from the reused use case model can be designated as partial (generic), and therefore can be mapped during the reuse process to (reuse context-specific) model elements in the reusing use case model. For example, the generic *User* actor of the *Authentication* concern might be mapped to a specific *BankManager* during reuse. Furthermore, in order to support composition of behaviour for reuse scenarios B, C and D, we must allow the concern users to specify in which

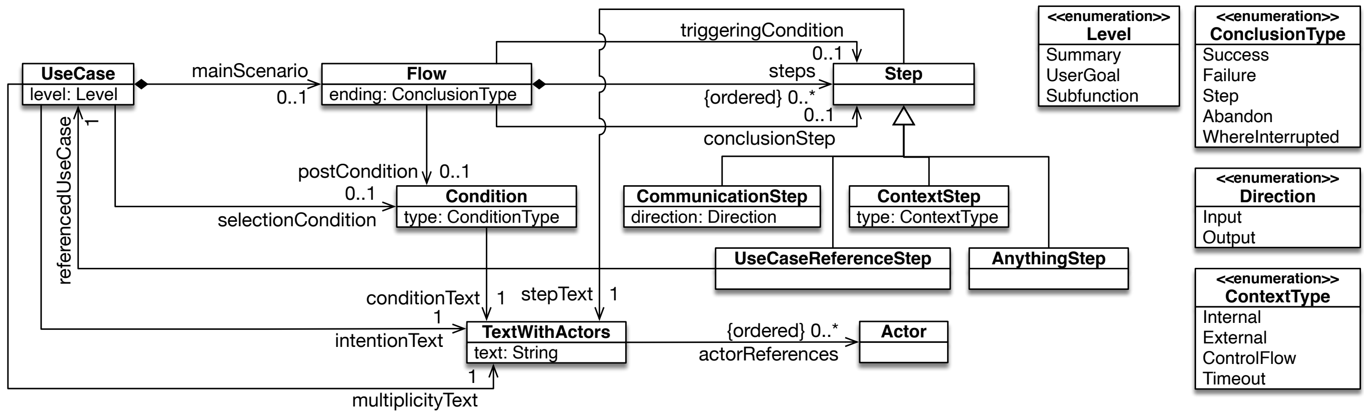

Figure 6 CoUC Metamodel - Use Case Textual Details

order the steps of the flows of the reused and reusing use case are combined. This can be achieved by establishing mappings between flows, and mappings between steps.

In the CI of a CoUC use case model, we therefore include:

1. *Actors* and *UseCases*, to specify which structural elements of two use case models should be composed.

2. *Flows* and *Steps*, to designate the behavioural elements that should be combined.

Usage Interface The UI for CoUC is straightforward, since the only usable elements in a use case model are the use cases. By default, all use cases defined in a concern are *public*, i.e., they are visible and accessible when the concern is being reused. However, we allow the modeller to change the visibility to *concern* if so desired. This makes sense, for example, when a model defines subfunction level use cases that should not be invoked by a concern user.

4.4. Concern-Oriented Use Case Composition

This subsection describes the CoUC use case composition algorithm, which in accordance with aspect-oriented modelling terminology is called a *weaver*.

4.4.1. Weaving of the Use Case Structure In CORE, a weaver takes as an input a *source* model, a *target* model, and a set of customisation mappings. The algorithm composes all model elements from the source model into the target model according to the customisation mappings, thus modifying the target model. When weaving across concern boundaries, the reused model is the source model, and the reusing model is the target. When weaving within a concern, the reusing model is the source, and the reused model is the target.

Previous work has shown that structural weaving can be reduced to a *merge* operation: when two structural model elements A and B are to be composed, they are replaced with a single model element that contains the union of the properties of both A and B (Kienzle et al. 2019). When the two model elements that are composed both have the same property that stores a single value, then typically one value is kept and the other discarded. If one of the model elements is *partial*, i.e., part of the customisation interface, then the value of the non-partial one is kept. In the case where none of them is partial, the weaving algorithm needs to know whether it is weaving with a concern or across concern boundaries to correctly determine which value to keep. In case of cross-concern weaving, the reusing model is more specific, hence the target model value takes precedence. In case of weaving within a concern, the child feature model value is more specific, hence the source model value takes precedence.

Algorithm 1 shows how this structural merge is achieved for CoUC using pseudo-code. Lines 2 to 10 copy use cases, actors and conditions that have not been mapped from the source model to the target model.

Lines 11 to 30 describe how a use case is merged. In the case where the algorithm composes within a concern and the source use case has not been designated as partial, the first two lines copy the name, intention and multiplicity values of the source into the target. Otherwise the name, intention and multiplicity of the target use case are kept. In case the target use case has no parent, the following two lines copy the inheritance relation from the source to the target, if any. Then, the primary actors, secondary actors and facilitator actor references from the source are added to the ones already existing in the target.

Finally, the algorithm iterates through all the flows in the source use case in hierarchical order, i.e., starting with the main success scenario and then going through the nested flows. If a flow is not mapped, no composition of behaviour is actually required. This is the case for reuse case A in Figure 2. The reusing model includes a placeholder use case (i.e., a use case which does not have a main success scenario), and the placeholder is mapped to the use case in the reused model. It suffices therefore in this case to copy the main success scenario (and any contained alternate flows) from the source into the target.

For correctly dealing with reuse situations B, C, and D, however, behavioural composition is required. In this case, the source flow is mapped to a flow in the target, and depending on the situation the concern user has also specified mappings from steps of the source to steps of the target. For these cases, the structural weaving algorithm triggers the behavioural weaving

Algorithm 1 Structural Use Case Weaving

```
 1: procedure WEAVEMODEL(targetModel, sourceModel, mappings, isWithinConcern)
 2:     for unmapped sourceUseCase in sourceModel.useCases do
 3:         newUseCase ← copy of sourceUseCase
 4:         add newUseCase to targetModel
 5:     for unmapped sourceActor in sourceModel.actors do
 6:         newActor ← copy of sourceActor
 7:         add newActor to targetModel
 8:     for unmapped sourceCondition in sourceModel.conditions do
 9:         newCondition ← copy of sourceCondition
10:         add newCondition to targetModel
11:     for mapped sourceUseCase in sourceModel.useCases do
12:         targetUseCase ← get mapped use case for sourceUseCase from mappings
13:         if isWithinConcern and sourceUseCase is not partial then
14:             copy name, intention, multiplicity from sourceUseCase to targetUseCase
15:         if targetUseCase intention / multiplicity is not set then
16:             copy intention / multiplicity from sourceUseCase to targetUseCase
17:         if targetUseCase has no parent use case and sourceUseCase has a parent then
18:             targetUseCase.parent ← sourceUseCase.parent
19:         for primaryActor in sourceUseCase.primaryActors do
20:             add primaryActor to targetUseCase.primaryActors
21:         for secondaryActor in sourceUseCase.secondaryActors do
22:             add secondaryActor to targetUseCase.secondaryActors
23:         for facilitatorActor in sourceUseCase.facilitatorActors do
24:             add facilitatorActor to targetUseCase.facilitatorActors
25:         for sourceFlow in all flows contained in sourceUseCase do
26:             if sourceFlow is mapped then
27:                 targetFlow ← get mapped flow for sourceFlow in mappings
28:                 WEAVEFLOWS(targetFlow, sourceFlow, mappings, isWithinConcern)
29:             else
30:                 copy sourceFlow to targetUseCase
31:     for mapped sourceActor in sourceModel.actors do
32:         targetActor ← get mapped actor for sourceActor from mappings
33:         if isWithinConcern and sourceActor is not partial then
34:             copy name and multiplicity from sourceActor to targetActor
35:         if targetActor has no parent and sourceActor has a parent then
36:             targetActor.parent ← sourceActor.parent
37:     for unmapped sourceCond in sourceModel.conditions do
38:         targetCond ← get mapped condition for sourceCond from mappings
39:         if isWithinConcern then
40:             copy text from sourceCond to targetCond
```

algorithm by calling `weaveFlows` as shown in line 28. The specifics of composing flows are described in section 4.4.2.

Once the use cases and their flows have been composed, lines 31 to 36 merge actors that are mapped. Again, in case an extension is being woven and the source actor is not partial, the name and multiplicity values from the source replace the ones in the target. If the target actor has no parent actor but the source has, then the parent relationship is also copied. Finally, lines 37 to 40 copy the source condition to the target use case in case an extension is being woven.

4.4.2. Behavioural Weaving of Flows

Previous work has stipulated that behavioural weaving can be reduced to *event scheduling*: two behavioural models that each define a partial order of events are composed by establishing a combined partial order respecting the original order as well as any additional constraints introduced by the composition instructions (Kienzle et al. 2019).

In CoUC the events are steps. Each flow is a sequence of steps, and the customisation mappings designate the steps that the concern user has designated as "equal". As a result, and since we are weaving the source flow into the target flow, the algorithm only has to *determine for each unmapped step in the source flow where to insert it into the target flow*. The insertion points in the target are either right before or right after a mapped step.

Algorithm 2 Behavioural Weaving of Flows

```
 1: procedure WEAVEFLOWS(targetFlow, sourceFlow, mappings, isWithinConcern)
 2:     currentTargetIndex ← 0, stepsBetweenMapped ← false, anythingFound ← false
 3:     for i = 0; i < sourceFlow.size; i++ do
 4:         sourceStep ← get step at index i from sourceFlow
 5:         if sourceStep is mapped then
 6:             mappedTargetStep ← get corresponding mapped step from mappings
 7:             assert( not stepsBetweenMapped or anythingFound )
 8:             stepsBetweenMapped ← false, anythingFound ← false
 9:             currentTargetIndex ← index of mappedTargetStep
10:             if isWithinConcern and sourceStep not partial then in targetFlow
11:                 replace mappedTargetStep with copy of sourceStep in targetFlow
12:         else if sourceStep is AnythingStep then
13:             currentTargetStep ←get step at index currentIndex from targetFlow
14:             nextMappedStep ←get next mapped step in targetFlow
15:             while currentTargetStep != nextMappedStep do
16:                 currentTargetStep ←get step at index currentTargetIndex
17:                 currentTargetIndex++
18:             anythingFound ← true
19:         else
20:             add copy of sourceStep to targetFlow at position currentTargetIndex
21:             stepsBetweenMapped ← true
22:         currentTargetIndex++
```

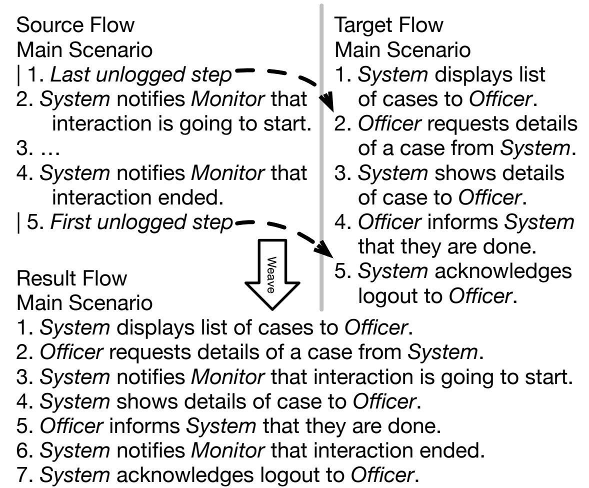

Figure 7 Example Weaving of Flows

When multiple steps are mapped, the placement of the *Anything* step is required to determine the insertion point unambiguously as illustrated in Figure 7. In the example, a generic monitoring interaction pattern is applied to a user goal use case describing an interaction between a police officer and a juridical system. It illustrates reuse case C (if the monitoring pattern is reused from a concern) or case D (if the monitoring pattern is a feature of the juridical system). The first step from the customisation interface |*1. Last unlogged step* was mapped to step 2 in the target model, and |*5. First unlogged step* is mapped to step 5. If the *Anything* step would not be present in the source flow, it would not be clear whether only step 2 from the source should be inserted after step 2 in the target, or whether step 2 and step 4 should be inserted after step 2. The *Anything* step matches the steps between 2 and 5 in the target, i.e., steps 3 and 4. Hence step 2 of the source must be inserted just after step 2 in the target, and source step 4 just before target step 5.

Algorithm 2 presents the pseudo code of the behavioural weaving algorithm that composes a *sourceFlow* into a *targetFlow* according to provided composition mappings. The algo-

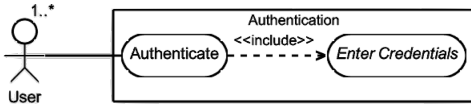

Figure 8 *Authentication* Root Model

rithm starts at the beginning of the target flow by initializing *currentTargetIndex* to 0. Then it loops successively through the steps of the source flow. There are three possible cases:

1. If the source step is mapped to the target flow, then a check is made that the composition specification is unambiguous (line 7). If the check passes, and the algorithm is composing within a concern and the source step is not partial, then the step content from the source replaces the target step, as it contains the more specific description (line 11). Otherwise nothing needs to be done, as the target step already contains the more specific description of the step.

2. If the source step is an *Anything* step, then the algorithm skips over steps in the target until the next mapped step or the end of the flow is reached by incrementing *currentTargetIndex* (lines 15 to line 18) and then remembering that an *Anything* step has been encountered.

3. In any other case, the unmapped source step is simply added to the *targetFlow* at position *currentTargetIndex*. The algorithm then remembers that it has encountered steps in the source that were not mapped.

5. Validation

To validate CoUC, we have implemented the language and the composition algorithm in the TouchCORE tool (*TouchCORE Website* 2023). We used TouchCORE to design three example concerns. They showcase *all of the reuse cases* elicited in section 3, as well as hierarchical reuse. All of the figures shown in this section are screenshots of models created or generated by TouchCORE.

5.1. Authentication Concern

The *Authentication* concern that was already partially introduced in the previous sections demonstrates model reuse between features. The root model for *Authentication*, shared between all features, is shown in Figure 8. It defines the base *Authenticate* use case, which calls the abstract *Enter Credentials* use case as shown in the use case details presented in Figure 8.

Figure 9 shows how the use case model of the *Password* feature defines a concrete child use case *Enter Password* and a new *Keyboard* actor. This model illustrates reuse scenario B3 of Figure 2. This model only requires structural weaving, as no flows are mapped. The figure also shows the use case diagrams for the *Retinal Scan* and *Voice Recognition* features, but the textual details have been omitted for space reasons.

Figure 10 shows the use case model of the *AutoLogoff* feature. It illustrates case A, because the feature-specific use case *Authenticated Interaction* includes the *Authenticate* use case that is defined in the root feature. It also illustrates case C, because the *Authenticated Interaction* use case defines a generic, *AutoLogoff*-specific pattern. When *Authentication* is reused with the *AutoLogoff* feature enabled, the concern user will map the partial *Authenticated Interaction* use case and in particular step 1 "*System is in an authenticated action.*" to all the steps in the reusing model that are to be performed in an authenticated state. When the reuse is woven, steps 1a, 1a.1 and 1a.2 are added as alternative flows to all the mapped steps of the reusing model.

Figure 11 shows the use case diagram of the *Access Blocking* and *Password Expiry* features. The textual details of *Access Blocking* illustrate reuse case B1. The *Authenticate* use case is not aware of the feature-specific addition to it, i.e., it does not specify extension points. The mapping step 1 $\rightarrow$ step 1 instructs the weaver to insert after the step that requests the credentials a check to determine whether the authenticating user's account is locked. If it is, a new alternative flow (2a) notifies the user about the situation and ends the use case in failure.

A second step mapping (2a -> 2b) ensures that the weaver adds a new step 2b.1 into the original alternative 2a that keeps track of the number of failed authentication attempts. It also adds an extension to the newly introduced step that deals with notifying the user that their account is blocked in case three unsuccessful attempts have been made.

Finally, the *AccessBlocking* model also introduces a completely new, stand-alone *Unlock User Account* use case. The textual details are not shown for space reasons.

The result of weaving all features of the *Authentication* concern (*Password*, *Password Expiry*, *Retinal Scan*, *Voice Recognition*, *Access Blocking* and *Auto Logoff*) is shown in Figure 1. The result of composing the textual details is not shown here for space reasons, but the interested reader can consult them in (Languay 2022).

5.2. Online Payment Concern

The *Online Payment* concern allows a user to pay for goods using any of the common payment paradigms accepted in typical e-commerce applications today. Figure 12 shows the feature model of the concern, depicting all the different payment variants that are supported.

The use cases for this example have been adapted from the use case maps described in (Siow 2018). For space reasons, we only show the use case model for the *Third Party* feature. The interested reader is invited to consult (Languay 2022) to look at the other detailed use case models for this concern.

The textual details of the *Third Party Payment* use case in Figure 13 reuse the *Authentication* concern described in the previous subsection with all of its features. This illustrates reuse case A with hierarchy support, since the generic *Authenticate* use case and *User* actor are reused, and then exposed as *Authenticate Customer* and *Customer*, respectively, to users of the *Online Payment* concern. This reuse also showcases reuse case C. By mapping the generic step "*1. System is in an authenticated action.*" from the *Auto Logoff* feature (see Figure 10) to step 3 and 4 of the main success scenario, an alternate flow is added to the *Third Party Payment* main success scenario that automatically logs customers out of the system in case they are idle for too long during the payment process.

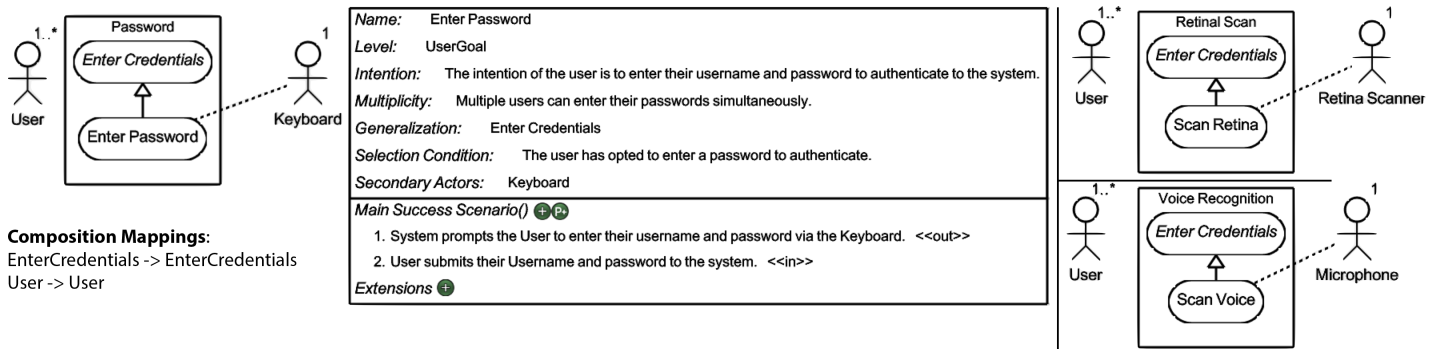

Figure 9 *Password* Feature of *Authentication* (and Structure of *Retinal Scan* and *Voice Recognition* Feature

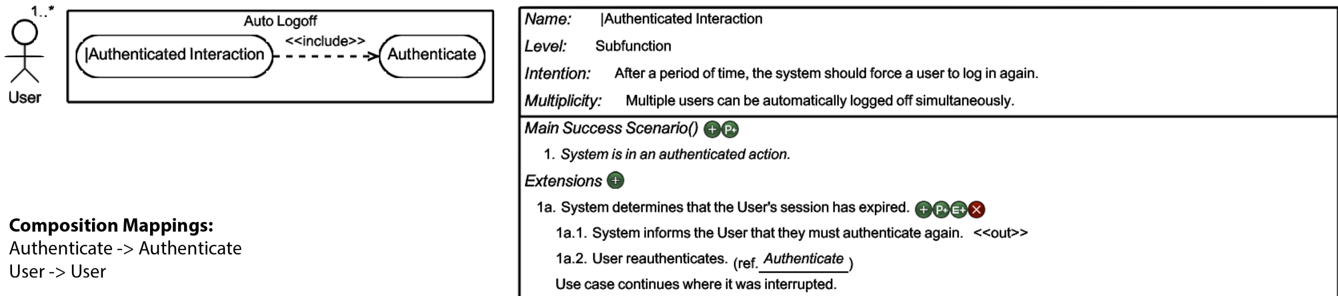

Figure 10 *AutoLogoff* Feature of *Authentication*

5.3. Elfenroads Product Line

Elfenroads (Rio Grande Games 2010) is a board game designed by Alan R. Moon The Elfenroads box set provides a version of the original base game, Elfenland, as well as the expansion Elfengold. The objective of Elfenland is to traverse the board using roads to visit as many towns as possible within for rounds. Roads can be made traversible through the use of transportation markers placed by any of the players, and on a player's moving phase they can traverse any adjacent marked road using travel cards. Elfengold adds the concept of gold to the game. Gold is awarded when visiting a town, and it can be used to purchase transportation markers in an auction at the start of each round. Elfengold also introduces spells, which can be used, e.g., to immediately transport a player to a particular town.

In McGill University's Software Engineering Project class for 2021/2022, students were tasked with designing and implementing a networked version of Elfenroads. The game had to be playable by multiple players, and support both the Elfenland ruleset and the Elfengold ruleset. Players should be allowed to create their own game lobby, which can then be joined by other players to play the selected game. To assist in the implementation, a generic game lobby service was provided to all students (Schiedermeier 2021). This lobby service offers functionality for user authentication and game session management, including creation, joining, starting, ending, saving and loading sessions.

We used CoUC to design the use cases of an Elfenroads SPL with an Elfenlands and an Elfengold feature. Figure 14 shows the woven use case model for a game configuration where both features were selected. The colouring of the use cases and actors illustrates the provenance of each model element. The white use cases and actors are defined in the root model, i.e., are common to both versions of the game. Elfenland and Elfengold have very different rules regarding the distribution of resources at the beginning of a round, hence an abstract use case *Distribute Resources* is specified in the base, planning for feature-specific extensions. Following the same reasoning, an abstract *End Round* use case is defined to deal with the different game rules that are applied at the end of a round depending on the chosen game version. The orange use cases are concrete child use cases related to the Elfenland feature (Case B3), whereas the purple use cases are introduced by the Elfengold feature (Cases A and B3). The purple stripe in the two use cases *Plan Route* and *Move Boots* defined by the root model indicate that the Elfengold feature used behavioural weaving to compose some additional behaviour and alternate flows into the root model to deal with the interactions required by the Elfengold game rules (Case B1). Finally, the green model elements come from a reused concern that encapsulates the functionality and interactions of a generic game lobby service (Case A).

For space reasons we do not show the individual use case models for the *Elfenroads* product line with the textual details. The interested reader is referred to (Languay 2022).

Elfenroads SPL Evolution Alan R. Moon subsequently released another variant of the game called Elfensea that is played on a different board with more towns. As an experiment, a student who had not worked on the Elfenroads SPL was asked to add an additional Elfensea feature to it to see how CoUC deals with this *unplanned addition*.

Again, the way Elfensea distributes resources and the rules

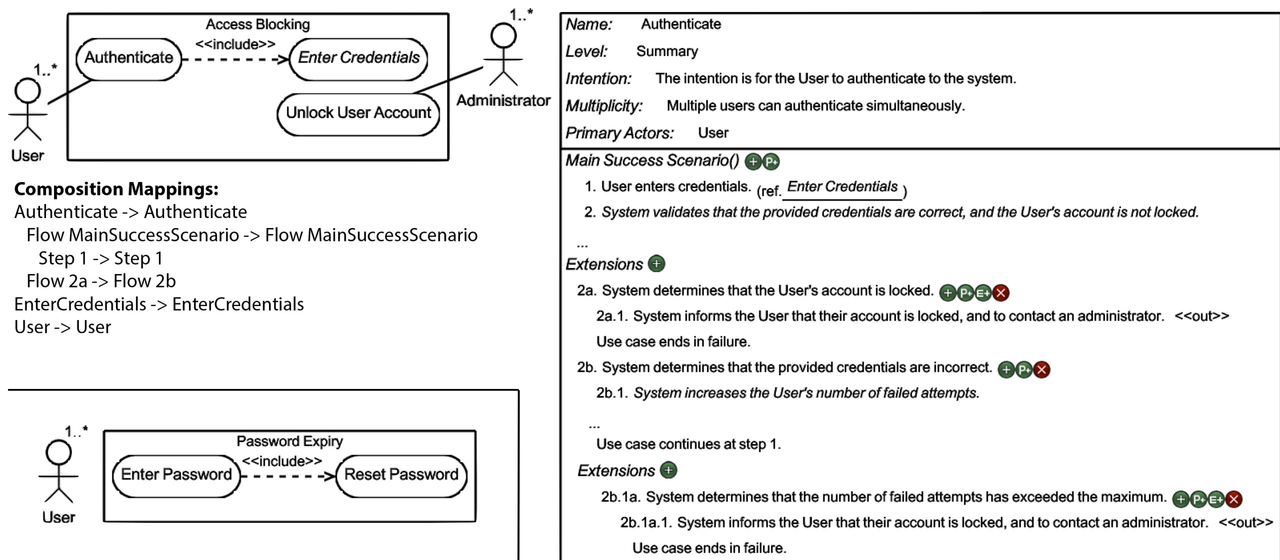

Figure 11 *AccessBlocking* Feature of *Authentication* (as well as Structure of *Password Expiry*)

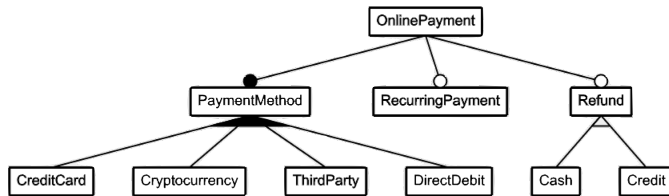

Figure 12 Feature Model of the *Online Payment* Concern

for ending a round differ significantly from the other two versions of the game. It was easy to deal with this by adding Elfensea-specific child use cases to *Distribute Resources* and *End Round*, respectively (Case B3).

Elfensea also introduces bonus markers which can be obtained when travelling to a town, and discarded during travel to obtain additional travel cards. To integrate these interactions into the SPL, the *Move Boot* use case was augmented to include additional alternative flows for obtaining and discarding bonus counters (Case B1 and Case D).

There was only one case that unfortunately required restructuring of the base use case. It turns out that while both Elfenland and Elfengold use obstacle markers, Elfensea does not. To maximize reuse between features, the *Plan Route* use case of the base model of the 2-feature version of the SPL included an interaction step for placing obstacle markers. This step had to be removed from the base model. To still maximize reuse, this step was moved into a new model that is shared only by the Elfenland and Elfengold features, and adds obstacle markers back into *Plan Route* (Case B1).

5.4. Discussion

The presented examples have been chosen because they illustrate all of the identified reuse scenarios. The CoUC metamodel, the customization mappings and weaver were powerful enough to allow the concern designer to modularize use case models according to features, and make them reusable within a concern in all three examples. Both the *Online Payment* and *Elfenroads* examples showed that it is possible for a concern user to reuse generic concerns and adapt them to a concrete context. Finally the *Online Payment* example demonstrated hierarchical reuse.

The examples also showcase the significant reuse potential of CoUC thanks to the integration of CORE. The *Authentication* concern, for example, contains 7 use case models. With the help of the CoUC weaver and the CORE reuse process, 72 different configurations of the *Authentication* use case model can be generated automatically, simply based on a feature selection of the concern user. For *Online Payment*, which also contains 7 use case models, 120 configurations can be generated. Since the *Third Party* feature of *Online Payment* reuses *Authentication*, a combined total of 8640 use case models can be generated.

Finally, the *Elfenroads* example shows the benefits of CoUC for non-technical stakeholders. Thanks to the modularization provided by CoUC it is possible to generate a textual description of the game interactions that can be read by the players either for just one version of the game, or for all game versions combined.

We also demonstrated that CoUC supports SPL evolution, as we were able to add all additional functionality of Elfensea to the Elfenroads SPL easily. The only difficulty encountered was due to the fact that some interactions that were shared by all features in the first version of the SPL were not required for the new Elfensea feature. CORE only supports additive composition (positive variability), and hence to deal with this situation the unneeded interactions had to be removed from the existing use case models. In case CORE in the future also supports model slicing (negative variability) then this might not be necessary any more.

6. Conclusion

This paper presented concern-oriented use cases (CoUC), a use case modelling language designed to support flexible reuse of

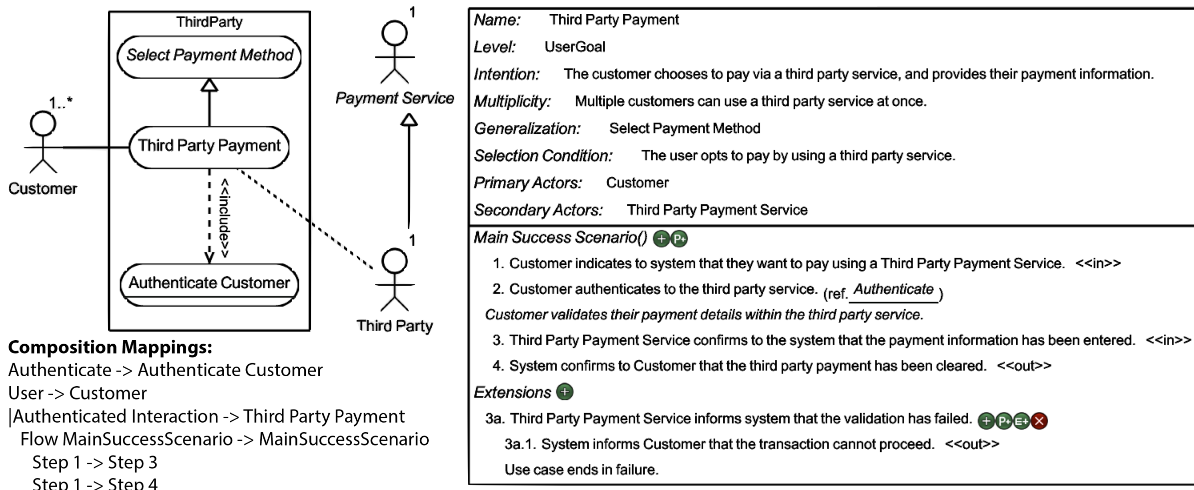

Figure 13 Use Case Model of the *Third Party* Feature of the *Online Payment* Concern

use case structure and behaviour. We presented a metamodel for the CoUClanguage, and detailed the algorithm of a model weaver capable of composing the use case structure and behaviour of two use case models, covering all possible reuse situations that can occur during standard and software product line development.

Following the concern-oriented paradigm, reusable use cases are encapsulated within a concern that exposes the different features the concern designer has planned for with a variation interface. Thanks to CoUC, the concern designer can share use case structure and behaviour among features (Case A), specify additional functionality introduced by a feature (Case B), or apply interaction patterns introduced by a feature consistently (Case D). The concern designer can also reuse other concerns internally if needed (hierarchical reuse). When a concern user reuses a concern, they can customize generic elements to specific ones, refer to the reused behaviour when needed (Case A), or apply reusable partial behaviour to one or many places in their use case models (Case C).

We validated CoUC by providing a complete implementation of the language, a GUI editor and a use case weaver in the TouchCORE tool (Schöttle et al. 2015). All figures in this paper were modelled using TouchCORE. We used this implementation successfully to model three different example concerns: an *Authentication* concern, an *Online Payment* concern that reuses the *Authentication* concern, and a software product line for the *Elfenroads* series of board games.

References

Alam, O., Kienzle, J., & Mussbacher, G. (2013). Concern-Oriented Software Design. In A. e. a. Moreira (Ed.), *Models* (pp. 604–621). Berlin.

Araujo, J., & Moreira, A. (2003). An Aspectual Use-Case Driven Approach. *Jisbd Journal*(January), 463–468.

Aspect-Oriented Modeling Workshop Series. (2002–2010).

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide* (2nd ed.). Addison-Wesley.

Burden, H., Heldal, R., & Whittle, J. (2014). Comparing and contrasting model-driven engineering at three large companies. In *Esem.* ACM.

Eriksson, M., Börstler, J., & Borg, K. (2004). Marrying Features and Use Cases for Product Line Requirements Modeling of Embedded Systems. In *Serps.* doi: 10.1.1.438.6729

Herrmann, C., Krahn, H., Rumpe, B., Schindler, M., & Völkel, S. (2007). An algebraic view on the semantics of model composition. In *Proceedings of the 3rd european conference on model driven architecture-foundations and applications* (p. 99–113). Berlin, Heidelberg: Springer-Verlag.

International Telecommunication Union (ITU-T). (2012). *Recommendation Z.151 (10/12): User Requirements Notation (URN) - Language Definition.*

Jacobson, I., Christerson, M., Jonsson, P., & Overgaard, G. (1992). *Object-oriented software engineering: A use case driven approach.* Addison Wesley.

Jacobson, I., & Ng, P.-W. (2004). *Aspect-Oriented Software Development with Use Cases.* Addison-Wesley.

Jacobson, I., Spence, I., & Bittner, K. (2011). *USE-CASE 2.0 The Guide to Succeeding with Use Cases.* Ivar Jacobson International.

Jacobson, I., Spence, I., & Kerr, B. (2016, apr). Use-case 2.0. *Communications of the ACM*, *59*(5), 61 – 69.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990, November). *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (Technical Report No. CMU/SEI-90-TR-21). Pittsburgh, Pennsylvania, USA: Carnegie Mellon University.

Kienzle, J., Mussbacher, G., Alam, O., Schöttle, M., Belloir, N., Collet, P., ... Rumpe, B. (2016, June). VCU: The three dimensions of reuse. In *Icsr 2016* (pp. 122–137). Springer.

Kienzle, J., Mussbacher, G., Combemale, B., & Deantoni, J. (2019, Jan 03). A unifying framework for homogeneous model composition. *Software & Systems Modeling*, *18*(5), 3005–3023.

Krueger. (1992). Software reuse. *CSURV: Computing Surveys*, *24*.

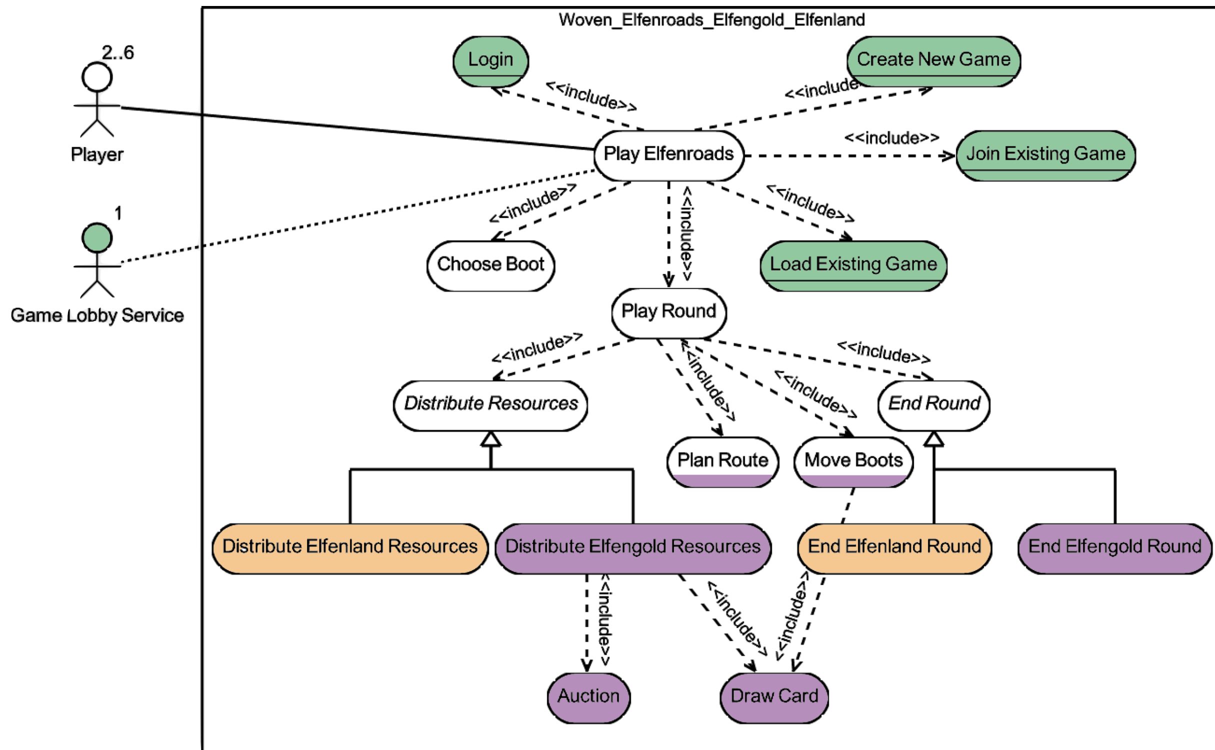

Figure 14 Woven Use Case Diagram for the Elfenroads Software Product Line

Languay, R. (2022). *Concern-Oriented Use Case Modelling* (Masters thesis). Montreal, Canada: McGill University. (https://escholarship.mcgill.ca/concern/theses/rn3016607)

Lu, C., & Song, I.-Y. (2008). A Comprehensive Aspect-Oriented Use Case Method for Modeling Complex Business Requirements. In *Lncs 5232* (pp. 133–143).

McIlroy, M. D. (1969). "Mass Produced" Software Components. In P. Naur & B. Randell (Eds.), *Software engineering* (pp. 138–155). Brussels: NATO.

Object Management Group. (2014). *Business Process Model And Notation.* Retrieved from https://www.omg.org/spec/BPMN/2.0.2/PDF

Object Management Group. (2017). *OMG Unified Modeling Language.* Retrieved from https://www.omg.org/spec/UML/2.5.1/PDF

Pohl, K., Böckle, G., & van der Linden, F. J. (2005). *Software product line engineering: Foundations, principles and techniques*. Secaucus, NJ, USA: Springer-Verlag.

Rio Grande Games. (2010). *Elfenroads.* Retrieved 2021-12-22, from https://www.riograndegames.com/games/elfenroads/

Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley. Retrieved from www.awl.com/cseng/

Schiedermeier, M. (2021). *LobbyService: Generic board game functionality.* Retrieved 2021-12-22, from https://github.com/kartoffelquadrat/LobbyService

Schöttle, M., Alam, O., Kienzle, J., & Mussbacher, G. (2016, mar). On the Modularization Provided by Concern-Oriented Reuse. In *Companion proceedings of modularity* (pp. 184–189). ACM. doi: 10.1145/2892664.2892697

Schöttle, M., Thimmegowda, N., Alam, O., Kienzle, J., & Mussbacher, G. (2015). Feature modelling and traceability for concern-driven software development with TouchCORE. In *Modularity 2015 Demonstration* (pp. 11–14).

Sendall, S., & Strohmeier, A. (2000). From Use Cases to System Operation Specifications. *Lecture Notes in Computer Science*, *1939*, 1–15.

Siow, C. C. (2018). *Concern-Oriented Use Case Maps* (Tech. Rep.). Montreal, Canada: McGill University.

Somé, S. S., & Anthonysamy, P. (2008). An approach for aspect-oriented use case modeling. In *Icse* (pp. 27–33).

Svahnberg, M., Van Gurp, J., & Bosch, J. (2005). A taxonomy of variability realization techniques. *Software: Practice and Experience*, *35*(8), 705–754.

Touchcore website. (2023). https://djeminy.github.io/touchcore/.

W3 Computing. (2020). Use Case Modeling. In *IS Management Handbook* (pp. 519–530).

Whittle, J. (2012). *The Truth about Model-Driven Development in Industry - and Why Researchers Should Care.* http://www.slideshare.net/jonathw/whittle-modeling-wizards-2012/.

Yue, T. (2010). *Restricted Use Case Modeling Approach User Manual.*

Yue, T., & Ali, S. (2012). *A Practical and Scalable Use Case Modeling Approach to Specify Crosscutting Concerns: Industrial Applications* (Tech. Rep.). Oslo, Norway: Simula Research Laboratory.

Yue, T., Briand, L. C., & Labiche, Y. (2009). A Use Case Modeling Approach to Facilitate the Transition towards Analysis

Models: Concepts and Empirical Evaluation. In *Models* (pp. 484–498). Denver: Springer.

About the Authors

Ryan Languay holds a master degree in Computer Science from the School of Computer Science of McGill University, Montreal, Canada. You can contact the author at Ryan.Languay@mail.mcgill.ca or visit https://www.linkedin.com/in/ryan-languay/.

Nika Prairie is an undergraduate student of the School of Computer Science of McGill University, Montreal, Canada. You can contact the author at Nika.Prairie@mail.mcgill.ca or visit https://www.linkedin.com/in/nika-prairie-5b0b0317a/.

Jörg Kienzle is full professor at the School of Computer Science of McGill University, Montreal, Canada. He holds a Ph.D. and Engineering Diploma from the Swiss Federal Institute of Technology in Lausanne (EPFL). His current research interests include model-driven engineering, concern-oriented software development, reuse of models, software development methods in general, aspect-orientation, distributed systems and fault tolerance. You can contact the author at Joerg.Kienzle@mcgill.ca or visit http://www.cs.mcgill.ca/~joerg/.