Aspect-Oriented Multi-View Modeling

Jörg Kienzle, Wisam Al Abed
School of Computer Science, McGill University
Montreal, Canada
Joerg.Kienzle@mcgill.ca,
Wisam.Alabed@mail.mcgill.ca

Jacques Klein
Laboratory of Advanced Software Systems
University of Luxembourg, Luxembourg
Jacques.Klein@uni.lu

ABSTRACT

Multi-view modeling allows a developer to describe a software system from multiple points of view, e.g. structural and behavioral, using different modeling notations. Aspect-oriented modeling techniques have been proposed to address the scalability problem within individual modeling notations. This paper presents RAM, an aspect-oriented modeling approach that provides scalable multi-view modeling. RAM allows the modeler to define stand-alone reusable aspect models using 3 modeling notations. The aspect models support the modeling of structure (using UML class diagrams) and behavior (using UML state and sequence diagrams). RAM supports aspect dependency chains, which allows an aspect providing complex functionality to reuse the functionality provided by other aspects. The RAM weaver can create woven views of the composed model for debugging, simulation or code generation purpose, as well as perform consistency checks during the weaving and on the woven model to detect inconsistencies of the composition.

Categories and Subject Descriptors

D.2.2 [**Software: Design Tools and Techniques**]:

Keywords

aspect-oriented modeling, class diagrams, sequence diagrams, state diagrams, aspect dependencies, instantiation, binding

1. INTRODUCTION

Multi-view modeling allows a developer to describe a system under development from multiple points of view, e.g. structural and behavioral, using different modeling notations. As a result, the developer can use the modeling notation that is most appropriate to describe the individual relevant facets of the system under development. In practice, multi-view modeling faces two important challenges: *scalability* and *consistency*. Models of complex applications tend to grow in size, to a point where even individual views are not readily understood anymore. Keeping different views of a system consistent is also challenging for the developer, and even sophisticated tools face algorithmic challenges when attempting to analyze models of considerable size.

Aspect-oriented techniques have been successfully used to identify and separate crosscutting concerns, which allows a developer to reason about each concern individually. Aspect-orientation also draws special attention to the composition of concerns, which allows the developer to focus on the intricacies of concern interactions and conflicts. Aspect-orientation therefore has the potential to address the scalability and consistency issues in multi-view modeling.

Existing aspect-oriented modeling (AOM) approaches have mostly focussed on separation and composition of models within the same modeling notation, e.g., within class diagrams, sequence diagrams, state diagrams, SDL, live sequence charts, etc. [1]. In the context of multi-view modeling, these existing techniques can be applied within individual views to achieve scalability. Unfortunately doing so makes guaranteeing consistency between views even harder, since each AOM approach defines its own way of model weaving or model composition.

This paper presents Reusable Aspect Models (RAM), an aspect-oriented multi-view modeling approach that 1) integrates existing class diagram, sequence diagram and state diagram AOM techniques into one coherent approach; 2) packages aspect models for easy and flexible reuse; 3) supports the creation of elaborate aspect dependency chains; 4) performs elaborate consistency checks to verify correct aspect composition and reuse; 5) defines a detailed weaving algorithm that resolves aspect dependencies to generate independent aspect models and ultimately the final application model[1].

The rest of the paper is structured as follows. Section 2 presents the AOM techniques that RAM is based on. Section 3 introduces the core concepts of RAM. Section 4 shows how we applied RAM to the AspectOptima case study. Section 5 presents related work and the last section draw some conclusions.

2. AOM BACKGROUND

This section briefly introduces the class diagram, sequence diagram and state diagram weaving approaches that our reusable aspect model approach is based on.

2.1 Composition of Class Diagrams

The symmetric model composition technique proposed by

[1]An initial version of RAM addressing only class and sequence diagrams has been introduced in [18].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

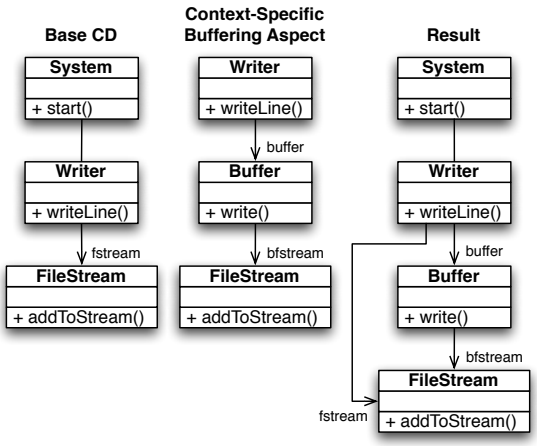

Figure 1: Merging Class Diagrams with Kompose

France et al. [10, 20] supports composition of model elements that present different views of the same concept. This composition technique has been implemented in a tool called *Kompose* [9, 2]. The model elements to be composed must be of the same syntactic type, that is, they must be instances of the same meta model class. An aspect view may also describe a concept that is not present in a target model, and vice versa. In these cases, the model elements are included in the composed model.

The process of identifying model elements to compose is called *element matching*. To support automated element matching, each element type (i.e., the element's meta-model class) is associated with a signature type that determines the uniqueness of elements in the type space: two elements with equivalent signatures represent the same concept and thus are composed. Currently, Kompose focuses mainly on the merging of class diagrams.

Fig. 1 shows an example of class diagram composition (taken from [20]) using Kompose. In the example, a modeler creates a target model in which an output producer (an instance of *Writer*) sends outputs directly to the output device to which it is linked (instance of *FileStream*). The modeler then decides to incorporate a buffering feature into the model by composing a buffering aspect model. The aspect model describes how entities that produce outputs (represented by instantiations of *Buffer*) are decoupled from output devices through the use of buffers. The result of the composition is the class diagram entitled *Composed Class diagram* where the model elements with the same name have been merged. Note that France et al. also propose a language of *directives* to modify the models before and after the composition step. This language is useful to adapt a generic aspect model to a specific target model or to improve the composed model.

2.2 Weaving of State and Sequence Diagrams

In RAM we use the state and sequence diagram weaving technique provided by GeKo [19] (Generic weaving with Kermeta), a generic model weaver that can easily be used to weave any kind of models. In GeKo, an aspect is defined as a pair of models. For instance, if we want to weave an aspect state diagram into a target state diagram, the aspect state diagram is composed of a pair of state diagrams: one state diagram for the pointcut (specification of the behavior to detect), and the other state diagram for an advice representing the expected behavior at the join point. Similar to *AspectJ*, where an aspectual behavior can be inserted '*around*', '*before*' or '*after*' a join point, an advice in the GeKo weaving approach may extend the matched behavior, replace it with a new behavior, or remove it entirely.

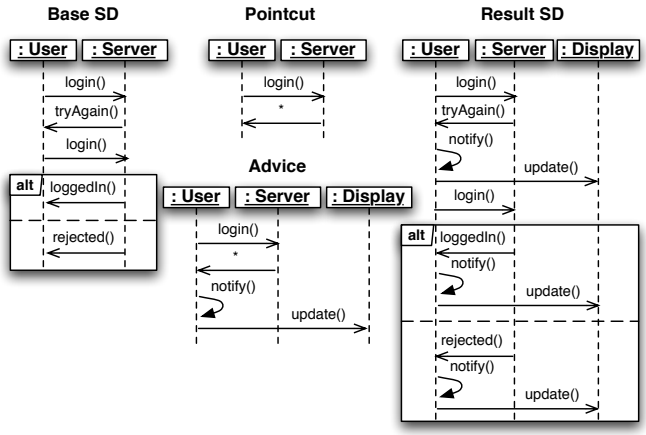

Figure 2: Sequence Diagram Weaving Example

GeKo defines a two-phased process weaving: 1) a generic detection where the pointcut is used to determine all the join points in the target model and 2) a generic composition mechanism where the advice model is composed with the target model at the join points previously detected.

An example of the use of GeKo is illustrated on sequence diagrams in Fig. 2. The base/target sequence diagram shows a possible interaction sequence between a user and a server. The user first sends a `login` message to the server. The server answers with `tryAgain` and the user performs a new attempt. The sequence diagram then shows an `alt` compartment that describes what messages are sent next, depending on whether the login succeeds or fails.

The aspect specified in Fig. 2 consists of a pointcut and an advice. The pointcut states that any message exchange between a user and a server starting with the message `login` is of interest. Note that in the specification of the pointcut, it is possible to use regular expressions on the message names: a message labeled with a `*` means that *any* message from a server to a user is of interest. The advice states that a message `notify` and a message `update` on an object of type `Display` are to be added after the return message from the server. The result of the weaving is shown on the right hand side of Fig. 2.

3. REUSABLE ASPECT MODELS

This section describes the core concepts of RAM. Before going into details, it is important to define what we call an *aspect*. In our approach, any *concern* or *functionality* that is *reusable* is modeled as an aspect. Even if an aspect is only used once in the *same* application, it is (or can be) reused again in other applications. Therefore, the structure and behavior models of a reusable aspect cut across the models of the application(s) in which the aspect is reused.

3.1 Aspect Packaging - Grouping Structure, State and Message Behavior

In RAM, the model of a concern or functionality contains up to 3 different kinds of views – a structural view, state views and message views – which are grouped together in an *aspect model*, a special UML package.

Structural View. The *structural view* is the first compartment of the aspect model. It is expressed using a UML class

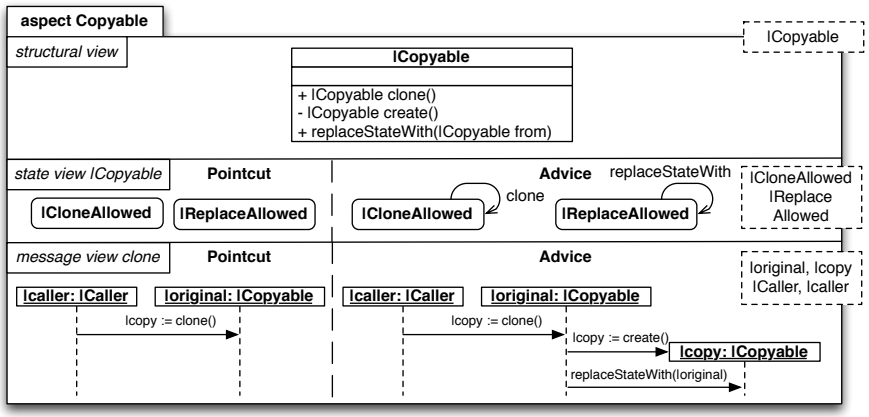

Figure 3: The *Copyable* Aspect Model

diagram, and therefore contains classes with attributes and methods, and associations that relate the classes. Public methods, i.e. methods that can be used from *outside the aspect package*, should be annotated with the "+" character. The classes in the structural view do not necessarily have to be complete, i.e., they only need to specify the attributes, methods and associations *that are relevant within the concern that is modeled*. The classes may later be composed by the weaver with other classes when the aspect is instantiated or bound to a base model (see subsection 3.2 for details on instantiation and binding) to yield a complete class. Incomplete classes, i.e. entities that are *not* directly or indirectly bound to model elements of some other aspect model, and methods whose name and signature are yet to be determined, are termed *mandatory instantiation parameters*. Mandatory instantiation parameters are identified by prepending a "|" character to their name, and have to be rendered prominent by depicting them as UML template parameters on the right hand side of the structural view compartment.

Fig. 3 defines an aspect called *Copyable* that provides the functionality of making identical copies of objects, often also called *cloning*. The structural view defines one incomplete class named `|Copyable` with one public method called `clone` that provides the cloning functionality. `|Copyable` is designated as a mandatory instantiation parameter of the structural view and hence shows up as a UML template parameter.

State View. In the aspect package, the structural compartment is followed by several *state view* compartments, one for each class (complete or incomplete) defined in the structural view. Using a UML state diagram, the state view of an entity describes the internal states of that entity that are relevant within the concern. A state is relevant if it affects the messages that the entity is capable of processing. In UML terms, the state view compartment describes the *usage protocol* of the entity. To be complete, the state diagram must contain each method defined in the structural view for the entity at least once.

When a state view describes the protocol of a standard class defined in the structural view, it takes the form of a standard state diagram. For incomplete classes, however, an *aspect state diagram* has to be defined. An aspect state diagram consists of two parts: a *pointcut* and an *advice*. The pointcut defines the states and transitions that have to exist in the target state diagram, i.e. the state diagram with which the aspect state diagram is composed. The advice part defines the (refined) state diagram that replaces the occurrence of the pointcut in the target state diagram. Just like in the structural view, states that are not directly or indirectly bound to states defined in a standard state diagram are *mandatory instantiation parameters* of the state view compartment, highlighted by prepending a "|" character to their name and emphasized as UML template parameters on the right hand side of the compartment.

The *Copyable* aspect in Fig. 3 only has one state view since the structural view only defines one entity. This entity being a mandatory instantiation parameter (and therefore an incomplete class), the state view takes the form of an aspect state diagram. The pointcut in the state view states that there are two relevant states within the *Copyable* aspect. The advice states that the relevant states are the states in which a call to `clone` or `replaceStateWith` are possible. The states `|CloneAllowed` and `|ReplaceAllowed` are designated mandatory instantiation parameters of the state view, meaning that they have to eventually be mapped to states in a standard state diagram using instantiation.

Message View. After the state view compartments, the aspect package contains *message view* compartments, at most one for each *public* method defined in the structural view. Each message view describes, using a UML sequence diagram, the sequencing of message interchanges that occur between entities when providing the functionality offered by the public method. Hence, if the functionality does not involve any message exchanges, but only computation internal to the entity, no message view compartment is shown for that method.

A message view compartment contains an aspect sequence diagram, which again has two parts: a *pointcut* and an *advice*. The pointcut defines the entities and message exchanges that have to exist in the target sequence diagram, whereas the advice specifies the sequence diagram that replaces the occurrence of the pointcut in the target diagram. Typically, for standard methods, the pointcut shows a template caller that calls the method on a template instance of the entity that defines the method. The advice then shows the details of the execution of that method. Nevertheless, in special cases, the pointcut can represent more complex behaviors, e.g., sequences of messages between several objects. In these cases, the advice shows how additional messages are added to the behavior specified in the pointcut or even how the matched messages are replaced.

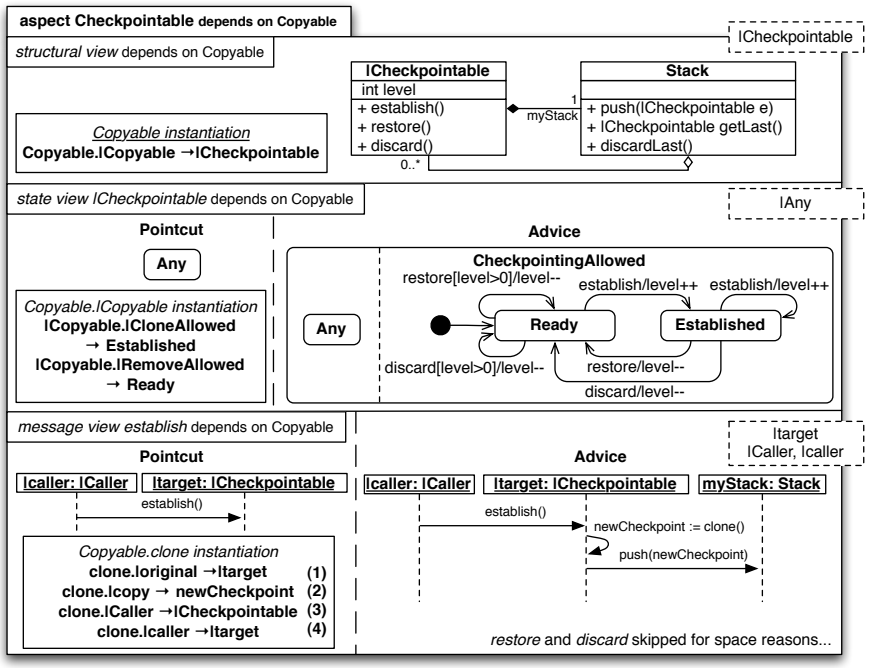

Figure 4: The *Checkpointable* Aspect Depends On *Copyable*

The *Copyable* aspect has two public methods, but only the `clone` method involves message exchanges between objects. This is the reason why Fig. 3 only contains one message view. Since the sequencing defined in the `clone` message view is likely to be reused many times, potentially with different target objects, caller instances and classes, and return references, the message view defines 4 mandatory instantiation parameters.

3.2 Aspect Dependencies and Reuse - Binding and Instantiation

One of the goals of our approach is to address the scalability problem of multi-view modeling. To keep our aspect models reasonably small, aspects providing complex functionality have to be able to reuse simpler functionality provided by other aspects. If an aspect A reuses models provided by an aspect B, then A *depends on* B. Dependencies have to be listed in the heading of an aspect package.

Fig. 4 describes the aspect *Checkpointable* that provides objects with the functionality to *establish*, *restore* and *discard checkpoints* of their state. To do that, it depends on the `clone` and `replaceStateWith` functionality provided by *Copyable*. Therefore, the *Checkpointable* aspect has to declare the dependency on *Copyable* in the aspect package heading.

Instantiation. In the RAM approach, if A depends on B, A *explicitly* states that it reuses the functionality provided by B by *instantiating* B, one or several times, if needed. Within the structural view, A must provide at least one *instantiation directive* that maps at least all mandatory instantiation parameters from B's structural view to entities in A's structural view. Classes in B that are not designated mandatory instantiation parameters can be instantiated, but do not need to be. For each incomplete class Y (or |Y) of B that is merged with an entity X (or |X) of A, A must also map all mandatory template states from B's state view of Y to states in A's state view of X using an instantiation directive in the state view X. Finally, all mandatory template parameters from B's message views that are reused within one of A's message views have to be mapped to the corresponding entities with one or multiple instantiation directives in A's message views[2]. The instantiation directive for one template parameter is of the form:

`SourceAspect.IncompleteEntityName` → `EntityName`[3]

Fig. 4 illustrates how the *Checkpointable* aspect can reuse the functionality offered by *Copyable*. First, the structural view of *Copyable* is instantiated into the structural view of *Checkpointable* by mapping the `|Copyable` incomplete class to the `|Checkpointable` incomplete class. The semantics of this is the following: all instances of `|Checkpointable`, in addition to providing the `establish`, `restore` and `discard` methods also provide the methods, attributes and associations defined by `|Copyable`. The */Checkpointable* state view specifies that an instance of the incomplete `|Checkpointable` class accepts any number of calls to `establish` followed by at most the same number of calls to `restore` or `discard`. The instantiation that maps `|CloneAllowed` to `Established` ensures that `clone` can only be called after a call to `establish`. Likewise, `replaceStateWith` can only be called after a call to `restore` or `discard`. Finally, the instantiation directive in the message view *establish* illustrates how the messaging specified in the message view *clone* can be reused. By mapping both `|caller` and `|original` to `|target`, `|Caller` to `|Checkpointable`, and the `|copy` return parameter to `newCheckpoint`, the weaver elegantly inserts the message sequencing specified in the *clone* message view into the *establish* message view at the point where the `|target` object invokes `clone` on itself. Note that since the message view `clone` is an aspect sequence diagram, if the method `clone` would have appeared several times in the `establish` mes-

[2]In case of multiple instantiations within the same state or message view, increasing numbers specify the order in which the instantiations shall be performed by the weaver.

[3]Of the model where the instantiation is located.

sage view on the `|target` object, then the message sequencing specified in the `clone` message view would have been woven several times into the `establish` message view.

Binding. In the case where an aspect A depends on an aspect B, it can happen that an incomplete class X (or |X) in the structural view of an aspect A needs to be composed with a complete class Y defined in B (or in one of the aspects that B depends on). In this case, the state view X in A might also need to refine the state view Y in order to take into account the functionality of A. Likewise, A might need to refine or override the message sequencing specified in a message view of the aspect that defines Y to take into account the functionality provided by A. In this case, A has to define a *binding directive* that maps the incomplete entities of A's structural view, state view or message view into the structural view, state view or message view of the aspect defining Y. The binding directive syntax for one template parameter is as follows:

`IncompleteEntityName`[4] → `Target.EntityName`

Whenever a view contains a binding directive, the bound model elements *cannot* appear at the same time as mandatory instantiation parameters of the view. This makes perfect sense: since there are binding directives that tell the weaver how to map the incomplete model elements to complete model elements, instantiation directives that specify the mapping are not mandatory anymore.

Instantiations and binding directives can be one-to-many or many-to-one, if needed. In this case, wild cards can be used as a shortcut to instruct the weaver to perform pattern matching on model elements at weave time to determine the set of model elements that are to be used in the directive.

Reuse. One of the main goals of RAM is to allow the modeler to design highly reusable aspect models. Reusability is at the very heart of aspect-oriented modeling: it should be possible to reuse an aspect providing a simple functionality within a base model or an aspect model that provides a more complex functionality whenever and wherever the simple functionality is needed, thus preventing scattering of model elements providing related functionality, and tangling of model elements providing different functionalities.

To make reuse possible, it is important that instantiations and bindings observe strict rules: if an aspect A provides a functionality whose design needs a simpler functionality provided by an aspect B, then A depends on B. In this case, and only then, A is allowed to instantiate views of B, or bind A's model elements to model elements defined in B. Circular dependencies are forbidden.

If these simple rules are followed, individual reuse of aspects is possible. In our example where A depends on B, it is possible to reuse B in isolation, or reuse A (which implies that indirectly B is also reused). Indirect dependencies of aspects are hidden from the user of an aspect: when a developer reuses A by instantiating it, the weaver takes care of all the indirect instantiations and bindings.

To fully exploit the benefits of reuse, aspect dependencies should be kept unresolved until the aspects are woven with the final application model. Only then the full potential of reuse is achieved: if A depends on B, then a change that is made to B is automatically propagated to A.

[4]Of the model where the binding is located.

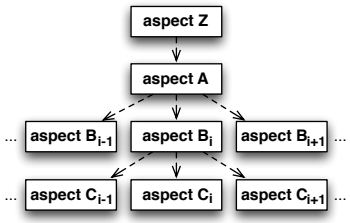

Figure 5: Aspect Dependencies

3.3 Aspect Weaving

Before an aspect A that depends on other aspects can successfully be woven with a base application model or reused in another aspect, the weaver first has to create an *independent* model of A, i.e. an aspect model that contains all the structural entities, states and message exchanges defined in the aspects it depends on. In general, an aspect A can have complex dependencies in form of a directed acyclic graph (DAG). In oder to resolve these complex dependencies, our weaving algorithm is *recursive*. It processes the DAG step by step in depth-first order. A weaving step is always executed on a pair of aspects: if A depends on B, then B is woven with A in order to yield a model of A that is independent of B. The weaving directives are obtained by combining the instantiation and binding directives.

In order to formally describe the weaving algorithm, we need the following definitions:

- The *higher aspect set* of an aspect A is composed of the aspects that (directly or indirectly) depend on A.
- The *dependency set* of an aspect A is composed of all the aspects that A directly or indirectly depends on.
- The *direct dependency set* of an aspect A is composed of all the aspects that A depends on directly.
- The *indirect dependency set* of an aspect A is composed of all the aspects that are in the dependency set of the aspects in the direct dependency set of A.

For example, in the context illustrated in Fig. 5, the *higher aspect set* of A includes Z, the *dependency* set of A includes all Bi and all Ci, the *direct dependency set* of A includes all Bi (but not any Ci), and the *indirect dependency set* of A includes all Ci (but not any Bi).

When asking to create an independent aspect model of A, Z provides as a parameter a set of binding directives D_{higher} defined within Z (or the higher aspect set of Z) that apply to aspects in the dependency set of A. To obtain an independent aspect model of A conforming to these binding directives, the weaving algorithm processes as follows:

Recursive Weaving Algorithm

Initialize the set of binding directives $D_{updated}$ to an empty set. For each aspect Bi within the direct dependency set of A perform the following 6 steps:

1. If there are binding directives defined in A that have the same left hand side as a binding directive within D_{higher}, perform a *consistency check* (as described in subsection 3.4), then discard the binding defined in A.
2. Assemble the set of binding directives D_{lower} to be passed on to Bi. D_{lower} is composed of all binding directives defined in A that bind into aspects within the dependency set of Bi *and* all binding directives in D_{higher} that apply to aspects within the dependency set of Bi. For each binding directive in D_{lower} that

on the left hand side of the binding refers to a model element X in A and for which A also defines an instantiation directive that maps a model element Y defined in Bi to X, *apply the instantiation directive in the inverse direction to the left hand side of the binding* to obtain a binding directive that refers to Y. For example, assume that D_{lower} contains the directive `A.X` $\rightarrow$ `C.Z`, and A defines the instantiation `B.Y` $\rightarrow$ `X`, then applying the instantion directive in the inverse direction results in the new directive `B.Y` $\rightarrow$ `C.Z`.

3. Create an independent aspect model of Bi (i.e. apply this algorithm recursively) using the binding directives D_{lower}. In addition to creating an independent aspect model of Bi, the recursive application of the algorithm produces an updated list of bindings $D_{lower\_updated}$, that now only contains bindings to model elements defined in Bi.
4. Assemble the set of binding directives D_{weave}. D_{weave} is composed of A's binding directives to Bi, if any, and any binding directives within $D_{lower\_updated}$ that bind from A to Bi. Remove the latter binding directives from $D_{lower\_updated}$.
5. Weave the views of the independent aspect model of Bi obtained in step 2 with the views of A as follows:
   - For each view Y in Bi, apply A's instantiation directives to the model elements of Y.
   - For each view Y in Bi, apply the binding directives D_{weave} to the model elements within Y.
   - For each view X in A, weave the views Y in B that X depends on with X using class diagram, state diagram or sequence diagram weaving techniques.
6. For each binding directives in $D_{lower\_updated}$ of step 4 that on the right hand side binds to a model element Y in Bi and for which A also defines an instantiation directive that maps Y to a model element X in A, *apply the instantiation directive to the right hand side of the binding* to obtain a new binding that binds to X. Add the obtained binding to $D_{updated}$.

Now that the independent aspect model is created, additional consistency checks are performed as described in subsection 3.4. If no consistency violations are detected, return the aspect model and the set of binding directives $D_{updated}$ to Z.

Fig. 6 shows a simple example in which the above weaving algorithm creates an independent aspect model of an aspect A that depends on B which in turn depends on C. First, an independent aspect model of B is created by weaving C with B according to the binding defined in B. Then, the independent aspect model of A is created by weaving B with A according to the instantiation and binding directives defined in A. For space reasons, only the structural view is shown.

3.4 Consistency Checks

In multi-view modeling it is important to ensure that the different views are consistent with each other. In RAM consistency checks are performed at multiple levels.

The first level of consistency checks is performed *within each individual aspect model* separately. Each class in the structural view of an aspect model has to have a corresponding state view. The state view has to define the complete protocol state diagram describing the acceptable sequencing of *all* the methods that the class declares in the structural

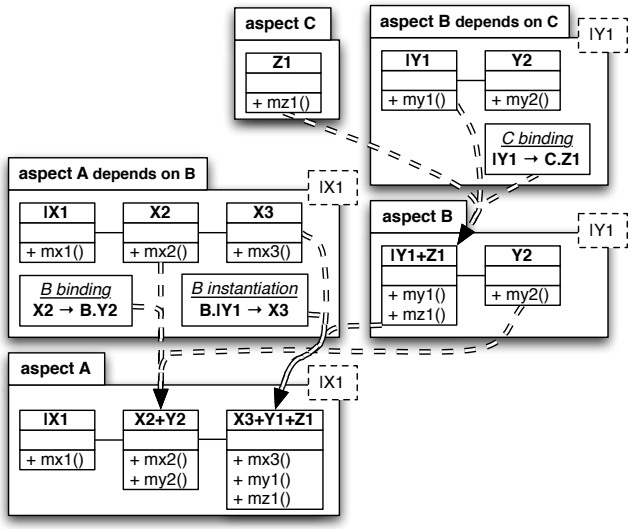

Figure 6: Structural Weaving Example

view. Any fields used in the state view diagram (e.g., `level` in Fig. 4) have to be declared as attributes of the corresponding class in the structural view. For each public method of a class in the structural view, the aspect model has to define a corresponding message view. Finally, it is possible to compare the behavior expressed in the message views and the state views within each aspect model. For each object life line in a sequence diagram, the incoming messages to that object are presented in sequence to the (partial) state diagram describing the protocol of the corresponding class. If the state diagram refuses a message, then the two views are inconsistent. This first level of check is not novel. Other multi-view modeling approaches suggest similar checks.

The second level of consistency check is performed *between aspect models* by checking the adherence to the instantiation and binding rules. The weaver makes sure that when A instantiates B, all mandatory instantiation parameters are supplied. Furthermore, the weaver ensures that the bindings in A conform to eventual bindings declared in B. This check is performed during step 1 of the weaving algorithms described in section 3.3. For structural model elements, if the instantiation directive in A maps an entity Y in B to X in A, then the binding in A has to map X *to the same entity* as the binding in B maps Y to. For states, the binding in A must be a sub-binding of the binding in B. A binding $X \rightarrow S$ (or $X \rightarrow \cup S_i$) is a sub-binding of a binding $X \rightarrow T$ (or $X \rightarrow \cup T$) if either $S = T$ or S is a substate of T (or $\forall S_i \exists T_i (S_i = T_i$ or S_i is a substate of $T_i)$). This level of consistency check is novel. For message views, the binding in A must map to the same or a subset of the objects used in the binding defined in B. These conformance rules make sure that A reuses B correctly, i.e. that the expected operation conditions of the reused aspect B are preserved within A.

The last level of consistency checks is performed *within the independent aspect model* and within the *final base model*, where the sequencing of message exchanges expressed in the final sequence diagram is checked against the final protocol state machines in the same way in which it was done for each individual aspect. For each object life line in the sequence diagram, the incoming messages to that object are presented in sequence to the state diagram describing the protocol of the corresponding class. If the state diagram re-

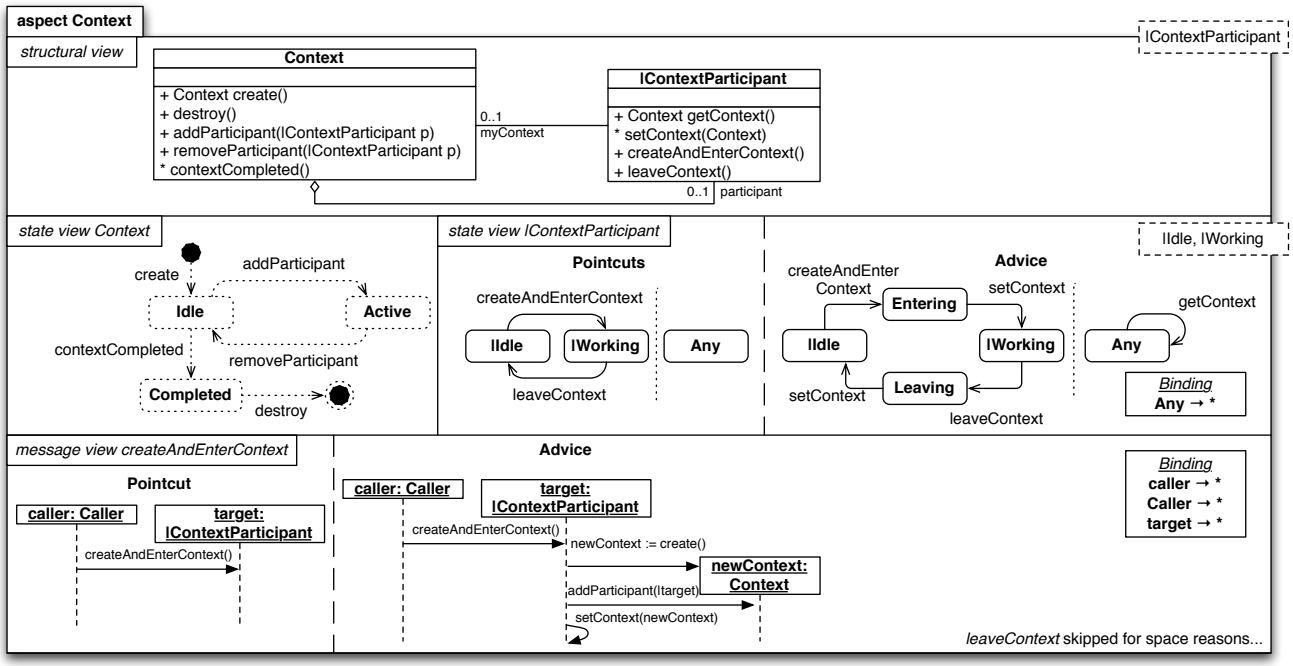

Figure 7: The Context Aspect Models

fuses a message, consistency is violated. This signals to the developer that the instantiations and bindings (or the ordering of the instantiations and bindings) of the state and message views contradict each other and have to be revisited. To our knowledge, no AOM approach has so far proposed such a powerful consistency check.

4. CASE STUDY: ASPECTOPTIMA

AspectOptima [15, 14] is an aspect-oriented framework providing customizable transaction support to applications. The current AspectJ implementation of AspectOptima [6] consists of 29 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 RAM, we applied our approach to model the design of parts of the AspectOptima framework. The feature diagram model that specifies the possible ways to build a transaction is shown in Fig. 8. Each of the features has been modeled as one RAM aspect model. For space reasons we can not present all the aspect models in this paper. We therefore concentrate on a subset of the aspects, in particular on *Recovering*. *Recovering* implements the atomicity property of transactions, which states that either all the changes performed in a transaction are reflected in the application state, or none is, i.e., the application state is identical to the state that was valid before the transaction started. Recovery can be implemented in two ways: based on a technique called in-place update and checkpointing, or using deferred update. We will concentrate on the checkpointing alternative. *Recovering* therefore depends (directly or indirectly) on *OutcomeAware*, *Checkpointing*, *Tracing*, *Checkpointable*, *Copyable*, *Traceable*, *AccessClassified* and *Context*. In the following subsections we present parts of these aspects, and then demonstrate how a base model can be woven with the *Recovering* aspect.

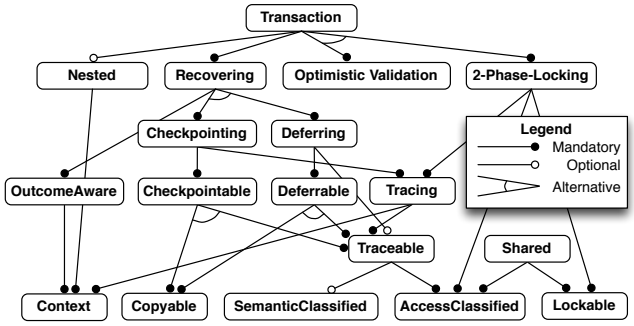

Figure 8: Feature Diagram of AspectOptima

4.1 AspectOPTIMA Aspects

Context. The *Context* aspect is one of the base aspects of AspectOptima. A context are best described as an *area of computation*. Contexts structure the execution of an application. They give identity to the set of operations executed by threads on objects over a given period of time in the pursuit of a goal. A transaction is an example of a context that exhibits additional properties, i.e., the Atomicity, Consistency, Isolation and Durability (ACID) properties [11].

The structural view of our RAM context model presented in Fig. 7 defines a `Context` class, together with a permanent association to a (to be determined) `IContextParticipant` class. The *createAndEnter* message view describes the functionality that allows a context participant to instantiate a context and add itself as a participant. The */ContextParticipant* state view shows that any class that wants to participate in a context must eventually follow each call to `createAndEnter` with a call to `leaveContext`. The *Context* state view describes the usage protocol of the `Context` class: it specifies, for instance, that in order to complete there must be no active participants.

OutcomeAware. *OutcomeAware* shown in Fig. 9 extends the functionality offered by *Context*. According to the struc-

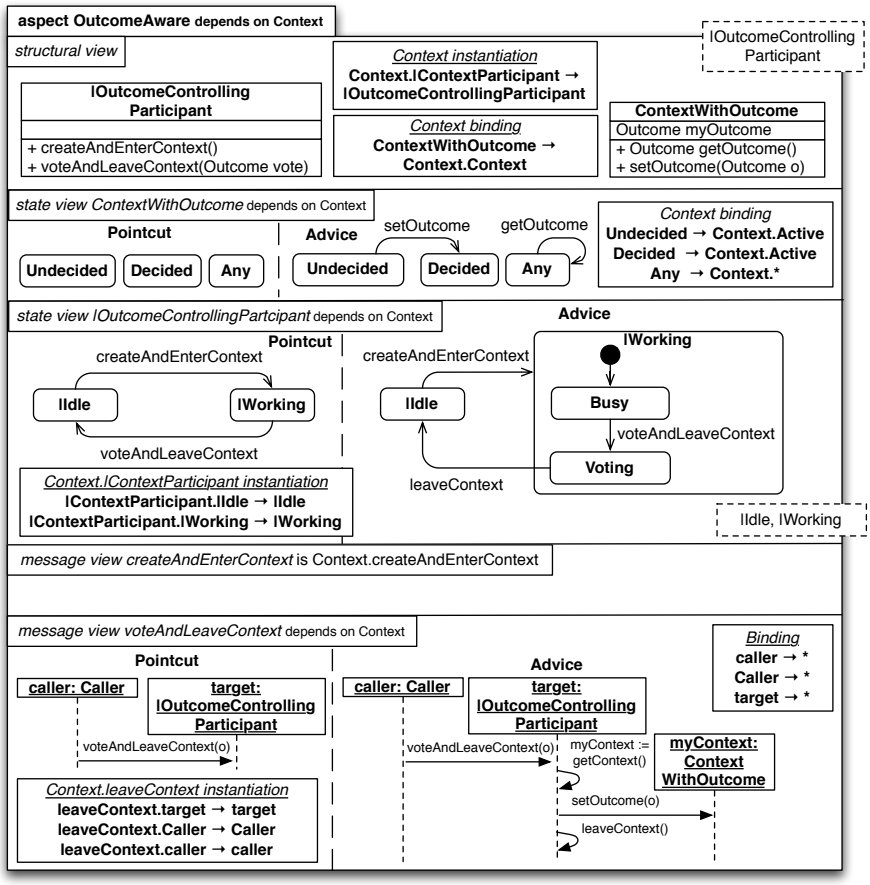

Figure 9: The OutcomeAware Aspect Model

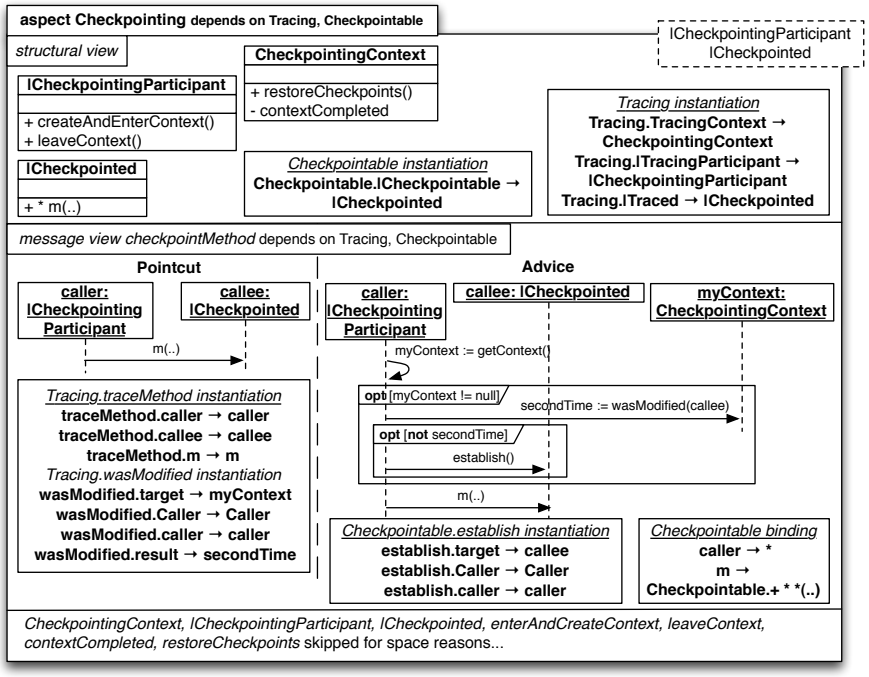

Figure 10: The *Checkpointing* Aspect Model

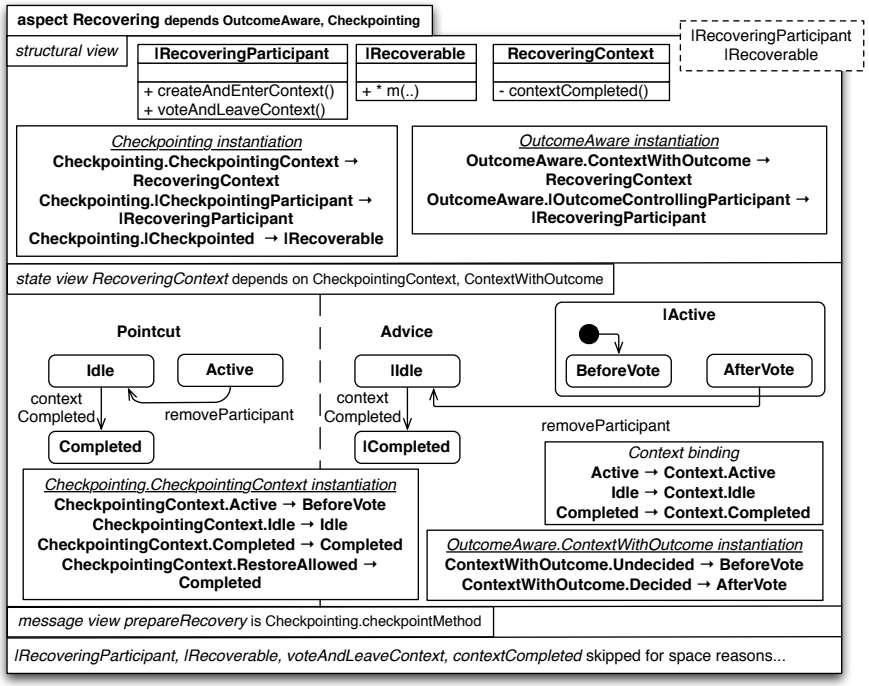

Figure 11: The Recovering Aspect Model

tural view, a `ContextWithOutcome` has an outcome (positive or negative), and offers operations to query or set the outcome. The state view shows that a context is first in an `Undecided` state, and then moves to a `Decided` state when the outcome of the context is set. An `|OutcomeControlling-Participant` must decide on the outcome of the context when leaving by invoking the `voteAndLeaveContext` method. After setting the outcome, the *voteAndLeaveContext* message view reuses the `leaveContext` functionality provided by *Context*.

AccessClassified, *Traceable* and *Tracing* are not shown for space reasons. *Checkpointable* and *Copyable* have already been presented in section 3.

Checkpointing. *Checkpointing* shown in Fig. 10 makes sure that whenever a modifying operation is invoked on a *Checkpointable* object for the first time within the current context, a checkpoint is established. To do this, *Checkpointing* depends on *Tracing* to keep track of the operations that are invoked, and on *Checkpointable* to establish the actual snapshot of the state of the *Checkpointable* object. How this happens is modeled in the message view *checkpointMethod*. Whenever an operation `m` is invoked by a checkpointing participant on a checkpointed object, the current checkpointing context, which is also a tracing context, is asked if the target object has already been modified from within the context. If not, the `establish` method of the checkpointable object is called before proceeding with the call to `m`. Note that the binding directive `m → Checkpointable.+ * *(..)` makes sure that all calls to public methods of checkpointable objects are taken care of.

Recovering. *Recovering* shown in Fig. 11 finally depends on *Checkpointing* and *OutcomeAware*. The structural view instantiates the `RecoveringContext` to also be a `CheckpointingContext` as well as a `ContextWithOutcome`. The *RecoveringContext* state view integrates the two dependent state views into its own state view. The message view *prepar-*

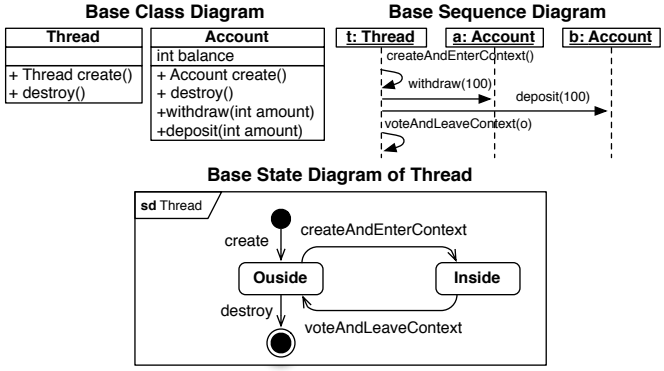

Figure 12: A Simple Base Model

eRecovery makes sure that all method calls to recoverable objects are checkpointed.

4.2 Applying Recovering to a Base Model

Fig. 12 depicts a simple base model of a banking application where a thread object `t` transfers some money from account `a` to account `b`. By instantiating the *Recovering* aspect into this base model, the transfer can be made atomic. The instantiation directives are as follows:

Recovering.|RecoveringParticipant → Thread
Recovering.|Recoverable → Account
|RecoveringParticipant.|Idle → Outside
|RecoveringParticipant.|Working → Inside

If the modeler does not want to add the instantiation directives directly to the base model, it is also possible to completely separate the base model from the *Recovering* aspect *and vice versa*. This can be done by modeling a *Recovery-Introduction* aspect that instantiates *Recovering* and binds it into the base model.

The resulting application structural view, 2 of the interesting application state views and the beginning of the application sequence diagram that are obtained after the weaving

of these instantiation instructions are shown in Fig. 13. In all three views it is (graphically) obvious that the different concerns (context, tracing, checkpointing, recovery) are tangled. Looking, for example, at the structure diagram it is not obvious to say which entities and methods are involved in, for example, establishing checkpoints. The annotations on the left hand side of the sequence diagram also show that, for example, functionality provided by *Tracing* is scattered throughout the diagram. Finally, the generated models are considerably bigger than the individual aspect models. The complete final sequence diagram, for example, is 4 times longer, but had to be truncated to fit on to the page.

It is also interesting to note here that during our modeling effort, we were able to detect subtle errors in our models thanks to the consistency checks between the final sequence diagram and the state views. In our sequence diagrams, *Recovering* specified that the restoration of checkpoints was to be performed after the context completes. We had forgotten that *Checkpointing* specified that all checkpoints should be discarded when the context completes. We discovered that mistake because the state view of */Checkpointable* refused to accept a `restore` message after a `discard` had already been processed (see Fig. 4).

5. RELATED WORK

Our RAM approach is based on the class diagram composition approach [10] (called *Kompose*) and the generic aspect model weaving approach (called *GeKo*) [19] to weave both state and sequence diagrams as presented in section 2. However, we believe that the RAM approach could be easily adapted to run on other model weavers that support model composition and model weaving. The major related aspect-oriented modeling tools and approaches are briefly described in this subsection.

Clarke and Baniassad [7] define the *Theme/UML* approach. It introduces a theme module that can be used to represent a concern at the modeling level. Themes are declaratively complete units of modularization, in which any of the diagrams available in the UML can be used to model one view of the structure and behavior the concern requires to execute. In Theme/UML, class diagrams and sequence diagrams are typically used to describe the structure and behavior of the concern being modeled. Just like in our approach, the binding to a base model is done by template parameter instantiation. In contrast to our approach, Theme/UML does not support model weaving, and hence does not allow the weaver to perform consistency checks between different views.

Similarly to our approach, Whittle and Araujo [22] represent behavioral aspects with scenarios. Aspectual scenarios are modeled as interaction pattern specifications and are composed with specification scenarios. The weaving process is performed in two steps. First state machines are generated from the aspects and from the specification. The weaving is then performed by composing the obtained state machines. Their approach differs from ours since it focusses on sequence diagrams only. We propose in [17, 16] a semantics-based weaver for sequence diagrams. This weaver allows the detection of join points which cannot be detected if only the syntax of the model is used.

The Motorola WEAVR approach [8] and tool have been developed in an industrial setting. Behavior is modeled with SDL, a formalism related to state diagrams and activity diagrams. In order to be able to reuse aspects, *mappings* have to be defined (equivalent to our instantiations) that link a reusable aspect to the application-specific context in which it is to be deployed. The WEAVR approach differs from our approach since it exclusively focusses on SDL.

Whittle and Jayaraman [23] have recently proposed an interesting aspect-oriented modeling tool called MATA. This tool uses graph transformations to specify and weave aspects at the modeling level. MATA can be applied to any modeling language with a well-defined meta model, and some case studies have been proposed using mainly class, sequence, and state diagrams. With MATA, both pointcut and advice are specified on the same model, whereas with our approach both pointcut and advice are separately specified. At first glance, the composition and weaving mechanisms offered by MATA seems powerful enough to implement the RAM approach. Similarly, Groher and Voelter [12] have proposed a weaver based on the *Eclipse Modeling Framework* (EMF) *Ecore* meta meta model [4]. This means that the approach can weave models that are instances of *Ecore* (meta models). XWeave weaves crosscutting concerns encapsulated as aspect models into (non-AO) base models. This is a form of asymmetric model weaving (similary to GeKo), where there is a designated base model into which a number of aspect models are woven (as opposed to symmetric weaving, where there is no designated base model). Weaving is done based on matching names of elements in the aspect and the base model. Additionally, pointcuts based on the *openArchitectureWare* (oAW) expression language [3] can be defined to select sets of model elements as join points. XWeave cannot remove, change, or override existing base model elements using aspects. XWeave thus currently supports essentially only additive weaving, where additional elements are added to the base model.

In [21], Stein et al. introduce a way to express various conceptual models of pointcuts (called JPDDs for Join Point Designation Diagrams) in aspect-oriented design. Structural and behavioral modeling is achieved by employing for instance class diagrams, state charts, and sequence diagrams. In contrast to our approach, their objective is not to perform the weaving at the modeling level, but rather to generate code for aspect-oriented programs from an aspect-oriented design as shown in [13]. Again, it would be possible to change the notation used in our approach and express pointcuts using JPDDs, if modelers find them more intuitive.

6. CONCLUSION AND FUTURE WORK

This paper presented Reusable Aspect Models (RAM), an aspect-oriented multi-view modeling approach. The main contributions of RAM are:

- RAM is the first AOM approach that integrates class diagram, sequence diagram and state diagram AOM techniques. As a result, RAM aspect models can describe the structure *and* the behavior of a concern under study.
- Reuse of aspect models in RAM is simple and flexible. Flexibility is achieved by allowing any model element to optionally be composed or extended through bindings. Correct reuse is enforced by the weaver, which makes sure that compatible model elements are provided for all mandatory instantiation parameters when an aspect is instantiated, and eventual bindings defined in a higher aspect are compatible with the bindings in the reused aspect.

- RAM supports the creation of elaborate aspect dependency chains. This makes it possible to model aspects that provide complex functionality by decomposing them into aspects that provide simpler functionality. Vice versa, aspects providing simpler functionality can be reused in several aspects of complex functionality. As a result, scattering and tangling of models can be prevented at all complexity levels.
- The RAM weaver performs extensive consistency checks during the weaving and on the final woven model to ensure that the composition directives of the state and message views are consistent.
- RAM defines a detailed weaving algorithm that resolves aspect dependencies recursively to generate independent aspect models and ultimately generate the final application model. Dependencies are resolved at weave-time only in order to maximize the benefits of reuse.

We have shown that the RAM approach can handle the modeling of complex aspect frameworks by applying it to model AspectOptima, an aspect-oriented framework for the generation of transaction middleware. Even though we only modeled the part of AspectOptima that deals with providing the atomicity property of transactions, the obtained RAM models of the individual aspects were a magnitude smaller than the woven final models.

The case study leads us to believe that RAM provides scalable and consistent multi-view modeling. Support for aspect-orientation and for dependencies among aspects allows the developer to modularize concerns at multiple levels. Enforcement of correct aspect reuse, and model checking performed on the views of the final woven model ensure global view consistency. In order to make a conclusive statement, however, further experiments are necessary.

We already have implemented tool support for RAM in Eclipse based on Kompose [2] and GeKo [9]. It can be downloaded from [5]. So far, however, only class and sequence diagrams are supported. In this restricted context we have also developed a technique that allows the RAM weaver to automatically handle aspect conflicts for *users* of aspect models, if the *developer* of the aspect model has previously identified and resolved the conflict by providing an *aspect conflict resolution model*. We are currently working on extending the approach to also include state diagrams.

Finally, we believe that in order to be complete, RAM aspect models should be extended by adding yet another kind of view that describes the detailed execution paths for individual methods. Detailed method algorithms could be expressed, for instance, with UML activity diagrams or SDL. With this additional view, RAM would be capable of generating final application models that are fully executable.

7. REFERENCES

[1] Aspect-Oriented Modeling Workshop Series. http://www.aspect-modeling.org/.

[2] Kompose. http://www.kermeta.org/mdk/kompose/.

[3] openArchitectureWare. http://www.eclipse.org/gmt/oaw/.

[4] The Eclipse Modeling Framework. http://www.eclipse.org/emf/.

[5] AspectOPTIMA Webpage: http://aspectoptima.cs.mcgill.ca/, 2007.

[6] Bölükbaşı, G. Aspectual Decomposition of Transactions. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, 2007.

[7] Clarke, S., and Baniassad, E. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison Wesley, 2005.

[8] Cottenier, T., v.d. Berg, A., and Elrad, T. Stateful aspects: the case for aspect-oriented modeling. In *10th Aspect-Oriented Modeling Workshop* (2007), ACM Press.

[9] Fleurey, F., Baudry, B., France, R., and Ghosh, S. A generic approach for automatic model composition. In *11th Aspect-Oriented Modeling Workshop* (2007).

[10] France, R., Ray, I., Georg, G., and Ghosh, S. Aspect-oriented approach to early design modelling. *IEE Proceedings Software* (August 2004), 173–185.

[11] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[12] Groher, I., and Voelter, M. Xweave: Models and aspects in concert. In *10th Aspect-Oriented Modeling Workshop* (2007).

[13] Hanenberg, S., Stein, D., and Unland, R. From aspect-oriented design to aspect-oriented programs: tool-supported translation of JPDDs into code. In *AOSD* (2007), pp. 49–62.

[14] Kienzle, J., Duala-Ekoko, E., and Gélineau, S. AspectOPTIMA: A Case Study on Aspect Dependencies and Interactions. *TAOSD* (to be published).

[15] Kienzle, J., and Gélineau, S. AO Challenge: Implementing the ACID Properties for Transactional Objects. In *AOSD* (2006), ACM Press, pp. 202 – 213.

[16] Klein, J., Fleurey, F., and Jézéquel, J. M. Weaving multiple aspects in sequence diagrams. *TAOSD LNCS 4620* (2007), 167–199.

[17] Klein, J., Hélouet, L., and Jézéquel, J.-M. Semantic-based weaving of scenarios. In *AOSD* (2006), ACM Press, pp. 27–38.

[18] Klein, J., and Kienzle, J. Reusable Aspect Models. In *11th Aspect-Oriented Modeling Workshop* (September 2007).

[19] Morin, B., Klein, J., Barais, O., and Jezequel, J.-M. A generic weaver for supporting product lines. In *Early Aspects Workshop at ICSE* (2008).

[20] Reddy, R., Ghosh, S., France, R. B., Straw, G., Bieman, J. M., Song, E., and Georg, G. Directives for composing aspect-oriented design class models. *TAOSD LNCS 3880* (2006), 75–105.

[21] Stein, D., Hanenberg, S., and Unland, R. Expressing different conceptual models of join point selections in aspect-oriented design. In *AOSD* (2006), ACM Press, pp. 15–26.

[22] Whittle, J., and Araújo, J. Scenario modelling with aspects. *IEE Proceedings - Software 151*, 4 (2004), 157–172.

[23] Whittle, J., and Jayaraman, P. Mata: A tool for aspect-oriented modeling based on graph transformation. In *11th Aspect-Oriented Modeling Workshop* (2007).

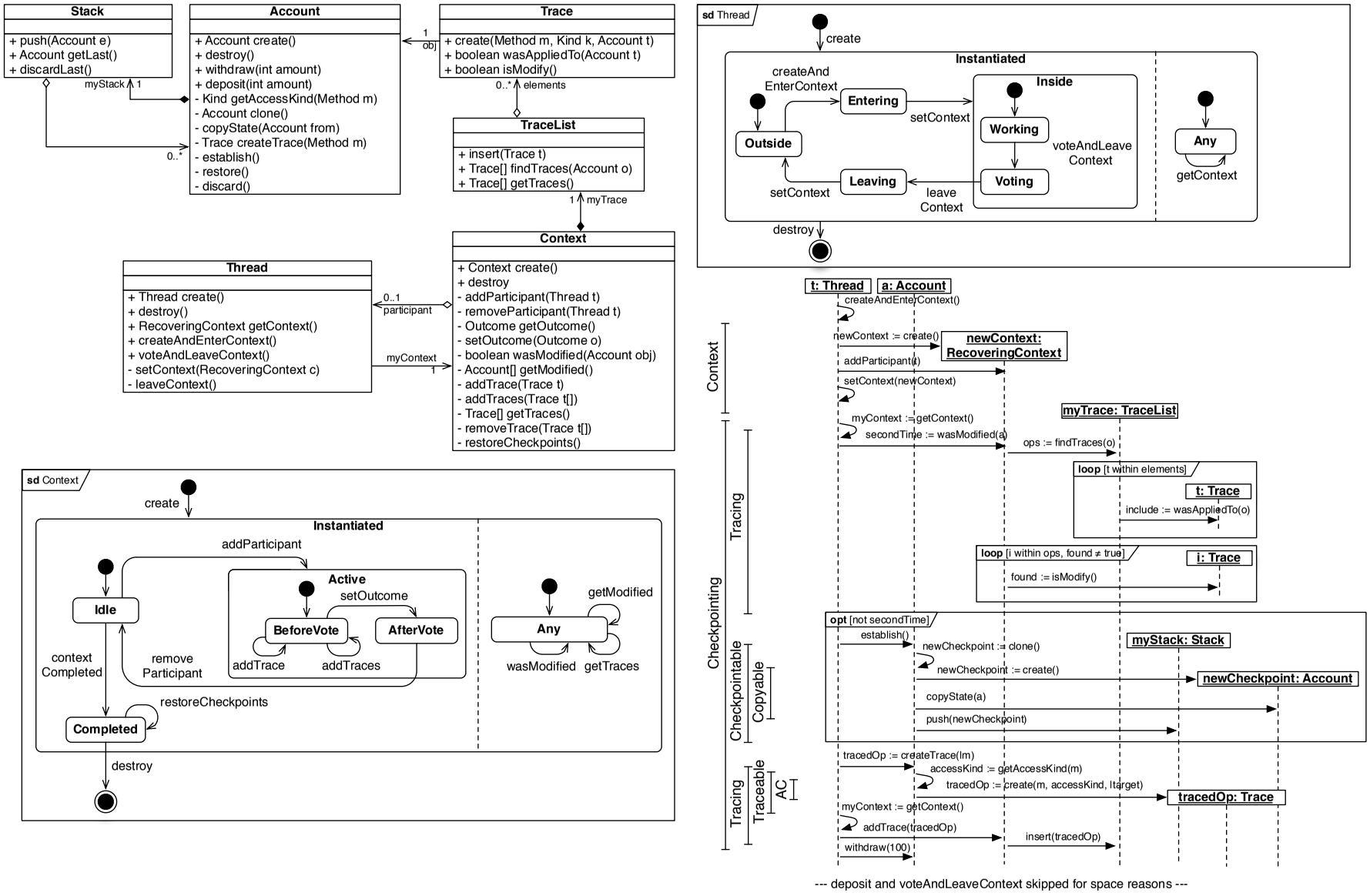

Figure 13: Woven Application Model