Aspect-Oriented Modeling with Integrated Object Models

Silvio Meier, Tobias Reinhard, Christian Seybold, Martin Glinz
Department of Informatics
University of Zurich

{smeier | reinhard | seybold | glinz}@ifi.unizh.ch

Abstract: With the advent of aspect-oriented programming, the need for adequate techniques for handling aspect-oriented artifacts in the early phases of the software engineering process has emerged. In this paper, we present an aspect-oriented language extension for an integrated modeling language based on object models. We present the way aspect constructs can be handled in requirements and architectural models, and identify the impact on existing modeling languages and models.

1 Introduction

Aspect-oriented programming (AOP) [KLM+97] is an emerging research field which deals with the handling of crosscutting concerns. Crosscutting concerns manifest as scattered and tangled code in the final system and result from the lack of additional decomposition dimensions which is also called the tyranny of the dominant decomposition [TOHS99].

The better the modularization of code artifacts is, the simpler they become and therefore the easier they are to understand. In [Lad03, KHH+01], the advantages of using AOP techniques are discussed: the resulting code artifacts have a clearer responsibility due to their better modularization and are therefore easier to reuse and to maintain. A late integration (also called weaving) of all the code artifacts makes it possible to have a clearer focus on them. This results in a clearer and better design and therefore leads to more stable systems. Overall, this lowers the system development costs.

Aspect-oriented software development (AOSD) propagates the application of aspect-oriented techniques to artifacts of the software engineering process in other stages than the implementation phase, namely to the requirements and architectural phases. This reduces the impedance mismatch between the traditional paradigms used in the early phases of the software process and the aspect-oriented paradigm used during the implementation. Furthermore, the advantages outlined above can also be gained from these early phases. Additionally, an early separation of crosscutting concerns facilitates the traceability of crosscutting software artifacts, which is especially important for the requirements phase. Hence, we aim at an early identification and separation of crosscutting concerns and a weaving of the crosscutting concerns and the core concerns as late as possible.

As stated in [MG05], introducing aspect-orientation into modeling languages has not only advantages, but also entails problems, in particular when requirements and sofware architectures are modeled:

i. The introduction of aspect-oriented constructs reduces the complexity when focusing (locally) on one or a few artifacts, but, at the same time, increases complexity when looking (globally) at the model as a whole. This is because of the additional separation dimension which results in the need for a linking mechanism (the so-called join point model) between the concerns which in turn introduces complexity.

ii. Today's aspect-oriented modeling approaches often use loosely coupled modeling languages like UML [OMG03] and extend them with aspect-oriented constructs. Loosely coupled languages consist of a set of sub-languages which are not well integrated with each other, neither in their visual representation nor in terms of their language design. Therefore, they need more redundant information and consequently introduce more consistency problems [GBJ02]. Also they demand a high degree of intellectual effort to integrate a system into one coherent model in mind. Thus, using loosely coupled modeling languages as a basis for an extended, aspect-oriented language amplifies the effect of complexity when trying to understand a model as a whole.

iii. Nowadays, aspect-oriented modeling approaches concentrate in the majority of cases on just one or two views, most often the behavior or the static structure of a system. For a comprehensive view of a system, it is necessary to have all the possible views (see Section 2). Having a language that does not coherently support aspect-orientation confuses both the modeler and the reader of such models rather than helping them to achieve better models.

For the early phases of software engineering, e.g. the requirements stage, it is necessary to satisfy quality criteria such as unambiguity, completeness, correctness, etc. (see [IEE98]). Expressed in more general terms, these standards have to be met every time a software artifact is used as a means of communication, e.g. between customer and engineer or between engineers. The qualities mentioned are strongly influenced by the issues described in items i) to iii) and therefore it must be the aim for a good aspect-oriented modeling language to solve the problems listed above.

In this paper, we propose a novel modeling approach for aspect-oriented models using an object-based, tightly coupled and integrated modeling language. The paper is organized as follows: Section 2 presents the modeling language ADORA, which is used as a basis for the approach. Furthermore, it discusses the problems occurring when modeling crosscutting concerns with conventional modeling techniques. Section 3 deals with the approach advocated in detail. Section 4 covers the related work and finally Section 5 gives some conclusions and a short overview of our future work.

2 Prerequisites

2.1 ADORA– An Object-Oriented Requirements Language

ADORA is a language for modeling requirements and architectural design specifications [GBJ02]. ADORA primarily models functional requirements. Non-functional requirements can be included as textual annotations in ADORA models. Alternatively, one can operationalize goals and non-functional requirements so that they can be modeled as functional requirements. Fig. 1 shows a library system described as a typical ADORA model, which is used as an example throughout this paper.

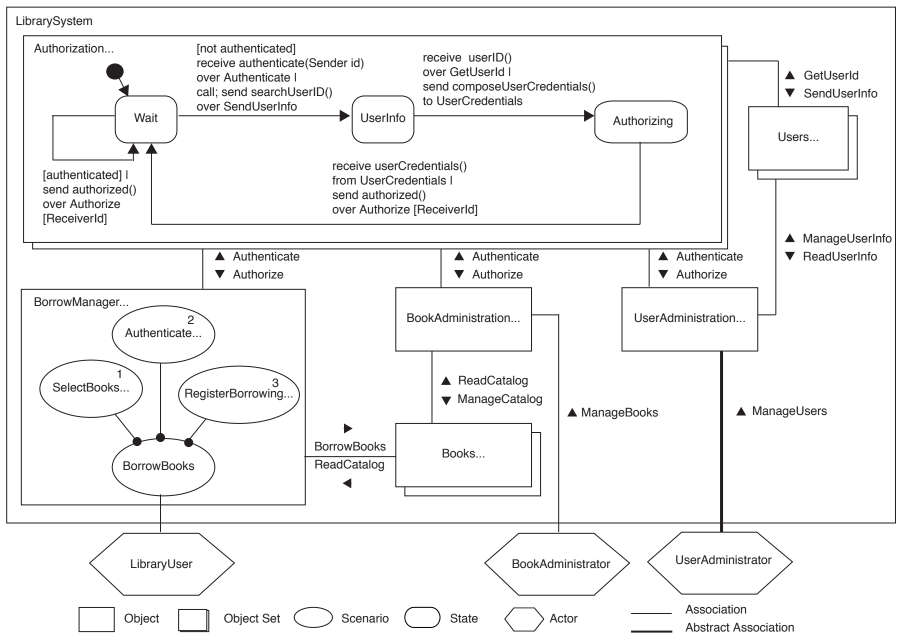

Figure 1: A part of a library system modeled in ADORA

A major difference between ADORA and other object-oriented modeling languages is that ADORA uses abstract objects (i.e. prototypical objects that have a name, but are not instantiated with attribute values) instead of classes as the basic modeling elements [GBJ02]. Using abstract objects and object sets allows hierarchical decomposition of models in a straightforward way with a simple and a clear semantics. Decomposition, in turn, yields abstraction and the possibility of visualizing components in their context, thus making models easier to understand and evolve.

In Fig. 1, all rectangles represent abstract objects. Hierarchical structure is modeled by nesting objects, which in turn implicitly describes part-of-relationships between objects.

Shadowed rectangles are object sets, i.e. they model collections of objects. For example, the library system in our example contains an abstract object *BookAdministration* and an object set *Books*.[1] For visualization, ADORA employs *views*. The so-called *base view* consists solely of the hierarchical structure built of the objects and object sets. Beside this base view, additional views describe other facets of the system. In Fig. 1, all views are visible in combination. The *structural view* comprises the associations between objects and/or object sets, and the associations between actors and scenarios. Associations model information flow. Hence, associations may model directed structural relationships as well as the communication between components. The *behavior view* integrates states (drawn as rounded rectangles) and state transitions into the object hierarchy at all places where the internal behavior of the system has to be modeled. Together with the decomposition hierarchy, we can define the semantics of the behavior view with a simplified version of the statechart semantics. In contrast to statechart semantics, events are broadcast only within the boundary of an object. From an originating object to a destination object, events have to be sent explicitly over associations or part-of-relationships. The *context view* shows the external actors (drawn as hexagons) of the system which are connected by associations to type scenarios[2] [GBJ02] in the *user view* (drawn as ovals). As interaction is frequently local, the scenarios are embedded in the object hierarchy at the position where they apply. Scenarios can also be decomposed, using an extended form of Jackson diagrams [Jac75] as notation.[3] The *functional view* describes the properties (i.e. the attributes and operations) of components. The functional view is not combined with other views; it is always represented separately in textual form. When there is only one object of a certain type, the complete type information is embedded in the object definition, otherwise the component's properties are provided by the type definition.

The ADORA language allows both semi-formal and formal modeling. Semi-formal means that some elements of a model don't abide by the formal semantics of the modeling language. A typical example is a state transition (a formal concept) for which the triggering condition is given in natural language. Furthermore, ADORA supports partial models, i.e. models containing parts that are intentionally incomplete [Xia04, SMG04]: some parts have not been modeled yet or will not be modeled at all. The difference to unintentional incompleteness is that the incomplete elements are known to be incomplete and therefore marked as such. Partial modeling is particularly useful in an evolutionary requirements modeling process, where we want to evolve a model in a controlled way through a series of iterations. In ADORA, we have two constructs for describing partial models: the first one is the so-called *is-partial* property which indicates that a component is incomplete. This is especially useful if a system part will still evolve or is incomplete at this time. The second construct is the so-called abstract association, which is represented as a bold line (see, for example, the association from *UserAdministrator* to *UserAdministration* in

[1]Abstract objects and object sets are also called components. A component is the superordinate concept of abstract objects and object sets.

[2]Type scenarios describe a set of system event sequences. Each of these sequences is initiated by a stimulus from an actor and ends with a system response. In contrast to type scenarios, instance scenarios just describe *one* example sequence of system events. Type scenarios are equivalent to use cases in UML.

[3]In our version, iteration is a property of all possible types of node. We added also parallel decomposition and made the notation layout-independent by numbering sequences of actions. Scenario nodes can be abstracted or marked as partial by three dots [Xia04].

Fig. 1). *Abstract associations* can be used if the modeler knows that there is some communication between components, but at the time of modeling it is not clear what the concrete communication will look like. Note that ADORA supports not only partial *models*, but also partial *views* [GBJ02]. Partial views are an abstraction mechanism that is used for deliberately hiding certain model elements or levels of detail from a diagram (for example, when a high-level, abstract view of a system is desired). ADORA supports partial views for abstract objects, object sets, scenarios, and states. Additionally, abstract relationships represent associations that are hidden in the current view. Partially viewed or modeled elements are indicated by names with three trailing dots (e.g. the object *BorrowManager*).

An ADORA model is the result of a process which could look like the one sketched in [SMG04]. This process makes it possible to evolve a requirements model semi-automatically by playing through scenarios.

2.2 Modeling Crosscutting Concerns with Conventional Modeling Techniques

The problem of crosscutting concerns will be explained in the context of the example described in Fig. 1. Suppose that there is a requirement to have a secure system. This requirement or goal manifests for example in the need to authenticate the users of the system and let them perform only operations that are permitted by the credentials for the particular user. This kind of security requirement is well known to be crosscutting in systems (e.g. see [CRS$^+$05]).

As an example, the functionality of borrowing books and the functionality of managing books provide operations that are crosscut by the authentication concern. Users utilizing these functionalities need to be authenticated first. We will use this problem as a running example to explain our approach. Fig. 2 contains the two components *BorrowManager* and *BookAdministration* of the library system implementing these functionalities[4]. Some chunks of the behavioral description for the authentication process can be found in *BorrowManager* and *BookAdministration*. The two *Authenticate* states with their outgoing transitions describe this kind of functionality. Additionally, the component *Authorization* is part of the authentication mechanism, which handles the authentication request and authorizes the particular user in the case of a successful authentication. The reflexive transition of state *Authenticate* sends an authentication request over the corresponding relationship to the *Authorization* component. *Authorization* processes the request. In case of success, it returns an authentication message containing the user credentials. Otherwise, it sends back a failure message, requiring the user to authenticate again.

The authentication mechanism described is scattered all over the system and tangled there with the core functionality of the system. Therefore the authentication mechanism is crosscutting. Any attempt to remodel the system in a better way will end up in situations like the one shown in Fig. 2. The resulting models will always be bloated by elements that seem to be redundant in some way. This kind of redundancy has a deep impact on the under-

[4]For the sake of simplicity, we will only refer to these two elements to explain the issues of our approach and hide the other elements of the system.

standability and therefore also on the validatability and verifiability of a model. In the next section, we will demonstrate how this problem can be overcome with an aspect-oriented model.

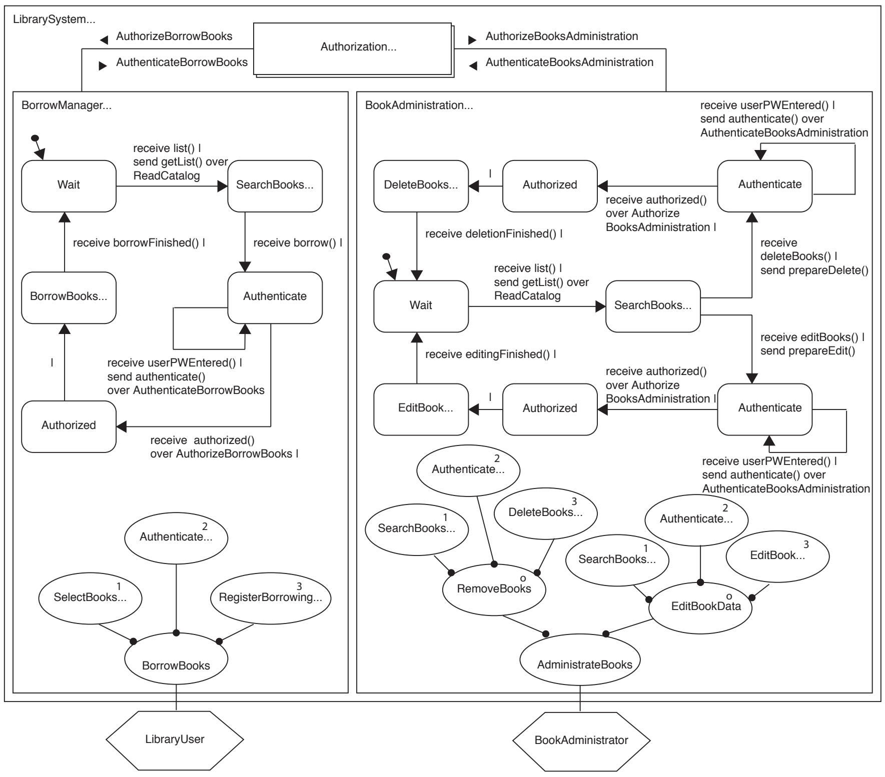
LibrarySystem...
AuthorizeBorrowBooks
AuthenticateBorrowBooks
Authorization...
AuthorizeBooksAdministration
AuthenticateBooksAdministration
BorrowManager...
Wait
receive list() I send getList() over ReadCatalog
SearchBooks...
receive borrowFinished() I
receive borrow() I
BorrowBooks...
Authenticate
I
receive userPWEntered() I send authenticate() over AuthenticateBorrowBooks
Authorized
receive authorized() over AuthorizeBorrowBooks I
Authenticate... 2
SelectBooks... 1
RegisterBorrowing... 3
BorrowBooks
BookAdministration...
receive userPWEntered() I send authenticate() over AuthenticateBooksAdministration
DeleteBooks...
I
Authorized
Authenticate
receive authorized() over Authorize BooksAdministration I
receive deletionFinished() I
receive deleteBooks() I send prepareDelete()
Wait
receive list() I send getList() over ReadCatalog
SearchBooks...
receive editBooks() I send prepareEdit()
receive editingFinished() I
receive authorized() over Authorize BooksAdministration I
EditBook...
I
Authorized
Authenticate
receive userPWEntered() I send authenticate() over AuthenticateBooksAdministration
Authenticate... 2
SearchBooks... 1
DeleteBooks... 3
RemoveBooks o
Authenticate... 2
SearchBooks... 1
EditBook... 3
EditBookData o
AdministrateBooks
LibraryUser
BookAdministrator

Figure 2: A part of the library system modeled conventionally in ADORA

3 Modeling with Aspects in Object Models

The goal of our approach is to provide an aspect-oriented modeling language for (functional) requirements and architectural design. In the following we will discuss the most important elements of our approach. Our approach explicitly deals with the three problems of aspect-oriented modeling described in Section 1. According to [MG05], problem i) can be mitigated by introducing different means for local and global validation / verification. We propose to use static methods like peer reviews for verifying / validating small parts of the model (local context), while we recommend dynamic techniques such as simulation on the global and woven level of the specification. For the sake of validation through simulation, we need the possibility to weave systems modeled in an aspect-oriented way and therefore we need a well defined weaving semantics for the aspect-oriented constructs in the model.

Problem ii) can be mitigated by using a tightly coupled and integrated modeling language. For this purpose, we use our object-oriented modeling language ADORA as a basis for an extension with aspect-oriented constructs. However, our approach is not constrained to ADORA; any other language with the same qualities could be used instead. With some restrictions and adaptations, it is also possible to use our concepts in loosely coupled modeling languages such as UML 2.0 [OMG03].

Problem iii) will be solved by creating a complete and coherent language which reflects the modeling of aspect-oriented artifacts in all affected views.

3.1 Aspect-Oriented ADORA

In the following, we describe our aspect-oriented extension of the language ADORA, which simplifies models by introducing new language elements for the modular separation of crosscutting concerns.

In contrast to the current (conventional) form of the ADORA language, the constructs of the aspect-oriented extension for ADORA do not have a defined execution semantics. For executing (e.g. simulating) aspect-oriented ADORA models, we need to transform them first to conventional ADORA models. This transformation process is called *weaving*. Therefore, in the following discussion of the aspect-oriented ADORA elements, each element will be described by its syntax, additionally needed language constraints and the weaving semantics.

We introduce what we call an aspect, which is a separated and modular description of a crosscutting concern. Aspects are not confined to behavioral descriptions only, but rather consist of a container with a static structure, relationships to other elements, etc. Therefore, an aspect manifests in almost all views of an ADORA model, namely in the *base view*, the *behavior view*, the *user view*, the *structural view* and also in the *functional view*.

3.1.1 Weaving semantics

As sketched in Section 3.1, aspect-oriented ADORA elements have no counterpart at the runtime of a model, i.e. they are not described by an execution semantics. Hence, they have to be transformed into conventional ADORA models by weaving aspectual and non-aspectual elements together. To enable this transformation, each aspect-oriented language element defines a set of transformation rules which describe how the aspect-oriented model is translated into a conventional ADORA model. The weaving of a model has an impact on every view of the model, except the context view.

3.1.2 Language Elements of the Aspect-Oriented ADORA Extension

Fig. 3 shows the system parts from Fig. 2 using the aspect-oriented version of ADORA. The crosscutting concern is shown as an aspect. The new elements and their weaving semantics are explained in the following.

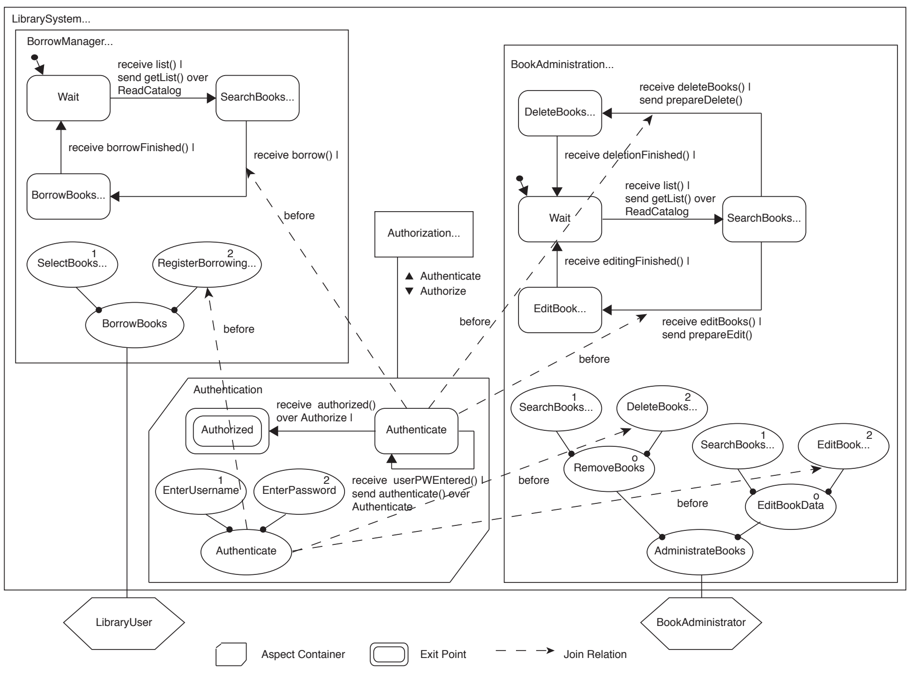

Figure 3: The example ADORA model described with aspect-oriented elements

Aspect Container An aspect container (just called aspect in the following) is a module that comprises all elements of a crosscutting concern. In ADORA it is drawn as a beveled rectangle (see Fig. 3). The name in the upper left corner indicates the name of the aspect. It can contain chunks of the crosscutting behavior and/or chunks of crosscutting scenarios (crosscutting behavior as it is seen by actors). An aspect can have attributes and operation definitions in its *functional view* which is woven into the *functional view* of the crosscut components.

The join relationships are directed relationships between the behavior or scenario chunks in the aspect and the element[5] that is crosscut, i.e. they indicate where to weave in crosscutting elements. The element which is crosscut is also called the target element.

When an aspect *A* is located in a component *C*, this does not mean that *A* has an is-part-of relationship with *C*, as this is the case with nested components. It rather indicates a certain

[5]Either a component or another aspect can be crosscut.

relationship which describes that *A* crosscuts either *C* or one or more child components of *C*. Join relationships to a direct or indirect parent element *P* of *C* are not allowed. Likewise it is not allowed to have a join relationship between *A* and a child of *P*, except with *C*. In Fig. 3 this means that no join relationship from the *Authorization* aspect is allowed to cross the border of *LibrarySystem*.

An aspect can contain components. When weaving an aspect *A* into the target element *T*, an identical copy (clone) of each component *C* contained in *A* is included in *T*.

Aspects can be partial, i.e. intentionally incomplete (see Section 2) and therefore indicate that their evolution is not finished. In this case the aspect name has trailing dots.

Join Relationships As described before, the join point model of Aspect-Oriented ADORA consists of a set of join relationships. A join relationship is denoted by a dashed arrow and can be attributed by a description of how to weave the elements. The semantics of join relationships for behavior is slightly different from that for scenarios (see the discussion below).

Join relationships can also be used to crosscut aspects. For example, suppose the situation where you would like to have an authentication aspect and a logging aspect, the latter logging certain operations in the system. If you want to have a logged authentication, you need to have the possibility of crosscutting an aspect by an aspect. Aspects that crosscut other aspects imply two constraints that must be fulfilled: (i) The weaving semantics for circular join relationships is not defined. Therefore cycles of crosscutting aspects are not allowed. (ii) For the same reason as in item (i), no reflexive join relationships are allowed.

Behavior of an Aspect Aspect behavior chunks are modeled by a statechart-like syntax and semantics. These chunks are no full-fledged statecharts, i.e. no start states are allowed in behavior chunks. There has to be exactly one entry point and one exit point for each of the aspect behavior chunks. The entry point of the crosscutting behavior denotes the state where the crosscut behavior enters the crosscutting behavior. It is drawn as a state with an outgoing join relationship. A unique exit point of the crosscutting behavior is denoted by a double lined state shape. An exit point can not have any outgoing transitions and can be named optionally.

A join relationship connects the entry point with the transition where the crosscutting behavior is woven in. The join relationship has to be attributed with one of the keywords *before*, *instead* or *after*. Fig. 4 describes the weaving semantics. On the left hand side, there is the aspect-oriented model, whereas on the right hand side, you will find the woven version:

- Fig. 4a describes the weaving semantics of a *Before* join relationship. The transition between *A* and *B* is crosscut by performing the behavior chunk given in the aspect before the action of the crosscut transition is executed. An additional state is introduced which is named either according to the name of the exit point or (if no name is given) an artificial name is generated. This state has an outgoing transition containing the action *b* from the crosscut transition.

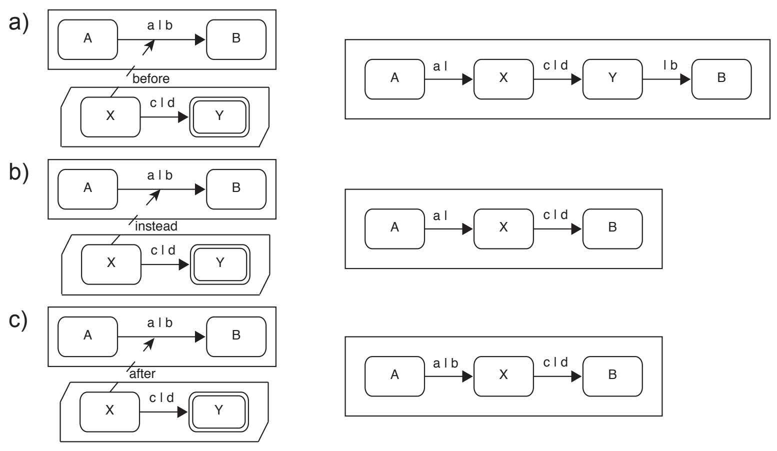

Figure 4: The weaving semantics for internal aspectual behavior

- Fig. 4b describes the weaving semantics of an *Instead* join relationship. This type of relationship indicates that the action *b* of the transition between *A* and *B* in the crosscut behavior is removed and the crosscutting behavior is added to the crosscut transition. The exiting transition of the crosscutting behavior is connected to the state *B*.
- Fig. 4c describes an *After* join relationship. The After relationship expresses that the crosscutting behavior is immediately inserted after the transition $a \mid b$ which is crosscut. The exit transition $c \mid d$ of the crosscutting behavior is connected to the state *B* of the crosscut transition.

Beside aspect behavior chunks, it is possible to model behavior that is executed concurrently with the crosscutting behavior. This can be modeled by a conventional ADORA statechart embedded in the aspect. When weaving, a clone of this concurrent behavior is copied into the crosscut element.

Scenarios Scenarios describe the protocol of the interaction between the actors in the context and the system (see Section 2.1). Like the internal behavior of the system, scenarios can also contain crosscutting behavior as for example discussed in [Jac03]. Crosscutting elements can be found in our example, as illustrated in Fig. 2. In this example, the *Authenticate* (sub) scenario appears at different locations (e.g. in *BookAdministration* and in *BorrowManager*), where it crosscuts the model. Therefore, there is a need to introduce an aspect-oriented notation for crosscutting scenarios.

Aspect scenario chunks can be described in Aspect-Oriented ADORA by modeling the scenario in the aspect which contains the corresponding (internal) crosscutting behavior described by a crosscutting statechart chunk. The root node of the crosscutting scenario tree is connected by a join relationship to the scenario node where the crosscutting scenario has to be inserted as a sub-scenario. This is depicted in Fig. 3, where the *Authenticate* scenario chunk is connected to the corresponding crosscut scenario trees. There is no need to specify an exit point for the crosscutting scenario as it is done for the crosscutting behavior statecharts. This is a consequence of the tree structure which results in a uniquely defined execution order for processing the scenario nodes.

The type[6] of the crosscutting scenario root node has to be of the same type as the target scenario node and its siblings. The join relationships can be attributed with one of the keywords *before*, *instead* or *after*. The keyword influences the way the model is woven but only if the root node of the scenario chunk is of the type *Sequence*. For root nodes in the crosscutting scenario chunk which have the type *Alternative* or *Concurrent*, all of the above mentioned keywords have no influence on the weaving, because scenario nodes of this type have no particular execution order.

Fig. 5 illustrates the weaving semantics for scenario chunks which are crosscutting a scenario sequence. On the left hand side, there is the aspect-oriented version, whereas the woven version can be found on the right hand side:

- Fig. 5a describes the *Before* join relationship. The join relationship causes the crosscutting scenario chunk having a root node of the type *Sequence* to be inserted before the target scenario node. The inserted node gets the sequence number of the target node. The sequence numbers of the subsequent nodes including the denoted one are incremented by one.
- Fig. 5b depicts the *After* join relationship. The root node of the crosscutting scenario chunk is inserted after the specified target node. The sequence numbers of all subsequent nodes are incremented by one.
- Fig. 5c explains the weaving semantics of the *Instead* join relationship. In this case, the target node in the crosscut scenariochart is replaced by the root node of the crosscutting scenario tree.

Functional View The weaving semantics for the functional view is an open issue in our research. In general, the *Functional View* of an aspect, i.e. the attributes and operations, etc. will be woven with the functional view of the target elements. The following two problems may happen during the weaving of an aspect's *Functional View* into target elements:

(i) Naming conflicts can occur between elements in the name space of the target element and the aspect's functional view. A mechanism which renames the conflicting *Functional View* elements in a unique way solves this problem. (ii) Aspect-orientation breaks the principle of information hiding [Par72] by injecting behavior which is not under control of the target element. This can result in software contract violations. The compliance of crosscutting concerns with contracts of crosscut elements is an open research topic in the AOSD community.

[6]The different types are: *Alternative*, *Sequence* and *Concurrent*.

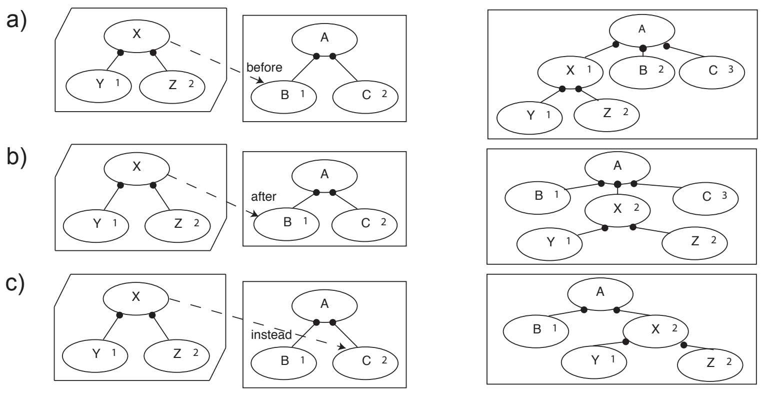

Figure 5: Weaving semantics for scenario chunks crosscutting a sequence of scenarios.

3.1.3 Reducing Cognitive Overhead in Aspect-Oriented ADORA Models

Join Relationships Other aspect-oriented approaches, e.g. AspectJ [KHH^{+}01], often use pattern matching mechanisms to identify join points. In our approach the modeler has to denote the join points explicitly. We deliberately do not use pattern matching, because not explicitly visualizing such important information would reduce the effectiveness of a visual model and increase the cognitive effort needed to read it. Apart from that, unintentional (i.e. wrong) matches may occur when using pattern matching.

When a model contains many join relationships, explicitly representing them can make the model more complex. This problem can be mitigated in Aspect-Oriented ADORA by using partial views (see Section 2) that hide the join relationships which are out of the focus of interest, i.e. it is possible to hide any number of join relationships.

Abstraction Mechanisms Furthermore, it is possible to reduce the complexity by hiding the inner structure of aspects and the crosscut elements. In this case, the join relationship between these elements will be abstracted, i.e. shown as a bold dashed arrow. Fig. 6 shows different types of situations that may occur. Fig. 6a shows the involved elements (in this case an aspect and a component) in a non-abstracted situation. The join relationship is shown as a normal dashed line. Fig. 6b shows the situation with the abstracted aspect, whereas Fig. 6c shows the situation where the component is abstracted. In Fig. 6d both the aspect and the component are abstracted.

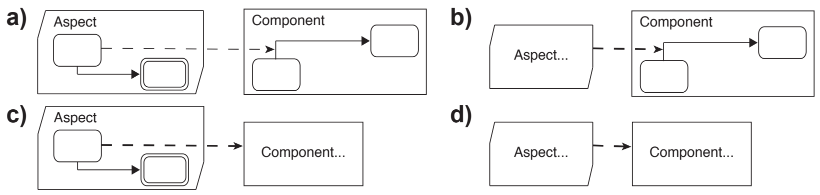

Figure 6: Influence of the ADORA abstraction mechanism on join relationships.

3.2 The Impact of Weaving on the Model Structure

Aspects are allowed to be connected by associations to abstract objects.[7] Associations between an aspect and an abstract object have to be taken into account when weaving an aspect-oriented model. In this case weaving has an impact on the *Structural View* and on the *Base View*. In the following, we call abstract objects connected by an association to an aspect *aspect-oriented server objects*[8] or just briefly *server objects*.

3.2.1 Structural View

Fig. 7 illustrates the change of the association structure in an aspect-oriented model after the weaving process. For the sake of simplicity, the scenarios and the inner structure of all the components except the behavior of *Authorization* and *BorrowManager* are hidden. Due to the fact that the behavior of an aspect is cloned and woven into different target objects, the communication channels from aspects to *server objects* have to be cloned too, because they are used by the crosscutting behavior to communicate with *server objects*. In Fig. 7 this manifests in the associations from *UserAdministration* to *Authorization*, from *BookAdministration* to *Authorization* and from *BorrowManager* to *Authorization*.

The number of associations that have to be introduced for each aspect woven in the model can be computed as follows. Let *O* be the *server object*. If *n* different target elements $T_{1...n}$ are crosscut by the aspect *A* and there exist *m* associations $a_{1...m}$ between *A* and *O*, there are $m * n$ new associations introduced.

3.2.2 Base View

A server object *O* is assumed to be in a certain state S_O to provide a service for the crosscutting behavior of the aspect *A*. As soon as the crosscutting behavior of *A* is woven into the target elements $T_1...n$, we have to make sure that the state S_O is preserved for each of the *n* cloned crosscutting behaviors. Therefore, the *server object* has to be cloned too, i.e. the woven model must contain a set of *server objects* with a cardinality of (n, n). This fact is illustrated by the object set *Authorization* in Fig. 7.

[7]For the sake of simplicity, *Object Sets* are not allowed to be connected to an aspect by an association.
[8]This is due to the fact that these objects provide a service for an aspect.

3.2.3 Adaption of Behavior

The crosscutting behavior has to be modified so that messages are addressed and sent to the correct object in the *server object set*. This is done by the ADORA reflection mechanism with its ability to address objects in object sets, which is not explained in detail here.

The behavior of the objects in the *server object set* has to be slightly modified too by storing the *object id* of the sender object. This *id* is used to send messages back. Without using this *id* responding to messages would result in a multicast[9] to all components that contain crosscutting behavior.

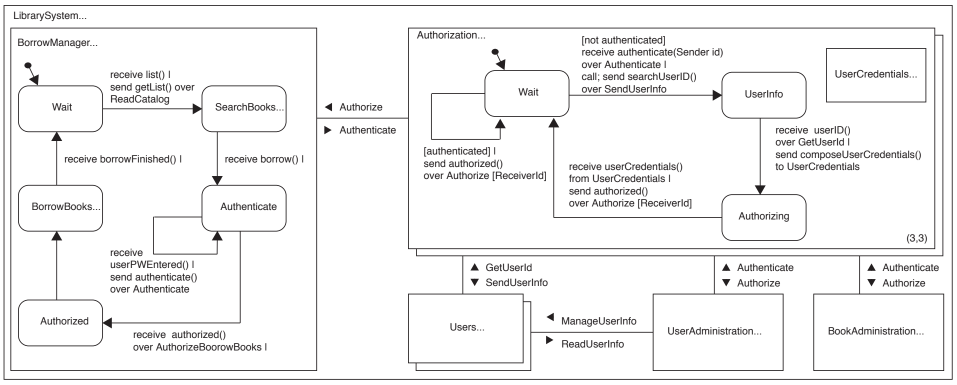

Figure 7: The woven system.

4 Related Work

Many aspect-oriented modeling approaches are discussed within the research community. Most of them, for example [SY99, GB04], are very close to AOP approaches.

There are only a few approaches dealing with aspect-oriented architectural design. In [AK03] stratified frameworks are discussed. Stratified framework are rather a viewing mechanism than an aspect-oriented approach. A discussion of further aspect-oriented architectural design approaches can be found in [CRS$^+$05].

There are some aspect-oriented approaches to requirements engineering. For example, [AWK04] describes an approach for weaving crosscutting behavior with core concern behavior that are both represented in terms of statemachines. These statemachines are generated from use cases. However, this approach considers only the behavioral view and neglects the others. In [YdPLM04] an approach for discovering aspects of requirements in *i** goal models is described, while [BM04] investigates aspects for the NFR (non-functional requirements) framework.

[9]This is due to the fact that the cloned associations have the same name.

A good survey of the currently existing aspect-oriented modeling approaches can be found in [CRS+05].

5 Conclusions and Outlook

In this paper, we presented an approach for modeling modularized crosscutting concerns for functional requirements and software architectures, respectively. We have shown that tightly coupled modeling languages are better suited for the introduction of an aspect-oriented extension than loosely coupled ones. We demonstrated that the understandability and therefore also the validatability and verifiability can be improved by using the newly introduced aspect constructs.

For an effective and efficient usage of ADORA and the aspect-oriented approach presented here, tool support is needed. We are currently extending our existing ADORA tool prototype with the aspect-oriented extensions presented in this paper.

The practical validation of the presented approach will be in the focus of our future research. Furthermore, we will research the problem of identification and separation of crosscutting concerns during the requirements phase. For this purpose, we plan to extend the process presented in [SMG04] which provides a use-case-driven method for evolving and simulating requirements.

References

[AK03] C. Atkinson and T. Kühne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1):81–89, 2003.

[AWK04] J. Araújo, J. Whittle, and D.-K. Kim. Modeling and Composing Scenario-Based Requirements with Aspects. In *12th IEEE Requirements Engineering Conference (RE'04)*, pages 58–59, 2004.

[BM04] I. Brito and A. Moreira. Integrating the NFR framework in a RE model. In *Early Aspects 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004)*, 2004.

[CRS+05] R. Chitchyan, A. Rashid, P. Sawyer, J. Bakker, M. Pinto Alarcon, A. Garcia, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of Aspect-Oriented Analysis and Design. In *R. Chitchyan, A. Rashid (eds.): AOSD-Europe Project Deliverable No. AOSD-Europe-ULANC-9.*, 2005.

[GB04] I. Groher and T. Baumgarth. Aspect-Orientation from Design to Code. In *Early Aspects 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004)*, 2004.

[GBJ02] M. Glinz, S. Berner, and S. Joos. Object-Oriented Modeling with ADORA. *Information Systems*, 27(6):425–444, 2002.

[IEE98] IEEE - The Institute of Electrical and Electronics Engineers. *IEEE Recommended Practice for Software Requirements Specifications. IEEE Std 830-1998*. IEEE Computer Society Press, 1998.

[Jac75] M. Jackson. *Principles of Program Design*. Academic Press, New York, 1975.

[Jac03] I. Jacobson. Use Cases and Aspects – Working Seamlessly Together. *Journal of Object Technology*, 2(4):7–28, 2003.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 1997)*, pages 327–353, 1997.

[Lad03] R. Laddad. *AspectJ in Action, Practical Aspect-Oriented Programming*. Manning Publications Company, New York, 2003.

[MG05] S. Meier and M. Glinz. Problems when Introducing Aspect-Oriented Constructs in Models of Functional Requirements and Possible Solutions to these Problems. In *Models and Aspects - Handling Cross-Cutting Concerns in MDSD, Workshop at the 19th Euorpean Conference on Object-oriented Programming (ECOOP 2005)*, 2005.

[OMG03] OMG. UML 2.0 Superstructure Specification. OMG document ptc/03-08-02. Tech. Rep., Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/2003-08-02, 2003.

[Par72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[SMG04] C. Seybold, S. Meier, and M. Glinz. Evolution of Requirements Models by Simulation. In *7th International Workshop on Principles of Software Evolution (IWPSE 2004)*, pages 43–48, 2004.

[SY99] J. Suzuki and Y. Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. In *Proceedings of the 3rd Aspect-Oriented Programming Workshop at the 13th Euorpean Conference on Object-oriented Programming (ECOOP 1999)*, pages 299–300. Springer, 1999.

[TOHS99] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Converence on Software Engineering (ICSE 1999)*, pages 107–119, 1999.

[Xia04] Y. Xia. *A Language Definition Method for Visual Specification Languages*. PhD thesis, University of Zurich, 2004.

[YdPLM04] Y. Yu, J. C. Sampaio do Prado Leite, and J. Mylopoulos. From Goals to Aspects: Discovering Aspects from Requirements Goal Models. In *12th IEEE Requirements Engineering Conference (RE'04)*, pages 38–47, 2004.