DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development*

Mónica Pinto, Lidia Fuentes, and Jose María Troya

Dpto. de Lenguajes y Ciencias de la Computación
University of Málaga, Málaga, Spain
{pinto,lff,troya}@lcc.uma.es

Abstract. Architecture description languages deal with the description, analysis and reuse of software architectures. This paper describes DAOP-ADL, a component- and aspect-based language to specify the architecture of an application in terms of components, aspects and a set of plug-compatibility rules between them. With the aim of connecting the specification of the application architecture to the implementation, we describe our language using XML and XML Schemas. The DAOP-ADL language was designed to be interpreted by DAOP, our own dynamic component- and aspect-oriented platform. DAOP provides a composition mechanism that plugs aspects into components dynamically at runtime. The software architect will use the DAOP-ADL language at design time to describe the architecture of the application. Later this architectural information is loaded into the DAOP platform, which needs it to establish the dynamic connections between autonomous components and aspects. Therefore, the use of DAOP-ADL closes the gap between design and implementation of component- and aspect-based applications.

1 Introduction

The emergence of Component-Based Software Engineering (CBSE) [1] imposes a new way of developing applications by plugging in standalone software components, rather than building them from scratch. In consequence, traditional methods provided by the Software Engineering should be adapted in order to evolve to more compositional architectures. The goal is the reduction of development times, costs, and efforts, while improving the flexibility, extensibility, and reusability of the final application due to the (re)use of software components already developed and verified.

However, everybody agrees that achieving an accurate functional decomposition of systems in context independent components is not a trivial task. Commonly, the same concern happens to be spread over different components creating undesirable dependencies among them. These dependencies make the functional decomposition of a system in autonomous components difficult. In this

* This research was funded by the CICYT under grant TIC2002-04309-C02-02

F. Pfenning and Y. Smaragdakis (Eds.): GPCE 2003, LNCS 2830, pp. 118–137, 2003.
© Springer-Verlag Berlin Heidelberg 2003

way, advanced separation of concerns is widely accepted as the most adequate approach to solve this problem, extending CBSE with new dimensions of concerns, beyond "objects" or "components". Aspect-Oriented Software Development (AOSD) [2] is a promising discipline based on the principle of separation of concerns. AOSD introduces a new dimension named *aspect*, which models those concerns presented along multiple components in a system that may change or evolve independently, resulting in more decoupled systems. Therefore, the techniques proposed by AOSD can be successfully applied in the context of CBSE, improving the modularity and in consequence the reusability and evolution of final applications. Our main goal is then, to combine CBSE and AOSD techniques to obtain all their mutual benefits.

Although aspects were originally defined only at the programming level [3], AOSD tries to cover all phases of the software development life-cycle, from requirements to implementation. Therefore, it is needed to define notations and languages that help software developers to specify and validate component- and aspect-based software architectures, enabling a more effective analysis and design of applications. Moreover, the resulting specification of the application architecture (AA) must be connected to the implementation to avoid inconsistencies and to promote software evolution.

Architecture Description Languages (ADLs) deal with the description, analysis and reuse of software architectures [4]. ADLs describe the structure of the system, which comprises software components, the externally visible properties of those components, and the relationships among them[5]. The purpose of this paper is to define a new ADL to specify the structure of a system in terms of components and aspects, and a set of composition rules. These composition rules will define how to plug software components among them and how and when to apply aspects to components to extend the system behavior with aspectual properties. Similar to other ADLs, our ADL is used to analyze and verify the correctness of a certain design, being possible to reason about the system structure and to evaluate different design possibilities, even before implementation.

While a number of different ADLs[4] have been defined, most of these languages have the problem of being loosely coupled to implementation languages, causing problems in the analysis, implementation, understanding, and evolution of software systems[6]. With the aim of avoiding this problem and facilitating the integration and interpretation of the information provided by the ADL at runtime (e.g. in a dynamic aspect platform), we describe our ADL using XML and XML Schemas. Most of the new ADLs are already using XML Schemas as the representation basis[8], as well as some of the existing ones such as Acme[7]. Our language (we call it DAOP-ADL), has been concretely designed to be "interpreted" by our own component- and aspect-based middleware platform (the "Dynamic Aspect-Oriented Platform", DAOP). The main characteristic of DAOP is that components and aspects are first-order entities that are dynamically composed at runtime. The DAOP platform and its supporting component and aspect model are described in detail in [9][10].

One of the main problems of current component platforms (e.g. CORBA, J2EE and .NET) is that the information about the AA is spread around a large number of classes [12] being difficult to verify the correctness of component composition. The DAOP platform copes with this problem by separating the information that describes the AA from the reusable components and aspects. In fact, the AA information that the platform consults at runtime is exactly the information generated using the DAOP-ADL language. With this approach we close the usual "gap", or loss of information, between the design and the implementation level - the information about the software architecture of an application is usually lost at implementation level.

The main focus of this paper is to present the XML Schemas used to specify a component- and aspect-based architecture and the role that DAOP-ADL plays at runtime in the DAOP platform. The DAOP platform will interpret a certain architectural description in DAOP-ADL, and will store this information as part of its internal data structures. The main benefit of using this approach is that components and aspects are really independent and autonomous entities, because the information about connections is not hard coded, it only resides inside the platform. The DAOP platform consults this information at runtime to perform dynamic weaving between components and aspects. Another important advantage is that the global configuration of the system is available throughout the software life cycle. So, apart from the analysis at design phase, the platform is able to detect at runtime if some component or aspect is violating some design rule. In order to simplify the software architect task, we provide a set of architecture description tools (DAOP-ADTools) that simplify the creation and verification of software architectures described with the DAOP-ADL language.

The structure of this paper is as follows. Firstly, in Section 2 we present some related work comparing our approach to other aspect-oriented systems. Then, Section 3 summarizes the main characteristics of our DAOP platform. Section 4 and Section 5 focus on the definition of the DAOP-ADL language and the creation and validation of a AA using the ADTools. Since we have experience in the implementation of virtual offices applications as part of a research project, we are going to use examples of collaborative applications throughout all the paper. Finally, in Section 6, we show how to integrate and use the AAs specified using the DAOP-ADL language in the DAOP platform.

2 Related Work

Aspects were introduced by [3] and are defined as system properties that tend to cut across functional components, resulting in more coupled systems (*code tangling*). AOSD technologies model components and aspects as two separate entities where the aspects are automatically weaved into the functional behavior of the system in order to produce the overall system.

In AOSD approaches, we can find different alternatives to tackle the separation of concerns issue, even though the final goal is the same. These approaches differ mainly as regards where the weaving information is placed (which aspects are applied to the primary code and when), and the phase in the software life

cycle in which the weaving process is performed (statically during compilation or dynamically at runtime). The variety of AOSD approaches is so extensive that it is not possible to cover all of them in this section. Then, we are going to compare our approach with those that we consider more relevant in relation to our work. A complete description of AOSD technologies can be found in [2]. Firstly, some works define AO libraries [13], which are characterized by offering static weaving and in which the functional code must include some information dependent of the aspects applied.

AO languages are the most common approaches. A broadly used AO language is AspectJ [14], which uses a slightly modified form of Java providing a new construction similar to Java classes to implement aspects. In AspectJ, the aspect encapsulates join points that indicate weaving information and advices that indicate aspect behaviors. The weaving process in AspectJ is static, mixing component and aspect codes at compile-time. Static composition provides high performance but separation of concerns is lost at runtime. Other similar approaches are [15] [16].

Another option is to use AO frameworks that provide template classes for modelling components and aspects and a more or less dynamic composition mechanism. Recently, new proposals on AO frameworks have emerged to be used instead of aspect languages and static weaving. Dynamic weaving is much more flexible than static weaving because the separation of concerns remains through all phases of the software cycle, enabling in some cases the late binding between components and aspects. These approaches are based mainly on the reflection mechanism to achieve application adaptability. Most runtime reflective systems are based on the ability to modify the application semantics while the application is running. This adaptability is commonly achieved by implementing a Meta Object Protocol (MOP) as part of the language interpreter that specifies the way a program may be modified at runtime.

For instance, PROSE [17] is an AO platform with dynamic composition that uses MOP-reflective interpreters on its back-ends. Its main contribution is that the platform weaves and unweaves aspects directly in the Java Virtual Machine (JVM), so this work is valuable but only applicable to Java applications. In addition, join points and advices are implemented in PROSE in the class representing the aspect, with the drawback of reducing the use of aspects advices in different contexts.

Another similar approach is JAC [18], an AO framework that uses BCEL [19], a load-time reflective system for Java, for adding aspects with a reflection mechanism. Using this mechanism, aspects in JAC are dynamically deployed and undeployed on top of running application objects. In contrast with PROSE and some aspect language approaches, in JAC, join points are not specified as part of the aspect definition but in a third-party entity available at runtime, making aspects more reusable. Another important feature of JAC is that it provides a development environment to help in the construction of aspect-oriented applications in JAC. This environment generates aspect configuration files to configure aspects, and program descriptors files to describe which aspects must be applied

to an application. These text files are loaded into the JAC architecture during application execution, and can be modified at runtime adapting the application functionality without code (re)compilation. In addition, JAC authors are working on a UML notation for aspect-oriented software design [20], although this UML design has not been connected with the JAC implementation yet. Basically, JAC is a client-server approach with all its advantages and disadvantages, and its main limitation is that it is again only applicable to Java applications.

Another proposal for load-time transformation of Java classes is JMangler [21]. JMangler is an AO framework which supports conflict-free composition of independently developed aspects. Aspects are implemented as JMangler transformer components and their main goal is to find an automatic way of combining black box components, avoiding unwanted side effects. Transformers are easy to configure in JMangler, as software developers can combine them just by editing a simple text file. JMangler supports all the transformations of class files that do not violate binary compatibility, for instance the addition of classes, interfaces, fields and methods, and it enables changing a method code. Similar to PROSE and JAC, JMangler modifies java classes when they are loaded in the Java Virtual Machine (JVM). However, JMangler modifies the base class of the Java class loader hierarchy, making the transformation independent of a specific class loader or JVM implementation. Its main limitations are again the dependency of the Java language and, like most dynamic approaches, its efficiency.

The Composition Filters (CFs) model [22] is another aspect-oriented programming technique where aspects are expressed in filters as declarative and orthogonal message transformation specifications. The CF model is a modular extension of the conventional object-based model. Therefore, all behavior is implemented by the manipulation of incoming and outgoing object messages. The main advantage of CFs are that filters are expressed using declarative specifications and, in consequence, they are not dependent of a specific implementation language. Secondly, the CF model provides the advantages of both static and dynamic approaches. Filters can be interpreted, providing runtime solutions, or they may be compiled, providing faster solutions.

Finally, current component platforms such as J2EE already offer a set of built-in services such as security, persistence and transaction services that can be incorporated to an application without programming effort. However, these services are limited to a predefined set and it cannot be considered that they fully support separation of any other concern at the application level. Some current approaches, such as the JBoss AOP framework [23], are trying to solve this limitation. Similar to PROSE, JAC and JMangler, the JBoss AOP framework supports load-time transformations of Java classes using Javassist, a load-time reflective system for Java that allows the modification of a class file before the JVM loads it. The JBoss AOP framework has been designed to be used on top of JBoss, which is a J2EE-based application server. They have developed several system-level aspects such as transaction, security and remoting, which use the "container" technology provided by J2EE.

Our approach is a sort of middleware AO framework. DAOP differs basically from other AO frameworks in that: (1) we define a new compositional model that overcomes the deficits of distributed component, mainly regarding the representation of the AA, while the rest of aspect approaches does not encompass components in the CBSE sense; (2) the platform performs dynamic composition instead of the static one, the most used in AOSD; (3) we specify component- and aspect-based software architectures using an ADL (the DAOP-ADL language) (4) the DAOP-ADL language closes the gap between design and implementation connecting the AA with the DAOP platform that loads it at runtime as part of its internal data structures (5) the contribution of our approach does not rely on taking advantage of a platform or programming language, for instance Java [18] [17] [21]. Although DAOP is currently developed using Java/RMI, the aspect-component model supported by this platform can be implemented in any general purpose language or component platform.

3 DAOP: Dynamic Aspect-Oriented Platform

In this section we summarize the main contributions of the DAOP platform. A complete description of our approach can be found in [9] [10].

Components and aspects exist at runtime. The DAOP platform is based on a component-aspect model where components and aspects are first order entities implemented in the same general purpose language. The composition between components and aspects is established during runtime interaction and is governed by a set of plug-compatibility rules in order to guarantee their correct interoperation. Although the model does not impose any restriction on the granularity of both entities, the way they are composed may impose may recommend not to define a few lines of code as components or aspects. The overhead normally introduced by dynamic composition moves us to consider both components and aspects as high-level, black-box, coarse-grained entities as in CBSE. This model is an extension of a previous work that only separates the coordination aspect [11].

The software architecture of an application (AA) is explicitly specified and stored in the platform. In our aspect-component model, the AA is described in terms of a set of components, aspects and the connections between them. The AA is defined by the software architect during the design phase, using the language to describe software architectures presented in next section, and it is stored in an application repository inside the DAOP server (see right side of Figure 1). More details about how the DAOP platform downloads the AA as part of the application launching are given in the last section of this paper.

Components and aspects have an unique and universal role name. Apart from the dynamic composition, another important contribution of DAOP is that the model detaches components and aspects interfaces from the final implementation classes. We use neither class names coupled to concrete implementations, or interface names as component and aspect identifiers. We assign a unique role name to reference both components and aspects. These role names are archi-

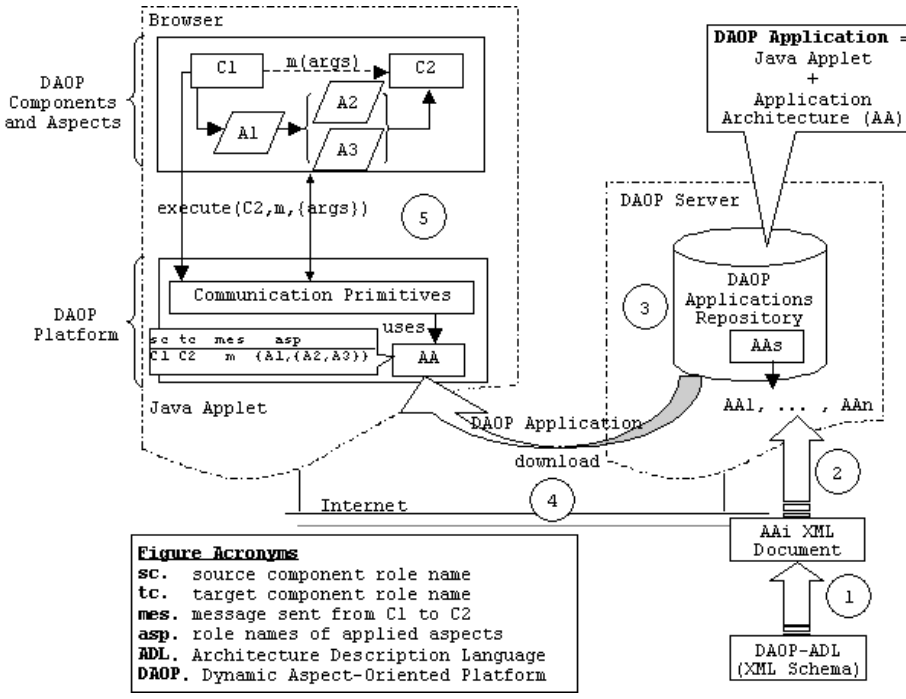


Fig. 1. Dynamic Aspect-Oriented Platform

tectural names that will be used for component and aspect composition and interaction, allowing loosely coupled communication among them - i.e., no hard-coded references need to be used for exchanging information, but just the role name of the target of a message. In addition, components can also be addressed by a unique component identifier that refers to a component instance, in case particular instances need to be addressed.

Component and aspect implementations can be modified at runtime without any code recompilation. By describing the AA in terms of role names, for instance components, are not enforced to implement a concrete implementation. We only require them to offer a set of methods. The benefits of this approach is that components or aspects with different interface and/or class names can play the same role name. Components and aspects with the same role name that are instantiated in different user sites provide the same behavior, so we are able to replace them by equivalent components or aspects according to their abstract definition inside the AA. This is very powerful, because different users of the same distributed application can collaborate through different implementations of, for example, a chat component. Furthermore, users may choose among a set of component implementations, the most appealing one.

Components do not have direct references among them. Another relevant feature of DAOP is that references between components are not hard coded

inside them. Instead of that, components use the role name to specify the target of an output message. In consequence, components in our platform are loosely coupled.

Component interactions are performed through the platform. The DAOP middleware layer defines different communication primitives for components to send messages and events to other components. For instance, in Figure 1 the component with role name C1 sends the asynchronous message *m(args)* to the component with role name C2 by invoking the *execute* primitive. The platform checks in the AA which component instance plays the C2 architectural role name and which aspects (aspects with role name A1, A2 y A3 in Figure 1) need to be evaluated before and/or after the invocation of that method in the target component. This decouples the composition information from the implementation classes making the system more extensible and configurable. Likewise, if a component needs to send a synchronous message it will use the *execsync* primitive. The *broadcast* primitive is used to send a message to all the components with the same role name, and events are sent using the *event* primitive.

Components and aspects are independent. In order to achieve component and aspect independence, they do not have any information about how they are composed, providing a powerful mechanism for late binding among them. This information is explicitly stated in the platform and consulted at runtime after the invocation of the middleware communication primitives. This means that components have no knowledge about the aspects they are affected by and also that the number and type of aspects applied to a component can vary dynamically. For each component and each method inside a component, the platform stores the information about the sort of aspects to apply and the order in which they have to be applied. We want to point out that component and aspect join points are neither part of the aspect implementation, nor of the aspect interface. As we already mentioned, they are stated in the AA inside the platform, so the resulting aspects are more extensible and reusable in different contexts.

The DAOP platform applies aspects at message and event interception. The AOSD community offers different approaches for weaving aspects, depending on the points where the pointcuts can be placed. Some approaches support the definition of pointcuts at any place of the code (e.g., before, after, around, ...), mainly because they are based on code insertion. Other approaches use different kinds of message interception, so the aspect evaluation is triggered by the delivery of a message or an event. This allows aspects to be applied to black-box components, closely to the CBSE philosophy. In the DAOP platform, aspects can be applied to the component's methods when a message or event is sent or received. We understand the meaning of messages and events in the CBSE sense, since events are messages with no information about the source and/or target component. In this case, the platform resolves the target component(s) by using a coordination aspect.

Context dependencies between components and aspects are solved defining properties. Since software components usually have some context dependencies,

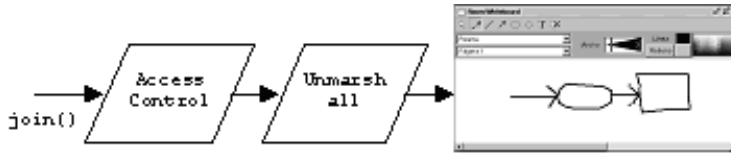


Figure 2.a. User joins the collaborative whiteboard application



Figure 2.b. User draws in the collaborative whiteboard application

Fig. 2. Collaborative Whiteboard Application

i.e, sometimes they need to share data, the DAOP platform defines the concept of property, which it is an entity that encapsulates this kind of data dependency. Properties can also be used to solve dependencies between non-orthogonal aspects. For instance, if an aspect needs any information generated previously by another aspect, it will take this information from the DAOP platform. That is, aspect output information is stored inside the platform, ready to be retrieved by any other aspect that needs it.

Summing up, the use of DAOP results in software components and aspects more extensible and reusable in different contexts. These benefits lie in the dynamic composition of components and aspects at runtime and in the explicit definition of the AA. In the rest of the paper we describe an ADL to specify and analyze the software architecture of an application developed following the component-aspect model of the DAOP platform.

4 DAOP-ADL: The Architecture Description Language of DAOP

In this section we present DAOP-ADL, a language to specify the software architecture of any application running on top of the DAOP platform. We avoid using complex and in some contexts not very popular formalisms for DAOP-ADL, so we decided to use XML Schemas [24]. XML is the standard language to represent structured information. A major advantage of using XML is the range of XML tools that can be applied both to write XML documents and to validate and analyze their content.

The DAOP-ADL language consists of two parts: the description of the interfaces of components and aspects and the description of their relationships. We had to define a new ADL since the existing ones, like UNICON, Darwin and

Rapide, are not able to express the concepts of our composition model, such as the separation between components and aspects and the rules that define the component and aspect composition. We are going to introduce an example of a shared whiteboard that will be used throughout the paper to illustrate our proposal. Figure 2 shows a *user* that wants to join a *whiteboard* application. Before executing the *join()* method, the *access control* and *unmarshall* aspects are evaluated (Figure 2 .a). The *access control* aspect makes it sure that the user only joins the application if he or she has the right permissions. The *unmarshall* aspect retrieves the persistent state of the component from a store, for example a LDAP directory server or an Oracle database. This is useful for *latecomer* users that join the application once there are other users working on it. When the user is drawing on the whiteboard the method *draw()* is executed, and the *marshall* aspect is applied to make the component state for other users persistent. We are going to describe the AA of this system with the DAOP-ADL language.

4.1 Description of Components and Aspects

Our component-aspect model as well as DAOP-ADL, consider components and aspects as two separate entities. So in this section we are going to present how to describe components and aspects in DAOP-ADL.

Similar to other component models such as CORBA CCM [25] and EJB [26], we specify components by a set of public interfaces. Conforming to our DAOP model, components are referred by the role name that they play in the software architecture. Therefore, the DAOP-ADL language names components by a XML-attribute that represents DAOP role names. In AOSD, components represent the functional behavior of a system. Traditionally, components publish a *provided* interface that describes this behavior. However, in order to define and reason about the software architecture of an application, a component model needs to make other kind of interfaces explicit. Therefore, DAOP-ADL defines, in addition to the provided interface, a *required* interface for specifying the output messages or events that the component is able to send. Figure 3 shows a XML schema for a component *required* interface. The *required* interface describes the events that the component may throw and the output messages indicating the method name and optionally the return value and a set of parameter types. The message delivery may be asynchronous or synchronous. The target component of each message is identified by the role name played by that component.

Aspects are properties of a system whose evaluation is automatically added to a component behavior, during component communication. Instead of a *provided* interface as for components, aspects have an *evaluated* interface describing those messages that are able to evaluate. It is possible that the signature of messages being evaluated by an aspect may change during evaluation. This transformation is specified in the *evaluated* interface. More details about this transformation are showed through some examples in the next section. In addition, during aspect evaluations, aspects communicate with components using the communication mechanisms provided by the platform. This means that in the *required* interface of aspects also goes the specification of output messages and events. In addition,

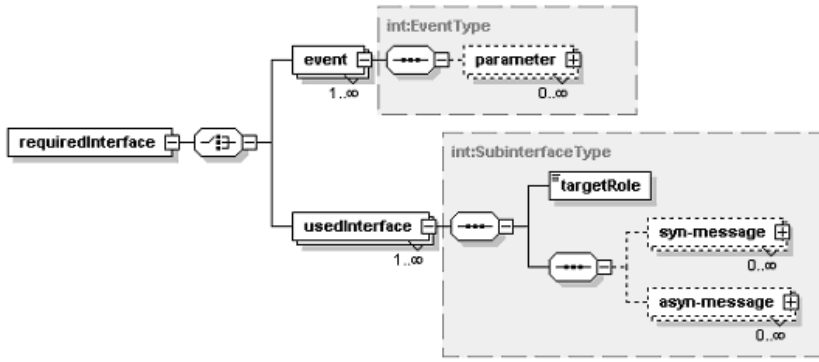


Fig. 3. DAOP-ADL Interfaces XML Schemas

aspects are able to handle events thrown by components, then they define a *target events* interface describing the events that they can capture.

Apart from the specification of abstract interfaces, the description of components and aspects includes other relevant information:

1. A list of possible *implementation classes*. Components that are candidate for playing a role name must implement the same *provided* interface for interoperability purposes. Likewise, aspect implementations of a certain role name must be able to evaluate all the messages described in the *evaluated* interface.
2. A list of input and/or output *properties*. Components and aspects can share information using these properties, which are stored into or retrieved from the platform. Each property is identified by a *name* and a *signature*. This is very useful, for instance, to solve parameter passing between independent aspects. They only have to know the public name of the property, and later they will be able to put a value in the platform or take it from the platform. The scope of these properties can be *usersite*, which means that the value is shared by all the components and aspects instantiated in the same middleware layer, and *serversite*, which means that the value is common to all the components and aspects that are part of the same distributed application.
3. A list of *dependencies*. Component and aspect dependencies are a list of names that match with the role names of the target components that appear in the *required* interface. Since components and aspects use role names to send messages in the implementation code, we use the dependency list for substituting formal parameters of role names by the actual ones.
4. In the aspects case, the software architect needs to specify if the platform has to create an instance of the aspect for each user or there must be a unique instance for that aspect in the application. In the latter case, the software architect may provide the URL where the aspect will be created.

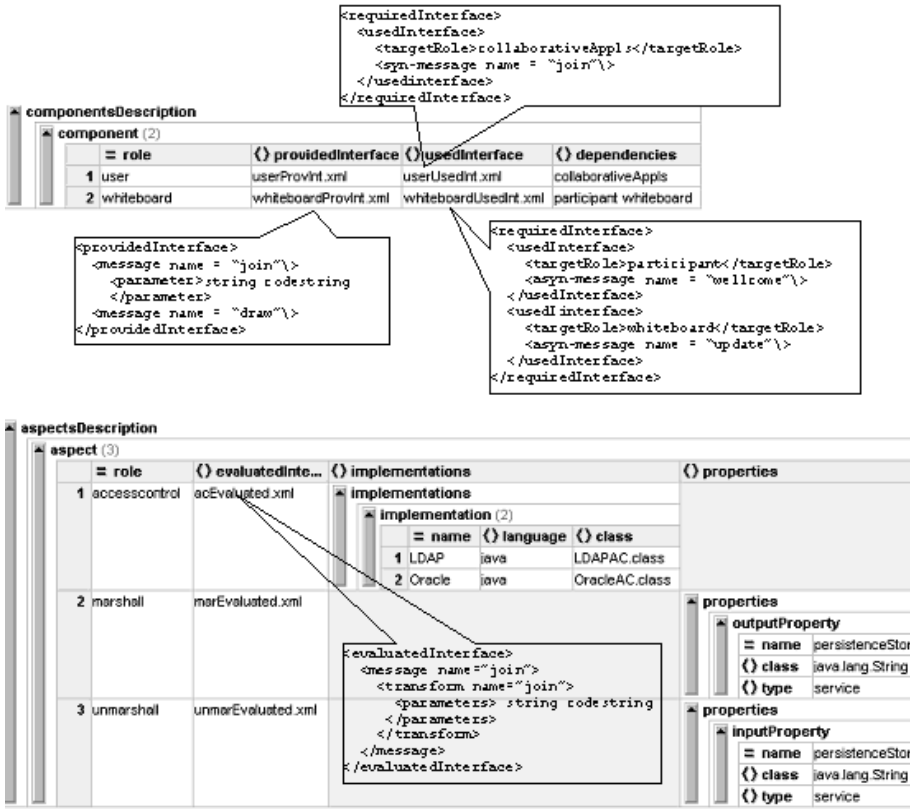


Fig. 4. Component and Aspect XML Description

An Example Figure 4 shows the description of the components and the aspects encoding the example presented in Figure 2. For clarity purposes we omit part of the description information for some of the components and aspects. There are two components in the system with role names *user* and *whiteboard*. We show the XML file describing the *provided* interface of the component with role name *whiteboard* having the *join* and *draw* methods. The *required* interface for the same component describes the output messages that are sent to interact with components with role names *participant* and *whiteboard*. These role names match the names in the *dependencies* list of the *whiteboard* component. The *whiteboard* component sends the asynchronous message *wellcome* to a participant that joins the application. Also, it will send the message *update* to other components with the same role name *whiteboard* to update the application state.

There are three aspects in the system with role names *accesscontrol*, *marshall* and *unmarshall*. In this example, the software architect provides two possible implementations of the *accesscontrol* aspect (based in LDAP or Oracle Database), the programming language (e.g. Java or CORBA), and the corresponding class

```
<compositionConstraints>
   <componentsCompositionRules>
     <componentRule role="user">
       <matchedPair>collaborativeAppls whiteboard
       </matchedPair>
     </componentRule>
     <componentRule role="whiteboard">
       <matchedPair>participant user
       </matchedPair>
       <matchedPair>whiteboard whiteboard
       </matchedPair>
     </componentRule>
       ...
   </componentsCompositionRules>

   <aspectsEvaluationRules>
       <aspectRule>
         <compRole>whiteboard</compRole>
         <inputAspects>
             <messages>joins</messages>
             <aspectlist>accesscontrol</aspectlist>
             <aspectList>unmarshall</aspectlist>
         </inputAspects>
         <outputAspects>
             <messages>draws</messages>
             <aspectList>marshall</aspectlist>
         </outputAspects>
       </aspectRule>
       ...
   </aspectsEvaluationRules>
</compositionConstraints>
```

Fig. 5. Description of Components and Aspects Composition

name. The *accesscontrol* aspect implementation may change even at runtime, since it is identified by a role name and never by anything related with implementation. The *evaluated* interface associated with the *accesscontrol* aspect says that this aspect transforms the message *join()* into *join(user,password)* upon evaluation. Also in the example, two aspects share the same property named *persistentStore*, containing the location of the persistent data associated with the collaborative application. *PersistentStore* is an output property for the aspect with role name *marshall*. This means that this aspect generates the value for this property and stores it in the platform. The *unmarshall* aspect uses the same property but as a consumer. This means that this aspect will obtain the value associated to this property from the platform. Although both aspects are related through the use of this property, they do not need to have any reference among them.

4.2 Description of the Component and Aspect Composition Rules

Using the first part of the DAOP-ADL language described above, the software architect describes all components and aspects that can be instantiated in the application. Now, we present the second part of the language that is used to express the relationships among components and aspects, and some information for adapting mismatched interfaces. As we told in the introduction, composition constraints are expressed in terms of a set of component composition rules and a set of aspect evaluation rules as showed in Figure 5.

Coming back to the example, component composition rules define the relationships between the *user* and *whiteboard* components described previously. Since component connections are established by mapping component role names, incompatibilities may appear as we mentioned in previous section. For each component we have to match role names used by those components in their interactions with role names assigned to components in the AA. The former role names are hard coded in the component implementations while the latter ones are established by the software architect. In Figure 4, in the *required* interface of the *whiteboard* component, the *participant* role name must be understood as a formal parameter that will be substituted by the platform with the *user* role name, that is, the actual parameter. In addition, you can notice that the *user* component sends the join message to a *collaborativeAppls* component, but again the platform will substitute this role name by the actual role name, that is *whiteboard*. Using the component composition rules in Figure 5, the platform will automatically adapt those role names at runtime.

On the other hand, aspect evaluation rules define when and how to apply aspects to components. There are three kind of aspects. *Input aspects* that are evaluated before the execution of a method in a component. *Output aspects* that are evaluated after the execution of a method in a component. And, *event aspects* that are evaluated when a component sends an event, needed to resolve which target components must receive the event. In Figure 5 we show the lists of input and output aspects for the *whiteboard* component. Before executing the *join* method in the component with role name *whiteboard* the aspects with role names *accesscontrol* and *unmarshall* are evaluated. Notice that there are two XML elements with the label *aspectlist*. This means that aspects contained in different *aspectlist* XML elements are evaluated sequentially, and those in the same list will be evaluated concurrently. After executing the *draws* method in the component with role name *whiteboard*, the DAOP platform will evaluate the aspect with role name *marshall* to make the drawing persistent.

Once again we want to highlight here that both component composition rules and aspect evaluation rules are expressed in terms of component and aspect role names, and we never use their interface or implementation classes. This is a relevant feature of our language because the AA of the running application can be adapted at runtime without code recompilation.

5 Analysis and Validation of the Application Architecture

Application developers need to understand the benefits of using aspects and how components and aspects interact and only after this they will be able to design their own component-aspect applications. Our approach provides some tools to guide designers through the development process of DAOP applications (Figure 6). We use the Xerces 2 Java parser [27], that is a fully conforming XML Schema processor, to parse the AAs described in DAOP-ADL. The Architecture Description and Validation Tool (DAOP-ADTool) of Figure 6 presents the typical functionality included in other development environments like visualCafe.

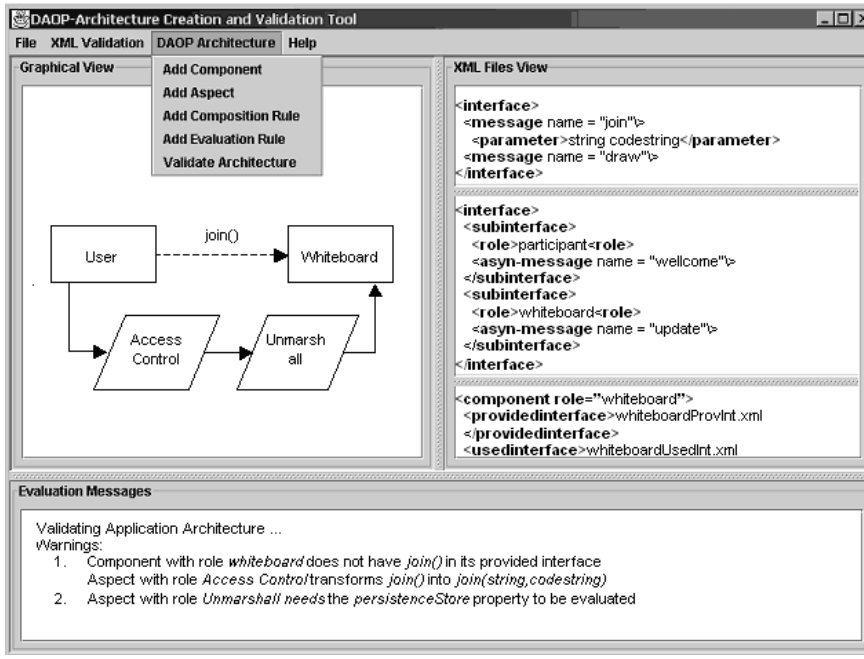


Fig. 6. DAOP Architecture Creation and Validation Tool

It has a visual editor for displaying available components and aspects and for interconnecting them. The DAOP-ADL (simply an XML Schema) serves as the back end of the visual representation.

We differentiate two levels of validation. Firstly, using the XML parser we check the well-formedness and validity of XML documents. An XML document is well-formed if it meets all the well-formedness constraints given in the XML 1.0 [28] specification. A well-formed XML document may in addition be valid if it has an associated XML Schema and if the document complies with the constraints expressed in it. Using the Schema formalism we can express syntactic, structural and value constraints applicable to the definition of XML documents describing AAs. The Schema formalism will allow a useful level of constraint, checking things like the type of the elements, the cardinality, default values, uniqueness, etc.

In addition, there are other constraints that the application must conform and cannot be expressed directly using an XML Schema. Then, the second level of validation is performed by our DAOP-ADTool. We classify the validation process according to the different parts of the DAOP-ADL language.

5.1 Validation of Component and Aspect Descriptions

The most relevant constraints checked by our DAOP-ADTool during the AA description process are related to the component and aspect *interfaces*, *implementations* and *properties*.

The software architect may propose different implementation classes for the same component or aspect. Component and aspect implementations are available through a *Component Repository* and an *Aspect Repository* respectively (as COTS components). The information needed to validate the correctness of the implementation list can be provided using different mechanisms. One of them is *introspection* as provided by JavaBeans [29]. Using introspection it is possible to recognize predefined patterns in method signatures and it is possible to *look inside* a component or an aspect to determine its properties and behavior. An alternative to introspection used by other component platform such as CCM and J2EE is packaging the component and the aspect with a set of XML description files that provide the same information obtained by introspection.

Using any of the above mechanisms we can validate that the interfaces implemented by classes conform the interfaces provided during components and aspects description. Other important constraints to validate are related to component and aspect properties.

DAOP-ADTool tool is able to verify that an aspect-component AA is correct by performing the following checks:

- Each component implementation class must implement at least the *provided* interface described for that component.
- The messages sent by an implementation class must all be described in the *required* interface for the component or the aspect. In addition, each message must be sent to the component that plays the appropriate role name in the AA. If the implementation class is able to send more messages than expected the tool will generate a warning. The software architect will decide wether this is critical or not. This is useful in an incremental development process and also when the designer really knows that a component will never send that critical message (for instance, it is a result of extra functionality)
- The *events* in the *required* interface must describe exactly the same events that the implementation class is able to throw. In other case, a warning will be generated, similarly to the last case.
- Each aspect implementation class must be able to evaluate at least all the messages described in the *evaluated* interface for that aspect.
- The component and aspect implementation classes must generate all the output properties described in the AA. If a component does not store in the platform an output property, an error message is generated. With input properties, we relax this constraint. The component could need or not the input properties described in the AA. In this case, only a warning message is generated by the validation tool.

Other constraints that are worthwhile to validate are the uniqueness of component and aspect role names and property names. The uniqueness of properties names means that two properties with the same name must have the same signature. In addition, for all input properties described in a component or aspect, there must exist another component or aspect that have the same property in its list of output properties. Otherwise, there would be at runtime a component or aspect needing a property that no other component or aspect will ever generate.

5.2 Validation of the Component Composition Rules

As explained before, component composition rules match role names used by components in their interactions with role names assigned to the components in the AA. We show the composition rules validation process through an example (see again Figure 5).

1. The formal parameters of role names must be in the *dependency* list of the component or aspect. The label *collaborativeAppls* must be in the *dependency* list of the component with role name *user*, and the same for the labels *participant* and *whiteboard* in the component with role name *whiteboard*.
2. The target component of output messages or events must exist inside the AA. There must be components with role names *user* and *whiteboard* in the AA description.
3. Given a formal role name for a target component in the *required* interface of a component or aspect, the set of messages associated with that role name must be contained in the *provided* interface of the component or aspect with the actual role name. For instance, the messages sent by the component with role name *user* to the component *collaborativeAppls* must be contained in the *provided* interface of the *whiteboard* component.

It is important to highlight the possibility of a mismatch between the *required* interface in the source component and the *provided* interface in the target component. This situation is showed in Figure 6, where the *user* component sends the *join()* message to the *whiteboard* component. However, the *whiteboard* components expects a message with the same name but the additional parameters *(string, codestring)* representing the user name and password. As showed in Figure 6 this provokes a warning message instead of an error message, because the validation tool is able to check that the *access control* aspect evaluated before executing *join(string,codestring)* transforms *join()* into *join(string, codestring)*.

5.3 Aspect Evaluation Rules Validation

A desirable feature of aspects is orthogonality. This means no dependencies among them. However, this is a hard task to achieve, and sometimes information generated during the evaluation of an aspect it is needed in the evaluation of following aspects. In DAOP we resolve this problem using input and output properties. With this mechanism of storing and retrieving properties from the platform, we achieve our goal of aspects independence. Dependent aspects do not interact directly, but through the platform using the same properties.

However, a set of aspects can be evaluated sequentially or concurrently depending on the information provided in the AA. When aspects are completely independent both approaches are valid. But, if there are dependencies among aspects, they must be evaluated sequentially. Otherwise, the platform cannot guarantee the correct behavior of the application. Then, if there is an aspect evaluation rule stating that two aspects will be evaluated concurrently, and those

aspects share the same property, and it is an input property in one aspect and an output property in the other aspect, the validation tool will generate a warning notifying the problem.

6 Putting It All Together

In this section we briefly illustrate how we integrate the description of a AA written in the DAOP-ADL language into the DAOP platform, bridging the gap between design and implementation. As explained before AAs in DAOP are described using XML Schemas and XML documents. The DAOP-ADL language provides an XML Schema defining the structure of a DAOP application. Then, XML documents conforming that schema represent AA instances (AA1, ..., AAn of Figure 1) describing concrete applications (Figure 1, step 1). In DAOP it is expected that the software architect defines the AA during the design phase and afterwards the resulting XML file is stored in a repository as part of the application registration in the DAOP Server (Figure 1, step 2 and 3).

Users initiate DAOP applications from a browser by downloading a java applet that will retrieve a concrete AA. Therefore, when the user downloads the application applet from a web server (Figure 1, step 4), the AA is downloaded at the same time and it is stored in the instance of the DAOP platform created for that user (Figure 1, step 5) at the client side. The AA definition will drive the dynamic composition of components and aspects (Figure 1, step 6) throughout application execution.

7 Conclusions and Future Work

In this paper we highlight the importance of explicitly describing the software architecture of complex distributed system to increase their configurability and adaptability. We have presented an architecture description language to describe and validate software architectures that conform to the component-aspect model of the DAOP platform. We have used XML Schemas because XML is the standard language for representing structured information and provides appropriate mechanisms to validate the correctness of the AA.

The main contributions of our DAOP-ADL language proposal are the following: 1) It defines the AA independent from programming languages and component platforms. This means the language is general enough to be used in other contexts besides DAOP; 2) The definition of aspect information is optional. Then, it is possible to describe the AA of any component-based application, including aspects or not; 3) The dependencies between components and aspects are explicitly defined. This makes possible to reason about them and to analyze the impact of changing, modifying or adding new components and aspects both statically and dynamically at runtime; 4) It allows the use of validation tools to assure the correctness of components composition and aspect evaluation. This system is currently being used in a virtual office developed in the TRACOM

project. Our next goal is to improve the DAOP-ADTool tools with code generation. Since the information that we specify in XML can be easily adapted to the UML language, we plan to use UML as the graphic notation. The AA defined now in UML could be stored according to our XML Schemas of the DAOP-ADL language, and then incorporated to the platform as we showed here.

Acknowledgment

The authors wish to thank Kim Mens for his insightful comments and suggestions in an initial version of this paper.

References

1. Heineman, G.T., Councill, W.T. Component-Based Software Engineering: Putting the Pieces Together. *Addison Wesley*, 1st Edition, 2001.
2. Aspect-Oriented Software Development Web Site. *http://www.aosd.net*.
3. Kiczales, G., Lamping, J., Mendhekar, A. et al. Aspect-Oriented Programming. In *Proceedings of the ECOOP'97*. June 1997.
4. Medvidovic, N., Taylor, R. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transaction on Software Engineering*, 26 (1), January 2000.
5. Garlan, D. Software Architecture. *Encyclopedia of Software Engineering*, J. Marciniak (Ed.), John Wiley & Sons, 2001.
6. Aldrich, J., Chambers, G., Notkin, D. ArchJava: Connecting Software Architecture to Implementation In *Proccedings of the ICSE'02*, May 2002.
7. xAcme: Acme Extension to xArch *http://www-2.cs.cmu.edu/ acme/pub/xAcme/*
8. Khare, R., Guntersdorfer, M., Oreizy, P., et al. xADL: Enabling Architecture-Centric Tool integration with XML In *Proceedings of the 34th Annual Hawaii ICSS'01*, January 2001.
9. Pinto, M., Fuentes, L., Fayad, M.E., Troya, J.M. Towards an Aspect-Oriented Framework in the Design of Collaborative Virtual Environments. In *Proceedings of the 8th IEEE Workshop on FTDCS'01*, November 2001.
10. Pinto, M., Fuentes, L., Fayad, M.E., Troya, J.M. Separation of Coordination in a Dynamic Aspect-Oriented Framework. In *Proceedings of the 1st International Conference on AOSD'02*, April 2002.
11. Fuentes, L., Troya, J.M. Coordinating Distributed Components on the Web: an Integrated Development Environment. *Software-Practice and Experience*, 31 (39), pp. 209-233, March 2001.
12. Krieger, D., Adler, R.M. The Emergence of Distributed Component Platforms. In *IEEE Computer*, 31 (3), pp. 43-53, March 1998.
13. Lieberherr, K., Orleans, D., and Ovlinger, J. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, 44 (10), October 2001.
14. Kiczales, G., Hilsdale, E., Hugunin, J., et al. An overview of AspectJ. In *Proceedings of the ECOOP'01*, June 2001.
15. Wichman, J.C. ComposeJ: The Development of a Preprocessor to Facilitate Composition Filters in the Java Language. MSc. thesis, Dept. of Computer Science, University of Twente, December 1999.

16. Ossher, H., and Tarr, P. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer 2000.
17. Popovici, A., Gross, T., Alonso, G. Dynamic Weaving for Aspect-Oriented Programming. In *Proceedings of the 1st International Conference on AOSD'02*, April 2002.
18. Pawlack, R., Seinturier, L., Duchien, L., Florin, G. Jac: A flexible and efficient framework for aop in java. In *Proceedings of Reflection'01*, September 2001.
19. Byte Code Engineering Library (BCEL). *http://jakarta.apache.org/bcel/*.
20. Pawlack, R., Duchien, L., Florin, G. A UML Notation for Aspect-Oriented Software Design. In *Proccedings of AO modeling with UML workshop at the AOSD'02 conference*, April 2002.
21. Kniesel, G., Constanza, P., and Austermann, M. JMangler - A Framework for Load-Time Transformation of Java Class Files *IEEE Workshop on SCAM'01*, November 2001.
22. Bergmans, L., and Aksit, M. Analyzing Multi-dimensional Programming in AOP and Composition Filters. *First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (at OOPSLA '99)*, November 1999.
23. Burke, B., Brock, A. Aspect-Oriented Programming and JBoss. *http://www.onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html*
24. XML Schema. *http://www.w3.org/XML/Schema*.
25. Wang, N., Schmidt, D.C., O'Ryan, C. Overview of the CORBA Component Model. In *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley, Editors G.T. Heineman and W.T. Council. 2001.
26. Blevins, D. Overview of the Enterprise JavaBeans Component Model. In *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley, Editors G.T. Heineman and W.T. Council. 2001.
27. The Xerces 2 Java Parser. *http://xml.apache.org/xerces2-j/index.html*.
28. Extensible Markup Language (XML) 1.0. *http://www.w3.org/TR/2000/REC-xml-20001006*.
29. JavaBeans Component Architecture Documentation. *http://java.sun.com/products/javabeans/docs/*.