Aspectual Connectors: Supporting the Seamless Integration of Aspects and ADLs

Thaís Batista[1], Christina Chavez[2], Alessandro Garcia[3], Uirá Kulesza[4], Cláudio Sant'Anna[4], Carlos Lucena[4]

[1]Computer Science Department, UFRN - Brazil

[2]Computer Science Department, UFBA - Brazil

[3]Computing Department, Lancaster University, United Kingdom

[4]Computer Science Department, PUC-Rio - Brazil

thais@ufrnet.br, flach@ufba.br, garciaa@comp.lancs.ac.uk

{uira,claudios,lucena}@inf.puc-rio.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. In this paper, we present the Aspectual Connector (AC), a special kind of architectural connector, as the only necessary enhancement to an ADL in order to support a seamless integration of AOSD and software architecture. We present AspectualACME, an extension to ACME, a well-known ADL that incorporates ACs in order to support the separation and composition of crosscutting concerns at the architectural level. We use an information system as a case study to illustrate the use of AspectualACME.*

1. Introduction

Aspect-Oriented Software Development (AOSD) (Kiczales et al. 1997) aims to provide systematic support for the identification, modularization, representation and composition of crosscutting concerns throughout the software lifecycle. At the architecture design level, a crosscutting concern could be any concern that cannot be effectively modularized using the given abstractions of an Architecture Description Language (ADL) (Shaw and Garlan 1996), leading to increased maintenance overhead, reduced reuse capability and generally resulting in architectural erosion over the lifetime of a system. Some aspect-oriented architecture description languages (AO ADLs) (Navasa et al., 2002) (Pérez et al., 2003)(Pessemier et al., 2004)(Pinto et al., 2005) have been proposed, either as extensions of existing ADLs or developed from scratch employing AO abstractions commonly adopted in programming frameworks and languages, such as aspects, joinpoints, pointcuts, advice, and inter-type declarations. 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.

In (Batista et al 2006) we have presented a reflection around seven issues that arise in this context. Among others things, we have discussed how and why extensions are required or not to traditional notions of interconnection ADL elements, such as interfaces, connectors, and architectural configurations. Our conclusion was that Software Architecture also promotes the principle of separation of concerns (SoC) by separating components and connections and the integration of software architecture and AOSD may take advantage of this SoC-based approach. The idea is to use the same abstractions used in the conventional ADL description, with minor adaptations to support effective modeling of crosscutting concerns without introducing additional complexity into the architecture specification. The central point is the composition mechanism built around the notion of an *Aspectual Connector* that is an extension of the traditional connectors to describe the interaction between crosscutting concerns and components.

In this work we apply the conclusions of our previous work and we present the Aspectual connector (AC) as the only necessary enhancement to an ADL that supports the seamless integration of AOSD into software architecture. In order to instantiate this concept in a well-known ADL, we incorporated the AC abstraction in ACME (Garlan et al, 1997). Therefore, we also present AspectualACME, an ADL that supports the seamless integration of these two separate pieces of technology. We illustrate the AspectualACME concepts with an information system that contains a persistence crosscutting concern affecting the functional components.

We have selected ACME as our base ADL because it presents a relatively simple core set of concepts for defining system structure and it captures the essential elements of architectural modeling (Medvidovic and Taylor 2000). In addition, unlike most ADLs, ACME is not domain-specific and provides generic structures to describe a wide range of systems. It comes with tools that provide a good basis for designing and manipulating architectural descriptions and generating code.

The remainder of this paper is organized as follows. Section 2 provides a background concepts involving AOSD and software architecture. Section 3 overviews existing AO ADLs according to our conceptual framework proposed in our previous work. This framework presents seven important issues related to aspects and architectural connection. Section 4 presents aspectual connectors. Section 5 illustrates how to incorporate ACs into ACME. Section 7 presents the final remarks.

2. ADLs and Aspect-Oriented Software Development

2.1 ADLs

Architectural concerns are typically expressed by using abstractions supported by Architecture Description Languages (ADLs). According to a well-known conceptual framework (Medvidovic and Taylor, 2000), the building blocks of an architectural description are components, connectors, and architectural configurations. *Components* are the units of computing, while *connectors* are the locus of interaction. 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. Component types are templates that encapsulate functionality into reusable blocks and can be instantiated many times.

Connectors model interactions among components and specify rules that govern those interactions. Similarly, connector types are templates that encapsulate component communication, coordination, and mediation decisions. 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. In some ADLs the set of interaction points specified in the connector's interface, where a component joins a connector, is referred to as *roles* which are named and typed. *Configurations* define architectural structure and how components and connectors are connected.

2.2 ACME

ACME supports the definition of (i) architectural structure, that is, the organization of a system into its constituent parts, (ii) properties of interest, information about a system or its parts that allow one to reason abstractly about overall behavior, both functional and nonfunctional, (iii) types and styles, defining classes and families of architecture, and (iv) constraints, guidelines for how the architecture can change over time (Garlan 2000).

Architectural structure is described in ACME with components, connectors, systems, attachments, ports, roles, and representations. *Components* are potentially composite computational encapsulations that support multiple interfaces known as *ports*. *Ports* are bound to ports on other components using first-class intermediaries called *connectors* which support so-called *roles* that attach directly to ports. *Systems* are the abstractions that represent configurations of components and connectors. A system includes a set of components, a set of connectors, and a set of attachments that describe the topology of the system. *Attachments* define a set of port/role associations. *Representations* are alternative decompositions of a given element (component, connector, port or role) to describe it in greater detail. Thus, the representation may be seen as a more refined depiction of an element. For instance, ports may have a representation to encapsulate a large set of API calls as a single port. Inside the representation, a set of ports is used to represent individual API calls.

Properties of interest are <*name, type, value*> triples that can be attached to any of the above ACME elements as annotations. Properties are a mechanism for annotating designs and design elements with detailed, generally nonstructural, information.

Architectural *styles* define sets of types of components, connectors, properties, and sets of rules that specify how elements of those types may be legally composed in a reusable architectural domain. The ACME type system provides an additional dimension of flexibility by allowing type extensions via the *extended with* construct.

Finally, ACME also provides special syntax for describing constraints. Armani (Monroe 1998) is an ACME extension used to express architectural constraints over ACME architectures. It is a FOPL-based sub-language[1] that can be used to express

[1] First order predicate logic

constraints on system composition, behavior, and properties. *Constraints* are defined in terms of so-called invariants which in turn are composed of standard logical connectives and Armani predicates (both built-in and user-defined) which are referred to as *functions*. The ACME fragment in Section 6 illustrates the main ACME/Armani concepts.

The architectural elements outlined above are sufficient for defining the structure of software architecture as a graph of components and connectors. However, they do not provide the adequate means to describe the composition mechanisms required for aspect composition as described in Section 4.

Figure 1 illustrates the example that we use in our discussion in this paper. The HealthWatcher (HW) system is a web-based information system developed by the Software Productivity research group from the Federal University of Pernambuco [ ]. It supports the registration of complaints to the health public system. It is composed of three functional components: (1) the *GUI (Graphical User Interface)* component provides a web interface for the system. (2) the *Business* component defines the business rules. (3) the *Data* component stores the information manipulated by the system. Figure 6 depicts the ACME description of this example.

Component GUI =
{Port GUI_RegisterUser
 Port GUI_RegisterComplaint
 Port GUI_ListComplaint
 Port UpdateComplaint
 Port SearchComplaint }
Component Business =
{Port BusinessComplaint
 Port saveInfo
 Port restoreInfo}
Component Data =
{Port storeInfo
 Port recoverInfo}
Connector C1, C2 =
{ Roles caller, callee }
Attachments
GUI.UpdateComplaint to C1.caller
C1.callee to Business.BComplaint
Business.saveInfo to C2.caller
C2.callee to Data.storeData

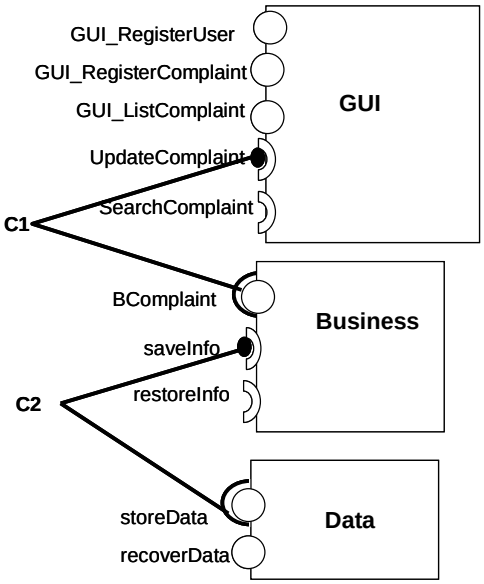

Figure 1. ACME Description of the HealthWatcher System

However, some architectural concerns cannot be modularly captured with traditional abstractions supported by ADLs, such as ACME. Some concerns are *crosscutting* even at the architectural design level, since they cannot be easily localized and specified with individual architectural units such as traditional interfaces, components, connectors, and configurations. Similar to the notion of aspect at the programming level (Kiczales et al, 1997), we say that these concerns crosscut the architectural units and denote the so-called *architectural aspects* (Araújo et al, 2005)(Baniassad et al, 2006)(Chitchyan et al, 2005)(Cuesta et al., 2005)(Krechetov et al, 2006). .

Three crosscutting concerns affects the components of the HW system: (i) Persistence – supports issues related to the data management in web-based systems (transaction management, data update, repository configuration); (ii) Distribution – supports the distribution of the Business component services; (iii) Concurrency – specifies mechanisms to apply different concurrency strategies to the functional components.

Very often, the crosscutting property of the architectural concerns remains either implicit or is described in informal ways leading to reduced uniformity, impeding traceability and hindering detailed design and implementation decisions

2.3 AOSD

Aspect-Oriented Software Development (AOSD) (Filmann et al, 2005) provides new *composition* mechanisms to support the explicit representation of aspects through software development stages, such as the software architecture design. The use of such new composition mechanisms allows for the encapsulation of crosscutting concerns into separated modular units, which are composed with other system modules at well-defined *join points*. Hence AOSD supports the *quantification* of structures and behaviors relative to a concern, which otherwise would be tangled and scattered through the representation of other concerns in software artifacts. Structural and behavioral enhancements can be typically applied *before*, *after* and *around* certain join points. Behavioral enhancements provided by aspects are also known as *advice*.

3. A Framework for Comparison of Aspect-Oriented ADLs

This section presents a conceptual framework that subsumes a set of core issues that need to be considered while dealing with architectural aspects. Our goal is to use such a conceptual framework to support the systematic comparison of existing aspect-oriented ADLs. The proposed framework is a result of a conceptual blend involving an AOSD glossary (van den Berg et al, 2005) and the well-known abstractions of ADLs (Section 2). We have aggregated and filtered the terms found in those sources, for marking out the most vital issues for the design of aspect-oriented ADLs. In order to identify the more relevant architectural concepts, such aggregation and filtering processes were based on a widely-recognized terminology for software architecture descriptions (Medvidovic and Taylor, 2000).

The conceptual framework was also derived from our extensive experience on: (i) the design of aspect-oriented software architectures in different application domains (Garcia et al, 2004)(Kulesza et al, 2004)(Garcia et al, 2006)(Kulesza et al, 2006)(Kulesza et al, 2006b), (ii) the development of modeling approaches to handle different categories of crosscutting concerns at the architectural stage (Chavez et al, 2006)(Garcia et al, 2006a)(Krechetov et al, 2006)(Kulesza et al, 2004), and (iii) analysis of the suitability of existing ADLs to support architectural aspects (Chitchyan et al, 2005)(Batista et al, 2006). As a result, our comparison framework is composed of seven main elements, which are described in Table 1. We recommend that the interested reader explores our extensive discussion on the issues that inspired the framework (Batista et al, 2006).

The first issue is dedicated to understand which architectural elements (e.g. components and interfaces) in an architectural description are typically affected by a

crosscutting concern. The following six issues correlate AOSD concepts with conventional abstractions of ADLs (Section 2). For example, the fourth issue is related to how to specify interfaces of aspects. The last issue is particularly concerned with the need of new abstractions for aspects at the architectural level.

Architectural Concept	**Description**
Base Elements	An AO ADL must define which architectural building blocks may be affected by aspects. As discussed in Section 2, the main architectural building blocks are components, connectors, and interfaces.
Composition	The ADL must support the composition involving architectural elements and aspects. The issue is where the aspectual composition should be defined.
Quantification	The ADL must provide some support for quantification of structural and behavioral elements, in order to avoid the need to refer to each join point explicitly in an architectural description.
Aspect Interfaces	The ADL should allow the explicit description of aspect interfaces. The issue is whether the traditional notion of architectural interfaces should be changed or not to express the boundaries of an "aspectual" component.
Join point Exposition	An AO ADL must support join point exposition. Architectural join points are the elements in an ADL-based specification that can be affected by a certain "aspectual" component. The issue is whether the base elements should have a different interface exposing the join points to the "aspectual" components.
Interface Enhancements	"Aspectual" components may enhance the component interfaces with new elements, such as services and attributes.
Aspect	An AO ADL must support the description of aspects. The issue is whether it should provide or not a new architectural abstraction for describing them.

Table 1. A Comparison Framework for Aspect-Oriented ADLs

In a previous work (Batista et al, 2006), we have used our conceptual framework to analyze several AO and non-AO ADLs. For each issue of the framework, we analyzed how the different ADLs offer support to address it and sketched a proposed solution whenever existing ADLs do not provide an adequate solution. As a result of our analysis, we have concluded that no extra architectural abstractions are needed to represent aspects. We have proposed the extension of the connector and configuration architectural abstractions to support the modeling of the composition mechanism used in the crosscutting concern representation at the architectural level. These extensions are related with the need to support new ways of composition, as well as the quantification essential property of AO approaches. Next section describes our proposal

of aspectual connectors which is the core of our extension proposal. Section 5 presents an extension of the ACME ADL with aspectual connectors.

4. Aspectual Connectors

As already stated, software architecture descriptions rely on a *connector* to express the interactions between components. A connector is a fundamental building block that can model simple or complex interaction protocol as discussed in the taxonomy of connectors (Mehta et al 2000). In fact, ADLs apply the separation of concerns (SoC) principle by explicitly distinguishing architectural elements used to specify computation (components) from those used to express interconnection (connectors) between components. This SoC approach can also play a key role in the integration of ADLs and AOSD. First of all, the component abstraction is enough to model crosscutting concerns. 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 components or multiple architectural elements. In this paper we use the term *aspectual component* just to refer to a component that implements a crosscutting concern. Second, as connectors are widely used for different interconnection purposes, they are enough to model the interaction between traditional components and components that represent a crosscutting concern. However, the way that an "aspectual component" composes with a traditional component is slightly different from the composition between traditional components.

A crosscutting concern is represented by a provided service of an aspectual component and it can affect provided or required services of other components. As in ADLs valid configurations are those that connect provided and required services, it is impossible to represent a connection between a provided service of an aspectual component and a provided service of a traditional component. In order to address this problem we propose an *aspectual connector* that is a traditional connector with a new interface. The aspectual connector interface contains: (1) a *base role*, (2) a *crosscutting role*, and (3) a *glue* clause. Figure 1 depicts a high-level view of the composition between an aspectual component and two components. C1 and C2 are aspectual connectors. The base role is specified to be connected to a port of the component and the aspectual role is specified to be connected to a port of an aspectual component. The pair base-crosscutting roles do not impose the constraint of connecting provided and required ports. A crosscutting role defines the place at which an aspectual component joins a connector. In Figure 2 the aspectual connector C1 connects a provided port of the aspectual component with a provided port of Component 1. C2 connects another provided port of the aspectual component with a required port of Component 2. The glue clause specifies the details about a connection such as the place where the advice must affect the component – after, before, around, and others - and also the details about the advice.

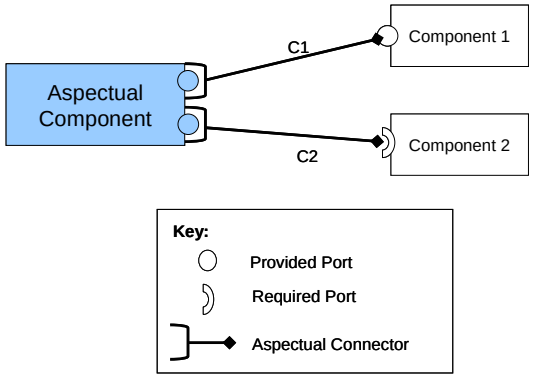

Figure 2. Aspectual Composition

In ADLs, the connection between components, connectors and aspectual components are defined in the *configuration* section. Thus, at the configuration description is defined join points at which an aspectual component acts. The join points are specified in the definition of the association between the base role 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. Thus, we are just taking advantage of this concept to identify the join points affected by a crosscutting concern.

In order to specify several join points in a single statement, wildcards and logical expressions can be used in the configuration part.

5. An Aspect-Oriented ADL

In this Section, we present the description of AspectualACME, an extension of ACME with the goal of supporting a seamless integration of aspects and ADLs. We propose the extension of the connector and configuration architectural abstractions to support the modeling of the composition mechanism used in the crosscutting concern representation at the architectural level.

5.1. Extending ACME

We address the integration of aspects and ADLs, in conformance to what we have discussed in Section 3, by extending ACME with aspectual connectors and quantification support at the configuration level. We call this extension AspectualACME. Additionally, the proposed extension to ACME is expected to support simplicity, expressiveness, and conservative extension (to foster reuse of ACME libraries and tools).

5.1.1. ACME extension for aspectual connectors

The first extension that we propose is a specialization of ACME's connector abstraction. This extension allows the expression of aspectual connectors and their inner constructs: base roles, crosscutting roles and the kind of composition between them (denoted by "glue").

We extend the connector interface in order to support the specification of *base roles* and *crosscutting roles*. The *base role* may be connected to the port of a component (provided or required) 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. An aspectual connector must have at least one base role and one crosscutting role. Figure 3a and 3b present examples of a regular connector and an aspectual connector in ACME.

```
Connector aConnector = {
  Role aRole1;
  Role aRole2;
}
```

Fig. 3a. Regular connector in ACME

```
Connector aConnector = {
  BaseRole aBaseRole;
  CrosscuttingRole aCrosscuttingRole;
  Glue glueType;
}
```

Fig. 3b. Aspectual connector in ACME

Fig. 3. Regular and Aspectual Connector in ACME

We also introduce a new construct -- the *glue* clause -- to specify details about the composition between components and aspectual components, such as the place where the crosscutting service from the aspectual component will affect the regular component. There are three types of aspectual glue: *after*, *before*, and *around*. The semantics are similar to that of advice composition from AspectJ. For binary aspectual connectors (only one crosscutting role and one base role), the glue clause is simply a declaration of the glue type (Figure 3b), but whenever more than one base role and one crosscutting role are declared inside an aspectual connector, the glue clause must be more elaborated (Figure 4).

```
Connector aConnector = {
  BaseRole aBaseRole1, aBaseRole2;
  CrosscuttingRole aCrosscuttingRole1,
                   aCrosscuttingRole2;
  Glue { aCrosscuttingRole1 before aBaseRole1;
         aCrosscuttingRole2 after aBaseRole2;
  }
}
```

Fig. 4. Glue Clause

5.1.2. ACME extension for quantification

The second extension addresses quantification, to avoid the need to refer explicitly to each join point in an architectural description. Since the Attachments part is the place where structural join points are identified, the quantification mechanism is defined by extending the notation used there, related to a system abstraction. We allow the use of wildcards to specify names or part of names of components and their ports. The quantification must be used in the attachment of a base role with target component(s). In Figure 5, wildcards (*) are used to specify that `aConnector.aBaseRole` is bound to all components that offer a port with a name that begins with `prefix`.

```
System Example = {
Component aspectualComponent = { Port aPort }
```

```
Connector aConnector = {
  baseRole aBaseRole;
  crosscuttingRole aCrosscuttingRole;
  glue glueType;
}
Attachments {
      aspectualComponent.aPort to aConnector.aCrosscuttingRole
      aConnector.aBaseRole to *.prefix* }
}
```

Fig. 5 ACME Description of the Composition

In ACME, attachments are declared in an attachment block. Figure 6a presents the BNF for it. Figure 6b presents the BNF for the provided extension. The component/port or connector/role is described as *Identifier* "." *Identifier*.

```
AttachmentsBlock ::=
Attachments "{"
(Identifier "." Identifier to
 Identifier "." Identifier
[ "{" ( PropertyDeclaration |
PropertiesBlock )* "}" ]
";" )*
"}" ";"
```

Fig. 6a Regular Attachments

```
AttachmentsBlock ::=
Attachments "{"
((Identifier "." Identifier to
  AttachmentTarget
| AttachmentTarget to
  Identifier "." Identifier)
[ "{" ( PropertyDeclaration |
PropertiesBlock )* "}" ]
";" )*
"}" ";"
AttachmentTarget ::=
ElementId "." InterfaceId
ElementId ::= Identifier | RegExp
InterfaceId ::= Identifier | RegExp
RegExp ::= …
```

Fig. 6b Extended Attachments

Fig. 6 Attachments

5.1.3. Example

In this section, we present the modeling of the Distribution and Persistence concerns in the context of the HealtchWatcher (HW) system (Section 2.2). We discuss two different configuration of the HW system architecture. This allows to illustrate the flexibility and expressivity of AspectualACME to represent different architectural decisions when modeling an architecture. Figure X and Y show the modeling of the two HW configurations using AspectualACME.

In the first system configuration only the Persistence is modeled as a crosscutting concern, the Distribution concern is specified as a non-aspectual component which allows the GUI component to access remotely the services provided by the Business component. The Persistence aspectual component existing in this configuration addresses: (i) the modularization of an update protocol to persist information modified by the GUI component; and (ii) the transaction demarcation of the services provided by the Business component using a transaction service available in the Data component.

Figure X depicts the AspectualACME description of the HW system including the Persistence crosscutting concern. The Persistence component affects the GUI component and the Business component. The composition of the Persistence component

with the GUI component is modeled by the *Persist* aspectual connector. In the attachments section, the *Persist* connector connects *UpdateStateControl* with *GUI_RegisterUser* and with *GUI_RegisterComplain* (both are referred by the * wildcard in the attachments description). The glue clause of *Persist* specifies that it uses the C3 connector and it acts after the method invocation. This interaction specifies that whenever a user or a complaint is registered, a persistence function is implemented by the *Persistence* component and, in order to implement this function, it needs to use the connection modeled by C3 – that uses a service of the *Distribution* Component. This interaction is expressed by the C3 connector. The composition of the *Persistence* component with the *Business* component is modeled by the *Trans* aspectual connector. It connects the *BComplain* with the *TransactionControl*. It defines that whenever *BusinessServices* is requested, a transaction control mechanism acts during this action. The transaction control mechanism of *Persistence* component uses the transactional operations (*begin_transaction*, *end_transaction*, *comit_transaction*, and *rollback*) provided by the *transactionService* of the Data component. This interaction is modeled by a traditional connector (C4) and it is referred in the glue clause of the *Trans* connector.

Component Persistence =
{ Port TransactionControl
Port UseTransaction
Representation beginTransaction
Representation endTransaction
Representation rollback
}
Component Distribution =
{Port RemoteUpdateStateControl
Port RemoteAccess
Port UseBusinessServices

Connector Persist=
{ baseRole sink
crosscuttingRole source
glue source after sink }
Connector Trans=
{ baseRole sink
crosscuttingRole source
glue {source around sink }
Connector Distrib=
{ baseRole sink
crosscuttingRole source
glue {source around sink }
Connector C3, C4, C5 =
{ Roles caller, callee }
Attachments
Persistence.TransactionControl to Trans.source
Trans.sink to Business.BusinessService
Persistence.UseTransaction to C5.caller
C5.callee to Data.transactionService

Distribution.RemoteUpdateStateControl to Persist.source
Persist.sink to GUI.GUI_Register*
Distribution.RemoteAccess to Distrib.source
Distrib.sink to GUI.*Entity
Distribution.UseBusinessSerivce to C3.caller
C3.callee to Business.BusinessService

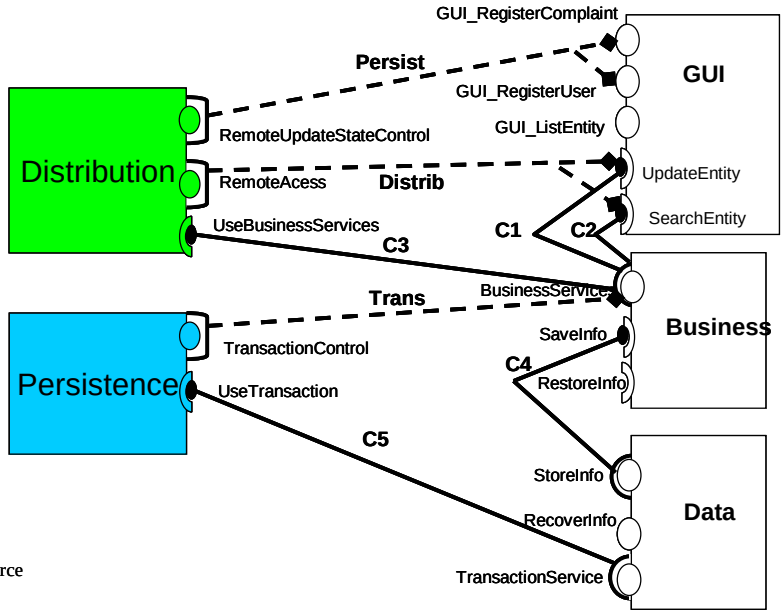

Figure 7. AspectualACME Description of the HealthWatcher System

The second configuration shows both Persistence and Distribution modeled as aspectual components addressing crosscutting concerns (Figure 7). This is the architectural modeling presented by an aspect-oriented implementation of the system [SoaresOOPSLA, Soares SPE]. The Persistence is now responsible by only the

transactional demarcation of the Business services. The Distribution aspectual component modularizes: (i) the transparent configuration of the calls from the GUI component to the Business to be realized through remote access; and (ii) the update protocol that persists information modified by the GUI component. This functionality is now implemented by the Distribution component because it requires to invoke remotely the Business component.

Figure X shows the AspectualACME description for the second configuration of the HW. To model the update protocol, mentioned above, the Distribution aspectual component affects the *GUI_RegisterComplaint* and *GUI_RegisterUser* by quantifying them using wildcard expressions (*GUI_Register**). It is realized by specifying the *Persist* aspectual connector. The *Persist* glue clause guarantees the C3 connector is invoked after the execution of the services affected in the GUI. The Distribution component also models the transparent distributed access of the Business component by the GUI component. The *Distrib* aspectual connector is responsible for this task. It affects the *UpdateEntity* and *SearchEntity* and defines a glue clause which redirects (using around) every invocation to the BusinessServices to be realized by means of the C3 connector. Finally, the Persistence aspectual component models the transaction control in the same way as in the first configuration.

5.2. Evaluation

Table 2 presents an evaluation of the proposed ADL according to the framework presented in Section 3. 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. AspectualACME 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 (Batista et al, 2006), 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.

	Our Approach
Base Elements	Aspects affect components
Composition	Modeled by **aspectual connectors** with base and crosscutting roles and by configurations

Quantification	Supported by wildcards defined at the configuration section
Aspect Interfaces	No extension required
Join Point Exposition	Components can expose their internal events
Interface Enhancement	Not supported
Aspects	Aspects are modeled by means of connectors, crosscutting roles and base roles.

Table 2. An Evaluation of the Proposed ADL

6. Related Work

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. In contrast, we argue that most of existing ADLs abstractions are enough to model crosscutting concerns. For this purpose, is just necessary to extend the connector concept.

Most AO ADLs are different from AspectualACME because they introduce a lot of concepts to model AO abstractions (such as, aspects, joinpoints, and advices) in the ADL. DAOP-ADL (Pinto et al. 2005) defines 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 aspects 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 (Perez et al 2003), 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). Pessemier et al (Pessemier et al. 2004) extend the Fractal ADL with Aspect Components (ACs). ACs are responsible for specifying existing crosscutting concerns in software architecture. 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.

Similarly to our proposal, FuseJ (Suvée et al. 2005) defines a unified approach between aspects and components. It provides the concept of a gate interface that exposes the internal implementation of a component and offers access-point for the

interactions with other components. In a similar way to our proposal, FuseJ concentrates the composition model in a special type of connector that extends regular connectors by including constructs to specify how the behaviour of one gate crosscuts the behaviour of another gate. However, differently from our work, our compositional model works in conjunction with the component traditional interface while FuseJ defines the gate interface that expose internal implementation details of a component. In addition, FuseJ does not work with the notion of configuration. It includes the definition of the connection inside the connector itself. This contrasts with the traditional way that ADLs works – that declares a connector and binds connectors' instances at the configuration section.

7. Conclusions

In this paper we have proposed the concept of aspectual connector as a central element to support the integration of crosscutting concerns in an ADL description. We have also instantiated this concept in the context of a general-purpose ADL – ACME – and we have illustrated the concept with a example that presents two crosscutting concerns.

Our proposal defines a composition model, centered on the concept of an aspectual connector, which takes advantage of traditional 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, 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.

Planned future work includes evaluating our ADL by modeling a large-size system.

Acknowledgements

This work has been partially supported by CNPq-Brazil under grant No.479395/2004-7 for Christina. Alessandro is supported by European Commission as part of the grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008. This work has been also partially supported by CNPq-Brazil under grant No.140252/03-7 for Uirá, under grant No.140214/04-6 for Cláudio. The authors are also supported by the ESSMA Project under grant 552068/02-0.

References

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

Araújo, J. et al. (2005) “Early Aspects: The Current Landscape.” Technical Notes, CMU/SEI and Lancaster University, 2005.

AspectJ Team. “The AspectJ Programming Guide”. http://eclipse.org/aspectj/.

Baniassad, E. et al. (2006) “Discovering Early Aspects”. IEEE Software, Jan/Feb 2006.

Batista, T. et al. (2006) “Reflections on Architectural Connection: Seven Issues on Aspects and ADLs.” Workshop on Early Aspects - Aspect-Oriented Requirements Engineering and Architecture Design, ICSE'06, May 2006, Shanghai, China.

Chavez, C., Garcia, A., Kulesza, U., Sant’Anna, C., Lucena, C. (2006). “Taming Heterogeneous Aspects with Crosscutting Interfaces”. Journal of the Brazilian Computer Society, SBC, Jan 2006.

Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto, M., Tekinerdogan, B., Clarke, S., Jackson, A (2005) “Survey of Analysis and Design Approaches”, AOSD-Europe Report, Deliverable D11, 18 May 2005.

Cuesta; C. et al. (2005) “Architectural Aspects of Architectural Aspects”. 2nd European Workshop on Software Architecture (EWSA), LNCS 3527, pp. 247-262, 2005.

Garcia, A., Kulesza, U., Lucena, C (2004). “Aspectizing Multi-Agent Systems: From Architecture to Implementation”. In: Software Engineering for Multi-Agent Systems III, Springer-Verlag, LNCS 3390, December 2004, pp. 121-143.

Garcia, A., Lucena, C. (2006) “Taming Heterogeneous Agent Architectures with Aspects”. Communications of the ACM, March 2006. (submitted)

Garcia, A., Batista, T., Rashid, A., Sant’Anna, C. (2006a) “Driving and Managing Architectural Decisions with Aspects”. Submitted to IEEE Software, 2006.

Garlan, D., Monroe, R., Wile, D (1997) “ACME: An Architecture Description Interchange Language”. Proc. CASCON '97, Nov. 1997.

K. van den Berg, J. M. Conejero and R. Chitchyan (2005), “AOSD Ontology 1.0 – Public Ontology of Aspect-Orientation”, AOSD-Europe Report, Deliverable D9, 27 May 2005.

Kiczales, G. et al. (1997) “Aspect-Oriented Programming”. European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer, Finland, June 1997.

Krechetov, I., Tekinerdogan, B., Garcia, A., Chavez, C., Kulesza, U. (2006) Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design. 8th Workshop on Aspect-Oriented Modelling (AOM.06), AOSD.06, Bonn, Germany.

Kulesza, U., Garcia, A., Lucena, C. (2004), “Towards a Method for the Development of Aspect-Oriented Generative Approaches.” Workshop on Early Aspects, OOPSLA'04, November 2004, Vancouver, Canada.

Kulesza, U., Garcia, A., Bleasby, F., Lucena, C. (2005) “Instantiating and Customizing Aspect-Oriented Architectures using Crosscutting Feature Models.” Workshop on Early Aspects, OOPSLA'05, November 2005, San Diego, USA.

Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., Lucena, C. (2006) "Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study". Submitted to ICSM.06, 2006.

Kulesza, U., Alves, V., Garcia, A., Lucena, C., Borba, P. (2006b). "Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming". Proc. Intl. Conf. on Software Reuse, June, 2006.

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

Mehta N., Medvidovic, N. and Phadke, S. (2000) Towards a Taxonomy of Software Connectors. Proc. of the 22nd Intl Conf. on Software Engineering (ICSE), Limerick, Ireland, pp. 178 – 187, 2000.

Monroe, R. T. (1998) Capturing Software Architecture Design Expertise with Armani. Technical Report CMU-CS-98-163, Carnegie Mellon University.

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

Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. (2003) PRISMA: Towards Quality, Aspect-Oriented and Dynamic Software Architectures. In Proc. of 3rd IEEE Intl Conf. on Quality Software (QSIC 2003), Dallas, Texas, USA, November (2003).

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

Pinto, M., Fuentes, L., Troya, J., (2005) A Dynamic Component and Aspect Platform, The Computer Journal, 48(4):401-420, 2005.

Shaw, M. and Garlan, D. (1996): Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall.

Suvée, D., De Fraine, B. and Vanderperren, W. (2005) FuseJ: An architectural description language for unifying aspects and components. Software-engineering Properties of Languages and Aspect Technologies Workshop @ AOSD2005.