AO-ADL: An ADL for Describing Aspect-Oriented Architectures

Mónica Pinto and Lidia Fuentes

Dpto. Lenguajes y Ciencias de la Computación, GISUM Research Group
University of Málaga, Málaga, Spain
{pinto,lff}@lcc.uma.es
http://caosd.lcc.uma.es/

Abstract. Architecture description languages are a sound and convenient approach to software architecture representation. The majority of well-known ADLs provide separation of computation and communication in components and connectors, respectively. However, computation and communication are not the only crosscutting concerns that may appear in a software architecture description. Traditional ADLs do not normally provide appropriate support to separate any kind of crosscutting concerns, which frequently result in poor architectures descriptions with highly coupled components. In this paper we present the AO-ADL language, based on a symmetric decomposition model that considers components and connectors as the basic structural elements (similar to traditional ADLs). We will show how aspects are treated as specific types of components that are composed by means of connectors. In order to cope with the separation of concerns we enrich the semantic and expressivity of traditional connectors to support either aspectual and non-aspectual component interactions.

Keywords: Aspect-oriented software architectures, ADLs, connectors, connector templates.

1 Introduction

Architecture Description Languages (ADLs) are a sound and convenient approach to software architecture representation. They have been recognized as an important tool for supporting the systematic reasoning about system components and their relationships early in the development process [1]. ADLs traditionally separate computation and communication into two different architectural elements, i.e. components to encapsulate computation, and connectors to encapsulate communication.

However, computation and communication are not the only crosscutting concerns that may arise during software architecture description. Other functional and extra-functional concerns, such as security, availability, etc., may also be tangled – i.e. several concerns are observed in the same component, or scattered – i.e. same concern is observed in more than one component. The lack of

A. Moreira and J. Grundy (Eds.): Early Aspects 2007 Workshop, LNCS 4765, pp. 94–114, 2007.

appropriate support to represent these crosscutting behaviors at the architectural level frequently result in poor descriptions of software architectures – with highly coupled components and loose of information about the interactions and dependencies among them.

On the other hand, the aim of Aspect-Oriented Software Development (AOSD) [2] is the identification and modeling of crosscutting concerns from requirements to implementation. Several approaches already exist for AO requirements, detailed design and implementation languages, but much more effort should be put in the specification of appropriate languages to describe AO architectures [3].

The answer to both challenges is the definition of ADLs able to support the separation and composition of any kind of crosscutting concerns by extending the expressivity of traditional ADLs. In this paper we present the AO-ADL (Aspect Oriented-Architecture Description Language) language [4]. Similar to traditional ADLs, AO-ADL considers components and connectors as the basic structural elements. Thus AO-ADL preserves the main architectural blocks of traditional ADLs that will be shown as enough to specify AO architectures. Notice that AO-ADL is not a completely new ADL. It is a generalization and evolution of our previous work named DAOP-ADL [5], which was specifically described for specifying component- and aspect-based software architectures running on top of the CAM/DAOP [6] platform. Other existing aspect-oriented ADLs are detailed in the related work section.

AO-ADL considers that components model either crosscutting (named *aspectual component*) or non-crosscutting behavior (named *base component*) exhibiting a symmetric decomposition model. Instead of defining a new entity to model aspects or extending the component with new kind of interfaces, as other aspect-oriented ADLs [5,7], a component is considered as an aspect when it participates in an aspectual interaction. This approach increases the reusability of components, which may play an aspectual or non-aspectual role depending on the particular interactions in which they participates. The symmetric nature of AO-ADL is one of the contributions of this language.

Following a symmetric approach, the crosscutting nature of a component only depends on the connections with other components, which in ADLs are specified in connectors. Thus, the second contribution of AO-ADL is the extension of the semantic of traditional connectors to represent the crosscutting effect of 'aspectual' components. This means that AO-ADL connectors provide support to describe different kinds of interactions among components – not only typical communication as in traditional ADLs, but also crosscutting influences among them. That is, AO-ADL connectors specify how 'aspectual' components are weaved with 'base' components during components' communication. In order to show the main characteristics of the AO-ADL language, we use the ATM model problem taken from the software architecture literature.

After this introduction, in section 2 we discuss the related work on aspect-oriented ADLs. In section 3 we describe the ATM case study that we will follow throughout the paper. Then, the AO-ADL language is presented in section 4. We briefly discuss components to then focus mainly on the definition of connectors.

We finish this section presenting the complete AO-ADL description of the ATM system. After the presentation of AO-ADL, in section 5 we discuss the main advantage of using our approach. The last section presents our conclusions and future directions.

2 Related Work

Non aspect-oriented ADLs have been extensively described and compared in the bibliography. Concretely, a comparison of some of the most relevant ones, like ACME [8], C2 [9] and Rapide [10], can be found in [1]. As stated in the introduction, these ADLs lack of support to specify crosscutting concerns and aspectual interactions. So, in this paper we focus on the related work about aspect-oriented ADLs.

Concretely, in order to define an aspect-oriented ADL, there are at least two questions that need to be answered: (1) which are the main architectural blocks of the language?, and (2) how composition between 'base' and 'aspectual' components is represented?

With respect to the first issue, there is not a consensus between existing AO architectural approaches [3,11]. While some of them add a new building block named aspect to model crosscutting concerns, others represent aspects as components with a different semantic, or as components that provide or require special 'crosscutting' interfaces.

In this sense, DAOP-ADL [5,6] (our previous work) and Fractal [7,12] extend traditional components with new kind of interfaces. In DAOP-ADL aspects are fist-order entities defining an *evaluated interface*. This interface specifies how aspects affect component's interfaces. Similarly, in Fractal a new kind of component named *Aspectual Component* is defined that contains a special *aspect component interface* that provides a set of methods to introspect a join point[1] Finally, in PRISMA [13] aspects are a new abstraction used to define the internal structure of both components and connectors. For a more extensive comparison of these proposals see [3,11].

Considering the lessons learned from our previous work DAOP-ADL (a asymmetric ADL), we concluded that the distinction between components and aspects at architectural level drastically decreases the possibilities for components reuse. Instead, the same building block (e.g. a component) can be used to represent either non-crosscutting and crosscutting concerns. Another proposal following this approach is AspectualACME [14], which is an aspect-oriented extension of ACME where aspects are modeled as ACME components. FuseJ [15] is also a symmetric approach that combines components and aspects and includes the concept of XML-based configurations to specify the weaving information. The main difference is that FuseJ is a Java based programming model and not an

[1] AO ADLs do not always share the same definition of symmetry. Notice that we consider that an aspect-oriented ADL is symmetric when the same building block (without new kind of interfaces or other structural elements) is used to model both base and aspectual behaviors. Otherwise, we consider that the ADL is asymmetric.

ADL. Finally, in [16] aspect-oriented software architectures are described in LEDA, where aspects are LEDA components; the support for aspect-orientation is provided by including special kind of components to the architecture.

With respect to the second issue, though there are differences on existing aspect-oriented ADLs, most of them agree on that the semantic of the compositions has to be somehow extended to incorporate aspects to an ADL. Once again, an interesting analysis and comparison of this issue has been performed in [3,11]. In [16] the semantic of connectors is not modified since in addition to the components modeling aspects, a new kind of component with the role of 'coordinators' have to be included in order to describe an aspect-oriented software architecture. These components behave as an aspect manager modeling the interception of join points and the invocation of aspects. In PRISMA [13], since aspects are part of either the components and connectors specification, also the composition specification follows a different approach, being specified inside both components and connectors.

In DAOP-ADL [5], the antecedent of AO-ADL, connections are defined outside either components and connectors, in a specific composition section. They are defined in terms of a set of composition rules, which specify component communication, and a set of aspect evaluation rules, which specify component and aspect weaving. This is basically the same information that AO-ADL encapsulates as part of the specification of connectors. A similar approach is followed by Fractal and FuseJ that define an XML-based aspect binding section and linklets specification, respectively, in order to describe components and aspects compositions. These specifications are equivalent to those in DAOP-ADL. Thus, the main difference between AO-ADL and DAOP-ADL, Fractal and FuseJ is that in the later the concept of connector is not explicit and the component and aspect bindings play the role of both connections and configurations. Beside that, the composition information in AO-ADL is not completely new, since it is an evolution of the information described in the DAOP-ADL composition rules.

In AspectualACME [14], the specification of connectors model composition by defining base and crosscutting roles and configurations, being very similar to AO-ADL connectors. This proposal extends ACME connectors to support crosscutting roles. The main difference with respect to our approach is on the definition of pointcuts. While in AspectualACME pointcuts are defined as part of ACME's attachments, in AO-ADL pointcuts are defined inside connectors. Other difference is that, though both languages allow the use of quantifications[2] in the specification of pointcuts, in AO-ADL the roles of connectors can also be specified using quantifications. As shown along the paper, this feature of AO-ADL improves the adaptability and reusability of connectors.

Finally, also the ConcernBASE proposal [17] is similar to ours in that it extends the semantic of connectors to capture new kind of interactions among components, though they follow a different approach. Concretely, ConcernBASE connectors defines a context to represent various kinds of interactions simultaneously. It defines a set of crosscutting interaction categories that can be specified

[2] We understand quantifications as the use of wildcard and binary operators.

inside connectors, such as data access, arbitrator, linkage, distributor and adaptor. The main contribution of this work is that it extends the taxonomy of software connectors proposed by Medvidovic in [18], though the main shortcoming is that the list of crosscutting categories is limited and therefore it does not provide support to separate any kind of crosscutting concern.

3 A Case Study: The Automated Teller Machine System

As mentioned in the introduction, in order to show the contributions of our approach we will use the automated teller machine model problem, taken from the software architecture literature. In order to illustrate the separation of concerns in software architectures, additionally to the ATM system, we have also experience using the AO-ADL language for describing the software architecture of the Health Watcher [19] and the Auction system [20] case studies.

A simplified description of the system[3] is the following: *"In the automated teller machines (ATMs) system each bank in a consortium provides its own computer to maintain its own accounts and process transactions against them. Human cashiers enter account and transaction data. Automated teller machines communicate with a central computer which clears transactions with the appropriate banks. An automated teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires appropriate security provisions. The system must handle concurrent accesses to the same account correctly. The banks will provide their own software for their own computers."*.

A partial description of the software architecture of the ATM system, where mainly the core functionality is represented, is shown in Fig. 1. According to Fig. 1, the ATM-GUI component represents the user graphical interface and interacts with the ATM component, which encapsulates the main behavior of an ATM. The bank (Bank component) is always an intermediary of the interactions between the ATM (ATM component) and the customers' accounts (Account). Finally, in order to maintain a backup copy of the accounts, the Account component is responsible for updating the BAccount component. This description has been partially extracted from [21], where a non-aspect oriented description of the ATM model problem is described[4],[5]

This description needs to be completed with the representation of extra-functional or non-functional requirements, which are of our interest to show the main contributions of our approach. Concretely, throughout the paper we focus on four of the non-functional requirements specified in [21] for the ATM system:

[3] As it appears in http://www.cs.cmu.edu/∼ModProb/

[4] We have not used the solution in [21] since it was not described with an ADL.

[5] To the best of our knowledge, a complete specification of the ATM system using an ADL is not available; and normally these descriptions do not address extra-functional concerns, the most important ones to illustrate the benefits of our AO approach.

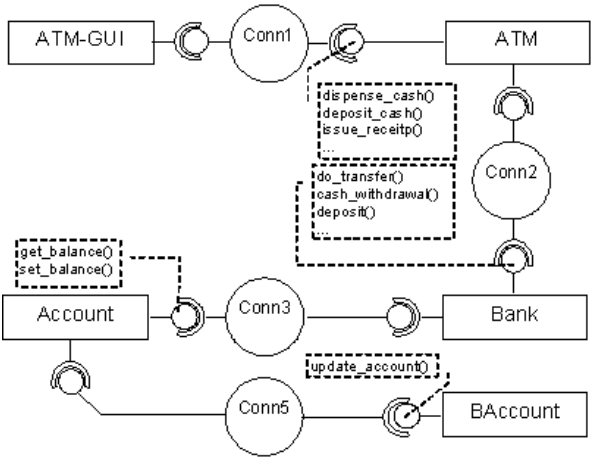


Fig. 1. Partial software architecture of the core functionality of the ATM System

- NFR1. *Heterogeneity*. In [21] this is interpreted as providing inter-communication mechanisms between two heterogeneous banking system.
- NFR2. *Concurrency*. The system must handle concurrent accesses to the same account preserving the integrity of the data account.
- NFR3. *Security*. In [21] security implies the *authentication* of customers that interact with the ATM, the *authorization* in the access to the bank's accounts and the *confidentiality* in the interactions between the ATM and the bank accounts. We extend *confidentiality* as a requirement to be satisfied by all the interactions between components in the ATM system.
- NFR4. *Availability*. Customer services should be available 999/1000 requests. In [21] availability is achieved by maintaining a replica of the accounts.

In the following sections we first describe the AO-ADL language and how to represent crosscutting concerns in the Component and Connector architectural view. Then, we show the completed aspect-oriented solution for the ATM system specified in AO-ADL. We will make special emphasis on the advantages introduced by the enriched semantic of AO-ADL connectors.

4 The AO-ADL Language

AO-ADL is a new XML-based architecture description language specifically well-suited for describing aspect-oriented architectures. Likewise traditional ADLs [1] the main architectural elements of AO-ADL are components and connectors. Instead of extending ADLs with concepts introduced by aspect-oriented programming (AOP) languages, we integrate all these concepts (e.g. join points, pointcuts, etc.) under the definition of components and connectors. We do not consider those concepts specific of AOP languages and close to implementation

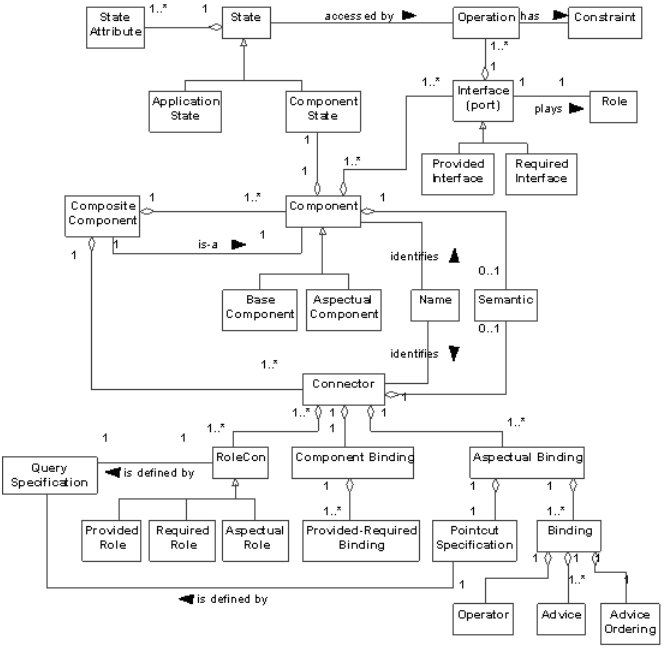


Fig. 2. AO-ADL Metamodel

such as introductions, since they not always help to capture the fundamental nature of software architecture descriptions.

As we already stated above, the main difference between crosscutting and non-crosscutting concerns is merely in the role they play in a particular composition binding and not in the internal behavior itself. Therefore, instead of inventing a new structural element, we redefine the connector and extend its semantics with aspectual bindings. Notice that a software architect has also the possibility of specifying a traditional connector without including the specification of neither aspectual roles nor aspectual bindings. This means that when there is not crosscutting behavior to be specified, the software architect can use connectors as in traditional ADLs. Moreover, software architects non-experts in aspect-orientation would not need to learn AO concepts. Instead, connectors can be specified in two iterations. First, provided and required roles and component bindings are specified. Then, this connector is extended to incorporate aspectual roles and aspectual bindings.

Notice that we have opted for defining a new language, even when many ADLs already exist that could have been extended with AO characteristics [1]. Even when the extension of an existing language would require less effort than the definition of a new one from scratch, the prior approach has also significant drawbacks. Concretely, the main problem of extending a language is obscuring the semantics of the elements of the new language. For instance, ACME [22] defines the *property* concept to extend a component or connector with new characteristics. If we want

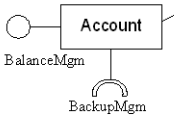
to extend ACME for AO-ADL, all AO-ADL distinctive features would be ACME properties. This means that the semantics of all AO-ADL new elements would be hidden under a unique concept, the property. A similar reasoning can be done for xADL [23], a highly-extensible XML-based ADL that provides a set of extensible schemas[6]. From our experience it was not worth extending an existing ADL, since AO-ADL required a number of innovative characteristics the semantics of which we wanted to keep clear.

Figure 2 shows the meta-model of our language. The different parts of the meta-model are explained in next subsections using the ATM system case study.

4.1 AO-ADL Components

As discussed above, the first contribution of AO-ADL is the representation of both components and aspects, without distinction, with a single architectural element, the component (Component in Fig. 2). In AO-ADL the specification of components is the same than in other ADLs [1]. From the case study described before, the Account component is represented in AO-ADL as specified in Fig. 3.

As shown in Fig. 3, components are identified by a required unique name inside a particular architectural design (Name in Fig. 2). They are described by means of their provided (ProvidedInterface in Fig. 2) and required (RequiredInterface in Fig. 2) interfaces (the *ports* of the components). The interfaces are defined by the <interface> tag (see Fig. 3). When reusing an interface definition as part of a component the software architect assigns a unique identifier to each interface, the role name (Role in Fig. 2), that refers to the role the component can play or require in a particular architectural design. For example, the Account component in Fig. 3 plays the role of managing the balance of the account (BalanceMgm) and requires the role of managing backup copies (BackupMgm).




```
<component name="Account">
    <provided-interface role="BalanceMgm"
        uri="//interface[@name="BalanceManagement"]/>
    <required-interface role="BackupMgm"
        uri="//interface[@name="BackupManagement"]/>
    <state name="AuctionState">
        <attribute name="balance"/>
        <attribute name="credit_limit_type"/>
    </state>
    <behaviour>
        BalanceMgm.setBalance ->
        BackupMgm.updateAccount
    </behaviour>
</component>
<interface name="BalanceManagement">
    <operation name="getBalance"/>
    <operation name="setBalance"/>
</interface>
<interface name="BackupManagement">
    <operation name="updateAccount"/>
</interface>
```

Fig. 3. AO-ADL description of the Account 'base' component

Components can also expose their public state (State in Fig. 2). For the Account component, the balance and credit_limit_type state attributes have been defined.

[6] We extended this language for our previous work DAOP-ADL (DAOPxADL [24]) but the resulting architectural specifications are more complex and awkward.

Finally, components can optionally expose their semantics (Semantic in Fig. 2), which is a high-level model of a component's behavior. This behavior is expressed in terms of the interaction protocol between components. The interaction protocol specifies the ordering of the occurrence between the reception of operations in provided interfaces and the sending of operations through required interfaces. In the Account component the behavior section specifies that once the setBalance operation is received in the provided interface with role name BalanceMgm, the updateAccount operation of the required interface with role name BackupMgm is sent. Note that AO-ADL allows embedding of domain-specific languages for specifying components semantics. This feature is used by other ADLs as well (e.g. ACME). Its main benefit is the possibility of reusing existing standard or well-adopted (formal) notations. Other concepts of AO-ADL, which are out of the scope of this paper, are composite components and application state.

Following the symmetric approach, the decision about if a component specification defines a crosscutting or non-crosscutting behavior will be taken during the architecture configuration definition. This means that the connector of a given components interaction will specify if a component is 'base' or 'aspectual'. Note that in the latter case, any operation defined as part of a provided interface of the component can be considered an *advice*.

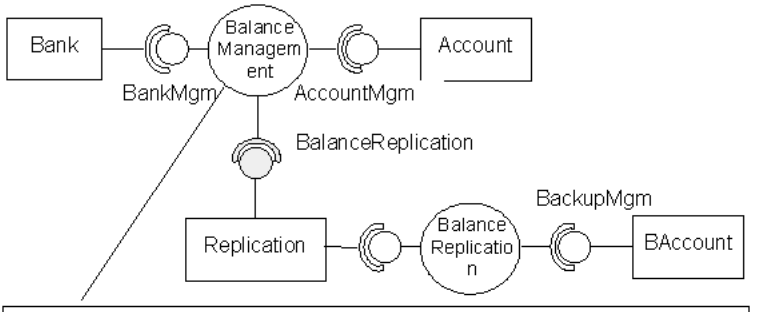
4.2 AO-ADL Connectors

In traditional ADLs connectors are the building block used to model interactions among components and rules that govern those interactions [1]. Similarly, the connector is the architectural element for composition in AO-ADL, but extended with additional features to support interactions with 'aspectual' components.

There are two main differences between the representation of connectors in traditional ADLs, and the representation of connectors in AO-ADL (Connector in Fig. 2). The first difference and most relevant one is the distinction between *component bindings* (interactions among 'base' components) and *aspectual bindings* (interactions between 'aspectual' and 'base' components). This means that components referenced in the aspectual binding section of a connector are playing the role of an 'aspectual' component in the specified interaction. Later in this section we show how the component bindings and the aspectual bindings differ due to the different interactions among them. Thus, Second difference is the use of quantifications to specify the connector roles.

Fig. 4 shows the BalanceManagement connector for the ATM system. The ATM requirements state that the Bank component (which required interface's role is specified in the provided-role clause of the connector in lines 2 to 6) is the responsible for setting the balance of the Account component (which provided interface's role is specified in the required-role clause of the connector in lines 7 to 11). The composition binding between these components is specified in the componentBinding clause of the connector in lines 17 to 22. Furthermore, the 'availability' requirement previously described in section 3 (NFR4) is a crosscutting requirement that affect the interaction between the Bank and the Account components. Basically, this requirement states that a backup account, represented by the

component BAccount, needs to be updated when a new balance is set in the Account component. Applying the AOSD principles, we consider Replication as an 'aspectual' component (which provided interface's role is specified in the




```
1 <connector name="BalanceManagement">
2   <provided-role name="BankMgm">
3     <role-specification>
4       /component/required-interface[@role="AccountMgm"]
5     </role-specification>
6   </provided-role>
7   <required-role name="AccountMgm">
8     <role-specification>
9       /component/provided-interface[@role="BalanceMgm"]
10    </role-specification>
11  </required-role>
12  <aspectual-role name="BalanceReplication">
13    <role-specification>
14      /component/provided-interface[@role="BalanceReplMgm"]
15    </role-specification>
16  </aspectual-role>
17  <componentBindings>
18    <binding name="BMBinding">
19      /connector[@name="BalanceManagement"]/provided-role[@name="BankMgm"] and
20      /connector[@name="BalanceManagement"]/required-role[@name="AccountMgm"]
21    </binding>
22  </componentBindings>
23  <aspectBindings>
24    <aspectual-binding  name = "BM-ReplicationBinding">
25      <pointcut-specification>
26        /connector[@name="BalanceManagement"]/componentBindings/
27                                    binding[@name= "BMBinding"] and
28        /operation[@name="setBalance"]
29      </pointcut-specification>
30      <binding operator="after">
31        <aspectual-component
32          aspectual-role-name="/connector[@name="BalanceManagement"]/
33                                    aspectual-role[@name="BalanceReplication"]
34          advice-label="/operation[@name="update-balance"]/>
35      </binding>
36    </aspectual-binding>
37  </aspectBindings>
38 </connector>
```

```
39 <component name="Bank">
40   <required-interface role="AccountMgm"
41     uri=//interface[@name="BalanceManagement"]/>
42 </component>
43 <component name="Replication">
44   <provided-interface role="BalanceReplMgm"
45     uri=//interface[@name="BalanceReplication"]/>
46 </component>
```

```
47 <interface name="BalanceManagement">
48     <operation name="setBalance"/>
49     ...
50 </interface>
51 <interface name="BalanceReplication">
52     <operation name="update-balance"/>
53     ...
54 </interface>
```

Fig. 4. Example of AO-ADL Connector

aspectual-role clause of the connector in lines 12 to 16) that affects the interaction between the Bank and the Account components (specified in the pointcut-specification of the connector in lines 25 to 29).

Notice that the above implies the removal of the required interface with role name BackupMgm from the Account component that was shown in Fig. 3. Instead, in an aspect-oriented solution this behavior is provided by the Replication component, acting as an 'aspectual' component. The main benefit is that the Account component is released of the responsibility of maintaining the backup copy, which was tangled with the core concern of the Account component.

In the next subsections we describe the different parts of the connectors[7].

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 the Encryption component to our architecture. Assuming that, according to the security requirement (NFR3) specified in section 3, all the interactions between components in the ATM system need to be encrypted, in Fig. 5.a the Encryption component is included as an 'aspectual' component that affects the interactions between the ATM-GUI and the ATM components, between the ATM and the Bank components and between the Bank and the Account component. As shown in Fig. 5.a these connections are specified by three different connectors. Consequently, an aspectual-role clause specifying the encryption 'aspectual' behavior and an aspectual-binding clause specifying how encryption is bound to the interaction among 'base' components have to be replicated in the three connectors. We have omitted the specification of Conn2 that is the same specification shown in Conn1 and Conn3[8].

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 (shown in Fig. 5.b) would specify the following: 'For all the interactions between

[7] A more formal statement of the connector semantics is part of our future work.
[8] Only the part of the connector that would be replicated is shown.

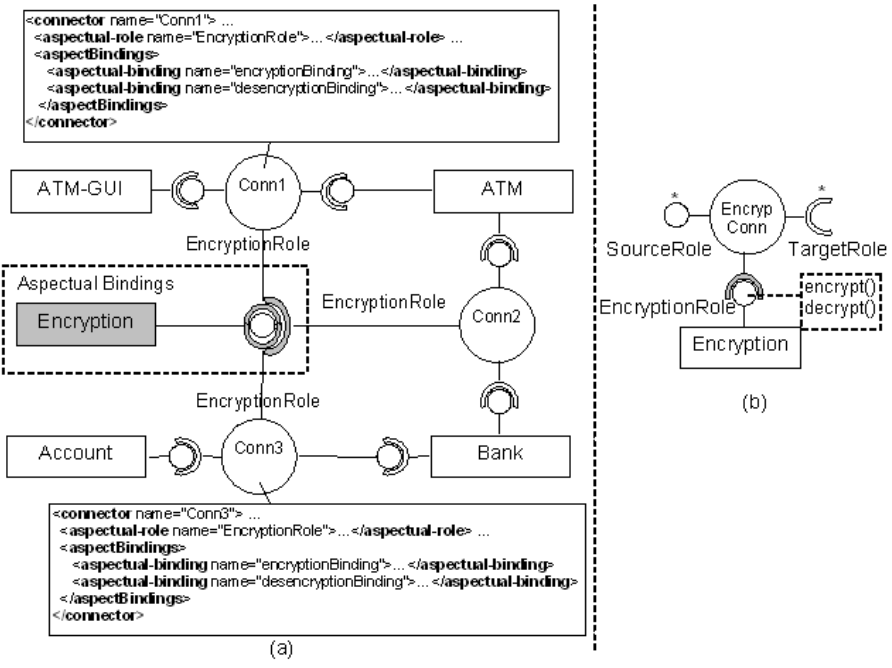


Fig. 5. Examples of (a)Scattered aspectual composition; (b)Use of quantified ports

any source and target component in the ATM system, a component playing the role EncryptionRole in one of its provided interfaces is attached'. In our case study that component is the Encryption component.

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.

Base Component Composition. The specification of the 'base' component bindings (ComponentBinding in Fig. 2) inside the connector describes the connections among 'base' components. For instance, the componentBindings clause in the connector shown in Fig. 4 describes the interaction between the Bank and

the Account components by connecting the corresponding provided and required roles previously described in the connector (component binding with name 'BM-Binding' in lines 17 to 23). Note that we do not discuss the kind of interactions or communication mechanisms among components that may be considered in AO-ADL, i.e. the type of connector (message-oriented infrastructures, pipes, filters, etc.). Interested readers are referred to [18,25] for such a discussion.

Aspectual Component Composition. This is the main novelty of AO-ADL connectors with respect to traditional connectors. The specification of the aspect bindings (AspectBinding in Fig. 2) inside the connector describes the connections between 'aspectual' components and 'base' components. For instance, in Fig. 4 the Replication component is an 'aspectual' component that affects the interaction between the Bank and the Account components. Concretely, in an aspectual binding (e.g. the aspectual binding with name 'BM-ReplicationBinding' in lines 23 to 36) the following information is provided:

1. **Pointcut Specification.** The description of the pointcuts identifying the join points that will be intercepted. As mentioned before these join points are defined in terms of: (1) a provided role of the connector; (2) a required role of the connector, or (3) a composition binding previously described in the same connector. In the first two cases the pointcut is expressed omitting the other party in the communication. The example of Fig. 4 is an instance of the third case, where the join point captured by the pointcut specification is the interaction between the components connected to the provided-role and required-role of the connector (which in our example would be the Bank and the Account components respectively) by means of the setBalance() operation (specified in the pointcut specification in lines 25 to 28 in Fig. 4). Pointcuts can be specified using wildcards and binary operators. The only requirement is that the pointcut specification will only refer to join points previously exposed in the roles of the connector.
2. **Binding.** A different binding section is defined for each kind of advice (before, after, around) 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.
   (a) **Operator.** The kind of advice is specified in the operator. Basic AO-ADL operators are before, after, around and concurrent-with. Operators take the form 'X before Y', where X is an advice and Y is a join point. A detailed description of all the possible AO-ADL operators as well as the semantic of each operator is out of the scope of this paper, but it is similar to other approaches. Thus, we will explain those that are used in the examples as they appear. For instance, in Fig. 4, the operator is after, indicating that the Replication component is injected always after the interaction among the Bank and the Account component.
   (b) **Constraint.** A constraint restricting the injection of aspects under certain circumstances. Constraints can be expressed as pre-conditions, post-conditions or invariants.

(c) **Advice.** For each kind of advice, the list of 'aspectual' components to be composed is specified. This makes reference to the aspectual-role clause previously defined in the connector (lines 12 to 16 in Fig. 4) and to the name of one of the operations in the referenced interface. The behavior specified by this operation is the behavior to be composed at the matched join points. In Fig. 4, the update-balance operation (line 34) of a component with a provided interface with role name BalanceReplMgm (line 14) is specified as the advice to be executed (line 32-34).

(d) **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 a set of aspects may be different depending on the context in which they are composed. The operators before, after, around and concurrent-with previously described are also used to indicate the order of execution of advice. Once again the complete list of ordering operators is out of the scope of this paper.

Notice that this composition information is defined in an aspect-oriented fashion, including common terms of aspect-orientation such as *pointcut specification*, *advice* and *advice ordering/sequencing*. Consequently, it is possible the specification of new relationships (aspect-oriented relationships) between components,

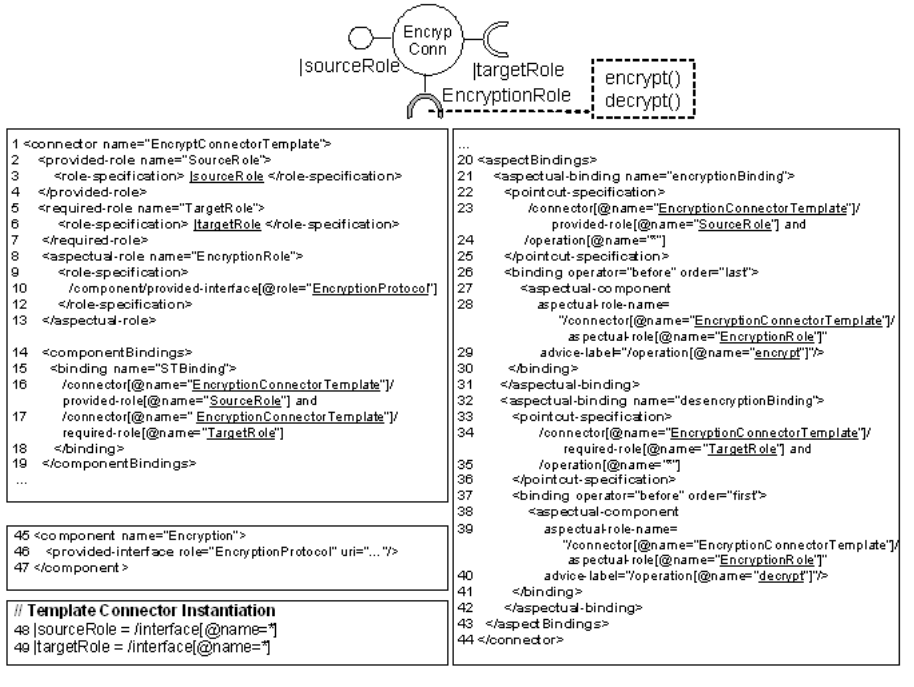


Fig. 6. Architectural Encryption Pattern

which are not supported by the component bindings usually specified inside connectors by non-aspect-oriented ADLs.

4.3 Connector Templates

Another interesting feature of AO-ADL is the definition of connector templates. The idea behind connector templates is that the definition of the connections between 'aspectual' and 'base' components from scratch is not a very practical solution since it requires a lot of effort and, potentially, it is an important locus of errors. Also, it would be very useful to have a mechanism to reuse some recurrent aspect-oriented architectural solutions. An alternative option would be that the software architect has available a set of aspect-oriented architectural patterns or templates for well-known crosscutting concerns. Then, these templates would have to be instantiated in particular software architectures.

For instance, in Fig. 5.b we have shown how to satisfy the confidentiality part of NFR3 in AO-ADL by adding the Encryption 'aspectual' component. However, since encryption is a well-known recurrent crosscutting concern, an alternative to the inclusion of this component from scratch would be the use of an AO-ADL template that indicates us how to compose this component with the rest of components in our software architecture. Concretely, we may have reused the connector template shown in Fig. 6.

The template is interpreted as follows: The behavior of the Encryption 'aspectual' component is the same independently of the particular components connected to the provided and required role of this connector (notice the use of |sourceRole and |targetRole formal parameters in lines 3 and 6 respectively). The template states that, always before sending a message the encrypt advice of the component connected to the encryption port is executed (lines 26 to 30). In case where other aspects have to be injected simultaneously to the encryption aspect, the encrypt advice will be the last one in being evaluated (see the ordering information in line 26). This makes sense since the content of the message should not be encrypted if other aspects need to access to it. Similarly, always before receiving a message, the decrypt advice is executed (lines 37 to 41). When other aspects also need to be injected in the same join point, the ordering information in line 37 indicates that the decrypt advice is the first one to be evaluated. This makes sense since the content of the message should be decrypted before any other aspect access to it.

Then, the |sourceRole and |targetRole formal parameters need to be substituted by particular interface names or queries in order to instantiate the pattern, as shown in the lower part of Fig. 6 (lines 48 and 49). Notice that the shown instantiation, using wildcards to specify both the source and the target components, corresponds to the connector previously described in Fig. 5.b.

In this example, we are assuming that all the interactions among the components need to be encrypted (notice the use of the * wildcard to specify the intercepted messages). However, this is not always the case. In general, in order to define more complex interactions, and still be able to reuse the pointcuts specification, we need to also parameterize those operations of the provided and

required roles of the connector that are going to be intercepted by the aspectual components. This is possible in AO-ADL using queries to specify the connector's roles (section 4.2).

For instance, analyzing NFR2, the Account component is a critical resource that can be simultaneously accessed by several instances of the Bank component in order to read (the Bank as a consumer) and write (the Bank as a producer) the balance of the account. This can be identified by the software architect as a typical producer/consumer synchronization problem, for which AO-ADL will provide a connector template. In this case the connector template would define: (1) a required role |reader with at least a |...read() operation; (2) a required role |writer with at least a |...write() operation; (3) a provided role |criticalResource with at least the |...read() and |...write() operations, and (4) an aspectual role |RWsynchronization with the |...beforeRead(...), |...afterRead(...), |...beforeWrite(...) and |...afterWrite(...) operations[9].

Then, in order to instantiate the template: (1) the |reader and the |writer parameters would be instantiated with a required interface of the Bank component containing the get_balance() and set_balance() operations respectively; (2) the |criticalResource parameter would be instantiated with a provided interface of the Auction component also containing the get_balance() and set_balance() operations; (3) the |RWsynchronization with a provided interface of the Concurrency component; (4) the |...read() parameter with the get_balance() operation; (5) the |write(...) parameter with the set_balance() operation, and so on. Since the specification of the pointcuts inside the connector is defined in terms of these parameters, it can be reused for different instantiations of the connector template.

Finally, notice that the use of the notation |param to identify the template's parameters is only a syntactic sugar used to simplify the description of the templates. We are now implementing an Eclipse plug-in that uses the Java Emitter Templates (JET)[10] technology for defining and instantiating the AO-ADL connector templates. JET is part of the Eclipse Modeling Framework Technologies (EMFT) and uses a JSP-like syntax to write templates, as well as the instantiation parameters[11].

4.4 The Complete Architecture of the ATM System in AO-ADL

To finish with our example we will show the complete software architecture of the ATM system presented in section 3.

Fig. 7 shows the aspect-oriented software architecture of the ATM system, which includes the components of Fig. 1 plus the components representing crosscutting concerns[12]. The 'aspectual' components are shown in grey color and

[9] The ... in |...m(...) means any parameters of any type.

[10] Web Site: www.eclipse.org/emft/projects/jet/

[11] More information about this can be found in http://caosd.lcc.uma.es/AO-ADL

[12] In this particular example the 'aspectual' components model the non-functional requirements described in section 2. In other scenarios, functional requirements may also be modeled as 'aspectual' components.

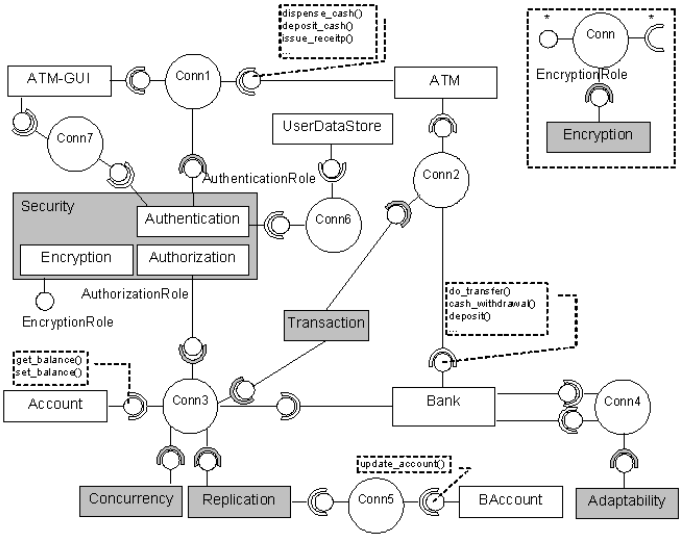


Fig. 7. AO version of the architecture of the ATM System

are the Security, the Transaction, the Concurrency, the Replication and the Adaptability components. We have omitted the XML descriptions for simplicity.

Notice that this is a high-level abstraction of the software architecture. More refined versions of these components may be also specified in AO-ADL, by decomposing each component in a set of components and interconnections among them (though it is out of the scope of this paper).

5 Main Features of AO-ADL

In this section we will outline the main contributions of the AO-ADL language and assess the advantages of these characteristics in the specification of AO architectures.

The modularity of components is improved, for either 'base' and 'aspectual' components. This is achieved by modeling crosscutting concerns as separated components (i.e. 'aspectual' components). In this sense it is possible to specify aspect-oriented architectures, solving the tangled and the scattered behavior problem earlier, at the architectural level.

The pointcut specification is not part of the 'aspectual' component definition. This is shared with most AO ADLs nowadays, including our previous work DAOP-ADL. Thus, the main contribution is that in AO-ADL the pointcut definition is part of the connector aspectual binding specification. Also, when a component plays the role of an 'aspectual' component the advice can be any operation defined as part of its provided interface. All this improves the reusability and evolution of 'aspectual' components.

AO-ADL defines a symmetric decomposition model. As previously mentioned, this feature of our language increases the possibilities of reusing a component, which may play an aspectual or non-aspectual role depending on the particular interactions in which that component participates. That means that the same component can be (re)used either as a 'base' component or as an 'aspectual' component in different software architecture specifications.

Explicit representation of all the dependencies of each aspect. Likewise components in traditional ADLs and IDLs, the 'aspectual' component specification explicitly shows all the dependencies that the 'aspectual' component has with other components by means of its provided and required interfaces. Moreover, AO-ADL provides an homogeneous way of representing such dependencies, without making distinction between 'base' and ' aspectual' components. This information is very useful for the study of the interactions among aspects [26].

Single architectural view to represent crosscutting and non-crosscutting concerns and its interactions. It is possible to clearly identify which are the concerns modeled in a particular software architecture just by looking at the component and connector view (see Fig. 7). This will improve the communication with stakeholders, as well as the evolution and maintainability of both functional and non-functional concerns. Notice that other solutions, such as the one presented in [21], use different levels of abstraction to represent different kinds of concerns. For instance, while *security* is represented as UML constraints in a class diagram, *availability* is represented by adding a new component and required interface into the system and *heterogeneity* is modeled by the use of a design pattern. Consequently, there is not a single architectural view in which all the functional and non-functional concerns of the system can be clearly identified. Notice that it is not our intention to say that other architectural views are not necessary to complete the specification of the software architecture, but that the homogeneous representation of all the concerns in the component and connector view helps to improve the understanding of software architectures and the traceability of concerns among views.

AO-ADL connector specification promotes the refinement of components interactions. Core functionality can be specified first or independently of crosscutting and non-functional concerns. Then, 'aspectual' components can be added or removed from a software architecture without affecting the connections among 'base' components. Since the binding information for 'base' and 'aspectual' component is organized in different sections of the connector, only the 'aspectBindings' section needs to be modified to add/remove 'aspectual' components. An example of this are Fig. 1 and Fig. 7, being the later one an extension of the prior, in which 'aspectual' components were added. The interconnections among components in Fig. 1 remains the same in Fig. 7.

Connector templates. Existing catalogues of crosscutting concerns define non-functional or extra-functional concerns that are typical of an application domain. Based on these catalogues, the software architect can identify aspect-oriented architectural solutions for certain recurrent crosscutting concerns. In AO-ADL

these aspect-oriented architectural solutions for recurrent crosscutting concerns can be captured by the specification of connector templates. These templates can then be reused in different architectural configurations as shown in Fig. 6.

6 Conclusions and Future Work

In this paper we have presented AO-ADL, an XML-based aspect-oriented architecture description language. The main contributions of AO-ADL are: (1) The symmetric decomposition model, which defines the component as the architectural block to model both functionality and aspectual behavior, and (2) The extended semantic of connectors to specify aspectual composition information.

Another interesting feature of AO-ADL, which improves the definition and evolution of aspect-oriented software architectures, is the definition of connector templates. Currently, we are working on the definition of a library of aspect-oriented architectural patterns. We are mainly focusing on the definition of connector templates for modeling well-known crosscutting concerns. This will allow reusing these concerns in different software architectures.

The use of XML has important advantages, such as using the built-in XML tool support to define and manipulate architectural descriptions, or the possibility for a runtime execution environment of using this information during the application execution [6]. However, it also has an important drawback regarding the readability of the software architecture in the communication with stakeholders. In order to cope with this shortcoming we are now defining a tool suite that will provide support to define, manipulate and reuse AO-ADL specifications. This tool makes use of the XML-to-Ecore mapping technology provided by the Eclipse Modeling Framework (EMF)[13] in order to provide an ECore model of AO-ADL. An additional goal is the definition of a mapping process, in the MDD (Model-Driven Development) sense, to define a correspondence between the AO-ADL ECore model and the UML 2.0 Ecore model. This correspondence bases on the mapping process from architecture to design defined in [27] and will provide a UML 2.0 standard representation of AO-ADL software architectures.

Finally, we would like to remark that AO-ADL is the aspect-oriented integrated ADL of AOSD-Europe[14]. Concretely, in this project we are now collaborating on the definition of COMPASS [20,28], an approach that offers a systematic means to derive an aspect-oriented architecture, described in AO-ADL, from a given aspect-oriented requirements specification, described in RDL (Requirements Description Language) [29]. This approach is a constituent part of the aspect-oriented mapping process that is being defined in this project from requirements to architecture and from there to design and implementation [27]. Concretely, the AO-ADL tool support discussed in the previous paragraph is being defined as part of this project.

[13] Web Site: www.eclipse.org/emft/projects/

[14] European Network of Excellence on AOSD.

Acknowledgment

We are very grateful to the anonymous referees for their insightful suggestions, that greatly helped improving the contents of the paper. This work is supported by European Commission FP6 Grant AOSD-Europe (IST-2-004349), European Commission STREP Project AMPLE (IST-033710), and Spanish Commission of Science and Technology (TIN2005-09405-C02-01).

References

1. Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transaction on Software Engineering 26(1), 70–93 (2000)
2. Aspect-Oriented Software Development Web Site, `http://www.aosd.net`
3. Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto, M., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A.: Survey of (Aspect-Oriented) Analysis and Design Approaches. AOSD-Europe project Report AOSD-Europe-ULANC-9 (2005)
4. Krechetov, I., Tekinerdogan, B., Pinto, M., Fuentes, L.: Initial Version of Aspect-Oriented Architecture Design Approach. AOSD-Europe project report AOSD-Europe-UT-D37 (February 2006)
5. Pinto, M., Fuentes, L., Troya, J.M.: DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. In: Pfenning, F., Smaragdakis, Y. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 118–137. Springer, Heidelberg (2003)
6. Fuentes, L., Pinto, M., Troya, J.M.: Supporting the Development of CAM/DAOP Applications: an Integrated Development Process. Software Practice and Experience 37(1), 21–64 (2007)
7. Pessemier, N., Seinturier, L., Coupaye, T., Duchien, L.: A Model for Developing Component-Based and Aspect-Oriented Systems. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, Springer, Heidelberg (2006)
8. Garlan, D., Monroe, R., Wile, D.: ACME: An Architecture Description Interchange Language. In: Proc. of CASCON 1997 (November 1997)
9. Medvidovic, N., Oreizy, P., Robbins J.E., Taylor, R.N.: Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In: Proc. of ACM SIGSOFT 1996, USA, pp. 24–32 (October 1996)
10. Luckham, D., et al.: Specification and Analysis of System Architecture Using Rapide. IEEE Trans. Soft. Eng. 21(4), 336–355 (1995)
11. Batista, T., Chavez, C., Garcia, A., SantAnna, C., Kulesza, U., Rashid, A., Filho, F.C.: Reflections on Architectural Connection: Seven Issues on Aspects and ADLs. In: Proc. of EA 2006, China (May 2006)
12. Pessemier, N., Seinturier, L., Duchien, L.: Components, ADL and AOP: Towards a Common Approach. In: Proc. of the Workshop ECOOP RAMSE 2004 (June 2004)
13. Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E.: PRISMA: Towards Quality, Aspect-Oriented and Dynamic Software Architectures. In: Proc. of 3rd IEEE Intl Conf. on Quality Software, USA (November 2003)
14. Garcia, A., Chavez, C., Batista, T., SantAnna, C., Kulesza, U., Rashid, A., Lucena, C.: On the Modular Representation of Architectural Aspects. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 82–97. Springer, Heidelberg (2006)

15. Suvée, D., De Fraine, B., Vanderperren, W.: A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development. In: Gorton, I., Heineman, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 114–122. Springer, Heidelberg (2006)
16. Navasa, A., Pérez, M.A., Murillo, J.M.: Aspect Modelling at Architecture Design. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 41–58. Springer, Heidelberg (2005)
17. Kandé, M.M., Strohmeier, A.: On The Role of Multi-Dimensional Separation of Concerns in Software Architecture. In: Proc. of OOPSLA 2000 Workshop on Advanced SoC in Object-Oriented Systems, USA (October 2000)
18. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proc. of the 22nd ICSE 2000, Ireland, pp. 178–187. ACM Press, New York (2000)
19. Pinto, M., Gámez, N., Fuentes, L.: Towards the architectural definition of the Health Watcher system with AO-ADL. In: Proc. of the Early Aspects Workshop at ICSE 2007, May, Minnesota, USA (2007)
20. Chitchyan, R., Pinto, M., Rashid, A., Fuentes, L: COMPASS: Composition-Centric Mapping of Aspectual Requirements to Architecture. Accepted for publication in TAOSD: Special Issue on Early Aspects
21. Choi, H., Yeom, K.: An Approach to Software Architecture Evaluation with the 4+1 View Model of Architecture. In: Proc. of the Ninth APSEC 2002 (2002)
22. Allen, R., Dounce, R., Garlan, D.: Specifying and Analyzing Dynamic Software Architectures. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, Springer, Heidelberg (1998)
23. Dashofy, E.M., Hoek, A., Taylor, R.N.: An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In: Proc. of the 24th International Conference on Software Engineering (ICSE'02), Orlando, Florida
24. Fuentes, L., Gámez, N., Pinto, M.: DAOPxADL: An extension of the xADL Architecture Description Language with Aspects. In: Proc. of the DSOA 2006 workshop collocated with JISBD 2006, Spain (2006)
25. Shaw, M., DeLine, R., Zelesnik, G.: Abstractions and Implementations for Architectural Connections. In: Proc. of (ICCDS 1996) (1996)
26. Sanen, F., et al.: Study on interaction issues AOSD-Europe Project Report, AOSD-Europe-KUL-7 (February 2006)
27. Chitchyan R., et al.: Mapping and Refinement of Requirements Level Aspects. AOSD-Europe project report No: AOSD-Europe-ULANC-24 (November 2006)
28. Chitchyan, R., Pinto, M., Fuentes, L., Rashid, A.: Relating AO Requirements to AO Architecture. In: Proc. of the Early Aspects Workshop(OOPSLA'05), USA (2005)
29. Chitchyan, R., Sampaio, A., Rashid, A., Sawyer, P., Khan, S.: Initial Version of Aspect-Oriented Requirements Engineering Model. AOSD-Europe project report No: AOSD-Europe-ULANC-17 (February 2006)