Reflections on Architectural Connection: Seven Issues on Aspects and ADLs

Thaís Batista[1] Christina Chavez[2] Alessandro Garcia[3]
Cláudio Sant'Anna[4] Uirá Kulesza[4] Awais Rashid[3] Fernando Castor Filho[5]

[1]Computer Science Department, Federal University of Rio Grande do Norte – UFRN, Brazil
[2]Computer Science Department, Federal University of Bahia – UFBA, Brazil
[3]Computing Department, Lancaster University, United Kingdom
[4]Computer Science Department, Pontifical Catholic University of Rio de Janeiro – PUC-Rio, Brazil
[5]Institute of Computing, University of Campinas – UNICAMP, Brazil
thais@ufrnet.br, flach@ufba.br, {garciaa, marash}@comp.lancs.ac.uk
{claudios, uira}@inf.puc-rio.br, fernando@ic.unicamp.br

ABSTRACT
Abstractions to express architectural connection play a central role in architecture design, especially in Architecture Description Languages (ADLs). With the emergence of aspect-oriented software development (AOSD), there is a need to understand the adequacy of ADLs' conventional connection abstractions to capture the crosscutting nature of architectural concerns. This paper reflects on seven issues pertaining to the interplay of crosscutting concerns and architectural connection abstractions. We review and assess the design of existing aspect-oriented (AO) and non-AO ADLs with respect to these issues. A case study is used to illustrate our viewpoints, claims, and proposals.

Categories and Subject Descriptors
D.2.11 Software Architectures: Languages (e.g., description, interconnection, definition)

General Terms
Design, Languages

Keywords
Software Architecture, Architecture Description Languages, Aspect-Oriented Software Development.

1. INTRODUCTION
Software Architecture Description Languages (ADLs) have been recognized as an important tool for supporting the systematic reasoning about system components and the connections between them early in the development process. Architectural connection comprises the elements involved in component interactions, such as interfaces, connectors, and architectural configurations. The goal is to provide software architects with means to express a plethora of heterogeneous, complex interconnection styles in a way that is agnostic to underlying composition implementation mechanisms, such as inheritance, method calls, and so forth.

With the emergence of AOSD [6], there is a need to reflect whether fundamental architectural connection abstractions provide the necessary means to modularize crosscutting concerns [16] ,[3]. A crosscutting concern at the architecture design level could be any concern that cannot be effectively modularized using the given abstractions of an ADL, leading to increased maintenance overhead, reduced reuse capability and generally resulting in architectural erosion over the lifetime of a system [19], [3]. Therefore, we need to understand to what extent software architects are able to modularly specify crosscutting concerns and their inter-connection with other architectural elements.

In order to support modularization of crosscutting concerns, some AO ADLs [13], [15] have been proposed, either as extensions of existing ADLs or developed from scratch employing AO abstractions such as, aspects, joinpoints, pointcuts, advice, and inter-type declarations, commonly adopted in programming frameworks and languages (e.g., [2]). Though these AO ADLs are interesting first contributions and viewpoints in the field, there is little consensus on how AOSD and ADLs should be integrated, especially with respect to the interplay of aspects and architectural connection abstractions. There is little reflection, to date, on how and why extensions are required to traditional notions of interconnection ADL elements, such as interfaces, connectors, and architectural configurations.

This paper presents our viewpoint on seven critical issues relating to the integration of AOSD and ADLs. Our goal here is not to come up with an entirely new ADL. Instead, we reflect on whether the presence of crosscutting concerns requires extensions to conventional architectural abstractions. For each issue, we sketch a proposed solution whenever existing AO and non-AO ADLs do not provide an adequate solution. We illustrate our arguments with a tourist guide system.

EA'06, May 21, 2006, Shanghai, China.

The remainder of this paper is organized as follows. Section 2 overviews a widely-accepted conceptual framework for ADLs, and existing non-AO and AO ADLs. Section 3 presents our reflections on the seven chosen issues related to aspects and architectural connection. Section 4 presents the final remarks.

2. Architecture Description Languages

In this section we present the basic concepts of Architectural Description Languages (ADLs) stated on the Medvidovic and Taylor [11] framework and introduce ACME, a well-known ADL that we will be used in our example. We also present some AO ADLs in order to discuss how they support AOSD concepts.

2.1 Non-AO ADLs

According to the classification framework proposed by Medvidovic and Taylor, the building blocks of an architectural description are components, connectors, and architectural configurations. Components and connectors may have associated interfaces, types, semantics and constraints, but only explicit component interfaces are a required feature for ADLs. A component's interface is a set of interaction points between it and the external world. It specifies the services (messages, operations, and variables) a component provides and also the services it requires of other components. Connectors model interactions among components and specify rules that govern those interactions. A connector's interface specifies the interaction points between the connector and the components and other connectors attached to it. It enables proper connectivity of components by exporting as its interface those services it expects of its attached components. Configurations define architectural structure and how components and connectors are connected.

ACME [8] is a general purpose ADL whose goal is to support the interchange of architectural descriptions. It was designed to consider the essential concepts common to different ADLs as well as to allow extensions to include other elements. The basic elements of ACME are components, connectors and attachments. Components are the computational elements whose interface is represented by *ports*. Connectors model interactions among components and have a set of interfaces named *roles*. The configuration of a system is defined by listing a set of attachments that bind component ports to connector roles. ACME elements may also be annotated with additional properties.

2.2 AO ADLs

Most AO ADLs are motivated by the integration of existing ADL concepts (such as, components, interfaces, connectors and configurations) with new AO abstractions (such as, aspects, joinpoints, pointcuts and advices) in order to address the modeling of crosscutting concerns in architecture.

Pinto et al [15] propose DAOP-ADL with components and aspects as first-order elements. Aspects can affect the components' interfaces by means of: (i) an *evaluated interface* which defines the messages that aspects are able to intercept; and (ii) a *target events interface* responsible for describing the events that an aspect can capture. The composition between components and aspects is supported by a set of aspect evaluation rules. They define when and how the aspect behavior is executed.

In the Prisma approach [13], aspects are new ADL abstractions used to define the structure or behavior of architectural elements (component and connectors), according to specific system viewpoints. Components and connectors include a weaving specification that defines the execution of an aspect and contains weaving operators to describe the temporal order of the weaving process (after, before, around, and others).

Pessemier et al [14] extend the Fractal ADL with Aspect Components (ACs). ACs are responsible for specifying existing crosscutting concerns in software architecture [4]. Each AC can affect components by means of a special interception interface. Two kinds of bindings between components and ACs are offered: (i) a direct crosscut binding by declaring the component references and (ii) a crosscut binding using pointcut expressions based on component names, interface names and service names.

Navasa et al [12] define a set of requirements which current ADLs need to address to allow the management of crosscutting concerns using architectural connection abstractions. The requirements are: (i) definition of primitives to specify joinpoints in functional components; (ii) definition of the aspect abstraction as a special kind of component; and (iii) specification of connectors between joinpoints and aspects. The authors suggest the use of existing coordination models to specify the connectors between functional components and aspects.

The above discussion shows that there is a diversity of viewpoints on how aspects (and generally concerns) should be modeled in ADLs. However, so far, the introduction of AO concepts into ADLs has been experimental in that researchers have been trying to incorporate mainstream AOP concepts into ADLs. Though this provides interesting insights into the problem of modeling crosscutting concerns at the architecture level, first and foremost we need to understand the various issues pertaining to such modeling. Any adaptation of existing ADLs or engineering of new ADLs needs to be based on a clear understanding of new challenges aspects pose at the architecture level that cannot be handled with existing ADL abstractions. In the following, we highlight seven issues as a roadmap for integration of AO concepts into ADLs.

3. Seven Issues on Aspects and Architectural Connection

We focus on seven key issues that arise when relating crosscutting concerns and ADL abstractions. Of course, there are other issues that need to be studied and analyzed. However, we have chosen to focus on these seven because (i) they involve elements (e.g., module interfaces), that have been discussed in AO implementation approaches (such as [1],[17]), which are also architecturally relevant, (ii) they have recurrently been a hotspot or point of controversy in the existing AO architectural solutions. We also concentrate on revisiting traditional definitions of architectural connection abstractions in the presence of crosscutting concerns. The first issue (Section 3.1) is dedicated to discussing which interconnection elements in an architectural description are typically affected by a crosscutting concern. This discussion provides the foundation for the following six issues (Sections 3.2 – 3.7), which examine the adequacy of using conventional notions of connection abstractions in the presence of crosscutting concerns. The last issue is concerned with the need of new abstractions for aspects at the architectural level.

All the last six debated topics follow a similar structure: (i) they revisit the well-known definition of the abstraction being discussed according to Medvidovic and Taylor's framework, (ii)

they present our position and arguments on whether extensions or redefinitions of conventional concepts are needed due to the presence of crosscutting concerns, (iii) they illustrate our viewpoint using the running example to be presented below, (iv) they analyze solutions adopted by existing ADLs, and (v) they present a preliminary proposal whenever existing AO and non-AO ADLs do not provide an adequate solution according to our perspective. We have decided to use ACME as our base ADL due to its generality.

Figure 1 introduces the example of a context sensitive tourist guide that we will be using throughout our discussions. The tourist guide is available on a handheld device.

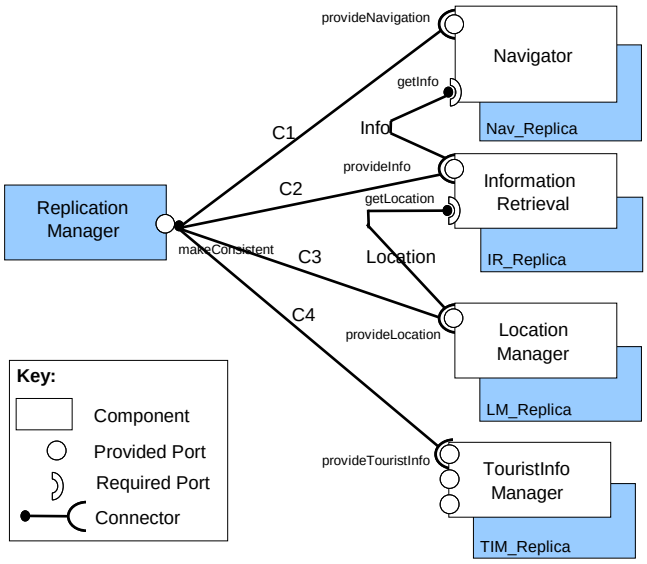

Fig. 1. Tourist Information Guide (TIG)

The visitor uses a *Navigator* to create a customized tour, to navigate through a tour and to update information about the navigation preferences. The *Navigator* component contacts the *InformationRetrieval* component to retrieve information from the system. The *LocationManager* component provides the identification of the current location of a visitor. This identification is used by *InformationRetrieval* to provide tourist information according to his/her current location.

The *TouristInfoManager* component allows the tourist centre to update information in the system. Availability is a crosscutting concern as it affects these four components. In order to support availability it is necessary to replicate components and to make the replicas consistent. *ReplicationManager* implements these tasks. It implements a synchronization protocol that synchronizes the primary component with its replica. The *makeConsistent* port applies this protocol.

Figure 2 illustrates the ACME description of the example. Note that ACME, as a lot of non-AO ADLs, does not provide support to avoid the duplication of the architectural connections in the specification. This problem affects the readability and comprehension of the architectural description. Furthermore, it also lacks primitives for describing temporal issues such as when a composition must be applied.

3.1 Issue 1: Which ADL elements can have crosscutting concerns?

ADL elements support the explicit specification (and possibly modularization) of some architectural concerns through the use of different categories of architectural elements such as components, connectors and architectural configurations. According to Medvidovic and Taylor framework, components, connectors, configurations and component interfaces constitute a minimum set of required features for ADLs (Section 2.1). Therefore, we adopt them as candidate ADL elements that can be affected by architectural crosscutting concerns at well-defined points. In order to discuss our viewpoint, we will resort to the description for the Tourist Information Guide (TIG) system (Figure 2).

```
Component Navigator = {
 Port provideNavigation
 Port getInfo }

Component InformationRetrieval = {
 Port provideInfo
 Port getLocation }

Component LocationManager ={
 Port provideLocation  }

Component TouristInfoManager = {
 Port provideTouristInfo  }

Component ReplicationManager= {
 Port makeConsistent }

Connector Type RemoteInvocation =
 { Roles caller, callee }

Connector Info, Location: RemoteInvocation = new RemoteInvocation;
Connector C1, C2, C3, C4: RemoteInvocation = new RemoteInvocation;
Attachments {
 Info.caller to Navigator.getInfo
 InfoRetrieval.provideInfo to Info.callee
 InfoRetrieval.getLocation to Location.caller
 LocationManager.provideLocation to Location.callee

 ReplicationManager.makeConsistent to C1.callee
 Navigator.provideNavigation to C1.caller
 ReplicationManager.makeConsistent to C2.callee
 InformationRetrieval.provideInfo to C2.caller
 ReplicationManager.makeConsistent to C3.callee
 LocationManager.provideLocation to C3.caller
 ReplicationManager.makeConsistent to C4.callee
 TouristInfoManager.provideTouristInfo to C4.caller   }
```

Fig. 2. ACME Description

The specification of an architectural configuration for the TIG system, expressed in ACME (Figure 2), includes the definition of components and their ports connected by different connector instances, the definition of a connector type and a listing of attachments that bind component ports to connector roles. *ReplicationManager* (*RM*) is defined as an ACME component with one port *(makeConsistent*). These ADL elements provide good support for the separation of some architectural concerns. The connector type *RemoteInvocation* localizes the communication protocol among the primary components and their distributed replicas, promoting reuse and enhancing comprehension. *RM*, for instance, includes a synchronization protocol that could be separated and modularized by a new connector type.

However, there are some situations that suggest that different ADL elements may be the subject of tangling and scattering of concerns and therefore crosscutting concerns may potentially exist and affect them. Suppose that all the information transmitted between the replica and the replicated using *RemoveInvocation* connectors must be compressed. This requirement would demand the composition of connectors, a feature that is not supported by

most ADLs. The designer could refine the *RemoteInvocation* connector type, keeping its roles but modifying its glue specification, so that the outgoing information is compressed before it calls *RM* and decompressed before being delivered. This requirement is an example of an architectural concern that may cut across one or more connector elements. Additionally, a new requirement may demand that several ADL elements are subject to new constraints. These constraints may be scattered and tangled up within components and connectors. Finally, since architectural configurations can be regarded as composite components, they may also be affected by crosscutting concerns.

Existing AO ADLs vary on the decision about the kinds of ADL elements that can be affected by crosscutting concerns. A related issue that must be considered is the coverage of these ADLs. Lack of coverage means that only a subset of the required features are considered by the ADL for explicit architectural description. DAOP-ADL does not consider connectors and configurations as building blocks, only components. The Aspect Components approach supports components and configurations, but not connectors; crosscutting concerns may affect components. Prisma covers components, connectors, and systems. Crosscutting concerns can affect components and connectors. Table 1, at the end of the paper, summarizes these decisions.

3.2 Issue 2: Composition

In software architecture there is a consensus that a *software connector* is the element that mediates interactions between components. By providing distinct architectural abstractions to specify computation and interconnection between components, software architecture descriptions promote the idea of separating concerns (SoC). The integration of software architecture and AOSD may take advantage of this SoC approach and use connectors to model interaction between two parts, regardless of the nature of the two parts involved: two traditional components or a traditional component and a component that represent a crosscutting concern. Connectors can model simple or complex interaction protocols and they are used in various contexts.

Based on the wide use of connectors for different interconnection purposes, we propose a *connection-based approach* in order to model the composition between "regular" components and "aspectual" components. In this approach connectors and configuration explicitly support the compositional model.

In ADLs valid configurations are those that connect provided and required services or event announcer and event receiver. As a crosscutting concern is represented by a provided service of an "aspectual" component (in our example the *makeConsistent* port is a provided port) and as it can affect provided services of other components (in our example the provided ports of the components are affected by the *makeConsistent* port), the traditional semantics of architectural connection cannot be applied in this case. In Fig. 2 these connections were represented because ACME does not distinguish provided and required ports. But most ADLs make this distinction and it is impossible to represent such kind of connection.

Our proposal is to extend the *connector interface* in order to specify a *base role* and a *crosscutting role* and also to define a *glue* clause to specify details about the connection such as the place where the advice must affect the "regular" component – after, before, or around. The base role may be connected to the port of a component and the crosscutting role may be connected to a port of an "aspectual" component. The distinction between base and crosscutting roles addresses the constraint typically imposed by many ADLs about the valid configurations between provided and required ports. The *base-crosscutting roles* dichotomy does not impose such semantic constraint. A crosscutting role defines the place at which an "aspectual" component joins a connector to affect a component. This connector with extension to support composition of "regular" components and "aspectual" components is called *aspectual connector*. As the same crosscutting concern can affect several elements in a different way, an aspectual connector can have a crosscutting role and multiple base roles.

The *configuration* also has an important function in our compositional model. It defines the connection between components, connectors and "aspectual" components. Thus, at the configuration description are defined the join points at which an "aspectual" component acts. The join points are specified in the definition of the association between the *baseRole* of a connector with a given element of the component interface. This element of the component interface is the join point where the advice acts. In fact, the concept of configuration already defines the point where a component joins a connector. In our approach we are just taking advantage of this concept to identify the join points affected by a crosscutting concern.

For instance, Figure 3 illustrates the composition of *ReplicationManager* (*RM*) with other components. The *Synchronizer* connector is defined to mediate the interactions between the "aspectual" component - *RM* - and the other components. It defines a *base role* and a *crosscutting role* and the glue that specifies where the "aspectual" component will affect the join points (*around*). The attachments section defines that the *base role* of Synchronizer is linked to all components with a port whose name begins with *provide*. This means that the synchronization protocol implemented by *RM* (via its *makeConsistent* port) is applied during the invocation of these ports in order to synchronize the components and their replicas.

Component ReplicationManager =
{*Port* makeConsistent}
Connector Synchronizer =
{ *baseRole* sink
crosscuttingRole source
glue around}
Attachments {
ReplicationManager.makeConsistent to Synchronizer.source
Synchronizer..sink to *.provide* }

Fig. 3. ACME Description of the Composition

Although there is no consensus about how to model the aspectual composition, there are some works that also advocate the use of connectors [9] for this purpose. Fractal is a component model with an XML-based ADL that models binding between components and aspects in the same way two components are bound. Navasa et al. also advocate the use of connectors for composition purposes. However, they propose the definition of primitives to specify join points in functional components. In contrast, we argue that the configuration part already supports the definition of join points. Thus, no new primitive is needed for this task. DAOP-ADL defines a new XML element - a set of

aspect evaluation rules - where the composition is defined. In contrast, we argue that existing SA abstractions are enough to model the composition. In PRISMA the composition of aspects is defined inside components or connectors, in the weaving specification. We consider that this approach adds complexity to the architectural description by scattering the weaving information inside the architectural elements that the aspect affects. This approach contrasts with our proposal that follows the traditional way of modeling composition in ADLs: using connectors and configuration description.

3.3 Issue 3: Quantification

In order to avoid the need to refer to each join point explicitly in an architectural description, it is necessary to use a quantification mechanism [7] that provides a single statement to reach several join points. As the configuration part is the place where static structural join points are identified, the quantification mechanism is defined in this part. Some means to support quantification must be defined including wildcards and logical expressions. The quantification must be used in the connection of the base role with the target component. In Figure 3 the connection between the Synchronizer connector uses wildcards (*) to specify that the *sink* role is linked to all components that offer a port whose name begins with provide.

In Fractal ADL with Aspect Components, pointcut expressions may use component names, interface names and method names. Navasa states that the quantification depends on the coordination model adopted by the ADL. Prisma and DAOP-ADL do not address quantification.

3.4 Issue 4: Aspect Interfaces

Is the traditional notion of interfaces used in ADLs appropriate to represent the “contract” between an “aspectual” component and the affected ones? Hence the question is whether (i) the connection-based extensions, as discussed in the Sections 3.2 and 3.3, are enough to properly capture the crosscutting connection, or (ii) interfaces of “aspectual” components also need to expose extra information relative to their crosscutting nature and, as a consequence, need to be extended.

This question is of paramount importance because interfaces play a central role in architectural modular reasoning. A component's interface is a set of interaction points. An interface thus defines computational commitments a component can make and constraints on its usage (Section 2). Modular reasoning [10] about an architectural component X means being able to make decisions about X while looking only at its interfaces, and the description of connectors and architectural configurations that describe the association of X with other components..

In this context, our position is that the notion of architectural interfaces should not be changed to express the boundaries of an “aspectual” component. From our point of view, the contract of an architectural component with its surrounding environment, independently from the nature of the concerns it addresses, should not be impacted by the way it collaborates with the rest of the architecture. At the architectural level, we already have suitable connection abstractions for specifying such a collaboration protocol. The computational commitments expressed by an interface should not specify "how" and "with whom" the associated component should be connected to. It is the interaction of a component with others, in the specific context of a system, that typically determines the "crosscuttingness" of a given concern. A certain concern can be crosscutting according to its involved interconnections for some systems' architectural specifications, but may be not in others.
In our case study, there is a unique “aspectual” component in charge of managing the creation and synchronization of the replicas (i.e. the crosscutting concern). There is no need for having extra information in the new component's interface for the sake of expressing its commitments with the external world: provided services express the creation and synchronization capabilities, constraints can be used to define a limit on the number of replicas, events could depict certain relevant state transitions in the replication process, and so forth.
Concluding, the presence of crosscutting concerns in an architecture design leads to new aspectual connectors, but interfaces and components remain the same. And, once the design of the aspectual connectors is known, the component interfaces can be identified, and, modular architectural reasoning is achieved. In our opinion, this perspective meets the underlying principles of software architecture and is simpler than some more radical proposals. As discussed in Section 2.2, DAOP-ADL and Fractal adopt a different notion of interfaces. Also, in our previous work [5], we have defined a UML-based architectural specification language for representing architectural aspects. The language included the concept of crosscutting interfaces [5] as a means to capture the crosscutting influence of certain components. However, this new notion of interfaces emerged from the fact that UML 1.4 provided no abstractions (e.g. connectors), to support the proper representation of crosscutting collaborations, as commonly supported by existing ADLs.

3.5 Issue 5: Join Point Exposition

As previously mentioned, architectural crosscutting concerns are represented by components. We claim that the notion of ADLs’ interface is rich enough and does not need to be changed in order to expose the crosscutting nature of the component. An “aspectual component”, which represents a crosscutting concern, affects other components by means of connectors. In this context, the issue is whether interfaces of the affected components need to expose extra information to allow the connection with “aspectual components” and, as a consequence, the traditional concept of interfaces in ADLs needs to be extended. In other words, is the traditional notion of interface appropriate to expose the join points in the affected components where the “aspectual components” will be connected?

In order to discuss this issue, we will revisit well-known works [1], [17] related to the exposition of component/module join points in the context of aspect-oriented programming (AOP). Some approaches [1], [17] criticize the obliviousness property [6] and propose the preparation of the base code for the application of aspects. Sullivan et al [17] propose the definition of interfaces between aspects and advised code. These interfaces are based on design rules, which govern how base code has to be written to reveal specified join points and how aspects can use these interfaces. Aldrich [1] proposes Open Modules, as modules whose interfaces, besides exporting data structures and functions, also export join points denoting internal semantic events.

Following this idea, we claim that, at the architectural level, the architect should prepare the component to be affected by

"aspectual components" by exposing in the components' interface the information necessary for composing an "affected component" with other components representing an architectural crosscutting concern. Hence, the ADLs should support the exposition of join points in the component interfaces.

According Medvidovic and Taylor's framework, a component's interface specifies the services a component provides. ADLs may also provide facilities for specifying component needs as required interfaces. Navasa et al claim that, to expose join points in components, ADLs should provide new primitives. However, we advocate that the concept of interfaces, as defined by Medvidovic and Taylor and supported by existing ADLs, already provides an expressive way to externalize join points, because it supports the exposition of a component's internal events. Thus, the concept of interface does not need to be extended.

Suppose that in our case study, the "aspectual component" *ReplicationManager* (*RM*) needs to synchronize the replica of the *LocationManager* (*LM*) component in order to maintain the consistency of the information about the localization of the tourists. In this way, the architect should create a port in *LM* that exposes the event of changing the location of the tourists. *RM* will be connected to this port to observe the occurrence of this event. Whenever the location of a tourist is changed in *LM, RM* listens the event and synchronizes the replicas.

To the best of our knowledge, existing ADLs do not propose extensions of the interface concept in order to expose component join points. DAOP-ADL exposes a component's join points by means of its required interfaces. This kind of interface specifies the output messages and events that a component is able to produce. In Fractal ADL aspects advise method calls and method execution in the interfaces. Therefore, the interfaces used in these two ADLs are complaint with Medvidovic and Taylor's concept of interface. In Prisma, aspects are used to define completely the structure or behavior of architectural elements. The Prisma approach differs from ours because their aspects have direct access to all properties of a component or connector in order to allow the behavior definition of that element.

3.6 Issue 6: Interface Enhancements

Some AO implementation approaches, such as AspectJ [2], also assign to the aspect the ability of changing the type and interface of the modules through the so-called *inter-type declarations*. From an architectural perspective, this would mean that "aspectual" components may enhance the component interfaces with new elements, such as services and attributes. Our viewpoint is that such a feature is not necessary to capture crosscutting concerns at architectural specifications. If a service or attribute relative to a concern is tangled with other services and attributes in a given component interface, we can easily rely on our notion of aspectual connector to express it in a more modular way.

From the approaches investigated, only Prisma proposes a model which seems to be related to the use of aspect introduction at the architecture level. It allows the refinement of the component and connector properties through the use of an aspect abstraction.

3.7 Issue 7: How to Represent Aspects?

Most AO ADLs and architecture modeling approaches propose the introduction of a new abstraction to represent aspects at the architecture level. The key question is whether the introduction of a new abstraction is essential given the fact that aspects are after all not that different from other components in the system. The key distinction between aspects and regular components is in the way aspects compose with the rest of the system – the scope of the composition is broad and affects multiple architectural elements. Do we really need to introduce yet another abstraction at the architecture level resulting in the need to train architects in the syntax and semantics of such an abstraction?

Aspect compositions can be modeled by means of connectors, crosscutting roles and base roles, without introducing new abstractions. In fact, our proposal is not out of sync with current trends in AOSD. Approaches such as JBoss and AspectWerkz represent aspects as classes with advice as methods in those classes. A composition specification (often in XML) captures how the "aspect classes" compose with other classes in the system. No new abstractions are introduced as is, for instance, the case for AspectJ. We can also observe a similar notion in DAOP-ADL. Though DAOP-ADL has an explicit abstraction as an aspect, at a conceptual level it is not different from the regular component abstraction apart from how aspects are composed. In fact, components and aspects extend the same abstract class in the DAOP-ADL model, hence indicating that components and aspects are substitutable wherever a component of their super-type is required. GluonJ [17] uses a unified abstraction to represent both classes and aspects. One might argue that an aspect abstraction is needed for typing purposes. We are of the opinion that such typing constraints are best captured in connectors and roles. Treating aspects the same as other components in the architecture also provides a uniform approach which is much closer to the notion of a multi-dimensional separation of concerns [19]. Such a multi-dimensional separation allows architects to undertake analysis of architectural trade-offs from multiple perspectives hence facilitating more informed architectural decisions to guide the design.

4. FINAL REMARKS

In this paper we discussed seven key issues about the integration of AOSD and ADLs. Our proposal advocates that no new architectural abstractions are needed to represent aspects. Regular components are used for this purpose. In addition, we have argued that no changes are required in components interfaces. Our proposal defines a composition model that takes advantage of existing architectural connection abstractions – connectors and configuration – and extends them to support the definition of some composition facilities such as a quantification mechanism. In this way, it avoids introducing complexity in the architectural description and comparing with the existing solutions (Table 1), we identified a reduced set of required extensions to deal with architectural crosscutting concerns. As a result the architects can model crosscutting concerns using the same abstractions, with minor adaptations, used in the conventional ADL description. As such our proposal is based on enriching the composition semantics supported by architectural connectors instead of introducing new abstractions that elevate programming language concepts to the architecture level. Our proposal, therefore, supports effective modeling of crosscutting concerns without introducing additional complexity into the architecture specification.

ISSUE >> ADL	Issue 1 Crosscutting Concerns	Issue 2 Composition	Issue 3 Quantification	Issue 4 Aspect Interface	Issue 5 Join Point Exposition	Issue 6 Interface Enhancements	Issue 7 Aspects in ADLs
Our proposal	Can be found in components, connectors and configurations	Modeled by connectors with base and crosscutting roles and by configurations	Supported by wildcards and logical operators defined at the configuration section	No extension required	Components and connectors can expose their internal events	Not supported	Aspects are modeled by means of connectors and aspectual and base roles.
Prisma	Can be found in components, connectors and configurations	Components, connectors and configurations are defined in terms of aspects.	Not addressed	Not addressed	Aspects have direct access to all properties of a component or connector.	Aspects allow the refinement of component and connector props.	Aspects define the semantic of components and connectors.
Fractal	Can be found only in components	Supports direct crosscut binding and crosscut binding using pointcuts.	Pointcut expression can be expressed in terms of component , interface and method names.	Interception interface allows binding between aspects and components.	Outgoing and incoming component calls can be exposed	Not supported	Aspect Components specify the crosscutting concerns.
DAOP-ADL	Can be found only in components	Supported by a set of aspect evaluation rules	Not supported	Evaluation interfaces Target events interfaces.	No extension required, interface components can expose join points	Not supported	A separated specification with a evaluation interface
Navasa Proposal	Can be found only in components.	Uses connectors between join points and aspects.	Depends on the coordination model adopted.	Only fix that aspects must have semantics different from components.	Creates new primitives to specify component join points.	Not supported.	Aspect is a new component with a different semantic.

Table 1. ADL Support for Aspect-Oriented Architectural Elements

5. REFERENCES

[1] Aldrich, J. Open modules: Modular reasoning about advice. In Proc. of the European Conf. on Object-Oriented Programming (ECOOP'05), 144-168, July 2005.

[2] AspectJ Team. The AspectJ Programming Guide. http://eclipse.org/aspectj/.

[3] Baniassad, E. et al. Discovering Early Aspects, IEEE Software, 2006.

[4] Bass, L., Clements, P., Kazman, R., Software Architecture in Practice: Addison-Wesley, 1998.

[5] Chavez, C. et al. Taming Heterogeneous Aspects with Crosscutting Interfaces. *Journal of the Brazilian Computer Society*, SBC, Jan 2006.

[6] Filman, R., Elrad, T.., Clarke, S., Aksit, M. Aspect-Oriented Software Development. Addison-Wesley, 2005.

[7] Filman, R. and Friedman, D. Aspect-oriented programming is quantification and obliviousness. In OOPSLA Workshop on Advanced Separation of Concerns, Minneapolis, 2000.

[8] Garlan, D. et al. ACME: An Architecture Description Interchange Language,º Proc. CASCON'97, Nov. 1997.

[9] Kandé, M. Strohmeier, A. Modeling Crosscutting Concerns using Software Connectors. ASoC3. Florida, 2001

[10] Kiczales, G., Mezini, M. Aspect-Oriented Programming and Modular Reasoning. In Proceedings of ICSE'05, 2005.

[11] Medvidovic, N., Taylor, R. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Soft. Eng., 26(1):70-93, Jan 2000.

[12] Navasa, A. et al. Aspect Oriented Software Architecture: a Structural Perspective. Workshop on Early Aspects, AOSD'2002, April 2002.

[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 - QSIC 2003, Dallas, November (2003).

[14] Pessemier, N., Seinturier, L., Duchien, L. Components, ADL and AOP: Towards a Common Approach. In Workshop ECOOP Reflection, AOP and Meta-Data for Software Evolution (RAM-SE04), June 2004.

[15] Pinto, M., Fuentes, L., Troya, J., A Dynamic Component and Aspect Platform, The Computer Journal, 401-420, 2005.

[16] Quintero, C., et al. Architectural Aspects of Architectural Aspects. Proc. of European Workshop on Software Architecture (EWSA)- Pisa, Italy, June 2005, LNCS 3527.

[17] Rajan, H. and Sullivan, K., Classpects: Unifying Aspect- and Object-Oriented Language Design, In Proc. of ICSE 2005, 2005, USA

[18] Sullivan, K.., Griswold, W., Song, Y. Cai, Y., Shonle, M. Tewari, N., Rajan, H. Information hiding interfaces for aspect-oriented design. In Proceedings of ESEC/FSE 2005, September 2005.

[19] Tarr, P. et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. Proc. ICSE'99, May 1999, 107-119.