Concern-Oriented Behaviour Modelling with Sequence Diagrams and Protocol Models

Wisam Al Abed, Matthias Schöttle, Abir Ayed, Jörg Kienzle

School of Computer Science, McGill University
Montreal, QC H3A 0E9, Canada
{Wisam.Alabed,Matthias.Schoettle,Abir.Ayed}@mail.mcgill.ca,
Joerg.Kienzle@mcgill.ca

Abstract Concern-Oriented REuse (CORE) is a multi-view modelling approach that builds on the disciplines of model-driven engineering, software product lines and aspect-orientation to define broad units of reuse, so called concerns. Concerns specify the essence of a design solution and its different variations, if any, using multiple structural and behavioural views, and expose the encapsulated functionality through a three-part interface: a *variation*, a *customization* and a *usage* interface. Concerns can reuse other concerns, and model composition techniques are used to create complex models in which these concerns are intertwined. In such a context, specifying the composition of the models is a non-trivial task, in particular when it comes to specifying the composition of behavioural models. This is the case for CORE message views, which define behaviour using sequence diagrams. In this paper we describe how we added an additional behavioural view to CORE – the *state view* – that specifies the allowed invocation protocol of class instances. We discuss why Protocol Modelling, a compositional modelling approach based on state diagrams, is an appropriate notation to specify such a state view, and show how we added support for protocol modelling to the CORE metamodel. Finally, we demonstrate how to model using the new state views by means of an example, and explain how state views can be exploited to model-check the correctness of behavioural compositions.

1 Introduction

Model-Driven Engineering (MDE) [21] is a unified conceptual framework in which software development is seen as a process of *model production*, *refinement* and *integration*. Models are built representing different views of a software system using different *formalisms*, i.e., modelling languages. The formalism is chosen in such a way that the model concisely expresses the properties of the system that are important at the current level of abstraction. During development, high-level specification models are refined or combined with other models to include more solution details, such as the chosen architecture, data structures, algorithms, and finally even platform and environment-specific execution properties. The manipulation of models is achieved by means of *model transformations*. Model refinement and integration continues until a model (or code) is produced that can be executed.

MDE, while successful in many areas, still faces important challenges in practise. One main challenge is *model reuse* [25]: typically, models for a system under development are created from scratch, rather than reusing already existing models. This makes modelling more cumbersome than coding, since most modern programming languages offer extensive libraries that facilitate code reuse. Furthermore, models of complex applications tend to grow in size, to a point where even individual views are not readily understood or analyzable anymore. This is particularly true for behavioural models that are executable or used as source models for code generation, since they need to specify the behaviour in great detail.

Concern-Oriented REuse (CORE) is a multi-view modelling approach aimed at addressing the reuse and scalability issues of model-driven engineering. CORE extends MDE with best practices from research in both software product lines and aspect-orientation to define broad units of reuse, so-called *concerns*. These concerns specify the essence of a design solution and its different potential variations using multiple structural and behavioural views. The modelling notations used within a concern offer aspect-oriented features that make it possible to separate and modularize crosscutting properties and functionality. Currently, CORE incorporates feature models, goal models, class diagrams, sequence diagrams, and – thanks to the research described in this paper – protocol state machines, all with aspect-oriented extensions.

Concerns can easily reuse other concerns thanks to their three-part interface (a *variation*, a *customization* and a *usage* interface), thus creating concern hierarchies with complex dependencies. Syntactic or semantic model composition techniques [20] are used to flatten concern hierarchies and create complex models in which the concerns are intertwined that can then be executed or from which code is generated.

Specifying this composition is a non-trivial task, in particular when it comes to specifying the composition of behaviour. Structural composition of class diagrams in CORE boils down to merging of model elements. For two classes, for instance, this yields a new class that has the properties from both of the merged classes. Experience shows that the symmetric merge operation is conceptually easy to master by modellers.

Behavioural composition of sequence diagrams in CORE is asymmetric in nature. One sequence diagram can invoke an operation that is defined by another sequence diagram, which means that the weaver has to insert the behaviour of the called operation into the sequence diagram that made the call. Furthermore, aspect message views can modify the behaviour of a sequence diagram by adding additional behaviour before and/or after the already existing sequence of interactions. Understanding the resulting behaviour becomes tricky, in particular if the different behavioural specifications are scattered in multiple models. Also, as for all asymmetric approaches, the order of composition matters in the case where several aspect models want to add behaviour at the same place.

The complexity of understanding aspect dependencies and interactions, unwanted or wanted, has been a subject of study for many years now [18], and

recognized as a major problem in aspect-oriented software development. This paper presents how we integrated protocol modelling (PM) [15] into CORE to allow the modeller to specify operation invocation protocols for class instances. We present some important properties that make PM an adequate notation for expressing state views in CORE design models, and show how we integrated a restricted form of PM into the CORE metamodel. We then discuss how this restricted PM can be used by a modeller to specify state views, and how it can be exploited by our TouchCORE tool [2,1] to perform consistency checking of behavioural specifications that cross model boundaries.

The remainder of the paper is structured as follows. Section 2 reviews some of the most important concepts of CORE. Section 3 enumerates the requirements that a modelling notation needs to fulfill in order to be useful in the context of CORE. Section 4 shows how we introduced a restricted form of PM into the CORE metamodel. Section 5 discusses how we envision modellers to use our new state views within CORE by means of the AspectOPTIMA [10] case study. Section 6 elaborates how the PM approach satisfies our requirements. Section 7 summarizes related work, and the last section draws some conclusions.

2 Background on CORE

In contrast to model-driven engineering's focus on models, the main unit of modularization, abstraction, construction, and reasoning in concern-driven software development (CCD) is the *concern*.

A concern is any domain of interest to a software engineer. It can (but does not have to) be a crosscutting concern as advocated by aspect-oriented software development. A concern is a unit of modularization that encapsulates a *set of models* describing all properties of that concern required to sufficiently understand and use the concern. Typically, the models within a concern span multiple phases of software development and levels of abstraction. The models are built using the most appropriate modelling formalisms to express the properties of the concern that are relevant at each level of abstraction. Consequently, a concern is typically described by many modelling notations, which may be object-oriented in nature, but typically also need to offer other language mechanisms (e.g., aspect-oriented features) in order to properly handle the crosscutting nature of certain properties encapsulated within a concern. Finally, a concern also encapsulates all relevant variations/choices that are available/encapsulated within a concern, together with guidance on how to choose among those variations.

The key concept of concern-orientation promoting modularity is the *three-part interface* [3] that every concern must provide:

- The *Variation Interface* describes the available variations of the concern and the impact of different variants on high-level goals, qualities, and non-functional requirements. The variations of a concern are represented with a *feature model* [7] (described in more detail in subsection 2.1) that specifies the individual features a concern offers, as well as their dependencies. The

impact of choosing a feature on soft-goals and system qualities is specified with goal models [6].
- The *Customization Interface* describes how a chosen variant can be adapted to the needs of a specific application. Each variant of a concern is described as generally as possible to maximize reusability. Therefore, some elements in the concern are only *partially* specified and need to be related or complemented with concrete modelling elements of the application that intends to reuse the concern. The customization interface is hence used when a specific variant of a reusable concern is *composed* with the application.
- The *Usage Interface* describes how the application can finally access the structure and behaviour provided by the concern.

2.1 Designing a Concern

Building a concern is a non-trivial, time consuming task, typically done by or in consultation with a domain expert; it requires a deep understanding of the nature of the concern to be able to identify and specify all *features* of a concern. In CORE, this is done with feature models (see subsection 2.1.1). Once identified, each feature of a concern has to be realized, i.e., the structure and behaviour of the functionality it encapsulates needs to be specified. In CORE, this is done with Reusable Aspect Models (RAM) [12,9], an aspect-oriented multi-view modelling notation based on UML class and sequence diagrams (see subsection 2.1.2).

2.1.1 Feature Model The features of a concern include all prominent or distinctive user-visible aspects, qualities and characteristics related to the concern. Once identified, the concern designer summarizes all relevant variations of the functionality pertaining to a concern, as well as all extensions of the functionality, with a *feature model* [7]. Feature models specify the relationships and dependencies that exist between features effectively and express them visually as a tree with different parent-child relationships (*mandatory*, *optional*, *xor* and *or*) and cross-feature dependencies (*requires* and *excludes*) that are not limited to just parent-child relationships.

Figure 1 shows the feature model of a low-level software design concern called *Association*. It occurs very frequently in object-oriented designs that an object of class `A` needs to be associated with other objects of class `B`. Implementing associations with multiplicity `0..1` or `1` is easy, since it simply requires the class `A` to store a reference to an instance of class `B`. Implementing an association where the upper bound of the multiplicity is greater than `1`, e.g., `0..*`, can be done in many ways, and it is the job of a designer to determine the most appropriate way. Typically, the design has to introduce an intermediate collection data structure that stores the instances of `B` and refer to it from within class `A`. Operations need to be provided that add and remove instances of `B` from the collection contained in the object of class `A`.

What kind of collection to use depends on the functional requirements of the association. For example, an `{ordered}` association has to be designed with

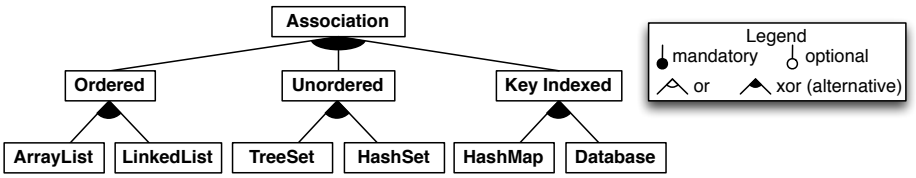


Figure 1: Association Concern Feature Model

a collection that orders the elements it contains, e.g., a queue (FIFO), a stack (LIFO), or a priority queue (sorted using some criteria). A qualified association has to be designed with some sort of dictionary or map that allows the retrieval of objects by means of a key. Furthermore, an abstract data structure may have many different internal implementations. For example, a queue can be structured internally as an array or a linked list, the choice of which affects the algorithms for insertion, deletion, and iteration. This ultimately impacts the non-functional properties of the application, e.g., memory usage and performance, which are expressed by means of the impact model (see subsection 2.1.5).

2.1.2 Realizing Features A concern can encapsulate complex functionality; however, this complexity is decomposed into several features that each modularize a coherent part of that functionality. The root feature of a concern, for instance, encapsulates structural and behavioural properties that are always present. Optional features modularize structure and behaviour that is only relevant when that particular feature of the concern is chosen.

In CORE, a feature is realized by a structural view (class diagram) and one or more message views (sequence diagrams). The class and sequence diagram notation used are extended with aspect-oriented features as described in the Reusable Aspect Models (RAM) approach [12,9]. This makes it possible for the realization models of one feature to augment the structure and behaviour of the realization models of their ancestor features.

Concretely, if feature A is the parent of feature B, then the model *BReal* that realizes feature B extends the model *AReal* that realizes the feature A. Because B is a sub-feature of A, the concern designer's intent is to add additional structural and/or behavioural model elements to *AReal* that provide *additional*, *alternative* or *complementary* properties to what already exists in *AReal*. By default, *BReal* has full visibility of all elements visible in *AReal*. As a result, *BReal* can use the structure and behaviour provided by *AReal* when needed. Furthermore, any model elements used in *BReal* that have the same name as an existing model element in *AReal* are considered to be the same, i.e., their properties are going to be merged by the TouchCORE tool using the weaving algorithms defined in [9] to produce a "woven" model that realizes both features. This allows *BReal* to augment the *customization* and *usage interface* of *AReal* with additional structure and behaviour.

2.1.3 Structural View – Class Diagrams To realize a feature, the concern designer specifies in the structural view the classes relevant to the feature, together with their attributes and operations, as well as any associations among classes. The notation used is UML class diagrams, with the additional possibility of marking classes, attributes and operations as *partial* by prefixing their name with a vertical bar: '|'. Partial model elements are included in the concern's customization interface, and designate model elements that are *general* from the point of view of the current concern, which means that they must be mapped to application-specific model elements before the concern can be used.

Because a concern is split into features, the structural view of a feature is of reasonable size (typically containing 1-6 classes with 1-10 public operations each). As a result, the complexity of a concern is split into several models that usually fit into the limited active *working memory* of a human [16]. This makes the elaboration, understanding and evolution of the models involved in the realization of the feature easy and less error prone [24] than if the entire concern model had to be specified in one model.

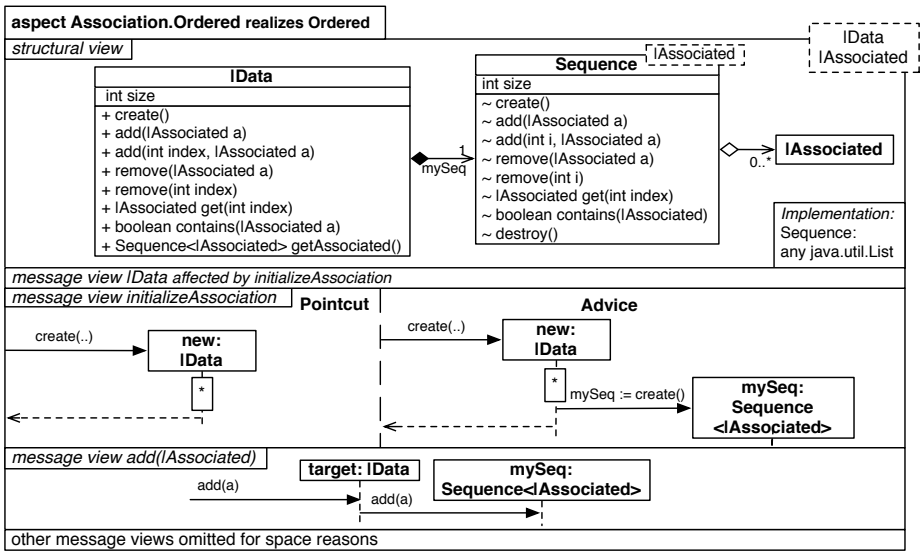


Figure 2: CORE-RAM Model realizing *Ordered* of the *Association* Concern

The top part of Fig. 2 shows the structural view of a CORE-RAM model that realizes the *Ordered* feature of the *Association* concern shown in Fig. 1. The structural view describes the structural design of an ordered association between two classes with a multiplicity of `0..*`. The structural view defines a partial class `|Data`, which uses a `Sequence` to link an instance of `|Data` to many instances

of the partial class `|Associated` in an ordered way. The class `Sequence` refers to a Java implementation of `java.util.List`.[1]

`|Data` and `|Associated` are partial classes, i.e., incomplete classes (highlighted by a vertical bar '`|`') that must be mapped to another class whenever this feature is used within another concern. These classes are partial since at this point it is not yet known what actual application classes will need to be associated with each other. All partial classes (and operations) are part of the customization interface of a CORE model: any model that wants to use the *Ordered* feature of the *Association* concern must map `|Data` and `|Associated` to two of its own classes.

In CORE, there are four types of visibility modifiers for operations: public (`+`), protected (#), private (`-`) and concern-private (`~`). All public operations are part of the usage interface of a CORE model, i.e., they are the operations that another model can invoke to trigger the behaviour realized by the feature. Protected operations can only be invoked from within the same class or subclass, private operations can only be invoked from within the same class, and concern-private operations can be called only by classes included in the same concern.

2.1.4 Message View – Sequence Diagrams Message views describe the behaviour of the feature being modelled. There is one message view for each public operation defined by a class in the structural view. Each message view describes the sequencing of message interchanges that occur between instances of classes of the concern when providing the functionality offered by the public operation.

In the *Association<Ordered>* model, the class `|Data` has eight public operations. All these operations involve interactions with an instance of the class `Sequence`. For space reasons, Fig. 2 only shows the message view for the constructor of `|Data` (`create`) and one of the `add` operations. The `add` operation illustrates how the call is forwarded to the `Sequence` class. Because the behaviour of the constructor of `|Data` is not known yet, it is advised using an *aspect message view* to initialize the `Sequence` class. The advice of `initializeAssociation` describes that after the behaviour of the specific constructor, the `Sequence` is created and assigned to `mySeq`.

2.1.5 Impact Model After realizing all the features of a concern, the concern designer has to specify the impact that the realization of each feature has on soft goals and system qualities. In CORE, this is done with impact models, which are based on goal modelling as defined by GRL [6]. For instance, the choice of *ArrayList* might provide better access performance than *LinkedList*. However, *ArrayList* uses more memory than *LinkedList*. Further details on how to define impact models are provided in [3].

[1] TouchCORE, the CORE tool, allows the reuse of existing classes provided by the programming language or frameworks being used.

2.2 Reusing Concerns

Once a concern has been designed, the expert knowledge and solutions encapsulated within can be reused whenever possible. While designing a concern is challenging, time consuming and requires in-depth domain expertise, reusing an existing concern is simple, and involves three steps:

1. A *concern user* must first *select the feature(s)* with the best impact on relevant soft-goals and system qualities from the variation interface of the concern based on the provided impact analysis. Based on this configuration, the TouchCORE tool then merges the models that realize the selected features to yield a new model of the concern corresponding to the desired configuration.
2. Next, the *concern user* has to adapt the generated concern realization model to the application context by mapping all customization interface elements, i.e., the model elements designated by a '|' prefix, to application-specific model elements.
3. Finally, the *concern user* can use the functionality provided by the selected concern feature's usage interface within his own application models. This typically consists of instantiating classes provided by the concern, and invoking their public operations from within the application-specific message views.

3 Requirements for Concern-Oriented Specification of Invocation Protocols

Specifying complex behaviour of a concern by composing several partial behaviours described in multiple message views is a non-trivial and error-prone task. Aspect message views can augment behaviour defined in other message views to include additional control flow directives and operation invocations. Furthermore, if several aspect message views are applied to the same base behaviour, the order in which the message views are applied usually changes the resulting behaviour.

In order to help the concern designer in defining correct behavioural specifications that extend the behaviour of parent realization models and to help the concern user to correctly invoke functionality provided by reused concerns, we decided to introduce an additional behavioural view into CORE-RAM that allows the concern designer to specify operation invocation protocols for the classes encapsulated within a concern. Using those protocol models, model checkers can verify that the behavioural specifications expressed in the sequence diagrams within a concern as well as the behavioural specifications obtained by composing models of reused concerns with the application model are valid, i.e., they do not violate any of the protocols defined for any of the features as well as for the application. This makes it possible to detect unwanted and incorrect behaviour resulting from feature and concern interactions and erroneous composition specifications.

In subsection 3.1 we list the requirements that a modelling notation needs to fulfill in order to be useful in the context of concern-oriented modelling. Then, in subsection 3.2 we briefly present the *Protocol Modelling* approach (PM), and show how we adapted it to fit our needs. Finally, in subsection 3.3 we highlight the key differences between protocol machines and state views.

3.1 Requirements for the Protocol Specification Notation

Based on experience gained from creating several software design concerns with CORE-RAM, the notation for expressing protocols must have the following properties:

1. **Expressiveness**: In CORE-RAM, the structural view presents the classes together with the operations they offer, and the message views present the interaction between objects when one of the public operations is invoked. This does not convey complete information on the order in which operations can be invoked on object instances. The notation we are looking for should support the specification of such invocation protocols, i.e., it has to be possible to state when an operation is allowed to be executed, and when it is forbidden.
2. **Conciseness**: The notation should be capable of specifying invocation protocols of class instances in a straightforward and concise way. This requirement is important to reduce accidental complexity [26].
3. **Diversity**: The notation should not be similar to sequence diagrams. This will force the modeller who is specifying the protocols to look at the design concern from a different point of view from how they specified the behaviour of the operations with sequence diagrams.
4. **Modularity**: To be useful in the context of concern-orientation, the modelling notation needs to be able to modularize the protocol of classes that belong to a feature of the concern. It should be possible to specify the protocols for the classes within a feature in isolation from the protocols of other classes of the concern.
5. **Composition**: Since a modeller can elaborate a complex design concern by composing multiple CORE-RAM feature realization models, the modelling notation for specifying protocols must support composition. Whether a child feature extends properties of parent features, or whether concerns reuse other concerns and therefore customize the realization model of the reused concern to their specific application, the protocols for object instances can change.
   To illustrate the kind of protocol compositions that the notation must support, imagine the following example: Feature A is the parent of feature B in concern R. The concern designer of R creates the realization model *BReal* that realizes feature B and extends the model *AReal* that realizes the feature A. The concern user of R selects A, and therefore customizes the model *AReal* by mapping its customization interface to model elements in the application-specific model *App* that he is building.
   The composition operator(s) of the protocol notation must support:

(a) **Adding New Operations**: Both the concern user (in *App*) and the concern designer (in *BReal*) might define new operations, which need to be integrated by composition with the protocol of the classes in *AReal*. In case of customization, the *public protocol* of classes in *AReal* need to be integrated with the protocol of classes in *App*. In the case of model extension, the *complete protocol* of *AReal* needs to be integrated with the complete protocol of *BReal*.
(b) **Adding Constraints**: Both the concern user and the concern designer might need to define additional constraints on the protocol of *AReal*. It should therefore be possible to restrict the public or the complete protocol of *AReal*, respectively, by forbidding the execution of an operation in certain cases.
(c) **Coupling Protocols**: A CORE-RAM model can depend on multiple other models. For example, *BReal* can both extend *AReal* and also reuse another concern *C*. In that case, it should be possible for the concern designer of *BReal* to specify a coupling between the protocols of *AReal* and *C* in the case where classes from *AReal* and *C* are mapped to the same class in *BReal*.

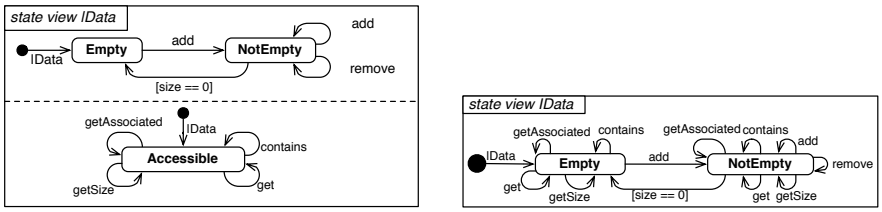
6. **Verification**: The protocol modelling notation should be appropriate for:
(a) **Verifying Internal Consistency** of CORE-RAM Models: It should be possible to verify that a *concern designer* is specifying the behaviour provided by the CORE-RAM model that realizes a feature in a consistent way, i.e., that there are no contradictions between the specified object invocation protocols and the interactions between objects specified in the message views.
(b) **Verifying Usage Consistency** for concern reuses: It should be possible to verify that a *concern user* is calling the public operations provided by the customized CORE-RAM model of a reused concern in the right order. In other words, the notation should support the definition of a *public protocol*.
(c) **Verifying Increment Consistency** for CORE-RAM models that extend parent realization models: It should be possible to verify that a *concern designer* that extends a CORE-RAM model realizing a parent feature is calling the operations provided by the parent in the right order. In other words, the notation should support the definition of a complete protocol that includes *public*, *concern-private* and *private* operations.
(d) **Verifying Composition Consistency** for woven CORE-RAM models: To verify that the concern designer or concern user have specified the behavioural compositions correctly, it should be possible to verify that all scenarios specified within woven message views are acceptable according to the combined protocol specifications of each model.

3.2 Specifying Protocols in a Concern-Oriented Way

Historically, protocols have been defined using state-based notations such as finite automata. State diagrams in general are also significantly different from sequence diagrams, which satisfies requirement 3. We therefore investigated several

state-based aspect-oriented approaches, including HiLa [27] and the framework designed by Elrad et al. [4] (details on these approaches and their limitations for expressing protocols are given in section 7). We ended up using the *Protocol Modelling* notation (PM) [15], which was specifically designed for modelling protocols and comes with a formally defined composition operator. In CORE-RAM, protocol models that specify invocation protocols are elaborated within separate *state views* (SV), one for each class that is present in the structural view.

The PM approach is based on the concept of a "Protocol Machine", a reusable behavioural component of the model that can either ignore, accept or refuse events that are presented to it. By associating a protocol machine with each class in a CORE-RAM model, we can therefore specify invocation protocols for all instances. This satisfies requirement 1. The PM approach is modular: a protocol model of a system is composed of a set of protocol machines and each machine describes a partial behaviour. This satisfies requirement 4. PM supports a highly compositional style of modelling; the partial behaviours are composed together to create the behaviour of the full system using a parallel composition operator (P || Q) as defined by Hoare in the Communicating Sequential Processes (CSP) approach [5]. CSP || composition has the advantage of being a well-defined and understood concept, and it enables the modeller to perform local reasoning on models. Furthermore, it gives the modeller the ability to deduce the properties of the whole system from the knowledge of the behaviour of the composed protocols [13].



(a) Orthogonal State Machines (b) Single State Machine

Figure 3: State View for the `|Data` Class in *Ordered*

Fig. 3a shows how we used PM to describe the protocol of the `|Data` class of the *Association<Ordered>* CORE-RAM model shown in Fig. 2. The state view of `|Data` has two state machines. The first one is describing the fact that calling an `add` operation changes the state of a `|Data` object from *Empty* to *NotEmpty,* and that `remove` operations should only be called after at least one `add` was invoked. Also, when removing the last `|Associated` object from the sequence of objects, the state of the `|Data` instance changes back to *Empty*. The second state machine describes the protocol for the getter and query operations. Since they do not alter the state of the object when called, there are no restrictions specified on their protocol. This is a great example on how PM supports

conciseness (requirement 2). Each state machine is simple: it only focuses on operation invocations and how they alter the object's logical state. To obtain the complete protocol for `|Data` instances, the modeller can apply the CSP operator mentally on the two state machines, which yields the composed state machine shown in Fig. 3b. Notice that the state view with two state machines, even for a simple class such as `|Data`, is easier to understand than the more cumbersome composed state view.

3.3 Differences between PM and State Views

PM is a powerful technique that was designed for a slightly different purpose. While our requirements are clearly focussed on documentation and verification, the goals of PM include system interaction modelling, protocol execution simulation, test and code generation. For that reason, we decided to adapt the main ideas of PM within CORE-RAM, but to omit some of the advanced features that we do not currently have a need for. In summary, the differences between the original PM approach and how we are currently using it in CORE-RAM are:

- While PM models specify any general event interchange, we only focus on modelling operation invocations.
- PM focusses on modelling interactions with a system, and therefore presents all the state machines of a system together. Consequently, all events are global (to the system being modelled), so they need to have unique names. To obtain the complete interaction protocol of the system, all state machines are composed with each other. In the case of a CORE-RAM model, only state machines that are contained in the same state view are related to each other (and logically composed with each other). Additional state view relationships are created when classes from different aspects are mapped together, which results in a logical composition of all the state machines that the respective views contain. Because of that, the events in CORE-RAM are not global to the whole system; they only need to be unique within all state views that are describing protocols of the same class. For example, the state view of `|Data` in Fig. 3 is composed of two state machines separated by a dashed line. These state machines are related to each other since they are in the same state view and CSP composition is logically applied to them. Conversely, the state machines of the state view `|Data` and those of `Sequence` (not shown in this paper) are independent, since they describe the behaviour of objects that are instances of different classes.
- PM introduces a new type of state – the *derived state* – which replaces transition guards. The main advantage of derived states is that they can be reused by different state machines, unlike guards, which are attached to transitions. Also, derived states offer the possibility to disallow events that could lead the system into an undesired state. For example, a derived state could be used to specify that a withdraw event should not be accepted on a bank account object if it would lead to a negative balance. So far, we did not need such expressive power in our case studies with CORE-RAM.

– Event abstraction in PM is realized by using special events called *generic events.* This kind of event is used to abstract away the difference between events that have the same effect to enable reuse of existing protocol machines in different contexts [14]. In the case of CORE-RAM, a similar kind of protocol reuse is achieved when renaming an operation in an instantiation directive.

4 Integrating PM into CORE-RAM

This section describes how we integrated our customized version of PM into CORE by extending the CORE-RAM metamodel.

The structural view and message view in CORE-RAM are based on *class diagrams* and *sequence diagrams* as defined by the *Unified Modelling Language (UML)* [17]. The metamodel for these two CORE-RAM views is, however, considerably simplified compared to the UML metamodel, for instance, [22] describes how this was achieved for message views. Following the same idea, we studied the metamodel for PM and looked at the UML metamodel for state diagrams and how it is integrated with UML class and sequence diagrams. Based on the integration strategy outlined in [23], we then defined a simplified metamodel for CORE-RAM state views as presented in the following subsections.

4.1 Current Metamodel of CORE

Before discussing the state view metamodel, an overview of the current metamodel of CORE-RAM is presented so that the reader can understand what existing model elements the state view metamodel can reuse/reference.

Overview

The unit of modelling in CORE is a *concern*. *COREConcerns* contain at least two *COREModels*, a feature model and an impact model. The feature and impact model part of the CORE metamodel, however, is not shown here for space reasons. For this paper, the most important model is the *RAMAspect*, which contains all other model elements that realize a feature directly or indirectly (see Fig. 4). An aspect is a *CORENamedElement* that has beside its *StructuralView* and the *AbstractMessageViews* many *Instantiations*. An *Instantiation* describes a dependency on some other aspect (which can either be a customization, if the instantiation is part of a *COREModelReuse*, or an extension, if the aspect extends an aspect that realizes a parent feature. The instantiation contains *COREMappings*, that describe which element from the external aspect is mapped to an element in the current aspect. *ClassifierMapping* and *OperationMapping* describe mappings for classes and operations, respectively.

Structural View

The *StructuralView* represents the class diagram and its basic structure, and its metamodel is shown in Fig. 5. This view contains a list of *Classifier* and *Association*. *Classifier* is a *Type*, i.e., an abstract class that has a name and contains a list of operations. An *Operation* has a name, a return type, may have

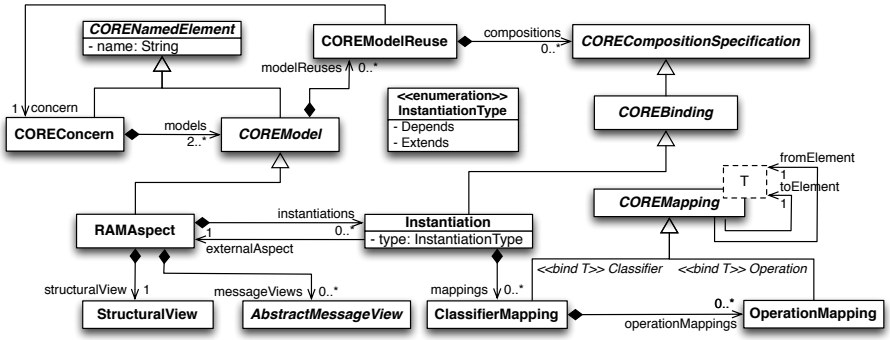


Figure 4: Overview of the CORE-RAM Metamodel

one or more parameters, and is described by four properties: *abstract*, *partial*, *static* and *visibility*. A *Parameter* has a type and a name. *Class* inherits from *Classifier,* has a list of attributes, and is described by two properties: *abstract* and *partial*. *Classifier*, *Operation* and *Parame*ter are all *MappableElements,* meaning that they can be mapped when customizing or extending another aspect.

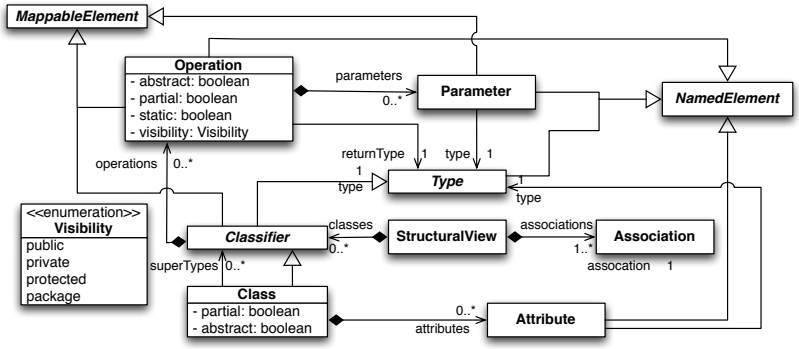


Figure 5: Structural View Metamodel

Message View

The *MessageView* represents the sequence diagram and an excerpt of its general structure is shown in Fig. 6. The sequence diagrams used in CORE-RAM, unlike UML sequence diagrams, describe only interchanges of messages in the form of operation calls. A *RAMAspect* can contain more than one *MessageView*. The latter is specified for a specific *Operation* (coming from the structural view) and contains the specification of the operation's behaviour. However this is not mandatory, because partial operations don't specify behaviour. *Interaction* describes the actual behaviour in the form of operation invocations. For

this purpose it contains, besides other entities, at least one *Message*. One of the properties of a *Message* is its return value, and this information is represented in the form of a *ValueSpecification*. The latter was inspired from the UML *ValueSpecification* which is "an abstract metaclass used to identify a value or values in a model. It may reference an instance or it may be an expression denoting an instance or instances when evaluated" [17].

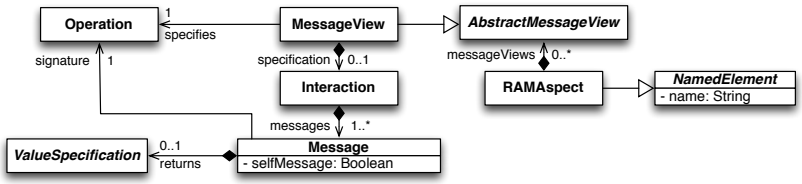


Figure 6: Simplified Metamodel of Message View

4.2 State View Metamodel based on PM

The CORE-RAM state view metamodel presented in Fig. 7 supports the simplified PM approach as described in section 3. Three entities from the existing CORE-RAM metamodel were reused in the state view metamodel: *Classifier* and *Operation* from the structural view metamodel, and *ValueSpecification* from the message view metamodel.

A *RAMAspect* can now have several *StateView*s, one *StateView* for each *Classifier* in the *StructuralView*. Because state views are mainly used for documentation and verification purpose, we do not make them mandatory. Hence, an aspect can have zero *StateViews* (which also makes the new metamodel backward compatible with the old one). Besides, partial classes sometimes do not have operations, thus a protocol cannot be defined for them. A *StateView* knows for which *Classifier* it specifies a protocol for, and it contains a set of *StateMachine*s. A StateMachine has at least one state, exactly one start state and at least one transition.

A *StateMachine* is the entity that defines the reusable protocol component of the model, which can be composed with other *StateMachines* using the CSP || operator. This entity is composed of a set of states, one of which is the start state, and a set of transitions.

A *State* represents a logical state of the object for which we are defining the protocol. A *State* has a name, a set of outgoing *Transitions,* i.e., allowed operation calls from this state, and a set of incoming *Transitions*, i.e., operation calls that led the object to be in this *State*.

A *Transition* connects two states: *startState* designates the state in which the object must be to accept an operation call, and *endState* is the new state of the object after the call has been made. *endState* and *startSate* can refer to the

same state, since some operation calls, e.g. getters, do not alter the state of the object. Since in RAM we are only interested in operation call events, a *Transition* has a signature of type *Operation,* i.e., it stands for calls to that operation only. Moreover, a *Transition* has at most one guard, which is a condition that has to evaluate to `true` for the protocol to accept a call to the transition's operation. *Guard* is of type *ValueSpecification*, a component that was borrowed from the *MessageView* metamodel, which in turn was inspired from the UML metamodel.

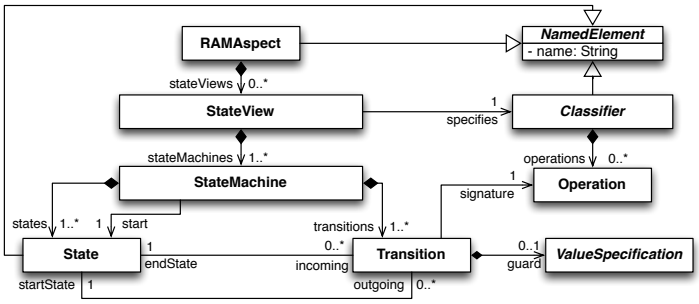


Figure 7: State View Metamodel

5 AspectOptima

AspectOptima [11,10] is an aspect-oriented framework providing customizable transaction support to applications. The current *AspectJ* implementation of AspectOptima consists of 42 aspects that modularize and implement critical transaction system features in a reusable way. The aspects can be combined in different ways to create different implementations of transaction models, concurrency control and recovery strategies.

To demonstrate the effectiveness of the new state views, we elaborated a concern-oriented design of parts of the AspectOptima framework. So far, we modelled 5 essential features of the *Transaction* concern: *ExecutionContext*, the root feature, and the optional features *Tracing*, *OutcomeAware*, *Checkpointing* and *Recovering*. The feature model is shown on the left side of Fig. 8. Each of the features has been realized in one CORE-RAM model. The dependencies between the realization models are depicted on the right side of Fig. 8. They all directly or indirectly extend the base feature realization model *ExecutionContext*. Also, some of them reuse other concerns, such as *Traceable*, *Checkpointable*, and *Association*. Indirectly, the *Copyable* and *AccessClassification* concerns are also reused.

For space reasons we can not present the complete CORE-RAM models in this paper. The interested reader can download the complete models together with our tool from [1].

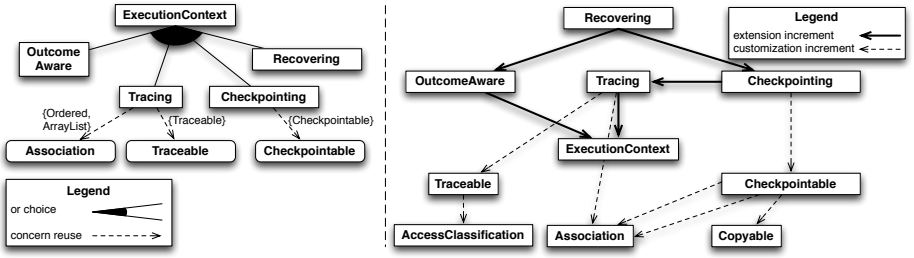


Figure 8: Feature Model (left) and CORE-RAM Realization Models with dependencies (right) of AspectOPTIMA

The base feature is called *ExecutionContext*, and its realization model is shown in Fig. 9. The main idea of *ExecutionContext* is that it allows instances of the class `IParticipant` to enter what is called a `Context` – an abstraction of an area of computation. When inside, the participant is associated with the context until it leaves the context again. The `IParticipant` class provides operations for entering and leaving, and querying the current context.

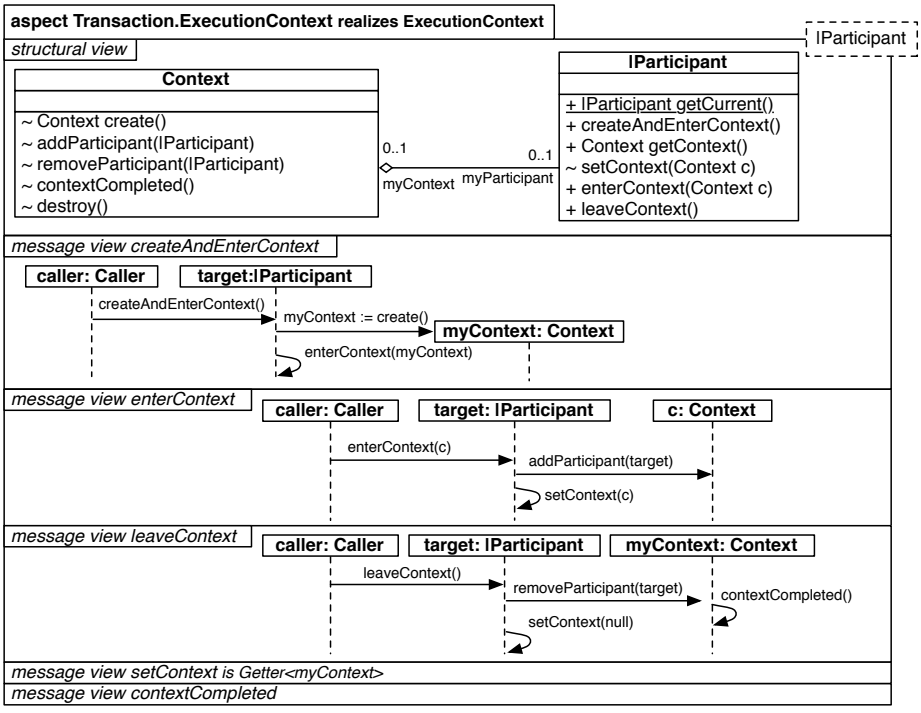


Figure 9: *ExecutionContext* CORE-RAM Model

An execution context on its own is not very useful. This is why the sub-features of *ExecutionContext* are related to it with an *or* dependency. At least one of the features must be chosen for an execution context to be of use. *Tracing* is one of those sub-features, and its structural view is presented in Fig. 10.

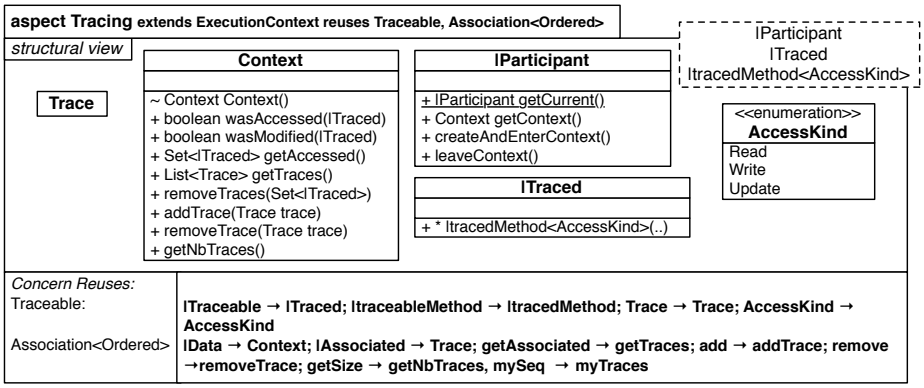


Figure 10: Structural View of the *Tracing* CORE-RAM Realization Model

Tracing depends on several other models to implement its behaviour. First, it is an extension of the *ExecutionContext* model, which already defines the classes `Context` and `IParticipant` together with the behaviour that allows a participant to enter and leave a context. *Tracing* adds additional behaviour that ensures that while inside a context, all operation invocations on instances of the class `ITraced` are recorded with the context. Such a feature can be useful for debugging or logging purposes. To achieve the desired behaviour, *Tracing* reuses the *Traceable* concern to provide the behaviour of creating a trace for a method invocation, and also *Association*<*Ordered*> (see Fig. 2) to associate an ordered list of traces with the context.

In subsection 3.1 we listed several consistency verifications that we would like to be able to conduct using our new state views. The following subsections illustrate some of them.

5.1 State Views for Public Operations

One requirement was *verifying usage consistency*, meaning that it should be possible to use the state view to ensure that a model user specifies behaviour in the sequence diagrams that call the operations of the customized aspect in the right order. Since a model user can only call public operations of the model it is customizing, it is in this case enough to define a state view that only specifies the protocol for the public operations.

For instance, *Tracing* customizes the *Association*<*Ordered*> model to enable a `Context` instance to store a list of `Traces`. In this case the *concern designer*

of *Tracing*, which is the *concern user* of *Association* needs to understand the public behaviour of the `|Data` class to be able to use it correctly. All invocations of the operations on the `|Data` class must respect the public protocol as specified in Fig. 3. For example, Fig. 11 shows the message view of the operation `removeTraces`. This operation, given a set of `Traced` objects, removes all the traces that belong to these objects from the context. The operations used by this message view, `getTraces` and `removeTrace`, are added to the `Context` class because the reuse instantiation directive maps `|Data` to `Context`, `getAssociation` to `getTraces` and `remove` to `removeTrace` (see reuse compartment in Fig. 10). Since there is no restriction for calling `getAssociated` before `remove` according to the public state view of *Association<Ordered>* (Fig. 3), the message view of `removeTraces` is using the operations of `|Data` class correctly. The CORE modelling tool should detect protocol violations and signal them to the concern users.

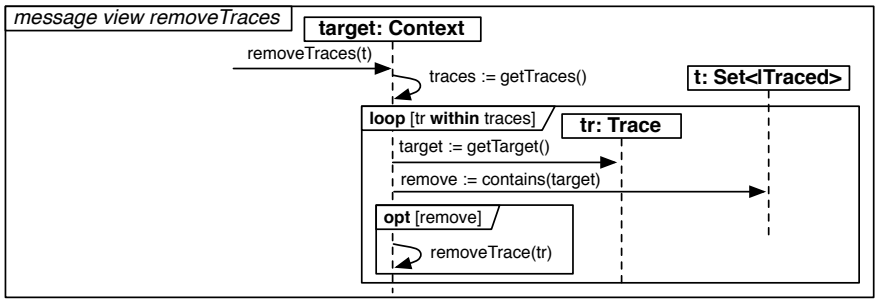


Figure 11: *removeTraces* Message View

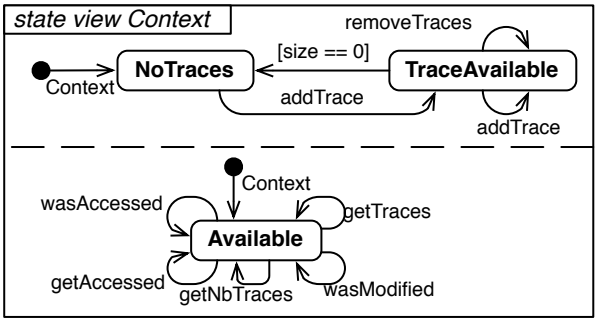


Figure 12: Public SV *Tracing.Context*

Applying the same idea to the next level, the *concern designer* should specify a public state view for *Tracing* that documents the correct use of the concern to the *concern users* and allows the modelling tool to verify its correct use. To model the public state view of the class `Context`, a *model designer* needs to consider the public operations of this class and determine for each operation the possible constraints for calling it. In our case, the major constraint for `Context` is that the operation `removeTraces` can not be called unless a trace was added previously through `addTrace`. Fig. 12 describes a public state view for `Context` that expresses this constraint.

5.2 Internal State Views

To be able to do a similar verification for model increments, a more elaborate "internal" state view needs to be defined that describes the invocation protocol detailing not only the public, but also the internal operation invocations that are acceptable during the life time of an object. This internal state view should describe how non-public operation invocations relate to public ones and to each other. This makes it possible to *verify increment consistency and composition consistency.*

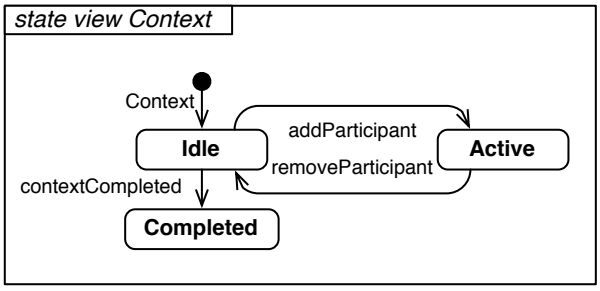


Figure 13: Internal SV *EC.Context*

For example, Fig. 13 shows the state view of the class `Context` of the *ExecutionContext* realization model (see structural view in Fig. 9). It states a participant can be added to a context and removed again multiple times in a row, if desired. However, once `contextCompleted` is invoked, no more operation calls are allowed on a context instance. The `Context` state view is an internal state view, since this class does not have any public operations. Because *Tracing* extends *ExecutionContext*, the `Context` class in *Tracing* is mapped to the `Context` class in *ExecutionContext*. The *model designer* of *Tracing* should hence specify any protocol restrictions that should be defined between the operations added to `Context` by *Tracing* and the operations that come from *ExecutionContext*.

One constraint that a `Context` object in *Tracing* must not violate is the fact that the operation `addTrace` should only be called when a participant entered

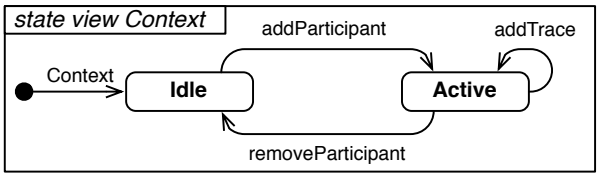


Figure 14: Internal SV of *Tracing.Context*

the context, i.e., when the context is in the state *Active* after `addParticipant` is called. Fig. 14 shows the state machine that expresses such a behavioural constraint.

Composing the public and the internal state views using CSP composition results in the state view shown in Fig. 15. The composed view can subsequently be used to verify the consistency of message views that were specified in models that reuse or extend *Tracing*.[2]

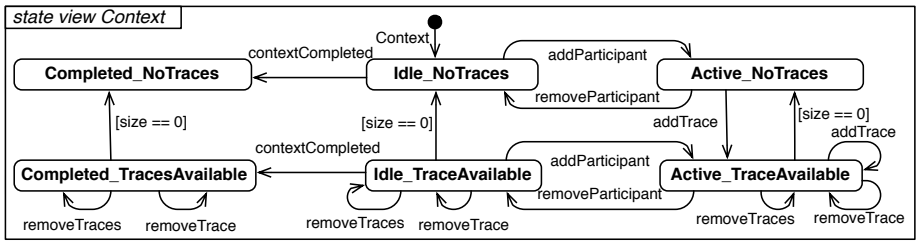


Figure 15: Composed SV of *Context* with features *ExecutionContext* and *Tracing*

6 Discussion

This subsection discusses how our new state views satisfy the requirements we detailed in subsection 3.1.

Expressiveness and Diversity: The structural view in a CORE-RAM model specifies the classes that a feature defines and *what* functionalities they offer. The message views show *how* instances of these classes interact with each other and with objects of other concerns to achieve this functionality. They also show for these scenarios *in what sequence* the operations of an object are called. For example, the *removeTraces* message view (Fig. 11) presents an overview of

[2] For simplicity and readability reasons, the state machine with the getters and query operations from *Association<Ordered>* were not added to Fig. 15. To create the actual woven model it suffices to add to each of the depicted states all self-transitions of the state *Available* from the state view in *Association<Ordered>*.

the interactions between `Context`, `Trace` and `Set<|Traced>` objects when the `removeTraces(Set<|Traced>)` operation is called, and shows the scenario where `getTraces` is called before `removeTrace(Trace)`. From a *diversity* perspective, the state views of *Context* (Fig. 12 and 14) complement the message views by giving information about *how* individual objects are to be used, i.e., the order in which an object's operations should be called, *from a state perspective*. For example, the fact that `addTrace` should be called after `addParticipant` is invoked is clearly expressed as a constraint in the *Context* state view in Fig. 14. We have so far not encountered a situation in which it was not possible to express a protocol using our new state views.

Conciseness: Measuring conciseness of a modelling notation objectively is difficult [19], and we therefore discuss conciseness of the protocol modelling notation only informally. The protocol of the objects can be concisely described using public and private state views. Woven state views that combine the protocol of several models can be generated on demand. To further increase conciseness, we made it optional to specify a protocol for operations that have no effect on the conceptual state of an object. In other words, if no transition is defined for an operation we assume that there is no restriction on its use. Finally, we added generic events to increase conciseness of state views, which can be used to group operations when operations affect the state of an object in the same way. For example, the getters and query operations in *Context* can be replaced by one event that can be called *getters_queries* as follows: *getters_queries = {wasAccessed OR getAccessed OR getTraces OR wasModified OR getNbTraces}*. As a result, only one transition needs to be shown in the state view, where otherwise five transitions with the same source and target states would have to be shown.

Modularity: CSP || composition allows the concern designer to specify the state views *for each feature within a concern independently*, and compose them together to form the complete description of the protocol of a class. Likewise, protocols of classes within a concern are modelled independently, and *the concern user can specify how to combine the protocols of the concern classes with his application classes* when customizing the model during the reuse process. For example, the *Tracing* feature was modelled separately of the *Association* concern, i.e., the protocols for `Context` and `|Data` are specified separately, and combined by the concern user by mapping `|Data` to `Context`. Modularity can even be exploited within a CORE-RAM model, since *the concern designer can specify the protocol of a class using multiple state machines* if he judges that using one state machine will be too complicated or cumbersome. For example, the getters and query operations of the class `Context` in *Tracing* were modelled in a separate state machine in the public state view to increase readability (see Fig. 12).

Composition: The CSP || composition operator offers a straightforward way to support adding of new operations, adding of constraints and coupling of protocols.

- **Adding new operations**: This kind of transformation is easily expressed by adding a new state machine that integrates the new operation into the

existing protocol. For example, *Tracing*.`Context` is using the operations of *Association<Ordered>*.`|Data` to manage the list of `Traces`, and additionally defines a new operation `removeTraces`. This operation affects the conceptual state of `|Data`, and therefore needs to be integrated in the protocol defined for `|Data`. For this reason, a state machine was defined (Fig. 12) to clarify the relationship between the behaviour of `|Data` and the new added operation. Other operations were added, i.e., `wasAccessed`, `wasModified` and `getAccessed`, but since they do not affect the state of a `|Data` object they were added to the queries state machine. Notice that `Context` of *Tracing* is extending the behaviour of `Context` coming from *ExecutionContext* by adding all the operations coming from `|Data` and all the newly defined operations. To determine the complete protocol of the new, composed `Context` object, the state views of the three classes, i.e., *ExecutionContext*.`Context`, *Tracing*.`Context` and *Association<Ordered>*.`|Data` are composed.

– **Adding constraints**: CSP || composition works by synchronizing state machines on events that are common in the alphabets of these entities. Regulation of the behaviour of an object by restricting operations is possible due to the ability of a composed state machine (M1 || M2) to refuse an event if M1 or M2 can not process this event in the current state. For instance, the feature *Recovering*, whose structural view is not shown here for space reasons, needs to change the protocol of `Context` defined in *ExecutionContext*. In EC.`Context`, a participant can enter a context, leave it, enter it again, and so on, as shown in Fig. 13. In a recovering context, once a participant is added, it has to set the outcome of the context before leaving. Furthermore, once this is done, no participants can be added anymore. Fig. 16 shows the protocol defined by *Recovering*, which is composed with the state views of *ExecutionContext*, *OutcomeAware*, *Checkpointing* and *Tracing*. Once the `Context` is created and `addParticipant`, `setOutcome` and `removeParticipant` operations are invoked, the context object will be in the state *Idle* in the machine of Fig. 13, and *ParticipantRemoved* in the machine of Fig. 16. According to the rules of PM, it is not allowed for `addParticipant` to be called again on the object. The machine in *ExecutionContext* allows processing `addParticipant`, but the machine of *Recovering* has the operation in its events but it does not allow processing from the state *ParticipantRemoved*. As a result, `addParticipant` is rejected by the composition. Only `contextCompleted` is allowed by the composed protocol.

– **Coupling Protocols**: Orchestrating the behaviour of the concerns that a model is extending and depending on is done simply by specifying the common behaviour in a separate state machine. For example, in *Tracing*, the class `Context` is mapped on the one hand to the class `Context` in *ExecutionContext* and on the other hand to the class `|Data` of *Association<Ordered>*. The behaviour of the object `|Data` is restricted by the behaviour of the object `Context` of the *ExecutionContext* concern as follows: adding traces should be done only when the `Context` is active, meaning the operation `addTrace` (coming from `add` in *Association<Ordered>*) should only be invoked after

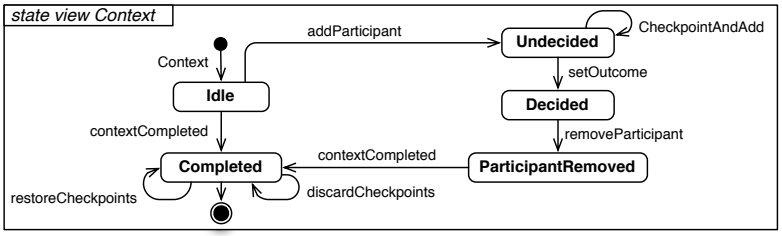


Figure 16: Adding Constraints Example

the invocation of `addParticipant` and before that of `removeParticipant`. The internal state view of `Context` shown in Fig. 14 presents the state machine needed to specify such an orchestration.

Sometimes orchestrating the behaviour of multiple objects can be tricky. Fig. 17 shows the design of a concern called *Checkpointable*. This concern can be used to add fault tolerance to a software application: it provides the functionality to create snapshots of the state of objects and restore the states in case of a failure. The class `|Checkpointable` represents the object that contains the state that needs to be recoverable and `Checkpoint` is the class responsible for handling the process. The `Checkpoint` needs to keep two lists of objects: the first contains references to the original "checkpointed" objects and the second contains the copies of the original objects at a specific moment of their life cycle. Whenever an object is "checkpointed", it is added to the first list and its copy is added to the second list. Therefore, *Checkpointable* needs the functionality offered by *Association<Ordered>* to manage these lists *twice*, and the protocols of the two lists need to be synchronized.

Fig. 17 shows the state view of `Checkpoint`. Adding a backup copy should always follow adding a checkpointable, and the same goes for removing. The behaviour of the two `|Data` objects needs to be orchestrated, and this is described by the bottom left state machine. The operation `checkpointAndAdd` is the public operation responsible for creating the copy and adding both the object and its copy to the lists. Calling this operation includes calling `addCheckpointable` and `addBackupCopy`,[3] which means that, according to the state view of `|Data`, `removeCheckpointable` and `removeBackupCopy` can be called at this point.

Verification: Section 5 discussed in detail how the CORE state views can be used to verify internal consistency, usage consistency, increment consistency and composition consistency.

[3] `addCheckpointable` and `addBackupCopy` are the same operation `add` of class `|Data` as it is shown in the instantiation compartment of the Fig. 17.

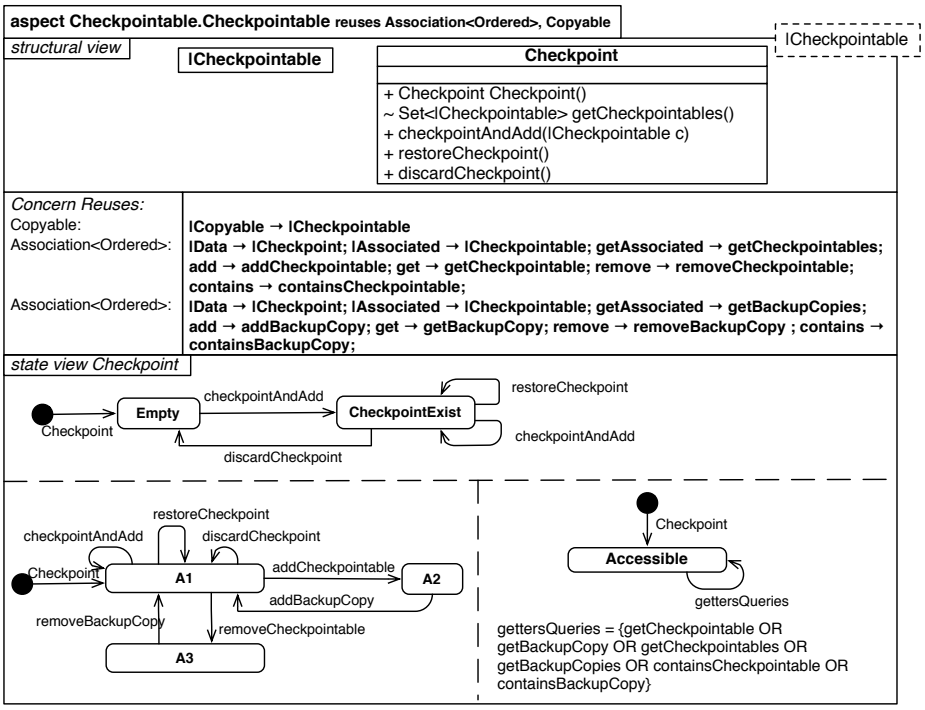


Figure 17: CORE-RAM Model of *Checkpointable*

7 Related Work

State transition modelling is an effective concept to capture software systems behaviour and protocols. In this section we describe some approaches that applied aspect-oriented modelling techniques in the context of state transition modelling.

The UML Superstructure document [17] describes several concepts for designing object-oriented systems. This modelling language provides different views to capture the static and dynamic behaviour of a software system. UML class diagrams represent the artifact used to model and describe the structural view of objects. This view is complemented by a state transition modelling artifact inspired from David Harel's statecharts. The document defines a *State Machine Package* where two kinds of state machines are described: *Behaviour State Machines* and *Protocol State Machines*.

Behaviour State Machines are used to specify discrete behaviour of a part of a designed system through finite state transitions. It can be attached to a "behavioured classifier" which is called its context. The latter defines which attributes and operations are defined for this state machine. State machines can have orthogonal regions.

Protocol State Machine is a specialization of Behaviour State Machine. It expresses the usage protocol or lifecycle of a classifier. It specifies the allowed call sequences on the classifier's operations.

In UML, a state machine can be extended, i.e., regions, vertices and transitions can be added and redefined. A simple state can be redefined to a composite state and a composite state can be extended by extending its regions or adding new ones. State machine extension was introduced following the example of class specialization. Unlike CORE-RAM, where the general and the specialized state machines can be composed together, the relationship between states that extend other states in UML is not clear. There is no explicit composition defined between state machines belonging to different classifiers, neither is event abstraction or event reinterpretation. Moreover, there is no tool, to our knowledge, that supports state machine inheritance.

The approach by Mahoney et al. [4] extends Harel's statecharts to create reusable orthogonal abstract statecharts. In order to be able to take advantage of existing CASE tools, their methodology uses UML semantics without adding any major extensions. The approach performs implicit weaving of statecharts based on orthogonality and event propagation. This makes it possible to adapt existing behaviour by adding aspects orthogonally, thus extending the model without impacting any of the other orthogonal regions. The approach also defines design guidelines that, when followed, enable traceability of crosscutting requirements from the design to code.

The main drawback of the approach that was noticed by the authors was the tight coupling between the core and the aspect statechart due to the explicit event propagation performed by the developer. To avoid such coupling, the authors introduced the concept of event reinterpretation, i.e., high-level declarations allowing an event in one statechart to be treated as a completely different event in another statechart.

A Java framework was implemented as a proof of concept that permits the translation of a statechart design into skeleton code for a class. However, the authors do not provide an integrated and concrete model view where aspects would already be woven into base classes.

Zhang et al. [27] propose the *High-Level Aspects for UML State Machines (HiLA)* approach, in which they significantly extend UML state machines with aspect-oriented modelling techniques. They use state machines to specify behaviour of base machines and aspect machines, which can be parameterized using UML template parameters similar to what is done during CORE-RAM customization. They provide several asymmetric pointcut-advice composition mechanisms that enable aspects to disallow and restrict transitions, describe mutual exclusion between two states in orthogonal regions and coordinate multiple state machines. This approach, while powerful for specifying detailed behavioural designs, is not adequate for our needs because of the complex composition semantics of the different composition operators.

8 Conclusion

In concern-driven software development, concerns are modelled separately, and model composition is used to create complex models in which these concerns are tightly coupled. In such a context, specifying the composition of the models is a non-trivial task, in particular when it comes to specifying the composition of behavioural models.

In this paper, we provided insight on the benefits that modelling of invocation protocols can have when used in combination with behavioural specifications expressed using sequence diagrams. Concretely, we showed how we applied this technique to augment the CORE approach, which expresses the structure of software design concerns within structural views based on class diagrams and the behaviour of software design concerns using sequence diagrams, with additional state views that describe invocation protocols. We detailed why Protocol Modelling, a compositional modelling approach based on state diagrams, is an ideal notation to specify such a protocol view, and show how we added support for protocol modelling to the CORE metamodel and the TouchCORE tool [1]. We explained that the new state views can be used to assist both the concern designer as well as the concern user in the model composition specification task. We outlined how the protocol view can be exploited to verify the correctness of compositions.

To demonstrate the effectiveness of our approach and to analyze its strengths and limits, we started the concern-oriented design of the AspectOptima case study. The paper partially presented some of the features of the *Transaction* concern. The complete models of the *Transaction* concern that includes *ExecutionContext*, *OutcomeAware*, *Tracing*, *Checkpointing* and *Recovering* can be downloaded together with our *TouchCORE* tool [1]. In the near future, we are planning to complete the design of the AspectOptima case study to include the remaining features needed for basic transaction support with optimistic and pessimistic concurrency control: *Nested*, *2-Phase-Locking*, *Deferring* and *Validating*. Finally, to fully support *Open Multithreaded Transactions* [8], we also need to add *Collaborative*, *EntrySynchronizing*, *ExistSynchronizing*, *SpawnSupporting*, *Closable* and *OutcomeVoting*.

References

1. TouchCORE Tool. `http://touchcore.cs.mcgill.ca`.
2. Al Abed, W., Bonnet, V., Schöttle, M., Alam, O., and Kienzle, J. TouchRAM: A multitouch-enabled tool for aspect-oriented software design. In *5th International Conference on Software Language Engineering - SLE 2012* (October 2012), no. 7745 in LNCS, Springer, pp. 275 – 285.
3. Alam, O., Kienzle, J., and Mussbacher, G. Concern-oriented software design. In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - MODELS 2013* (2013), vol. 8107 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 604–621.

4. Elrad, T., Bader, A., Mahoney, M., and Aldawud, O. Using aspects to abstract and modularize statecharts. *In the 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004* (2004).
5. Hoare, C. *Communicating Sequential Processes.* Prentice-Hall International, 1985.
6. International Telecommunication Union (ITU-T). Recommendation Z.151 (10/12): User Requirements Notation (URN) - Language Definition, approved October 2012.
7. Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, S. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, CMU, 1990.
8. Kienzle, J. *Open Multithreaded Transactions — A Transaction Model for Concurrent Object-Oriented Programming.* Kluwer Academic Publishers, 2003.
9. Kienzle, J., Al Abed, W., and Klein, J. Aspect-Oriented Multi-View Modeling. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development - AOSD 2009, March 1 - 6, 2009* (March 2009), ACM Press, pp. 87 – 98.
10. Kienzle, J., Duala-Ekoko, E., and Gélineau, S. AspectOPTIMA: A Case Study on Aspect Dependencies and Interactions. *Transactions on Aspect-Oriented Software Development 5* (March 2009), 187 – 234.
11. Kienzle, J., and Gélineau, S. AO Challenge: Implementing the ACID Properties for Transactional Objects. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development - AOSD 2006, March 20 - 24, 2006* (March 2006), ACM Press, pp. 202 – 213.
12. Klein, J., and Kienzle, J. Reusable Aspect Models. In *11th Aspect-Oriented Modeling Workshop, Nashville, TN, USA, Sept. 30th, 2007* (September 2007).
13. McNeile, A., and Roubtsova, E. Composition semantics for executable and evolvable behavioral modeling in mda. *BM-MDA'09: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture* (2009), 1–8.
14. McNeile, A., and Roubtsova, E. Aspect-oriented development using protocol modeling. *Transactions on aspect-oriented software development VII* (2010), 115–150.
15. McNeile, A., and Simons, N. Protocol modelling. A Modelling Approach that Supports Reusable Behavioural Abstractions. *SoSyM 5*, 1 (2006), 91–107.
16. Miller, G. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review 63*, 2 (1956), 81.
17. Object Management Group. *Unified Modeling Language: Superstructure (v 2.4.1)*, December 2011.
18. Rashid, A., and Ossher, H., Eds. *Transactions on Aspect-Oriented Development (TAOSD VI), Special Issue on Dependencies and Interactions with Aspects*, vol. 5490 of *LNCS*. Springer, 2009.
19. Rossi, M., and Brinkkemper, S. Complexity metrics for systems development methods and techniques. *Information Systems 21*, 2 (1996), 209–227.
20. Rumpe, B. Towards Model and Language Composition. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages* (New York, NY, USA, 2013), GlobalDSL '13, ACM, pp. 4–7.
21. Schmidt, D. C. Model-driven engineering. *IEEE Computer 39* (2006), 41–47.
22. Schöttle, M. Aspect-Oriented Behavior Modeling In Practice. M.Sc. Thesis, Department of Computer Science, Karlsruhe University of Applied Sciences, September 2012.

23. Schöttle, M., and Kienzle, J. On the Challenges of Composing Multi-View Models. *In the GEMOC'13 Workshop co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)* (October 2013).
24. Sweller, J. Cognitive load during problem solving: Effects on learning. *Cognitive science 12*, 2 (1988), 257–285.
25. Whittle, J. "the truth about model-driven development in industry - and why researchers should care". `http://www.slideshare.net/jonathw/whittle-modeling-wizards-2012/`, 2012.
26. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., and Heldal, R. Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In *Model-Driven Engineering Languages and Systems*, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, Eds., vol. 8107 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 1–17.
27. Zhang, G., and Hölzl, M. HiLA: High-Level Aspects for UML State Machines. In *Sel. Rev. Papers Wshs. at MoDELS'09* (2010), S. Ghosh, Ed., vol. 6002 of *Lect. Notes Comp. Sci.*, Springer, pp. 104–118.