Approach for Modeling Aspects in Architectural Views

ABSTRACT

This document presents the approach for modeling aspects in architectural views. It builds on earlier deliverables that have not explicitly covered the notion of architectural views as it is currently applied in architecture design community. We have selected the module view, the component and connector view and the deployment view. Views are described by viewtypes that are in essence metamodels. We provide a discussion on how to approach metamodeling for architectural views and define the (aspect-oriented) metamodels (viewtype) for the selected views.

Document ID:	AOSD-Europe-UT-D77
Deliverable No:	D77
Work-package No:	6
Type:	Deliverable
Status:	FINAL
Version:	1.0
Date:	February 27, 2007
Author(s):	Bedir Tekinerdogan (UT) Alessandro Garcia, Claudio Sant'Anna, Eduardo Figueiredo (ULANC), Mónica Pinto, Lidia Fuentes (UMA)

Table of Contents

Table of Figures

1. Introduction

In previous deliverables we have defined an aspect-oriented architecture design approach in which we have not made an explicit distinction between different architectural views. By separating and modelling different views for the different concerns of the stakeholders the complexity of architectural description can be reduced, and each view can be described using specialized notations. Views are described using viewpoints which provide the necessary modelling notations to develop individual views for particular stakeholders. Another recent term that is close the notion of viewpoint is the term *viewtype* as it is introduced in [10]. In essence, a viewtype can be considered as a category of views. From this perspective we can identify three different viewtypes: Module viewtype, Component and Connector Viewtype and Deployment Viewtype [10]. Different views can be developed from these three basic categories.

This document elaborates on our earlier work on aspect-oriented modeling of aspects at the architecture design level by considering the different views. For this we will consider three views from the three categories. More concretely this deliverable includes the following contributions:

- *Impact analysis of views on aspectual architectural modelling*. We provide an analysis on the impact of views on the architectural model. This implies that the current model will be enhanced and integrated in one of the selected views (see next points).

- *Updating the existing architectural model to represent different views.* We have defined three different viewtypes for representing an aspect-oriented architecture: module view, component-and-connector view and deployment view.

The remainder of this document is organized as follows:

Section 2 elaborates on modelling of aspects in architectural views. It presents the set of alternatives that can be taken to provide metamodels for architectural views.

Section 3 presents the metamodel of the module viewtype.

Section 4 presents the metamodel of the component and connector viewtype.

Section 5 presents the metamodel of the deployment viewtype.

Finally in section 6 we provide the conclusions.

2. Modelling and Meta-Modelling of Software Architecture Views

2.1 Metamodeling

Meta-modeling is the activity in which models are provided for models. The product of the metamodeling activity, called *metamodel*, is a model. In this document we adopt the following definition of metamodel:

"Meta-models are models that make statements about modeling. More precisely, a metamodel describes the possible structure of model – in an abstract way, it defines the constructs of a modeling language and their relationships, as well as constraints and modeling rules – but not the concrete syntax of the language. We say that a a metamodel defines the abstract syntax and the static semantics of a modeling language. "

T. Stahl & M. Voelter, Model-Driven Software Development [33].

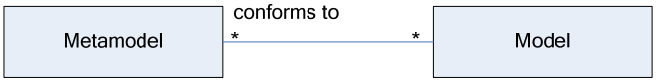

Figure 1. *Relation between model and metamodel*

Note that based on this definition a metamodel is not just a model of another model, but basically a model of a modeling language [21]. Hereby, it is in particular important to make a distinction between the abstract syntax and the concrete syntax of the language. This is illustrated in Figure 2.

The concrete syntax can be, for example, the textual or visual representation of the language. In a sense, it specifies what the parser of a language would accept [33]. The abstract syntax on the other hand, abstracts away from the possible representations and merely specifies what the structure of the language looks like. The concrete syntax can be considered as a realization of the abstract syntax.

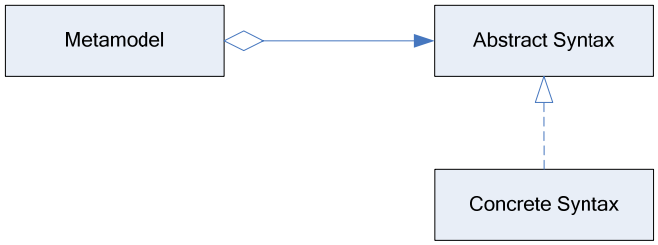

Figure 2. *Distinction between Abstract Syntax and Concrete Syntax*

2.2 Models and Meta-Models for Architectures

In the deliverables so far we have developed models for the architecture, that is, textual model as ADL and a visual model including graphical notations. For both models we have developed a meta-model and a first initial uniform meta-model. Since metamodels represent the language for expressing the architectural model, these can be considered as viewpoints. Figure 3 illustrates the relation between viewpoints and metamodels.

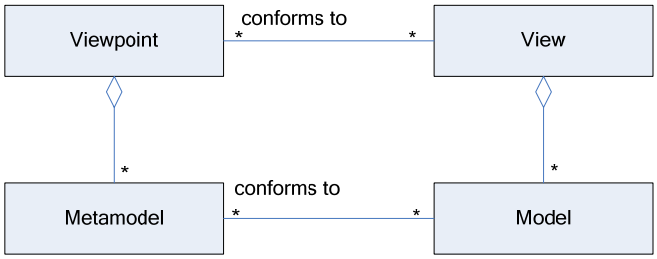

Figure 3. *Relation between viewpoints and metamodels [11]*

According to the IEEE Standard on Architectural Description a view consists of a collection of models. Figure 3 provides an enhancement to this standard by introducing the concept of metamodel [11].

If we consider multiple views of architecture then we need to elaborate on the previous modeling and meta-modeling approaches. In particular the question that needs to be answered is how to define the models and meta-models for the three selected view(types).

The approach that we will use for meta-modeling is based on the four-layer MOF metamodeling architecture. This architecture is a proven infrastructure for defining the precise semantics required by complex models. Hereby the modeling is decomposed into four layers. The M0 layer defines the real world entities.

M1 layer defines the classes. For example, “Account” and “Withdrawal” are concepts at the M1 layer. The primary responsibility of the model layer is to define a language that describes an information domain. In our approach the models are the specific views for a given application domain.

In the M2 layer, the metamodel, the constructs that are used in M1 model are defined. Elements of M1 are thus instances of elements at the M2 level. An example metamodel M2 is the UML metamodel. Viewtypes also reside at the metamodel level since they describe views. For instance, “Module” or “Component” are concepts at the viewtype level.

The meta-models at M2 will be instantiated from the meta-meta-model at the M3. The meta-metamodeling layer forms the foundation for the metamodeling architecture. The primary responsibility of this layer is to define the language for

specifying a metamodel, in this case, for modeling architecture. A meta-metamodel defines a model at a higher level of abstraction than a metamodel, and is typically more compact than the metamodel that it describes. A meta-metamodel can define multiple metamodels, and there can be multiple meta-metamodels associated with each metamodel. Here, the meta-metamodel defines the modeling of architecture independent of the various views. In this document we also assume that M3 is defined as MOF.

2.3 Viewpoints as Metamodels

Since we have decided to model different views of architecture this implies that we will need meta-model(s) for the different views. We have the following alternatives for doing this:

1. Provide one common metamodel (level M2) for all viewtypes.

2. Provide one metamodel for each viewtype (level M2), which share the meta-metamodel (level M3).

For both alternatives we have to consider different two types of representation: textual and visual. Also note that each alternative will have an impact on the way how meta-metamodel is represented.

In the following we discuss both alternatives.

2.3.1 Common Metamodel for viewtypes

Figure 4 presents the alternative with one shared meta-model. At the modeling level (M1) we will have different views of the architecture including the module view, C&C view and Deployment View. As you can see there is only one metamodel described for representing viewtypes. The implication of this decision is that the concepts of different viewtypes are integrated at the M2 level. This means that one viewtype describes the complete notation for representing the different elements. In case of the three represented view, we would then have an one architectural modeling language (textual or visual) for representing modules, components and nodes. In principle this is possible, and one might find it beneficial to have one metamodel only for representing architectural views.

However, there are also several disadvantages related to this alternative. First of all, the metamodel would then be more complicated because all the elements of the viewpoints have to be taken into account. It will be harder and more difficult to understand the metamodel.

Secondly, the metamodel would be constrained to the three viewtypes. In case we would introduce another viewtype then this would mean that we need to extend or change the metamodel later on.

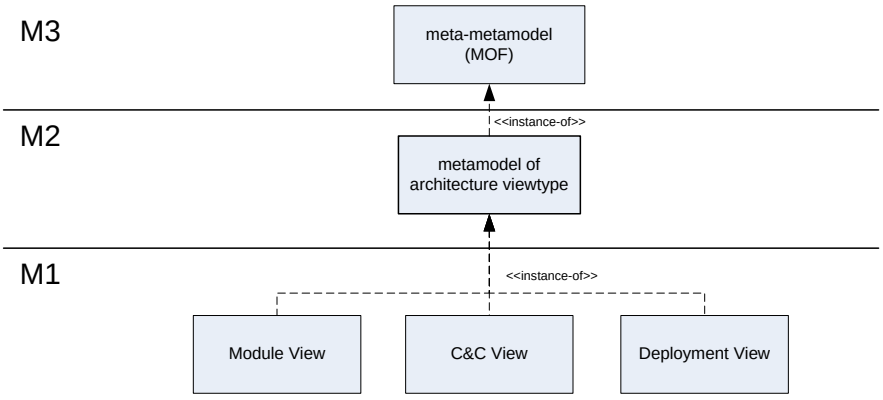

Figure 4. *The four metalevels for architectural views: one common meta-model for different viewtypes*

The meta-metamodel here can be represented using, for example, the MOF [ref], EMF ECore [ref] or KM3 meta-metamodel [ref].

2.3.2 Common Metamodel with Refinement for Different Viewtypes

Figure 5 provides an alternative to the metamodel of Figure 4. Here we have again one commone metamodel for the different viewtypes but enhancements (at the metamodel) are provided to customize the common metamodel for the particular viewpoints.

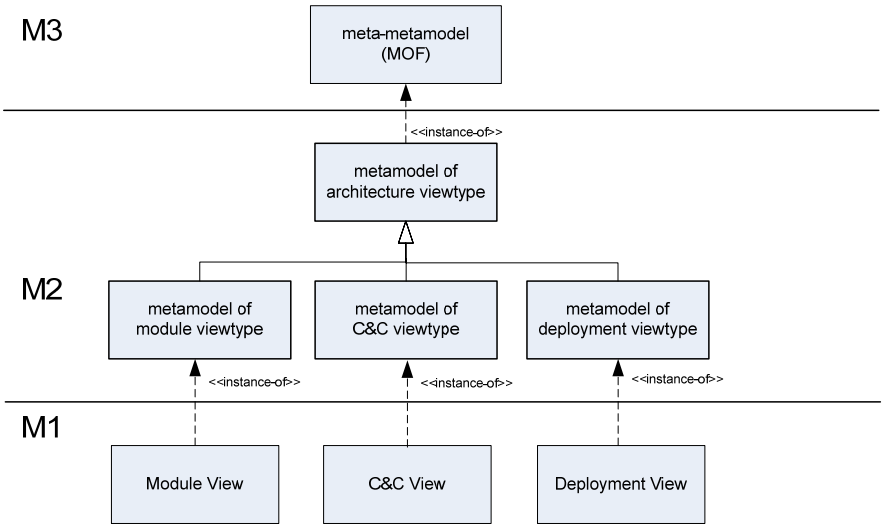

Figure 5. *The four metalevels for architectural views: one common meta-model for different viewtypes*

2.3.3 Different Metamodel for Different viewtypes

A second alternative to modeling views using a metamodeling approach is shown in Figure 6. The rationale for this alternative lies in the fact that the different views are conceptually different and as such require a different language, that is a different meta-model, for different viewtypes. In that case we have to define three metamodels for the three different viewtypes. Thus, we will have a metamodel for the module view, metamodel for the C&C view, metamodel for the deployment view. In fact these can be named as the module viewpoint, C&C viewpoint, and deployment viewpoint.

These metamodels are all instantions of the meta-metamodel (M3). Again, the meta-metamodel here can be represented using, for example, the MOF meta-metamodel [23].

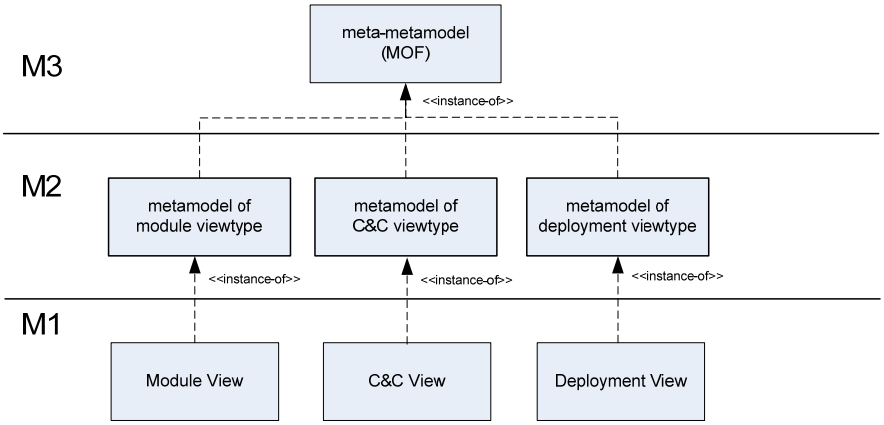

Figure 6. *The four metalevels for architectural view: different metamodels for different viewtypes*

2.3.4 Selecting a metamodeling approach

We have chosen for the last alternative in which we have to define three metamodels for the architectural views. In fact this means that we need to define the three viewtypes since in essence viewtypes are considered metamodels.

There is however an important issue here. As we have seen before we can model an architecture either textually or visually. In previous deliverables we have defined metamodels for the texual representation and metamodel for visual representation of the C&C view.

Based on our previous discussion the way how we represent architecture is actually independent of the metamodel itself, that is, on the concrete syntax. To repeat the previous discussion the *concrete syntax* is for example the textual or visual representation and defines what a parser for the language would accept. The *abstract syntax* merely specifies what the language's structure should like. Various concrete syntax forms can thus have one common abstract syntax. Put

another way, the metamodel (viewtype) of an architecture can be expressed in different notations, either graphically or textually.

If we would consider a separate metamodel for both the textual representation and visual representation (which is, again, conflicting with the previous discussion) then we would have the situation as depicted in the following figure.

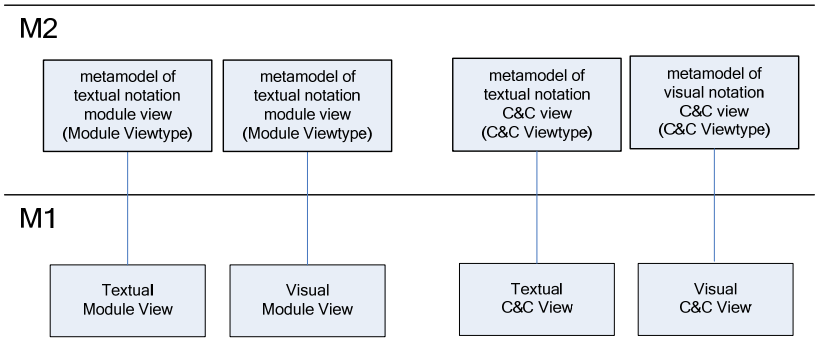

Figure 7. *Different metamodels for different representations (not chosen)*

In fact, this view is conflicting from the current ideas on meta-modeling. Metamodel is in fact independent on the concrete syntax. In modeling the metamodels we will thus need to focus on one abstract syntax.

3. Meta-Model of Module Viewtype

The module viewtype describes the language for module views in which elements are modules, which are units of implementation [10]. Since the way in which the software is decomposed has a direct impact on the structure the module view seems to be one of the most important views. In general every architecture description will include the module view.

The basic elements of the module view are the modules and the relations. The relations among modules include *is-a*, *is-part-of*, and *depends-on*. The *is-a* relation defines a generalization relation among modules. The *is-part-of* relation defines a part/whole relationship between two a part module and the parent module. The *depends-on* relation defines a dependency relationship between modules.

The metamodel for the module view is represented in Figure 8. From this figure it becomes clear that the metamodel for the module view is in essence a simplified version of the UML metamodel. This is not surprising because the UML also largely supports the structuring of implementation units and has been, as such, widely used to represent the module view of an architecture.

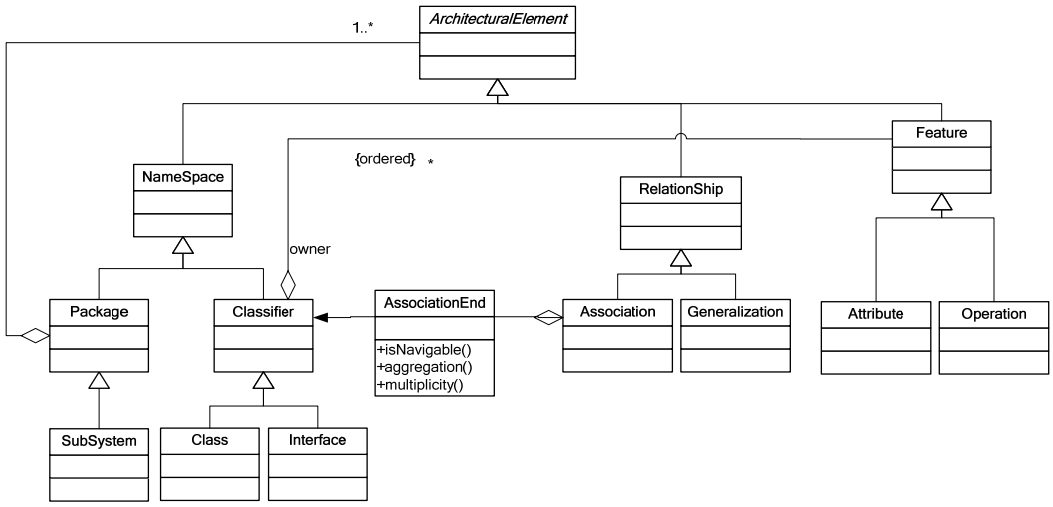

Figure 8. *The metamodel for the module view*

3.1 UML with extension mechanisms

Obviously we can not reuse the 'plain' UML for our purposes since UML does not provide explicit mechanisms for representing crosscutting concerns. However, UML does provide so-called extension mechanisms to adopt the existing mechanisms that are provided by default. These include *Stereotype, Constraint and Tagged Value,* which can be used separately or together to define new modeling elements that can have distinct semantics, characteristics and notation relative to the built in UML modeling elements specified by the UML metamodel. *Stereotype* is used to introduce

a new model element as extension of an existing base element. *Tagged Values* define properties of stereotyped elements and relations. *Constraints* are a set of well-formedness rules expressed in OCL or natural language to define new semantics to the extended model elements.

Several authors have indeed proposed to use the UML with extension mechanisms to represent the aspect-oriented concepts such as aspects, crosscutting, pointcut and advice [1][20][22][26][34][38]:

Aspect – A module that represents a crosscutting concern and which has a crosscutting relation with other modules.

Crosscutting – A new association relation to represent the crosscutting of an aspect over different modules

Pointcut – An expression in the aspect that defines the crosscutting relation

Advice – The operations that are imposed on modules that the aspect crosscut.

In general, notations (for module view) similar to as defined in Figure 9 are used in these approaches. That is, usually an <<aspect>> stereotype is applied to a the UML class element to represent aspects. Pointcuts are represented using the stereotype <<Pointcut>> or <<crosscut>> like names for operations or attributes. Finally, the advice concept is usually represented by using stereotypes <<advice>> for again the operations or attributes of a class. Regarding the *crosscut* relation of aspects to modules there are some slight differences. Some authors suggest just omitting the visual links in the diagram [22], while others define explicit notations using stereotyped associations like <<crosscut>> [34].

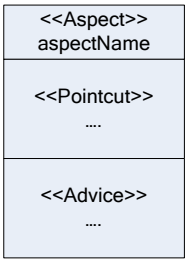

Figure 9. *Typical approaches for representing aspects with stereotypes*

The extension mechanisms together can be defined in a so-called UML Profile. In [1], for example, a UML profile is provided for representing aspects in UML.

3.2 Enhancing UML metamodel

Of course it is also possible to extend the UML metamodel by explicitly adding new metaclasses and other meta constructs. The key reason for this approach is to provide more expressive and native support for the aspect-oriented concepts. That is, aspects need to be represented as first class abstractions.

In the literature we can identify several attempts to enhance the kernel UML metamodel model with aspect-oriented abstractions. The model in Figure 10 represents a typical example for enhancement of the UML metamodel and as such should be considered in the same line of existing work.

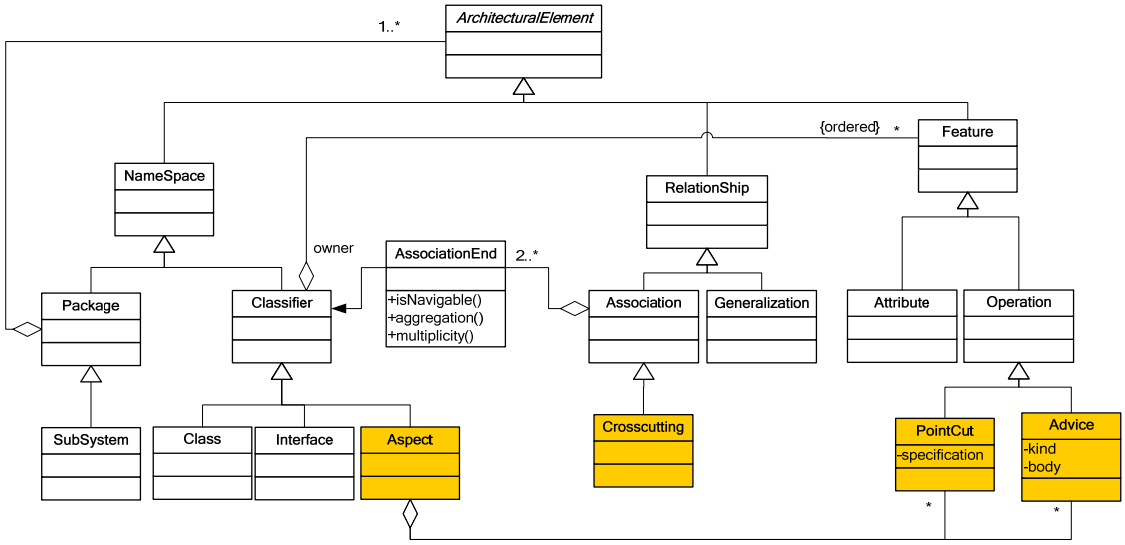

Figure 10. *Meta-Model for the Aspect-Oriented Module View*

Here, it should be noted that completely new model elements are introduced instead of "just" stereotyping existing UML elements. Typical enhancements to the metamodel include the concepts of *aspect, crosscutting, pointcut* and *advice*. *Aspect* is generally defined as a specialization (in the metamodel) of the *classifier* metamodel element. *Crosscutting* relation is usually inherited from the metaclass *Association*. *Pointcut* and *Advice* are usually direct or indirect subclasses from the metaclass *Feature.*

The metamodel in Figure 10 does not specify how the newly introduced concepts are represented. In principle the metamodel abstracts away from the type of representation and as such different representations can be used as instances of this metamodel.

3.2.1 Example – Theme/UML

In the current work we can, for example, identify Theme/UML [8] as an approach that requires enhancements to the kernel UML metamodel. In Theme/UML each model contains its so-called own *theme*, or design of an individual requirement, with concepts from the domain designed from the perspective of that requirement [8]. The extensions to the UML metamodel are needed for the composition of the thematic design models. This is achieved with a so-called *composition relationship*. A composition relationship specifies how models are to be composed by identifying overlapping concepts in different models and specifying how models should be integrated. Earlier deliverables in the project have provided detailed discussion on Theme/UML including the notations, and therefore we will not elaborate on this further. However, we will focus here on the impact on the metamodel [9].

The scope of the approach is basically the structural and collaboration diagrams in UML. The style for restricting the kinds of model elements that may participate in compositions is similar to the way that the UML defines the model elements that may participate in generalization relationships. In the UML, model elements that may participate in a generalization will inherit from the metaclass *GeneralizableElement*. Since Theme/UML focuses on composition a new abstract metaclass called *ComposableElement* is introduced to define which model elements may participate in a composition relationship (See Figure 11). Composable elements are divided into two types: primitives and composites. Primitives can be *Interaction*, *Attribute, Operation* and *Association.* Composition Elements are *Package, Classifier, Collaboration* and *Subject (*or *Theme*).

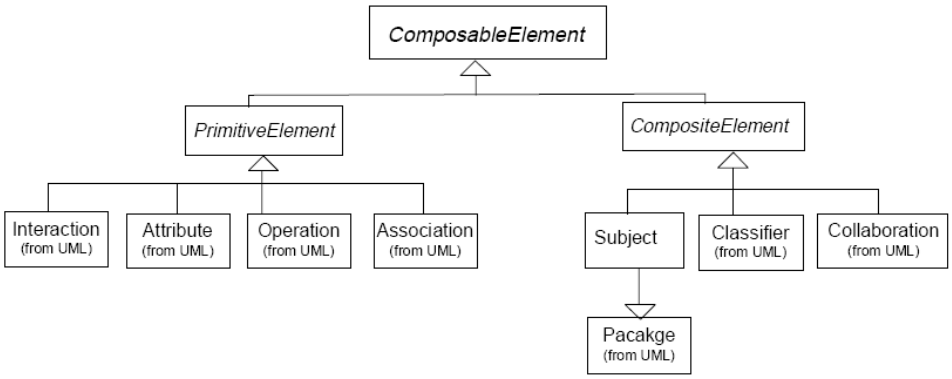

Figure 11. *ComposableElement metamodel as an enhancement to UML metamodel in the Themes/UML approach*

Since a composition relationship is also a new element for the UML, a new metaclass *CompositionRelation* has been introduced that is subclassed from the UML metaclass *Relationship* (see Figure 12). Composition relationships are defined between composable elements. A distinction is made between a primitive composition relationship, a composition relationship between primitives, and a composite composition relationship which is a composition relationship between composites.

Composition in Theme/UML involves the composition of two or more input subjects into an output subject. Composable elements explicitly related by a composition relationship are said to *correspond*, and are integrated based on the semantics of the particular integration strategy attached to the composition relationship. The *Match* metaclass provides a more general approach to identifying corresponding elements, because it supports the specification of match criteria for components of composite elements related by a composition relationship. For more detail on the enhanced metamodel we refer to [9].

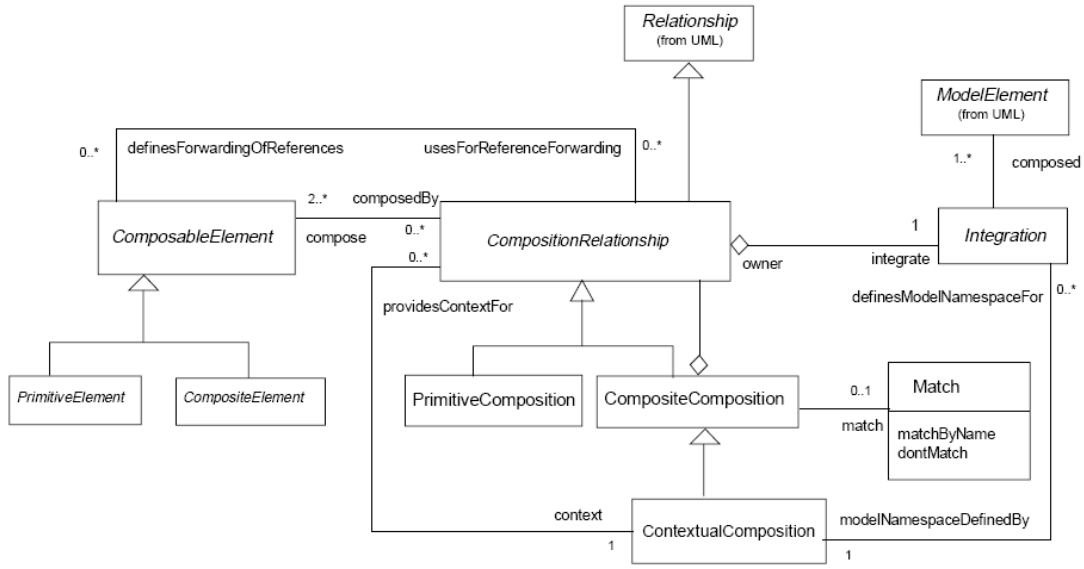

Figure 12. *CompositionRelationship in Theme/UML*

4. Meta-Model of C&C Viewtype

The C&C viewtype represents the views in which the elements are components and connectors. Components are principal units of computation; connectors are the communication means among components. The principle relation shown in C&C views is attachment between the components and the connectors. C&C views depict the major executing components and how do they interact.

The AOSD-Europe network has delivered two approaches for expressing component-and-connector views of aspect-oriented software architectures, namely a unified AO ADL [29][28], and an aspect-oriented visual notation [16]. Both approaches are very closely related and share a common language to describe the C&C view. This means that either the AO-ADL language or the visual notation can be indistinctly used to describe the C&C view type of aspect-oriented software architectures as part of the development methodology being developed in the project. Additionally, the mappings defined in D63 [7], from RDL to architecture, and from architecture to the unified design approach, are valid in both cases, as was discussed in D63.

In spite of sharing a common language, these approaches also present some differences in the final representation of these common concepts in order to facilitate:

(1) The communication and understanding of the software architecture between different stakeholders, especially in the case of the visual notation – i.e. we tried to maintain the visual representation of the architecture as concise and clear as possible, and

(2) The enhancing of the degree of information provided at the architecture level, especially in the case of the textual notation. We found this especially useful during the definition of the mapping process from requirements to architecture, since we realised that many times requirements provide useful information (e.g. partial behavioural tips) that is lost at the architecture level due to lack of appropriate support for modelling them. Consequently, we have extended the common metamodel with additional concepts to expose, for instance, the semantic and state information of components. An additional advantage is that these detailed information may also be used by execution environments in order to implement more adaptable and evolvable systems [14].

Notice that even when the AO-ADL metamodel put emphasis on additional concepts that were not explicitly incorporated to the visual notation metamodel, this is not a shortcoming of our visual notation since the graphical representation of these concepts can be easily provided using the visual notation. This notation is UML 2.0 compliant, and UML 2.0 provides behaviour diagrams for those purposes. Furthermore, the mappings presented in section 4.3 demonstrate the alignment between the metamodels of the AO-ADL and the visual notation.

In this section we describe the common concepts in the AO-ADL and the visual notation metamodels. Then, the metamodel for AO-ADL is provided in section 4.2,

while the metamodel for the visual notation is presented in section 4.3 (Figure 22). To reason about the metamodels the mapping of the two meta-models is described in section 4.4 (Table 1).

4.1 C&C common metamodel

In this section we describe the set of concepts that are common to both the AO-ADL and the visual notation metamodels. Notice that these are the minimal required concepts to describe aspect-oriented software architectures with our approach.

4.1.1 No specialized Components and Interfaces

A **Component** is the locus of computation and state [39]. A *component* is considered a modularity unit within a system architecture that has one or more *provided and/or required interfaces* (potentially exposed via ports). A component specifies a formal contract of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required interfaces. Its internals are hidden and inaccessible other than as provided by its interfaces. Such access constraints also apply to components playing the role of architectural aspects, i.e. those ones involved in a crosscutting collaboration with other components. If an architectural aspect needs to know any internal detail of a certain component, such a detail needs to be made available at one of its interfaces.

The meta-model is symmetric in the sense it does not define an explicit abstraction for an aspect. Both crosscutting and non-crosscutting concerns are represented by components. The distinction is made at the connector level, i.e. it is the way two or more components are composed that denote that a crosscutting composition is taking place. No new "aspectual" component interface is defined in the meta-model as we believe that "aspectual components" also offer services and expose events or attributes, like any other component.

Therefore, in our approach base components and aspectual components have identical characteristics:

- They are identified by a required unique name inside a particular architectural design.
- They are described by means of their ports. We consider two kinds of ports, which are the **provided** and **required interfaces** of components. The definition of these interfaces determines the different roles the components can play in a particular architectural design.
- They can interact with the rest of the components in the application, and consequently, can have different kinds of dependencies with the rest of the components in the application.

4.1.2 Connectors

The Connector is the architectural element for composition in the modelling of software architectures. Connectors are the building block used to model interactions among components and rules that govern those interactions [39]. Our position is that

connectors should provide support for the possible architecture-level crosscutting compositions. This support in AO-ADL and in the visual notation is discussed and specified in subsequent sections.

4.1.3 Base and Crosscutting Roles

In our approach connectors are basically formed by base and crosscutting or aspectual roles. These roles consist of two types of connector's interfaces, and define the role the connected components are playing in a crosscutting composition. A crosscutting or aspectual role defines which component is playing the role of "aspect" in the architecture decomposition, i.e. which component is encapsulating a crosscutting concern and needs to affect other architecture interfaces. A base role defines which components are playing the role of "component" as understood in traditional architectures.

4.1.4 Pointcut and Sequencing Specifications

Associated to a crosscutting or aspectual composition are the concepts of pointcut and sequencing. The pointcut specification is the description of the join points of interest that will be intercepted. These pointcuts are to be defined in terms of: (1) a base role of the connector, which can be associated to the provided or the required interface of a component, or (2) a composition between two or more components previously described in the architecture.

In addition, a sequencing operator is associated with each crosscutting or aspectual composition. It specifies when or how the aspectual component is affected the intercepted join points. Typical sequencing operators are before, after, and around though other operators could be used.

4.1.5 Quantification and Aspect Interaction

Our approach, both the AO-ADL and the visual notation, supports for specifying quantifications, i.e. describing in a single place which elements a certain aspectual component is affecting. It also provides elements for addressing aspect interactions. This is required when several aspectual components are affecting the same join point.

In the next subsections we describe how these concepts are represented in both the AO-ADL language and the visual notation.

4.2 AO-ADL Metamodel

The meta-model of AO-ADL is shown in Figure 13 where we can observe that the main architectural elements in AO-ADL are components and connectors. This meta-model is an evolution of the AO-ADL description in D37.

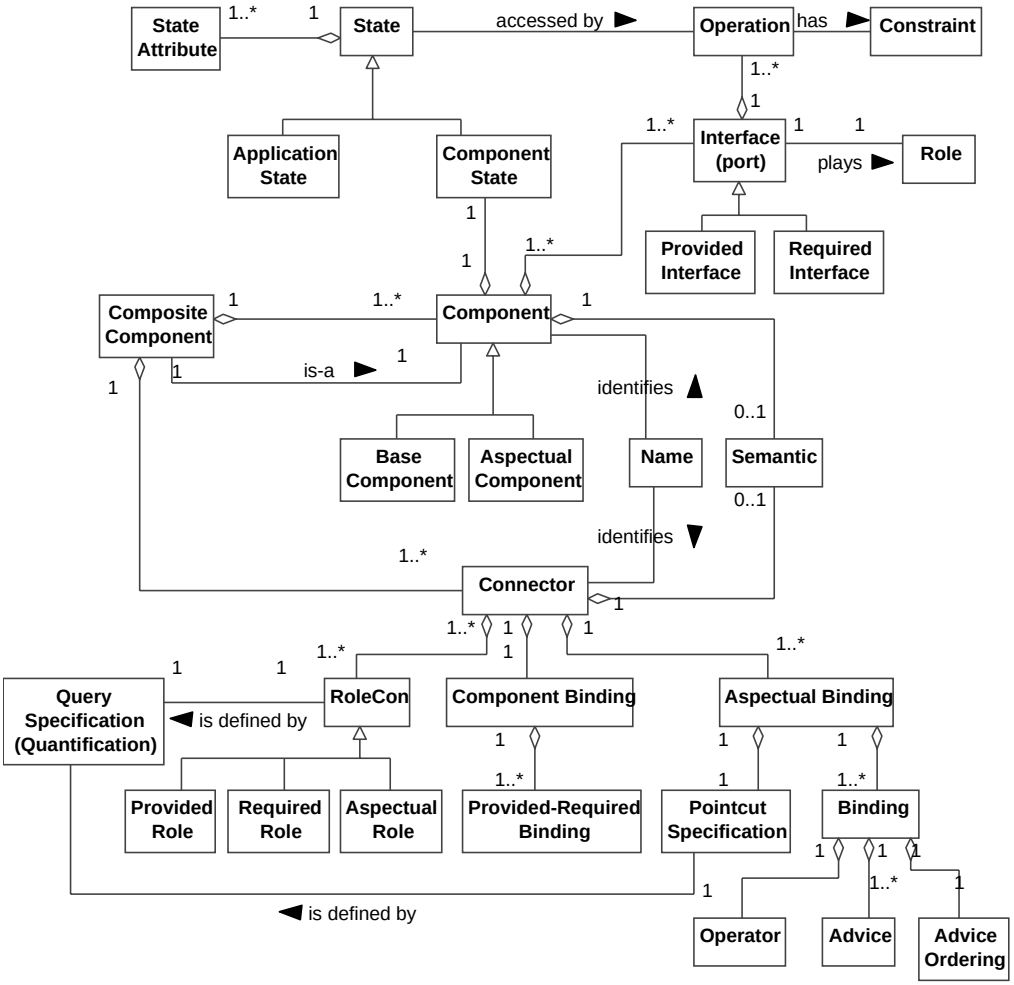

Figure 13. *AO ADL Metamodel for C&C Viewtype*

4.2.1 Components

As discussed above, we have chosen a symmetric approach for representing both components and aspects, without distinction, with a single architectural element. Therefore, in AO-ADL base components and aspectual components have identical characteristics. They both have a required unique name and are described by means of their ports (provided and required interfaces) as discussed in previous section.

In addition, the AO-ADL metamodel defines the following features of a component:

- They can both expose their *public state*.
- They can optionally expose their *semantics*, which is a high-level model of a component's behaviour. This behaviour is expressed in terms of the interaction protocol in the component's interfaces, reflecting the ordering of the occurrence between the reception of operations in provided interfaces and the sending of operations through required interfaces.

Note that as shown in Figure 13, in spite of using the same architectural entity, we offer the possibility of optionally identifying which are the base components and the aspectual components of an architecture design. This is useful in those cases where the software architect desires to explicitly express this role of the component during its specification.

The concept of **component state** is used here with the same meaning as in traditional IDLs of distributed systems. Thus, it represents the component's public state – e.g., the information that should be made persistent to be able to later restore the state of a component.

The concept of **application state** in AO-ADL identifies and models separately the data dependencies among both base and aspectual components. In the particular case of dependencies between aspectual components, the concept of application state allows to model, at the architectural level, certain types of interactions and dependencies among aspects. For example, consider the dependency between an authentication aspect and an access control aspect in the auction system. The first one obtains the name of the user during the authentication process. Later, the second one will need the identification of the user to check if s/he is allowed to do some actions in the application. The description of this information is external to any of the components described as part of the architecture of the application. Particular components only reference the attributes of the application state to indicate if they would generate this information (*outputState*) or if they would need this information (*inputState*).

A Composite Component in AO-ADL can be used to encapsulate a sub-architecture, or to define different levels of abstraction of a component. This kind of component is internally composed by a set of components and also connectors, which specify the interactions and relationships among them. Composite components play in AO-ADL the same role than representations in other ADLs, such as ACME.

Examples of part of the AO-ADL definition of components, interfaces and states are shown in Figure 14. In the auction system case study the user interacts with an auction as a buyer or a seller, among other roles that the user may have in the system. Thus, the AO-ADL description of the User component specifies the two roles – i.e. Buyer and Seller, played by the user by means of two required interfaces. These services are provided in the example by the Auction component. The state of the Auction component is also described in the example. Finally, the Security component is also specified which have a provided interface Log-on with a log operation on it.

```
<component name="User">
      <required-interface role="Buyer" name="Buy"/>
      <required-interface role="Seller" name="Sell" />
      …
</component>

<component name="Auction">
      <provided-interface role="BuyService" name="Buy"/>
      <provided-interface role="SellService" name="Sell"/>
      …
      <state name="AuctionState"/>
</component>

<component name="Security">
     <provided-interface role=" "/>
</component>

<interface name="Buy">
   <operation name="join"/>
   <operation name="bid"/>
</interface>

<interface name="Sell">
   <operation name="initiate"/>
   <operation name="activate"/>
   <operation name="close"/>
   <operation name="setReservePrice"/>
   <operation name="setMinimumIncrement"/>
   <operation name="setMinimumBidPrice"/>
   <operation name="setStartPeriod"/>
   <operation name="setDuration"/>
</interface>

<interface name="Log-on">
    <operation name="log"/>
</interface>

<state name="AuctionState">
      <attribute name="start-date"/>
      <attribute name="end-date"/>
      <attribute name="highest-bid"/>
      <attribute name="reserve-price"/>
</state>
```

Figure 14. Example of interfaces, states and components in AO-ADL

4.2.2 Connectors

The Connector is the architectural element for composition in AO-ADL. There are two main differences between the representation of connectors in traditional ADLs, and the representation of connectors in AO-ADL. One difference is the distinction between component bindings (interactions among base components) and aspect bindings (interactions between aspectual and base components). This means that components referenced in the aspect bindings are playing the role of an aspectual component in the specified interaction. The other difference is how the roles of

connectors are specified. In AO-ADL the role of connectors are defined specifying a query that identifies the component(s) that may be connected to that role. This means that instead of associating a particular interface to a role, a role may be described using quantifications by a query describing "any component playing a particular role", or "any component containing a set of particular operations in one of its provided interfaces". Additionally, and similarly to a component, a connector can also expose its semantics. In this case, the semantics will expose a high-level model of the interaction among components.

```
<connector name="UserAuctionSecurity">
<provided-role name="UserRole">
    <role-specification>
        component-name="User" and (interface-role="Sell" or interface-role="Buy")
    </role-specification>
</provided-role>
<required-role name="AuctionRole">
    <role-specification>
        component-name="Auction" and (interface-role="Sell" or interface-role="Buy")
    </role-specification>
</required-role>
<aspectual-role name="SecurityRole">
    <role-specification>
        component-name="Security" and interface-role="Log-on"
    </role-specification>
</aspectual-role>
<componentBindings>
    <binding name="UserAuctionB">
        <provided-role name="CustomerRole">
        <required-role name="AuctionRole">
    </binding>
</componentBindings>
<aspectBindings>
    <aspectual-binding name="security">
        <pointcut-specification>
            binding-name = "UserAuctionB" and operation-name="*"
        </pointcut-specification>
        <binding operator="before*" constraint="is_not_registered">
            <aspectual-component name="Security" role="Log-on" advice-label="log"/>
        </binding>
    </aspectual-binding>
</aspectBindings>
</connector>
```

Figure 15. *Example of a Connector in AO-ADL*

Figure 15 shows the UserAuctionSecurity connector for the auction system description in our AO-ADL. It describes the interaction between the User component (specified in the provided-role clause of the connector) and the Auction component (specified in the required-role clause of the connector). Since the interactions among these components have to be secure, a Security component is also connected to this connector (specified in the aspectual-role clause of the connector). This component behaves as an aspectual component and, therefore, the information to bind Security

with User and Auction differs from the information to bind User and Auction, as described below.

4.2.3 Role Specification

An important novelty of AO-ADL connectors is that the connector roles can be defined specifying a query that identifies the component(s) that may be connected to that role. This means that instead of always associating a particular interface to a connector role, this role may be described by a query matching 'any component playing a particular role', or 'any component containing a set of particular operations in one of its provided interfaces', or 'any component having a particular attribute as part of its public state'. This is particularly useful for the specification of aspectual bindings (described below). Since 'aspectual' components encapsulate crosscutting concerns, this means that the usual situation would be one in which the same 'aspectual' component affects different connections between 'base' components. However, without using queries to specify roles, it is not possible to avoid that the same aspectual composition rules would appear replicated through different connectors of the architecture.

An example of the replication of the same aspectual composition rules in different connectors occurs when we add, for instance, a new Encryption component to the architecture of the Auction System. Assuming that, according to the security requirements of our system, all the interactions between components in the Auction system would need to be encrypted, the Encryption component would be included as an 'aspectual' component that affects the interactions between the User and the Auction components as well as between any other two components in the system. Additionally, these connections would be specified by different connectors describing the base communication between components. Consequently, an aspectual-role clause specifying
the encryption 'aspectual' behaviour and an aspectual-binding clause specifying how encryption is bound to the interaction among 'base' components have to be replicated in the different connectors.

The above can be easily avoided using queries to specify roles. The main idea is to define an 'aspectual' connector which mainly focuses on the specification of the aspectual compositions, using wildcards and binary operators to specify roles. For the Encryption component previously mentioned, the new connector would specify the following: 'For all the interactions between any source and target component in the Auction System, a component playing the role EncryptionRole in one of its provided interfaces is attached'.

Furthermore, the connectors' roles establish the scope of the join points that can be referenced by the connector's pointcuts (described below). The idea is that we can easily represent different weaving scenarios with connectors by using queries on the definition of the connectors' roles. For instance, at the architectural level typical join points would be: the 'reception of an operation defined in the provided interface of the component', where the source component is omitted from the pointcut specification; the 'sending of an operation defined in the required interface of a component', where the target component is omitted from the pointcut specification; the 'interaction

among two components', where both the source and the target component are identified; the 'occurrence of a particular component's state or application's state', what is called a state-based join point, etc. The only requirement is that we need to expose, in the specification of the connector roles, all the information that we will then refer during the specification of the pointcuts.

4.2.4 Base component composition

The specification of the base component bindings inside the connector describes the connections among base components. For instance, the *componentBinding* clause in the connector shown in Figure 15 describes the interaction between the `User` and the `Auction` components by connecting the corresponding provided and required roles previously described in the connector.

4.2.5 Aspectual component composition

The specification of the aspect bindings inside the connector describes the connections between aspectual components and base components. For instance, in Figure 15 the `Security` component affects the interaction among the `User` and the `Auction` components.

Concretely, in an aspectual binding the following information is provided:

- *Pointcut Specification*. The description of the pointcuts identifying the join points that will be intercepted. These pointcuts are to be defined in terms of: (1) a provided role of the connector; (2) a required role of the connector, or (3) a composition binding between a provided role and a required role previously described in the same connector. In the example of Figure 15 the join points captured by the pointcut specification are all the operations in the interaction between the `User` and the `Auction` component (specified referring to the binding name `UserAuctionB`). Other possibilities would be to intercept the reception of a message in one of the provided interfaces of the `Auction` component, or the sending of a message through one of the required interfaces of the `User` component. This can be expressed by omitting in the pointcut specification the other party in the communication.
- *Binding*. A different binding is defined for each kind of advice (before, after, before*, after*, concurrent-with…) to be included in the definition of the aspect bindings. For instance, the set of aspectual components being composed "before" a join point may be different from the set of aspectual components composed "after" a join point.
- *Operator*. The kind of advice is specified in the operator. AO-ADL operators are *before*, *after, before*, after** and *concurrent-with*. They take the form: X before/before* Y, X after/after* Y and X concurrent-with Y, where X is an advice and Y is a join point. In Figure 15, the operator *before** indicates that the `Security` component is injected before the interaction among the User and the Auction component.

- *Constraint*. A constraint restricting the injection of aspects under certain circumstances. Constraints can be expressed as pre-conditions, post-conditions or invariants:
    - [pre-condition] X, the [pre-condition] must be satisfied before X
    - X [post-condition], the [post-condition] must be satisfied after X
    - [invariant] X, the [invariant] must be satisfied during X

  In Figure 15 the "`is_not_registered`" pre-condition is specified to avoid the execution of the corresponding `Security` advice once the user has already been logged on to the system.
- *Advice*. For each kind of advice, the list of aspectual components to be composed is specified. This specifies the name of the aspectual component, and the name of an operation in one of its provided interfaces. The behaviour specified by this operation is the behaviour to be composed at the matched join points. In Figure 15, the `log` method of the provided interface with role name `log-on` of the `Security` component is specified as the advice to be executed.
- *Advice Ordering*. Several aspects being composed at the same join point need to be appropriately ordered. This information is provided for each aspect binding, since the ordering among the same set of aspects may be different depending on the context in which they are composed. The operators before, after and concurrent previously described are also used to indicate the order of execution of advice.

4.3 AO Visual Notation Meta-model

The new symmetric AO Visual Notation replaces the old asymmetric Visual Notation, described in the deliverable D37. The key differences between the AO-ADL metamodel (Figure 11) and previous Graphical Notation (GN) metamodel were that: (1) the ADL meta-model contains more elements than the Graphical Notation meta-model, (2) components could have N-N relationships in the ADL, while only two components could be involved in the GN relationships, and (3) the notions of component state and application state were explicit architectural elements in the ADL metamodel.

In the new symmetric version of the GN metamodel, the conceptual differences described in (2) no longer exist; the solution for (2) is that now we have the notion of connector in the graphical notation (as defined in UML 2), which naturally allows for multiple components engaging in a certain collaboration (modularized in the connector). The divergence in the number and nature of meta-concepts (Difference (1)) has been significantly reduced due to both the new symmetric property of the GN metamodel and the explicit support for conventional connectors (defined in UML 2) and aspectual connectors. The next sections respective discuss the problems we have identified in antecessor asymmetric notations (Section 4.3.1), and the elements of the new symmetric notation (Section 4.3.2).

4.3.1 Issues on the Visual Representation of Architectural Aspects

This section reports some problems we have faced on the application of existing asymmetric AO architectural notations in several projects. The goal in these projects was to achieve visual representations of AO architectures that were simultaneously: (i) both easy to understand the occurrence of architectural aspects and intuitive to

grasp its different forms of composition with other architectural components, (ii) straightforwardly translating aspect-oriented concepts commonly supported by multiple AO architecture description languages (ADLs), (iii) undertaking appropriate high-level modularity measurements, and (iv) smoothly mapping requirements-level aspects to architecture-level aspects.

The HW Architecture. The Health Watcher (HW) system is a real-life Web-based information system [32] that supports the registration of complaints to the health public system. It will be used as our running example throughout this paper. Figure 16 illustrates a partial graphical representation of the HW architecture, which is based on a set of components mainly realizing an instance of the layered style. It is composed of seven architectural components; three of them are layers: (i) the GUI (Graphical User Interface) component provides a Web interface for the system, (ii) the Business component defines the business elements and rules, and (iii) the Data component addresses the data management by storing the information manipulated by the system. The aspect-oriented HW architecture also contains four architectural aspects: Persistence, Distribution, Concurrency and Error Handling (EH). For instance, the Distribution aspectual component externalizes the system services at the server side and supports their distribution to the clients. Figure 16 shows an asymmetric representation of the HW architecture, based on the AOGA language, which we have chosen to illustrate the limitations of existing visual notations and respective meta-models [20]. In AOGA, aspects are aspectual components that are represented by UML components with a diamond in the top.

Problem 1: Expressiveness Impairments. Because AOGA and other asymmetric notations create a specific symbol to represent an aspect, it is not possible to smoothly use the same representation for a component that plays the role of an aspect in certain architecture contexts, but not in others. This expressiveness bottleneck also hinders reuse of component representations across different projects, when the target component is an aspect in the original architectural design, but not in the second, or vice-versa. Such a dichotomised notation gives the wrong impression that a certain aspectual component cannot assume different roles defined by different styles [6], for example. For instance, in one of the HW releases [17], the Distribution component played the role of being both an aspect and a layer. With an asymmetric architectural notation, it would not be obvious to notice that Distribution was free to take part in other collaborations and play different architecture stylistic roles, i.e. “being a layer” in this particular case.

Problem 2: Inability to Represent Heterogeneous Aspectual Compositions. We have also observed that in existing approaches, there is no much visual means to graphically specify and distinguish different forms of collaborations between non-aspectual and aspectual components. For example, there is no possibility of clearly communicating the sequencing of a crosscutting composition; i.e. the order (e.g. before, after or around) in which the aspect computation will affect the base computation. In general, architects cannot easily check the composition sequencing at a glance, because they are only supported in the expanded view of component interfaces and, even worst, it is textually declared together with the crosscutting service. Figure 16 illustrates this problem in the context of the TransactionControl interface. Also, in asymmetric notations the sequencing is typically associated with a

service in a certain aspect interface, which in turn also reduces the component specification reusability. This also might cause problems when the same service is involved in different aspectual compositions, thereby affecting the target join points in distinct orders. Finally, some of these notations (AOGA is an example) typically use the same symbol to represent different composition mechanisms, even though they have different architecturally-relevant semantics. For example, crosscutting interfaces and relationships are used to denote both behaviour-based (pointcut-advice-like) and structural compositions (such as inter-type declarations or structure merges).

Problem 3: Limited Scalability. In the projects [17, 30] where we have used existing asymmetric architecture notations, a number of scalability issues were detected. Some examples are discussed in the following. First, they do not scale when a crosscutting interface affects several join points in the architecture, even via the same aspect interface. Figure 16 illustrates this problem for the TimestampControl interface that affects all interfaces of the Data component. Actually, this is a generic problem in AO design notations as crosscutting is a kind of relationship that often implies a plethora of links between the aspect and the affected elements. In other words, such notations suffer from not having visual resources to quantify such links. Second, they neither support graphical capabilities for representing certain inter-aspect dependencies. Also, they are typically textual and alternatively based on the use of stereotypes. Even though AOGA has a stereotype dedicated for annotating aspect precedence, the granularity is aspect-aspect level and cannot be tailored to certain compositions or particular architectural join points, such as a particular service.

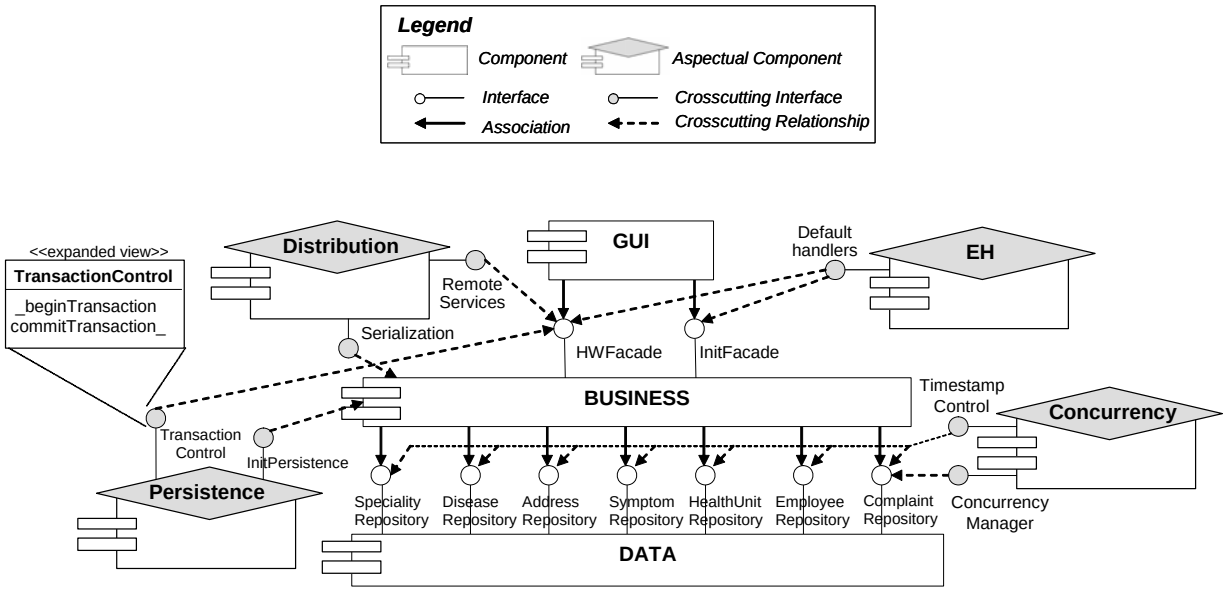

Figure 16. *Health Watcher Architectural Design with AOGA*

Problem 4: Hindering Architecture Modularity Assessment. A direct consequence of problems 2 and 3 is that architects are not able to effectively assess modularity properties of an AO architecture design. Differently from current practice in UML 2, where different kinds of connectors (delegators, dependencies, or assemblies) are

supported by the notation and the underpinning meta-model, existing AO architecture notations are not yet mature to serve as expressive artefacts to support early modularity assessment. We have experienced this problem in architectural assessment of 3 case studies [17, 30]. For instance, because the differences in the representation of certain architectural aspect compositions are not made explicit (problem 3), computation of specific architectural metrics such as afferent and efferent couplings [17, 31] is impaired.

Problem 5: From Requirements to Architecture. From our experience defining and applying the mapping process and guidelines presented in D63 to relate AO requirements specified using RDL) and AO architecture (specified using AO-ADL), we know that there is not always a 1-to-1 mapping between crosscutting concerns identified at the requirements and architectural level. Instead, the same crosscutting concerns can be mapped either to a non-aspectual or to an aspectual component, or even to an architectural decision depending on the application context. An example is Distribution that, as mentioned before, can play the role of being an aspect or a layer depending on the HW release. The problem is that asymmetric visual notations provide different abstractions to represent components and aspects and, as a consequence, force us to make the decision of mapping requirements to a component or to an aspect as part of the mapping process itself. This is neither necessary, since the decision can be postponed until a refined version of the mapped architecture, nor desirable, since emerging AO requirement proposals are symmetric and do not do such a distinction.

4.3.2 The Symmetric Visual Notation

The following sections present our new visual notation in terms of: (i) its meta-model (Section 4.3.3) with the key architectural abstractions currently being supported, and (ii) a set of graphical elements (Sections 4.3.4 to 4.3.8) to allow the representation of recurring aspectual compositions in component-and-connector models. Both meta-model abstractions and graphical elements were defined to address the limitations discussed in Section 4.2.1.

The proposed notation is an evolution of our previous work, rather than a totally new approach. It has being systematically derived from: (i) our previous systematic analysis of four modelling approaches, namely TranSAT, PCS Framework, AOGA, and CAM of DAOP-ADL, (ii) a primitive visual notation defined for an AO extension to the ACME language [15], (iii) a analysis of abstractions consistently manifesting across existing ADLs, such as AO-ADL (Section 4.2), AspectualACME [15], DAOP-ADL [27], and others [3]. The derivation of our current approach involved the "transformation" of a previous asymmetric notation, unified from the 4 approaches mentioned above in (i), into a new symmetric notation. Hence, Section 4.3.9 evaluates the benefits and drawbacks obtained in this transformation process.

4.3.3 Meta-Model

Figure 17 presents our notation meta-model. Our visual notation extends the set of architecturally-relevant abstractions and respective graphical elements of UML 2, such as services, components, interfaces, and connectors.

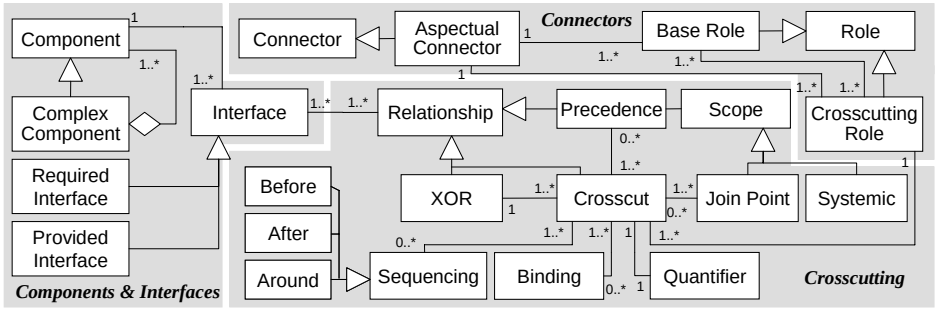

Figure 17. *AO Visual Notation for Metamodel for C&C Viewtype*

In fact, we use UML 2 as the basis without modifications to its existing visual elements. As a result, existing UML architectural models can be straightforwardly refactored to accommodate architectural aspects. It is not the goal of this document to discuss the integration of our notation's meta-model and UML 2 meta-model. Also, for the sake of simplicity we have omitted from the meta-model (Figure 17) some conventional architectural concepts in UML 2, such as ports. From this integration perspective, the discussion here is limited to evaluate on why specific UML connectors, with associated graphical representations, are not appropriate to represent a crosscutting composition. The meta-model focuses on the definition of new aspect-oriented concepts and their relations. The meta-model (Figure 17) subsumes 3 main categories of elements: (i) components and interfaces (Section 4.3.4); (ii) aspectual connectors (Section 4.2.5); and (iii) crosscutting relationships (Section 4.3.6).

4.3.4 No Specialized Components and Interfaces

As described before, a *component* is considered a modularity unit within a system architecture that has one or more *provided and/or required interfaces* (potentially exposed via ports). Both crosscutting and non-crosscutting concerns are represented by components. The distinction is made at the connector level (Section 4.3.5), and not new "aspectual" component interface is defined in the meta-model.

4.3.5 Aspectual Connectors

Our position is that a minimum of new abstractions and respective graphical elements should be supported by the visual notation. The reason is that architectural description languages, whether textual or visual, were conceived with the goal of being agnostic to specific architectural styles, such as layered and pub-sub architectures, or even object-oriented architectures. Hence, architecture design languages should be kept as small as possible, while accommodating support for representing architecture-level aspects.

In fact, UML 2 and other component-and-connector notations do not create specific graphical elements to denote that a certain component is a layer, a publisher, or an aspect. This visual distinction would be very counter-productive in large architecture designs since it is common to find single components playing multiple roles defined by different architectural styles. As discussed in Section 4.3.1, the Distribution component (Figure 16) is an example of this case in the HW architecture. Also, based on our experience, it is becoming increasingly clear that the key difference of an aspect-oriented architecting style is on the composition semantics [15, 3].

Hence, our position is that the visual representation of AO software architectures should provide support for the possible architecture-level crosscutting compositions observed in our case studies (Section 4.3.1). This should be rooted at the traditional notion of connectors (and attachments) of the software architecture discipline. The reason is that connectors are the locus of composition in architectural design. As a result, our visual notation supports the notion of aspectual connectors (Figure 17). This emphasis on aspectual connectors is not currently supported by the investigated visual notations for aspect-oriented software architectures. However, it is consistently becoming a common practice in recent AO textual description languages [15, 16]. Before describing how we represent aspectual connectors, we discuss first why conventional connector types, available in UML 2, are not appropriate to capture the notion of crosscutting compositions.

Connectors can define a wide range of composition styles, ranging from simple dependencies to complex collaboration protocols. For example, UML 2 defines three specialized connectors for interlinking components, namely dependencies, assemblies, and delegators. Different visual elements are associated with each of them. Crosscutting compositions cannot obviously be represented by dependencies; hence, we concentrate our discussions on assemblies and delegators.

Assembly vs. Aspectual Connectors. We cannot rely on assembly connectors to represent crosscutting compositions because the former implies on a simple relation between a required port and a provide port. In addition, an assembly connector must only be defined from a required interface (or port) to a provided interface (or port), which violates a typical composition property of crosscutting collaborations [15, 20]: an aspectual component and affected components can be linked through their both provided interfaces.

Delegation vs. Aspectual Connectors. In addition, we cannot reuse the notion of delegation connectors. They have a number of modelling constraints that do not match the requirements for aspectual compositions at the architectural level. The main problem is that they subsume a "forwarding" semantics. A delegation connector is a connector that links the external contract of a component (as specified by its ports) to the internal realization of that behaviour by the component's parts. Besides, a delegation connector must only be defined between used Interfaces or Ports of the same kind (e.g., between two provided interfaces or between two required interfaces). Aspectual compositions involve the identification of several join points (e.g. affected interfaces or services) to be connected to the component encapsulating a "crosscutting concern". They also specify composition operators on when or how those points are

being connected with other services provided by components encapsulating a "crosscutting concern".

Finally, architecture-level crosscutting compositions require that aspectual connectors might actuate directly over other connectors. Most architecture representation languages do not grant this property to connectors, which reinforces the need for a specialized type of connector. Figure 18 shows our visual representation for aspectual connectors. The use of the stereotype is optional and not motivated. The aspectual connector is a component-like graphical notation with elements to specify the "crosscutting collaboration" amongst involved architectural elements. A simpler notation is available in case connector internals are not relevant.

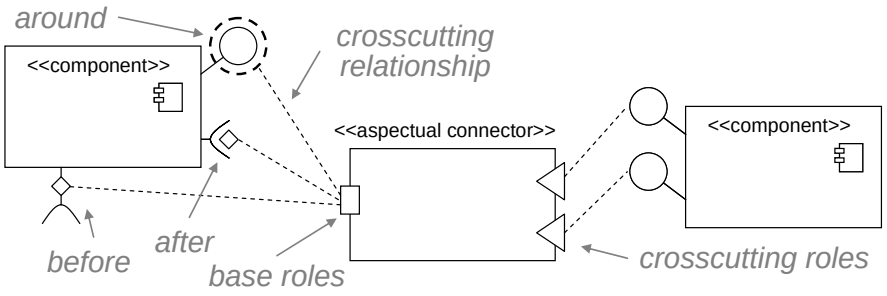

Figure 18. *Notation for Aspectual Connectors*

4.3.6 Base and Crosscutting Roles

Aspectual connectors (Figure 18) are basically formed by base and crosscutting roles. These roles consist of two types of connector's interfaces, and define the role the connected components are playing in a crosscutting composition. A crosscutting role defines which component is playing the role of "aspect" in the architecture decomposition, i.e. which component is encapsulating a crosscutting concern and needs to affect other architecture interfaces. Crosscutting roles are represented by triangles "cutting across" the connector boundaries. Base roles are associated with different join points affected by the components associated with the crosscutting roles. They are represented by small rectangles in the opposite extreme of an aspectual connector (Figure 18).

Crosscutting relationships define how the connectors and components are attached. In another words, they are equivalent to attachments in ADLs (ACME and xADL), and their visual representation is a dashed arrow. The arrows associate crosscutting or base roles with component interfaces. In the presence of multiple base and crosscutting roles, the dashed arrows can also cut across the aspectual connector representation in order to show how base and crosscutting roles are interlinked. This situation is illustrated in Figure 19, which is a symmetric visual representation of the HW system shown in Figure 16. The DataControl connector has multiple roles which are bound through the dashed arrows.

4.3.7 Pointcut and Sequencing Specifications

The set of join points of interest (i.e. pointcuts) in a certain crosscutting composition are conventionally indicated by visual (and sometimes, textual) elements associated with a crosscutting relationship. When a component interface is touched by an arrow, it means that one or more of the interface services are affected by an aspectual connector. If a precise indication of which service(s) are being connected, the name of the service(s) is attached to the arrow using stereotypes. Figure 19 illustrates an example of specific services being bound through the Synchronization connector. In addition, whenever it is required, a sequencing operator can be associated with a crosscutting relationship. It specifies when or how the connector is affecting the service(s). By now, the notation includes graphical elements for three sequencing operators: before, after, and around (Figure 18). However, other operators could be used. Some concrete examples for the HW architecture are presented in Figure 19.

4.3.8 Quantification and Aspect Interaction

Our visual notation provides support for specifying quantifications, i.e. describing in a single place which elements a certain aspectual connector is affecting. The goal is to overcome the problem 3 discussed in Section 4.3.1, i.e. visually support quantification and reduce the number of arrows for crosscutting relationships. An example is presented in Figure 19: the Synchronization connector affects all the Data interfaces. The notation used is: (i) a set of multiple grouped dashed arrows pointing to the direction of the affected elements, plus (ii) a label with an expression indicating more precisely a property that matches the affected elements. Figure 19 shows that the Synchronization connector is affecting all interfaces of the Data component. We have being defining also specific visual elements to represent certain recurring quantifications that we have observed in our case studies (Section 4.3.1), such as: “all the provided interfaces in…” and “all the required interfaces in…”. Due to space limitation, we cannot present them all here.

The visual notation also provides elements for addressing aspect interactions (problem 3 – Section 4.3.1). Figure 19 illustrates a scenario where we specify that the same aspectual connector is affecting (in an after fashion) the same join point, i.e. the interface HWFacade. As a result, two diamonds are on the top of this interface. However, priority is given to the element that is associated with the diamond closer to the interface circle. It means that TransactionControl has precedence over InitPersistence. The same semantics applies to before and around operators; in the case of two or more around operators actuating over the same join points, inner circles have priority over the enclosing ones. Graphical elements are also used to represent XOR and OR relationships.

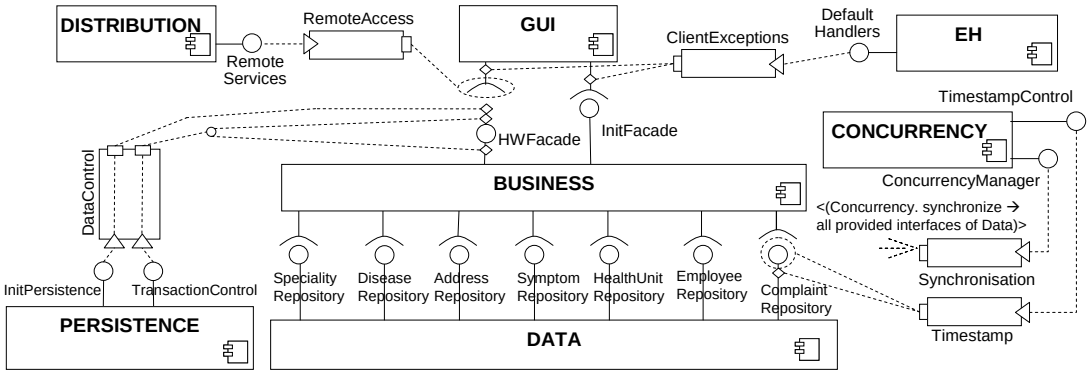

Figure 19. *Symmetric Representation of the Health Watcher Architecture*

4.3.9 Evaluation

This section briefly summarizes evaluation of the new visual notation undertook in the context of two case studies: (i) the complete specification of the HW architecture (Section 4.3.1), and (ii) the definition of an online auction system architecture [7] based on a AO requirements description. In particular, we have tried to observe to what extent the visual notation addressed the challenges discussed in Section 4.3.1.

First, the expressiveness problems were solved since we do not have a separate visual element for representing aspects (problem 1). Moreover, it is still straightforward to identify the set of components playing the role of aspects in the architectural design: it consists of all elements bound to crosscutting roles (the triangles in the aspectual connectors). The new visual notation also allows for more intuitive and clear representations of different kinds of aspectual compositions (problem 2). For instance, the symbols used for before, after, and around have demonstrated to be a nice addition both for communication and measurement purposes. The sequencing operators can be often inferred from use cases and/or AO requirements documents. They are useful to distinguish different forms of coupling early in the design process, thereby facilitating application of AO architecture metrics (e.g. [30]).

It is true that some points in the architecture model might aggregate a number of visual elements, such as join points that are shared by multiple aspectual connectors. For example, like the two interfaces between GUI and Business layers (Figure 19). However, it causes also a desirable effect: the architects, designers, and programmers should pay special attention in this part of the architecture since this is a point where multiple “aspects” interact. Visual means to express quantification were very useful in the HW architecture, where four cases of broadly-scoped aspectual connectors were identified. They were associated with synchronization, persistence, and serialization (distribution), and generic error handling issues. Hence, scalability-related impairments have been substantially reduced (problem 3). When more complex aspectual connectors were required, we have exploited the resource of internal representations available in UML.

Even though the resulting visual language is much richer than the original asymmetric notation, a number of simplifications were also achieved. The notation meta-model no longer has abstractions and visual elements dedicated for aspects, aspect interfaces (i.e. crosscutting interfaces in the aspectual components), advice, and inter-type declarations. These elements are also present in almost all the asymmetric notations we have analyzed. In addition, because we support the specification of multiple forms of crosscutting compositions, it also facilitates the transition of architectural design to detailed design. It is easier to identify which slice of the component boundaries is likely to be translated to a design or programming aspect.

Finally, we have also tried to analyze whether benefits or drawbacks were obtained from viewpoint of requirements-architecture transitions (problem 5). The symmetric visual notation is part of a wider research that targets the development of an end-to-end AO methodology. It has the objective of defining a single approach that, starting from aspect-oriented requirements, results in an aspect-oriented architecture specification. The Auction System case study is used in [7] in order to illustrate the integrated approach. We have omitted the requirements and architectural specifications and more details can be found in [7].

A typical concern in the auction system case study is security, with a requirement specifying that "*Users have to log on to the auction system for each session*". Following a symmetric decomposition model, security, user and auction are modelled at the requirements level using the same element (concern, viewpoint, goal). The requirement used as example states that the security concern is related to the interactions among the user and the auction concerns, modelling a user that needs to be authenticated before buying and selling in an auction. During the mapping from requirements to architecture these concerns are mapped to components in the visual notation. Figure 19 shows the User, the Auction and the Security components. The 'log-on' verb in the requirement of the security concern is mapped to an operation of a provided interface of the Security component.

In the analysis of the Auction System case study in [7] a crosscutting influence has been identified between security and the interaction among users and auctions, both at the requirements and at the architecture levels. Notice, however, that using the symmetric visual notation, there is not impediment to use Security as a non-aspectual component in other architectures. Thus, the Security component can be composed either as a non-aspectual or as an aspectual component with no difference in its component specification.

The decision of the role played by a component is taken during the specification of the connectors. Concretely, in the UserAuctionSecurity connector in Figure 20, the User and Auction components are connected to base roles of the connector, participating in the interaction as non-aspectual components, and the Security component is connected to a crosscutting role of the connector, participating in the interaction as an aspectual component. Notice that the crosscutting behaviour modelled by the Security component can be any operation defined as part of its provided interface (log-on interface in Figure 20). The kind of binding (called *sequencing* in the previous sections), 'before' in this example, is also represented in the visual notation, as shown in Figure 20.

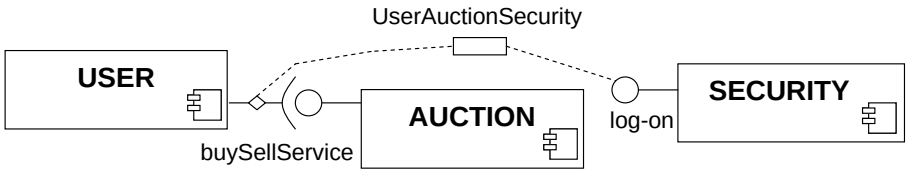

Figure 20. *Mapping of the security non-functional requirement to the symmetric visual notation*

As a next step, we are planning to enrich the visual notation with elements to express structural aspect-oriented compositions. In fact, this is a major limitation that we have identified in the current visual notation. In an industrial-strength case study [5], we have observed that more structural composition operators, such as merge and unification are also required in architecture specifications. We have being working in the definition of new operators as extensions to the xADL language [5], but not reflected much about visual representation for such operators. However, we have learned that the connector abstraction is potentially not the best abstraction to capture such structural compositions, as connectors have been historically explored for behavior-dependent architecture compositions.

4.4 Aligning the metamodels

So far we have made a distinction between a textual and visual metamodel of the C&C view. However, as we have noted before, a metamodel can be expressed in different notations but it is itself agnostic to the type of representation. As such, it does not seem to be relevant to have two distinct metamodels for the same view. In fact this would mean that we would then have two different languages.

To provide one metamodel for the different representations we need to align both metamodels. For this we will consider the relations among the elements in both metamodels.

Different from the set of mappings involving the AOSD-Europe Requirements Description Language (RDL) and the ADL metamodels, most of the mappings between the concepts of the ADL and the Graphical Notation (GN) metamodel are one-to-one mappings. Our practical investigation has resulted in the following initial mappings between the elements of these languages (see Table 1). The only ADL elements that do not have a counterpart (i.e. no mapping) in the GN metamodel are: role and semantic. While role is a concept necessarily always attached to interfaces, the use of semantic is always optional. In addition, the following mappings do not entail an one-to-one relationship: (1) the Component State in the ADL is mapped to a number of attributes in the GN component definition, (2) the Application State is mapped to a component that makes the global state variables available through a public interface to all the other components in the application architecture.

AO ADL Metamodel Elements	**Correspondence to the Graphical Notation Metamodel Elements**	**Type**
Component	Component	1 to 1
Component Name	Component Name	1 to 1
Interface	Interface	1 to 1
Provided Interface	Provided Interface	1 to 1
Required Interface	Required Interface	1 to 1
Composite Component	Composite Component	1 to 1
Operation	Interface Operation	1 to 1
Role	Provided Interface Name	1 to 1
Component Semantic	-	No
State	See below.	Partial
Component State	Part of Provided Interface (Attributes)	Partial
Application State	Component, Provided Interface, N Assembly Connectors	1 to N
State Attribute	Attribute in Provided Interface	1 to 1
Base Component	Component	1 to 1
Connector	Assembly Connector, Delegator Connector or Aspectual Connector ?????? how to decide??? Specify in the heuristics	1 to 1
Connector Name	Connector Name	1 to 1
Aspectual Component	Component	1 to 1
Component Binding		
Provided-Required Binding	Assembly Connector	1 to 1
Aspectual Binding	Crosscutting Relationship,	1 to 1
Poincut Specification	Stereotype in the Crosscutting Relationship	1 to 1
Binding		
Operator	Ordering Operator	1 to 1
Advice	Operation in Provided Interface and Crosscutting Role	1 to 2
Advice Ordering	Precedence	1 to 1
Connector Role	-	1 to 1
Required Role	Role (Aspectual Connector) or Port (Delegator Connector)	1 to 1
Provided Role	Role (Aspectual Connector) or Port (Delegator Connector)	1 to 1
Query Specification	Stereotype in the Crosscutting Relationship	1 to 1

Table 1: Mapping of the ADL meta-model elements to VN meta-model elements

5. Meta-Model of Deployment Viewtype

In the deployment viewtype the allocation of elements in the C&C and/or module view to non-software entities are shown. Non-software entities are in principle the hardware nodes on which the software will run, software development teams, or the development's environment file's structure. In principle there are different allocation views (or styles) possible but if we consider hardware nodes then we speak of deployment viewtype.

In the module view and the component and connector views we have introduced notations to represent aspects. The question arises of course whether we have also aspects in the deployment view. On first sight this does not seem realistic, because this would mean, for example, that a hardware node is scattered over different nodes. As such we do not think, at least not consider, *aspectual nodes,* whatever this would imply.

Thus, for representing the deployment view we will "just" utilize the deployment diagrams of UML. In UML2, the Deployments package specifies a set of constructs that can be used to define the execution architecture of systems that represent the assignment of software artifacts to nodes. Nodes are connected through communication paths to create network systems of arbitrary complexity. Nodes are typically defined in a nested manner, and represent either hardware devices or software execution environments. Artifacts represent concrete elements in the physical world that are the result of a development process [25].

A Deployment Diagram models the run-time architecture of a system. It shows the configuration of the hardware elements (nodes) and shows how software elements and artifacts are mapped onto those nodes. A node is either a hardware or software element. It is shown as a three-dimensional box shape. A node can contain other elements, such as components or artifacts. An example deployment diagram is shown in the following figure.

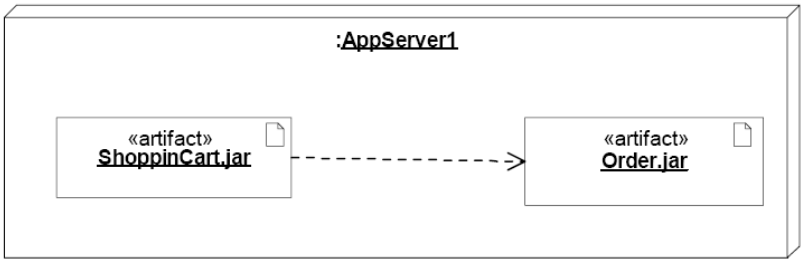

Figure 21. *Example Deployment Diagram [UML2]*

The metamodel for deployment view can be reused from the UML2. This is illustrated in Figure 22. Artifacts are mapped to deployment targets.

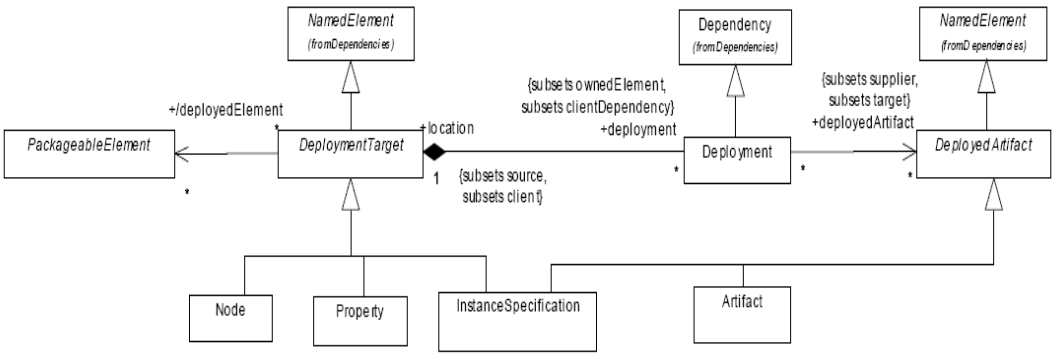

Figure 22. *Deployment relations in UML2 (metamodel)* (Figure 10.4 in [25])

The problem that is tackled is thus mapping elements of the module view and component and connector view to ordinary UML2 elements in the deployment view. If we use UML2 as discussed above we would not need to provide any new enhancements and we are in a sense ready. If we would also like to express it textually then this would simply mean to represent the deployment view in the ADL.

In the following subsection we first discuss the relationship between the elements of the component and connector view elements of the deployment view and then the relationship between elements of the module view and elements in the deployment view.

5.1 Relationship between C&C view elements and Deployment view elements

In order to use the UML 2.0 deployment view metamodel for our approach as it is, we need to check that it provides support to represent:

1) *Deployment of "aspects"*. Considering that our approach follows a symmetric decomposition model and that, as consequence, aspectual behaviors are modeled as components, there is no problem to model an "artifact" or "deployedArtifact" whose model element is an aspect. This is shown in Figure 23 for the User, Auction and Security components in the Auction System (see Figure 23). The "manifestation" relationship (<<manifest>> stereotype) indicates the set of model elements that are manifested in the Artifact. That is, these model elements are utilized in the construction (or generation) of the artifact [25].

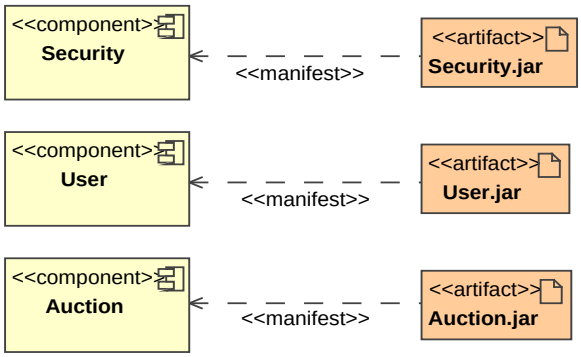

Figure 23. *Base and Aspectual Components Deployment*

2) A*spect "deployment information"*. Even when the representations of base and aspectual behavior are the same (i.e. they are modeled as components), the instantiation of aspects differ from the instantiation of components mostly in all AO approaches. This means that aspect-oriented execution environments needs to know "instantiation" information –i.e. how to instantiate/deploy an aspect. This information is mainly related to the number of instances of an aspect that will be created in a particular system. This is needed basically because aspects are not explicitly created by the application. Instead, it is usually the runtime execution environment the responsible of implicitly instantiated the aspects accordingly to the instantiation information provided during the application deployment.

Though the type of information may be different, the aspect instantiation mode may be compared with the information that is provided by the deployment descriptors of distributed component platforms such as CCM/CORBA and EJB/J2EE. Thus, the same mechanism used in the UML 2.0 deployment metamodel to provide information about deployment descriptors may be used to provide information about aspect instantiation modes.

Concretely, to provide support to represent deployment descriptor, the UML 2.0 metamodel defines the ComponentDeployments package. This package extends the basic deployment model with capabilities to support deployment mechanisms found in several common component technologies. In this package, a component deployment is the deployment of one or more artefacts instances to a deployment target, optionally parameterized by a deployment specification. A deployment specification specifies a set of properties that determine deployment and execution parameters of a component artefact that is deployed on a node. The only constraints imposed by the metamodel are: (1) the deployedElements of a DeploymentTarget that are involved in a Deployment that has an associated DeploymentSpecification is a kind of Component, and (2) the DeploymentTarget of a DeploymentSpecification is a kind of ExecutionEnvironment [25].

The first constraint it is not a problem in our approach since aspectual behaviours are modelled by Components. Neither the second constraint is a problem, since this constraint only implies that the aspect instantiation mode should be determined by the execution environment in which the application will finally be executed.

In fact, with respect to the second issue we think that two alternative approaches are possible. One of them is the definition of a set of aspect instantiation modes related to the elements in the C&C view. This is shown in Figure 24 and the different values would be:

- **singleton:** only an instance of the aspectual component is created in the system.
- **percomponenttype:** a different instance of the aspectual component is created for each "type" of a base component.
- **percomponentinstance:** a different instance of the aspectual component is created for each "type" and "instance" of a base component.
- **percomponenttype_port:** a different instance of the aspectual component is created for each "type" and "port" of a base component.
- **percomponentinstance_port:** a different instance of the aspectual component is created for each "type", "instance" and "port" of a base component.
- **perconnectortype:** a different instance of the aspectual component is created for each connector "type" to which that component may be attached.
- **perconnectorinstance:** a different instance of the aspectual component is created for each connector "type" and "instance" to which that component may be attached.

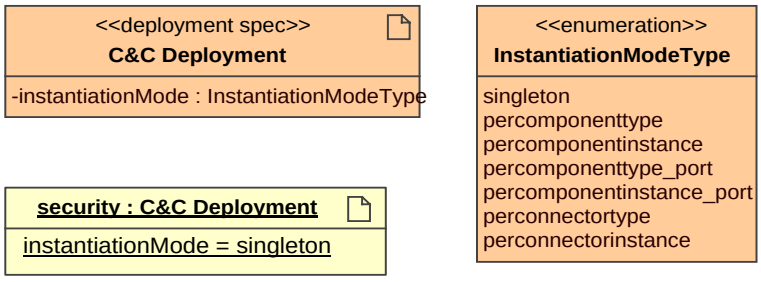

Figure 24. *C&C view related Aspect Instantiation Modes*

At this point we need to say that this is the broader range of instantiation modes that we may consider at the architectural level in the C&C view. However, no validation was made yet about the degree of usability of each of them. This validation is part of our ongoing work and will be done for at least two case studies. One of them is the Auction System case study used in D63 [47]. The other one is the Toll System case study provided by Siemens as part of the industrial case study "Application of the Integrated AO Architecture Design Method to the Toll System" that will be soon carried out by UMA, ULANC and Siemens in the context of the Applications Lab [24].

The advantage of using this approach is that the software architect can provide at the architectural level information about the expected number of instances for each aspectual component. The main inconvenience is that a mapping would need to be defined between these instantiations modes and the instantiations modes available by a particular execution environment.

In Figure 25 and Figure 26 we show two alternatives provided by the UML 2.0 deployment metamodel in order to deploy the Security component. In the Auction System this component plays the role of an 'aspectual' component. Notice that in Figure 25 we show the deployment specification previously defined, while in Figure 26 we show a more general approach in which it is indicated that the deployment specification for the Security component is specified in an XML file (this will be the XML representation of the deployment specification described in AO-ADL).

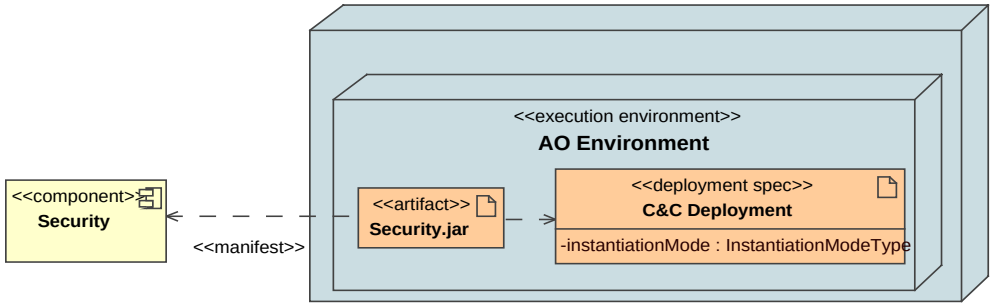

Figure 25. *Deployment of the Security component (Alternative 1)*

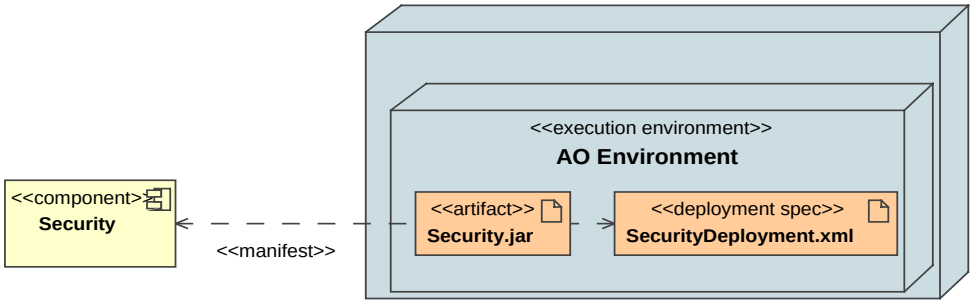

Figure 26. *Deployment of the Security component (Alternative 2)*

The more general representation in Figure 26 gives us support to specify the second approach previously commented. This consists on the deployment of the components in a particular aspect-oriented execution environment, where the deployment specification will be particular of that execution environment. As an example in Figure 27 we show the deployment specification for CAM/DAOP [14], a component and aspect distributed platform where the deployment information is provided in XML using the DAOP-ADL language (predecessor of AO-ADL). In CAM/DAOP components can be deployed locally (for each instance of the DAOP platform) or remotely (in a particular instance of the DAOP platform). Aspects can be instantiated as a singleton instance (ENVIRONMENT_BASED) or one instance for each instance of the DAOP platform (USER_BASED). Moreover, the same instance can be shared by all the components playing a particular role (ROLE_BASED), for all the instances representing the same resource in the system (INSTANCE_BASED) or one instance for each component based on a component's unique identifier (CID_BASED).

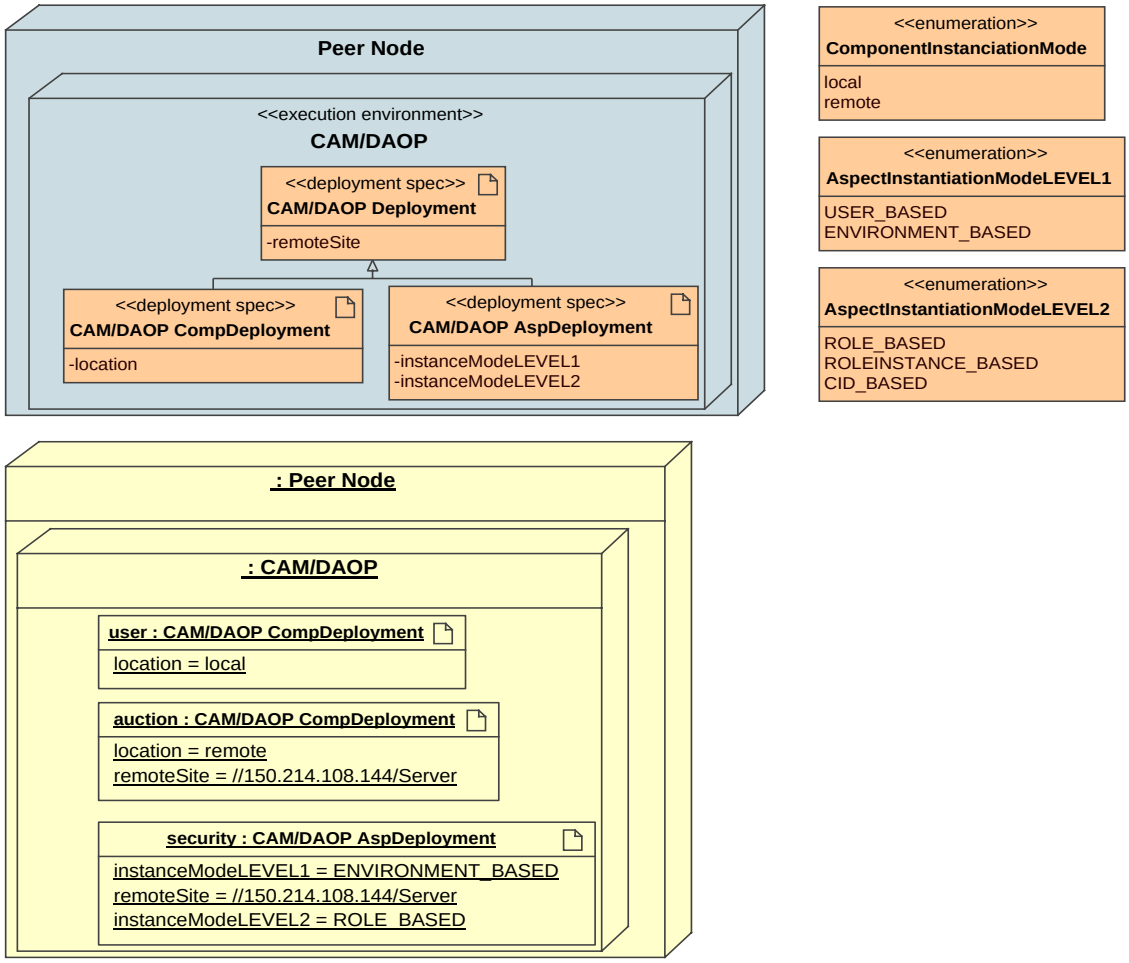

Figure 27. *Deployment specification for CAM/DAOP*

The important point here it not the particular instantiation modes set, but the idea that using different AO execution environment this information will change. For instance, in AspectJ an aspect can have the following instantiation modes: singleton, perthis(Pointcut), pertarget(Pointcut), percflow(Pointcut) and percflowbelow(Pointcut); while in JAsCo the following instantiation modes are possible: perobject, perclass, permethod, perall, perthread, per(custom aspect factory).

Notice that in some execution environments also the binding information (provided by connectors) may need to be deployed [14]. This scenario is shown in Figure 28 for the composition between the User, Auction and Security components shown in Figure 25. As an example, an additional deployment specification (BindingInformation.xml) has been added with a <<manifest>> relationship with the connector between the User and the Auction component. The same can be done for the aspectual binding information.

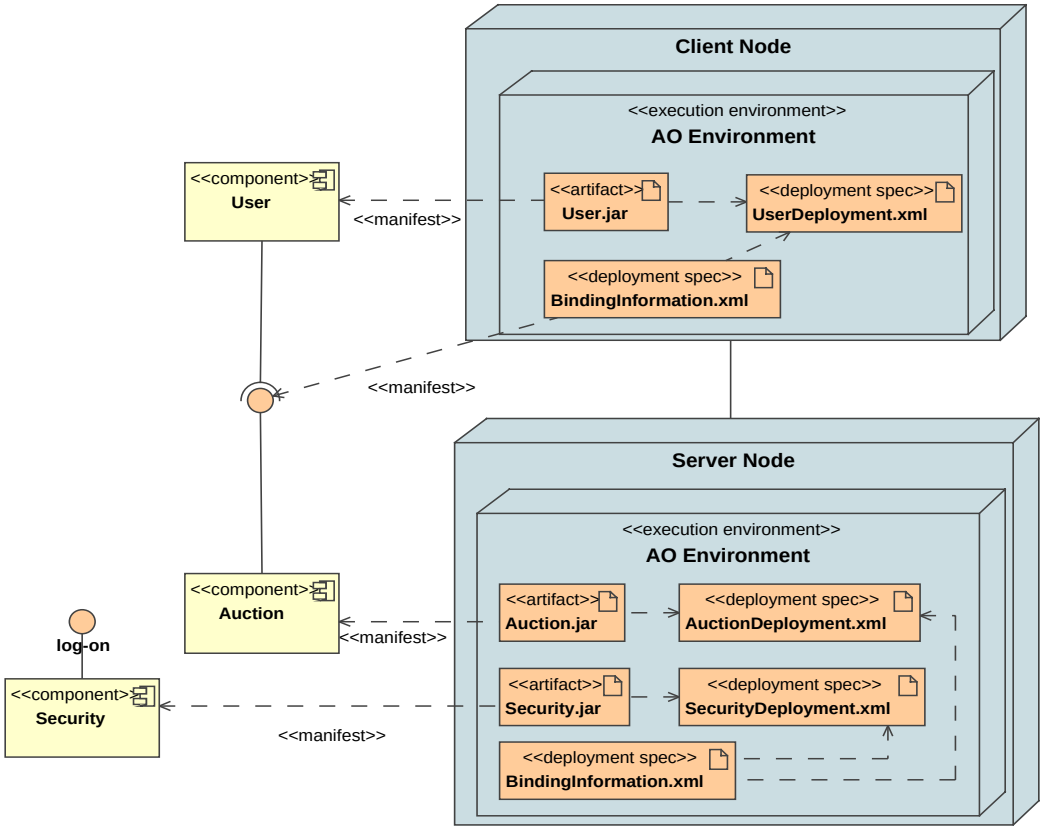

Figure 28. *Deployment of the User-Auction-Security scenario*

8. Conclusion

In this document we have provided an analysis and approach for modeling aspects in architectural views. In principle many different architectural views can be defined based on the concerns that need focus. We have used the categorization in [10] in which we have derived the three basic views including module view, component and connector view and allocation view. Views are represented using viewtypes (or viewpoints). To understand views and describe aspect-oriented versions of the three views we have defined different viewtypes. In essence a viewtype is a metamodel and therefore we have aimed to provide the metamodels for the different views.

It appears that we can approach the problem of defining different viewtypes in different ways. In section 2 we have provided an analysis on the alternatives and decided for one approach in which we define separate metamodels for the three viewtypes.

We have defined thus a metamodel for module view, C&C view and deployment view. The metamodel for the module view appeared to be in the same line of existing work on modeling aspects in UML The metamodel for the C&C view seemed to be the most in alignment with the existing deliverables. Finally, we have defined a metamodel for deployment view which maps the components (and aspects) in the C&C view to the hardware nodes.

References

1. O. Aldawud, T. Elrad, A. Bader. *Uml Profile for Aspect-Oriented Software Development*. In: Proceedings of Third International Workshop on Aspect-Oriented Modeling, 2003.
2. O. Barais et al. TranSAT: A Framework for the Specifcation of Software Architecture Evolution. Ws on Coordination and Adaptation Techniques for Software Entities, ECOOP, Oslo, Norway, 2004.
3. T. Batista, C. Chavez, A. Garcia, C. Sant'Anna, U. Kulesza, A. Rashid, F. Filho. Reflections on Architectural Connection: Seven Issues on Aspects and ADLs. Ws on Early Aspects, ICSE, 2006.
4. T. Batista, C. Chavez, A. Garcia, U. Kulesza, C. Sant'Anna, C. Lucena. Aspectual Connectors: Supporting the Seamless Integration of Aspects and ADLs. Brazilian Symposium.on Software Engineering, Brazil, 2006.
5. N. Boucke, A. Garcia, T. Holvoet. Composing Architectural Structures in xADL. Int'l Workshop on Early Aspects, AOSD, 2006, Canada.
6. C. Chavez, A. Garcia, T. Batista. Are Architectural Aspects Style-Dependent? Proc. International Workshop on Aspects in Architecture Descriptions (AARCH.07), AOSD Conference, Vancouver, Canada, March 2007.
7. R. Chitchyan et al. *Mapping and Refinement of Requirements Level Aspects*, NoE-AOSD Deliverable D63, November 2006.
8. S. Clarke & E. Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley, 2005.
9. S. Clarke. *Extending standard UML with model composition semantics*, Science of Computer Programming, Volume 44, Number 1, pp. 71-100(30), July 2002.
10. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.
11. J.M. Favre. *CacOphoNy: Metamodel-Driven Software Architecture Reconstruction.* in: Reverse Engineering, 2004. Proceedings. 11th Working Conference on, 2004.
12. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems* (IEEE Std 1471-2000). Piscataway, NJ: IEEE Computer Press, 2000.
13. D.S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, OMG Press, 2003.
14. L. Fuentes, M. Pinto, J.M. Troya, *Supporting the development of CAM/DAOP applications: an integrated development process*, Software Practice and Experience, 37:21-64, Willey InterSicence, 2007.
15. A. Garcia, C. Chavez, T. Batista, U. Kulesza, C. Sant'Anna, A. Rashid, C. Lucena. On the Modular Representation of Architectural Aspects. 3rd European Workshop. on Software Architecture, EWSA, France, 2006.
16. A. Garcia, E. Figueiredo, C. Sant'Anna, M. Pinto, L. Fuentes. *Representing Architectural Aspects with a Symmetric Approach*.

17. P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, A. Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP.07), July 2007, Germany.
18. A. Jackson, P. Sanchéz, L. Fuentes, and S. Clarke, *Towards traceability between AO architecture and AO design,* in Workshop on Early Aspects (held with ASOD '06), (Bonn, Germany), 2006.
19. M. Kandé, J. Kienzle, and A. Strohmeier. *From AOP to UML—A Bottom-Up Approach.* Workshop on aspectoriented modeling with UML at the AOSD conference, 2002.
20. U. Kulesza, A. Garcia, and C. Lucena. Towards a Method for the Development of Aspect-Oriented Generative Approaches. Ws on Early Aspects, OOPSLA'04, 2004, Vancouver, Canada.
21. I. Kurtev. *Adaptability of Model Transformations,* PhD Thesis, University of Twente, Dept. of Computer Science, Software Engineering Group, 2005.
22. P. Merson. *Representing Aspects in the Software Architecture – Practical Considerations*, In Early Aspects Workshop, OOPSLA 2005, San Diego, California, Oct. 2005
23. Meta Object Facility (MOF) Specification, Object Management Group, http://doc.omg.org/formal/02-04-03, 2003.
24. R. Meunier, et. al. "Study Plan for Empirical Studies", Deliverable D70, AOSD-Europe-Siemens-5.
25. OMG document, Unified Modeling Language: Superstructure, version 2.0, formal/05-07-04, 2005
26. R. Pawlak et al. *A UML Notation for Aspect-Oriented Software Design.* Workshop on aspect-oriented modeling with UML at the AOSD conference, 2002.
27. M. Pinto, L. Fuentes, J. Troya. DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. LNCS, Vol. 2830, 118-137, 2003.
28. M.Pinto, N. Gámez, L. Fuentes, "Towards the architectural definition of the Health Watcher system with AO-ADL", Early Aspect Workshop at ICSE'07, Minneapolis, USA, May 2007
29. M. Pinto, L. Fuentes, *AO-ADL: An ADL for describing Aspect-Oriented Architectures*, accepted for publication in the Proceedings of the Early Aspects Workshop (collocated with AOSD'07), 13 March, 2007, Vancouver, Canada
30. C. Sant'Anna, C. Lobato, U. Kulesza, C. Chavez, A, Garcia, C. Lucena. On the Quantitative Assessment of Modular Multi-Agent Architectures. NetObjectDays, September 2006, Germany.
31. C. Sant'Anna, E. Figueiredo, A. Garcia, C. Lucena. On the Modularity Assessment of Software Architectures: Do my architectural concerns count? Proc. International Workshop on Aspects in Architecture Descriptions (AARCH.07), AOSD.07 Conference, Vancouver, Canada, March 2007.
32. S. Soares, E. Laureano, P. Borba. Implementing Distribution and Persistence Aspects with AspectJ. Proc. OOPSLA 2002, pp 174 - 190.
33. T. Stahl & M. Voelter. *Model-Driven Software Development*. Wiley, 2006.

34. Stein, D., Hanenberg, S., and Unland, R. "A UML-based Aspect-Oriented Design Notation For AspectJ". Proceedings of the 1st International Conference on AOSD, 2002.
35. K. Sullivan, W.G. Griswold, Y. Song, Y.Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pages 166–175, New York, NY, USA, 2005.
36. B. Tekinerdoğan. *Synthesis-Based Software Architecture Design*, PhD Thesis, Dept. of Computer Science, University of Twente, The Netherlands, 2000.
37. B. Tekinerdogan, C. Hofmann, M. Aksit. *Tracing Aspects within and across Architectural Views*, submitted to Aspect-Oriented Modeling Workshop, Aspect-Oriented Software Development Conference, Vancouver, March, 2007.
38. Zakaria, A., Hosny, H, and Zeid, A. *A UML Extension for Modeling Aspect-Oriented Systems.* Workshop on aspectoriented modeling with UML at the AOSD conference, 2002.