A Dynamic Component and Aspect-Oriented Platform

MÓNICA PINTO, LIDIA FUENTES AND JOSÉ MARÍA TROYA

Dpto. de Lenguajes y Ciencias de la Computación, University of Málaga
Campus de Teatinos s/n, E29071, Málaga (Spain)
Email: {pinto,lff}@lcc.uma.es

Component-based software development (CBSD) represents a significant advance towards assembling systems by plugging in independent and (re)usable components. On the other hand, aspect-oriented software development (AOSD) is presently considered as a possible technology to improve the modularity and adaptability of complex and large-scale distributed systems. Both are complementary technologies, so it would be helpful to have models that combine them to take advantage of all their mutual benefits. Thus recent research has tried to combine CBSD and AOSD by considering aspects as reusable parts that can be woven and then attached to the individual components. Our contribution to the integration of these technologies is CAM, a new component and aspect model that defines components and aspects as first-order entities, together with a non-intrusive composition mechanism to plug aspects into components. The underlying infrastructure supporting CAM is the dynamic aspect-oriented platform (DAOP), a component and aspect platform that provides the usual services of distributed applications, as well as a composition mechanism to perform the plugging of software aspects into components at runtime.

Received 12 December 2003; revised 6 August 2004

1. INTRODUCTION

Nowadays, traditional methods provided by software engineering are not sufficient to cope with the complexity of current distributed systems. Component-based software development (CBSD) [1] and, more recently, aspect-oriented software development (AOSD) [2] have emerged for the purpose of improving the modularity and evolution of systems by plugging in independent and (re)usable entities.

CBSD relies on achieving an accurate functional decomposition of a system into truly independent components, sometimes produced by third parties, ready to be (re)used in different contexts. The goal is the reduction of development times, costs and efforts, while improving the flexibility, reliability and maintainability of the final application.

AOSD, on the other hand, is a promising discipline that supports the separation of crosscutting concerns by introducing a new dimension called aspect. An important aim of AOSD is to provide a solution to what has been identified as the code tangling problem [3], which refers to the difficulty crosscutting properties have in changing or evolving independently of the objects they affect. In this sense, in AOSD aspects usually encapsulate crosscutting properties (synchronization, coordination etc.) that are defined as part of several objects in a system. Aspects define join points, which are the points in the execution of a program that can be intercepted; pointcuts, which describe the objects that aspects are applied to; and advice, which describes aspect behaviour.

CBSD and AOSD are complementary technologies. In this sense, AOSD can help to improve the independence, reusability, evolution and maintainability of components by extracting crosscutting concerns from components and putting them into aspects. Those crosscutting concerns can then be managed separately without affecting the evolution of the core functionality of components. Consequently, it would be helpful to have models that combine CBSD and AOSD to take advantage of all their mutual benefits. In the resulting component and aspect models, aspects may become reusable parts, which can be composed (woven in the AOSD terminology) and then attached to the individual components. Current research about the integration of CBSD and AOSD is in its initial stages [4, 5], so there is much more work to be done in this direction, as described below.

Recently, component models, such as EJB/J2EE [6] and the new CCM/CORBA [7], have evolved to face the challenge of achieving the definition and the use of common services outside component implementation—i.e. consider common platform services as crosscutting concerns. However, they have not yet found a proper solution to this tangled code problem, as they only separate a limited number of commonly used services in distributed systems (e.g. security, transaction and persistence). In order to incorporate the concept of aspect, these platforms should offer additional mechanisms to separate any kind of crosscutting concern or replicated code.

THE COMPUTER JOURNAL Vol. 48 No. 4, 2005
presents a survey of current distributed component platforms. Files [7] only in the deployment phase and not at runtime. The difference is that these models use the assembly descriptor assembly of components explicitly, for instance CCM. The application. Other component models also describe the aspect of pointcuts will be separated from components and aspects, so aspects may be (re)used in different contexts. Regarding the different AOSD approaches, many of them [8, 9] do not treat aspects as context-independent and reusable entities. This means that in addition to the aspect behaviour, aspects usually hard-code the aspect pointcuts. This prevents aspects from being reused in different contexts. In addition, the join points that can be intercepted by current AOSD approaches are not always appropriate in the context of CBSD [9, 10]. Considering components as black-box entities, aspects should not intercept points that are part of the internal behaviour of a component. Instead, they should only intercept the interactions among components.

In this paper we propose a component and aspect model (CAM) that combines the benefits of both CBSD and AOSD disciplines, trying to solve the shortcomings presented above. CAM defines components and aspects as first-order entities, together with a non-intrusive composition mechanism for plugging aspects into components. These aspects may be common services of distributed applications—e.g. security, persistence, or any kind of crosscutting or replicated code, but both are added to software components in a homogeneous way. Therefore, CAM components can be configured to use an open-ended list of crosscutting services. In addition, in our model, the aspect code does not include information about the components that it crosses. This means that the definition of pointcuts will be separated from components and aspects, so aspects may be (re)used in different contexts.

The underlying infrastructure supporting CAM is a component and aspect platform (DAOP, a dynamic aspect-oriented platform) where the plugging of software aspects into components is performed at runtime. Another relevant feature of our approach is that the architectural information about connections between aspects and components is not hard-coded as part of component and aspect implementations as usual; it is stored within the platform. Our platform composition mechanism consults this information at runtime to establish the connections among components and aspects. This is particularly useful because we make components and aspects much more reusable, isolating the dependencies between them in the platform's internal structures. In addition, this information can be adapted at runtime, improving the flexibility and adaptability of the final application. Other component models also describe the assembly of components explicitly, for instance CCM. The difference is that these models use the assembly descriptor files [7] only in the deployment phase and not at runtime.

The rest of this paper is organized as follows. Section 2 presents a survey of current distributed component platforms and aspect-oriented approaches. CAM is presented in Section 3 and DAOP in Section 4. We use a running example to describe the main characteristics of both CAM and DAOP. After describing the main features of DAOP, in Section 5 we show the contributions of CAM/DAOP to other CBSD and AOSD approaches. As a proof of concept, Section 6 describes the CAM/DAOP prototype, which has been implemented based on Java/RMI. Finally, Section 7 draws some conclusions and outlines some further work.

2. COMPONENT AND ASPECT BASED SOFTWARE DEVELOPMENT

In this section we place our work in an appropriate context, providing a survey of current distributed component platforms and AOSD approaches.

2.1. Component platforms

In this section we present the results of a comparative study of the best-known component platforms. The criteria used in this comparative study is related to the most significant contributions of our work, to be discussed throughout the rest of the paper (see Table 1).

<table>
<thead>
<tr>
<th></th>
<th>CORBA</th>
<th>CCM/CORBA</th>
<th>EJB/J2EE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separation of crosscutting concerns</td>
<td>Limited (interceptors)</td>
<td>Limited (container)</td>
<td>Limited (container)</td>
</tr>
<tr>
<td>Explicit AA definition</td>
<td>No</td>
<td>XML assembly file (components + connections)</td>
<td>XML descriptors (only components)</td>
</tr>
<tr>
<td>AA use</td>
<td>No</td>
<td>Deployment</td>
<td>Deployment</td>
</tr>
<tr>
<td>Interface description</td>
<td>IDL (only provided interface)</td>
<td>IDL (provided + required + event interface)</td>
<td>Java interface (only provided interface)</td>
</tr>
</tbody>
</table>

Regarding the different AOSD approaches, many of them [8, 9] do not treat aspects as context-independent and reusable entities. This means that in addition to the aspect behaviour, aspects usually hard-code the aspect pointcuts. This prevents aspects from being reused in different contexts. In addition, the join points that can be intercepted by current AOSD approaches are not always appropriate in the context of CBSD [9, 10]. Considering components as black-box entities, aspects should not intercept points that are part of the internal behaviour of a component. Instead, they should only intercept the interactions among components.

In this paper we propose a component and aspect model (CAM) that combines the benefits of both CBSD and AOSD disciplines, trying to solve the shortcomings presented above. CAM defines components and aspects as first-order entities, together with a non-intrusive composition mechanism for plugging aspects into components. These aspects may be common services of distributed applications—e.g. security, persistence, or any kind of crosscutting or replicated code, but both are added to software components in a homogeneous way. Therefore, CAM components can be configured to use an open-ended list of crosscutting services. In addition, in our model, the aspect code does not include information about the components that it crosses. This means that the definition of pointcuts will be separated from components and aspects, so aspects may be (re)used in different contexts.

The underlying infrastructure supporting CAM is a component and aspect platform (DAOP, a dynamic aspect-oriented platform) where the plugging of software aspects into components is performed at runtime. Another relevant feature of our approach is that the architectural information about connections between aspects and components is not hard-coded as part of component and aspect implementations as usual; it is stored within the platform. Our platform composition mechanism consults this information at runtime to establish the connections among components and aspects. This is particularly useful because we make components and aspects much more reusable, isolating the dependencies between them in the platform’s internal structures. In addition, this information can be adapted at runtime, improving the flexibility and adaptability of the final application. Other component models also describe the assembly of components explicitly, for instance CCM. The difference is that these models use the assembly descriptor files [7] only in the deployment phase and not at runtime.

The rest of this paper is organized as follows. Section 2 presents a survey of current distributed component platforms and aspect-oriented approaches. CAM is presented in Section 3 and DAOP in Section 4. We use a running example to describe the main characteristics of both CAM and DAOP. After describing the main features of DAOP, in Section 5 we show the contributions of CAM/DAOP to other CBSD and AOSD approaches. As a proof of concept, Section 6 describes the CAM/DAOP prototype, which has been implemented based on Java/RMI. Finally, Section 7 draws some conclusions and outlines some further work.

2. COMPONENT AND ASPECT BASED SOFTWARE DEVELOPMENT

In this section we place our work in an appropriate context, providing a survey of current distributed component platforms and AOSD approaches.

2.1. Component platforms

In this section we present the results of a comparative study of the best-known component platforms. The criteria used in this comparative study is related to the most significant contributions of our work, to be discussed throughout the rest of the paper (see Table 1).

<table>
<thead>
<tr>
<th></th>
<th>CORBA</th>
<th>CCM/CORBA</th>
<th>EJB/J2EE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separation of crosscutting concerns</td>
<td>Limited (interceptors)</td>
<td>Limited (container)</td>
<td>Limited (container)</td>
</tr>
<tr>
<td>Explicit AA definition</td>
<td>No</td>
<td>XML assembly file (components + connections)</td>
<td>XML descriptors (only components)</td>
</tr>
<tr>
<td>AA use</td>
<td>No</td>
<td>Deployment</td>
<td>Deployment</td>
</tr>
<tr>
<td>Interface description</td>
<td>IDL (only provided interface)</td>
<td>IDL (provided + required + event interface)</td>
<td>Java interface (only provided interface)</td>
</tr>
</tbody>
</table>
separately supplies a runtime environment that provides access to platform common services. The main advantage of the container approach is that software developers do not have to include code for accessing these services inside the component implementation. However, the use of the container programming model is restricted to the list of services offered by the platform vendor. Thus, it cannot be applied to model, as separate services, any other properties (e.g., fault-tolerance, synchronization and domain-specific properties) that may also be intermingled with the code of the core components.

A non-standard approach that tries to overcome the component platform’s limitation is that components can only access a predefined list of common services through the container is given in [12]. The approach of this work consists of separating the behaviour of components in a non-limited set of aspectual services, describing them in the interfaces of components as required or provided services. Although with this approach the evolution of extra-functional properties can be addressed separately, its main drawback is that components have to access the aspects directly through an aspect manager component, making components dependent on aspects. This proposal is implemented by extending the JViews framework, an extension of the JavaBeans components.

Other criteria to compare current component platforms are the mechanisms they provide to describe explicitly the assembly of components [i.e. the application architecture (AA) description] and how they use this information throughout the whole software life cycle. As mentioned earlier, the AA is not explicitly stated anywhere, but is usually spread throughout the component implementation modules. Consequently, the platform has neither the information about the architectural constraints nor the control to reorganize the system in order to adapt it to different user preferences or runtime environments. It would thus be very useful to have component platforms with the capability of interpreting architectural information, especially at runtime. Additional benefits of our approach with respect to this issue are presented in the following sections.

The traditional CORBA platform does not specify anything about how to assemble a CORBA application from a set of independent CORBA objects. To address this and other limitations, the Object Management Group defined CCM, which extends the CORBA object model to support components. In addition to component description, CCM/CORBA provides a way to express the architecture of an application. CCM applications are packaged in assembly files, which are XML documents describing components and their interconnections. This is a significant advantage, although this information can be used only during the deployment phase and not at runtime.

On the other hand, the EJB/J2EE platform also uses XML to describe an application in terms of descriptions of a set of enterprise beans (EJB deployment descriptors) and an assembly descriptor. The assembly descriptor provides configuration information common to all the beans in the application. The main restriction of EJB/J2EE is that it only describes the beans that set up an application and not the relationships among them. Similar to CCM/CORBA, EJB/J2EE uses the descriptor files primarily in the deployment phase to deploy the application in the adequate component servers.

Finally, component platforms also differ in the information they put in the description of the component interfaces. For instance, CORBA and EJB/J2EE only describe the provided interface of a component, i.e., the messages that the component can manage. This makes it difficult to (re)use components in different contexts, as there is no information about the messages and/or events that a component can emit. CCM/CORBA tries to solve this limitation by extending the traditional interface description language (IDL) of CORBA to include the description of all the messages and events that a component can both receive and emit.

2.2. Aspect-oriented approaches

In existing AOSD approaches, we can find different alternatives to tackle the separation of concerns issue, even though the final goal is always the same. These approaches differ mainly with regard to when aspects are applied (before, after or around), the join points where aspects are applied (a method invocation, a field access etc.), where the weaving information is placed (inside or outside the aspect definition), whether the weaving process is static (performed during compilation) or dynamic (performed at runtime), and whether they follow an invasive or a non-invasive model. Other differences, which we consider less relevant in order to compare current AOSD approaches with our approach, have been omitted for the sake of clarity.

The variety of AOSD approaches is so extensive that it is not possible to cover them all in this section. Consequently, we present those that we consider more relevant with respect to our work. A complete description of AOSD technologies can be found in [2]. First, some works define aspect-oriented libraries [13], which are characterized by offering static weaving and in which the functional code must include some information dependent on the applied aspects.

The definition of aspect-oriented languages is the most common approach. A broadly used aspect-oriented language is AspectJ [10], which is a superset of Java providing a new construction similar to Java classes to implement aspects. In AspectJ, the aspect encapsulates advice indicating aspect behaviour and also pointcuts indicating where aspects are applied, which drastically reduce aspects (re)use. Advices in AspectJ can be applied before, after or around method calls, field accesses and even points that occur ‘within’ a method execution, regardless of Java access modifiers. In addition, AspectJ can modify the structure and the hierarchy of Java classes, adding new methods, fields etc., by means of introductions. Although AspectJ is very flexible since it is able to intercept all these join points, it is more appropriate to use an non-invasive approach (with no access to the private part of an object) when aspects are applied to black-box components.

Aspectual collaborations (AC) [14] is another aspect-oriented approach that tries to solve this limitation
TABLE 2. Related work in dynamic AOSD approaches.

<table>
<thead>
<tr>
<th>CBSD concepts</th>
<th>No</th>
<th>No</th>
<th>No</th>
<th>Yes (Beans)</th>
<th>Yes</th>
<th>Yes</th>
<th>J2EE</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate advice and pointcuts</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Composition policy file</td>
<td>XML files</td>
<td>Yes</td>
</tr>
<tr>
<td>External configuration of pointcuts</td>
<td>No</td>
<td>.acc or XML files</td>
<td>XML files</td>
<td>No</td>
<td>No</td>
<td>Decorator-like wrappers</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Invasive model</td>
<td>Yes</td>
<td>JVM Class Loader</td>
<td>Yes</td>
<td>HotSwap Sun technology</td>
<td>No</td>
<td>Traps + connector registry</td>
<td>Continer</td>
<td>No</td>
</tr>
<tr>
<td>Dynamism</td>
<td>No</td>
<td>XML files</td>
<td>XML files</td>
<td>XML files</td>
<td>XML files</td>
<td>XML files</td>
<td>XML files</td>
<td>XML files</td>
</tr>
</tbody>
</table>

providing a non-invasive model that combines modular programming [15] and aspect-oriented programming to obtain properties such as encapsulation, hierarchical composition and external assembly. In AC, crosscutting concerns are defined as a set of collaborations. The most important advantages of AC are that: (i) only the behaviour exported in collaboration’s interfaces can be intercepted by AC’s asp ectual methods; (ii) the composition of a set of collaborations is performed externally to constituent collaborations.

Another feature of both AspectJ and AC is that the weaving process is static, mixing the component and aspect code at compile-time. Static composition provides high performance, but separation of concerns is lost at runtime. Although they use introspection to provide reflective information about join points at runtime, the number and type of join points affected by an aspect cannot be modified after compilation. Other similar approaches are given in [16, 17].

Another approach is to define aspect-oriented frameworks that provide template classes for modelling objects and aspects and a more or less dynamic composition mechanism. Recently, new proposals on aspect-oriented frameworks have emerged as an alternative to aspect languages and static weaving. Dynamic weaving is much more flexible than static weaving because the separation of concerns remains at runtime, enabling, in some cases, the late binding between components and aspects. These approaches are mainly based on a reflection mechanism that offers 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.

Table 2 compares several aspect-oriented frameworks that provide dynamic composition of aspects into objects or components. With the aim of comparing them with the main contributions of CAM/DAOP later, we are particularly interested in: (i) whether these systems incorporate the CBSD concepts; (ii) the separation (or not) of advice and pointcuts in different entities; (iii) the mechanisms they provide to express pointcuts; (iv) whether they define an invasive or non-invasive model; (v) the mechanisms they use to perform dynamic composition of components (or objects) and aspects. The last column of Table 2 shows the contributions of our approach, which will be detailed throughout the paper.

PROSE [9] is an aspect-oriented platform with dynamic composition that was not really developed for CBSD. PROSE aspects can be applied before and after method executions, field accesses and modifications, join points within the code of methods, class load etc. Therefore, like AspectJ, PROSE aspects can intercept points that are part of the internal behaviour of objects. Its main contribution is that the platform weaves and unweaves aspects directly in the Java Virtual Machine (JVM). This work is valuable but has the problem that it is based on the use of the JVM debugger interface. This means that the JVM has to run in debug-mode, imposing important performance restrictions on the entire application, even if there is no aspect to be applied. Recently, PROSE’s authors have been working on weaving aspects and objects by inserting the aspect advice directly into the native code generated by the just-in-time compiler. This seems to reduce the performance overhead that is introduced by the PROSE class-loader weaver. In addition, pointcuts and advice are implemented in PROSE in the class representing the aspect, with the drawback of reducing the (re)use of the aspect advice in different contexts.

Another similar approach is JAC [18], an aspect-oriented framework that uses the reflexive API BCEL [19] for adding aspects. Using this mechanism, aspects in JAC are dynamically deployed and undeployed on top of running application objects using wrappers and aspect containers. A wrapper task is divided into two parts, the before part and the after part (before and after a method execution, application start etc.). In contrast with PROSE and some aspect language approaches [8, 10], in JAC pointcuts are not specified as part of the aspect definition but in a third-party entity available at runtime, making aspects more reusable. In addition, JAC uses AspectComponent configuration files (.acc files) or XML files to configure the aspect evaluation rules (or pointcuts) externally. Although JAC uses the term ‘component’ to refer to one of the entities of the model, it does not really define a component platform and its components cannot be considered software components in the CBSD sense [1]. In addition, JAC provides special mechanisms to
implement distributed protocols that allow objects in different containers to collaborate.

Another aspect-oriented framework that performs dynamic composition of objects and aspects is AspectWerkz [8]. Using the HotSwap Sun technology [20], aspects are composed with objects at runtime, by modifying the objects’ byte code after class loading. This is an interesting approach that uses XML files to define pointcuts separately from aspect implementations. However, aspects in AspectWerkz are applied to objects and not to components. In addition, it defines an invasive model that makes it possible to intercept any point in the internal behaviour of an object and not only the behaviour exposed through the explicit interfaces of Java objects. Furthermore, this work has been specifically developed for Java systems as its weaving mechanism relies completely on Java technology.

The following AOSD approaches focus on applying the AOSD principles to the development of component-based systems. In this sense, JAsCo [5] is an aspect-oriented implementation language that defines a new component model that is compatible with the JavaBeans component model. Aspects in JAsCo can be applied, adapted and removed at runtime. The JAsCo language introduces two concepts: aspect beans that encapsulate advice and connectors that define pointcuts. This separation has the advantage that both advice and pointcuts can evolve separately, increasing the reuse of advice. One important drawback of JAsCo connectors is that they have to be compiled, reducing their runtime adaptation. Anyway, dynamic connector loading and unloading is possible in the JAsCo connector registry. The connector registry is notified when aspects have to intercept Java Beans components or when a connector has been loaded or unloaded. Aspects interception is performed only before or after a method execution and, therefore, in JAsCo no join point intercepting the internal behaviour of components is defined. In addition, an aspect can ‘replace’ the normal execution of a method.

Lasagne [21] defines a platform-independent architecture for dynamic customization of component-based distributed systems using decorator-like wrappers. The Lasagne system models aspects as extensions, which consist of several wrapper definitions. These wrappers have to implement the interface of the component they decorate. According to CBSD philosophy, in Lasagne aspects only intercept the incoming and outgoing component messages and never the private state of components. In Lasagne, the composition logic responsible for integrating extensions into the core components is completely separated from the code of the components, and of the extensions as well, increasing their reuse in different contexts. This information is specified in composition policy files that can be dynamically attached to the system, promoting the evolution of final applications. The main difference between Lasagne and other AOSD proposals is that they compose extensions at the instance-level instead of at the class-level. This gives Lasagne a runtime performance overhead, although the composition mechanism is much more flexible.

Finally, the JBoss AOP framework [4] is built on top of a J2EE application server named JBoss and tries to solve the limitation imposed by J2EE of providing just a set of built-in services, as mentioned before. Aspect advice in JBoss AOP is implemented using interceptors, which can intercept method invocations, constructor invocations and field accesses. Even though JBoss AOP is developed to apply aspects to JavaBeans components, JBoss interceptors can access fields regardless of whether they are public or private, and can even introduce new methods and fields within a class. Once again, these features affect the non-invasiveness required in a component model. The JBoss AOP framework has the advantage that it separates advice and pointcuts in different entities, where pointcuts are configured using XML descriptor files. Its main limitation is that currently it is not possible to add interceptors to a class that was deployed (i.e. loaded by the Java class loader) without having an associated pointcut definition.

3. THE COMPONENT ASPECT MODEL

The basis of our approach is the component and aspect model (CAM) that defines how to describe the structure of an application in terms of components, aspects and communication mechanisms.

Figure 1 shows an unified modelling language (UML) diagram with the basic entities of CAM and the relationships that can be established among them. We consider this diagram as the UML profile for CAM, so the entity names are UML stereotypes for modelling applications in terms of CAM. Figure 2 shows part of the CAM diagram of a chat application using these stereotypes. By applying the separation of concerns principle we model the core behaviour of a chat application, separated from other crosscutting application requirements such as authentication, persistence or message filtering. This makes it easier to (re)use the chat component in applications that may or may not require these properties. These aspectual properties can also be (re)used in other contexts as standalone entities.

The authentication aspect is applied when a user wants to join a chat, and this implies that the user has to introduce some identification information. The local chat component instance is created only if the user is registered in the system. In case we want to develop a persistent chat, the persistence aspect, which stores the current state of the chat component in a data store, must be applied. This is useful for latecomer users who join the application once it has been initiated. Finally, sometimes we may want to apply a filter aspect to filter some messages, for instance, according to user preferences (e.g. receive only messages from ‘peter’).

We describe the relationships among the components and aspects that appear in Figure 2 in the rest of this section. A broader description of how to generate an application’s CAM diagram can be found in [22].

3.1. Components and aspects

The main entities of CAM are components and aspects, as shown in Figure 1. Although in principle there is no restriction on the granularity of these entities, the distributed
FIGURE 1. The component and aspect model.

FIGURE 2. The component and aspect model of a chat application.
nature of the applications we want to design with CAM, and the way they are composed, i.e. components are distributed and they interact by exchanging messages, and aspects are dynamically plugged into components, impose some recommendations regarding their size and level of encapsulation. In this sense, we consider both components and aspects as coarse-grained encapsulated entities that can only act as units of composition with contractually specified interfaces and explicit context dependencies. They may be deployed independently and are subject to third-party composition (component definition by Clemens Szyperski [1]).

Since in CAM aspects are treated as a ‘special’ kind of component, both components and aspects share some common features. For instance, they may have a set of StateAttributes that represent their public state, i.e. the information that should be made persistent to be able to restore the state of a component or aspect instance. This information will be used to implement some properties such as fault tolerance or persistence.

In order to detach component and aspect interfaces from their final implementations, we assign a unique role name (in class Role of Figure 1) to identify both component and aspect classes. A role name identifies a specific functionality and will be played by a component that implements this functionality. These role names are architectural 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 a role name identifying the target of a message. We obtained good results with this role concept in previous work [23, 24], considerably increasing the reusability of components. In the example in Figure 2 we have a component with role name ‘chat’ and three aspects with role names ‘authentication’, ‘persistence’ and ‘userfilter’.

However, it is possible that several components play the same role inside a distributed application (e.g. a user is participating in more than one chat at the same time). In this case the model distinguishes between different instances of a role by the RoleInstance. Figure 2 shows that the component chat fulfills the ‘chat’ role, but if the same user initiates, for example, two chats, each one will have different role instances (e.g. ‘chat_with_peter’, ‘chat_with_susan’). Every chat instance has the ‘chat’ role name; those belonging to the same chat will also share the role instance name and those from different chats have different role instance names. Moreover, each component instance is identified uniquely by the Component IDentifier (class CID in Figure 1).

Another important goal of CAM is that aspects do not have any information about the other aspects applied at the same time to a component. Even if there is some kind of dependency, the aspects should not be aware of this. We cope with this problem of composition of non-orthogonal aspects by using the class Property in Figure 1. Properties are identified by a unique name, their type and current value. Thus, in CAM, non-orthogonal aspects indirectly interact to resolve their dependencies by sharing properties with the same name and type. Then, aspects that are evaluated concurrently should not have any dependence among themselves, and their effect on components should not interfere. Otherwise, aspects should be evaluated in sequence. Likewise, properties also help to define truly independent components, putting any kind of data dependency as a shared property.

As an example of the use of properties Figure 2 shows that aspects with role name ‘authentication’ and ‘userfilter’ share a property with name ‘username’. Once the user has been authenticated, the authentication aspect sets the value of this property to the name of the user. Then the filter aspect gets the value of the property when it is needed. The relevant issue is that the former aspects do not have any interaction among themselves since properties are stored in a third-party entity.

We consider that the use CAM makes of properties provides a simple but very effective solution to the problem of data dependency among model entities, mainly for two reasons:

(i) By using properties CAM permits the description of data dependencies during the design phase. This information is very useful to develop more independent and reusable entities, since software developers are aware of the information that a component or an aspect shares with the environment in which they will be executed. They also know whether the information has to be generated to the environment, as in the case of the authentication aspect, or has to be consulted from the environment, as is the case of the filter aspect.

(ii) The information consulted by an aspect may have been generated not only by another aspect but also by a component; the same occurs for components. This provides a homogeneous mechanism for sharing information, not only among components and among aspects but also between components and aspects. In our example, the property called ‘username’, which is generated by the authentication aspect, may also be consulted by the Chat component, i.e. to show a welcome message that includes the user name.

The other important feature of CAM is the way in which the entities of the model communicate among themselves. Following the standard practices of CBSD, components interact by exchanging Messages and by throwing Events. We understand the meaning of messages and events in the CBSD sense: messages are sent to a specific target entity and events are messages with no information about the target component.

In this sense, we describe the class Message in Figure 1 as a composition of a MessageHeader class, which identifies the source and the target of the message, and a Body class,

---

1 Although we share the definition of component by Clemens Szyperski, for the sake of simplicity, in this paper, we will not distinguish between components and component instances, so we use the term component to mean a set of classes assembled to be deployed together and executed as a single software unit.

2 In this context, we do not consider the terms ‘role’ and ‘role instance’ as equivalent to the terms ‘components’ and ‘component instances’ in [1].
which contains the message information. On the other hand, the Event class contains the body of the event and an EventHeader that only identifies the source component CID, omitting the destination component(s). The handling of events is resolved at runtime by a coordination aspect (class CoordinationAspect in Figure 1) that encapsulates a component interaction protocol that will decide which are the target components of a given event, at runtime. More details about this and other functionalities of the coordination aspect can be found in [25]. The different mechanisms defined by CAM to address the target component(s) of a message are described in Section 3.2.

In CAM, we consider that aspects are applied to black-box components. Therefore, in our model we intentionally avoid the definition of join points that intercept the internal behaviour of a component, as we only have access to a component through its public interface. Thus, CAM aspects can be applied before and after (incoming and outgoing) messages and events, and also before and after the creation and destruction of component instances, as shown in the different ‘applies to’ relationships for aspects in Figure 1.

In our example, components with the role name ‘chat’ communicate by sending the ‘sendText’ message, which updates the state of the chat component for all users. This message is received by components that also have the role name ‘chat’ and role instance name ‘chat_with_peter’. Aspects are applied to the send and receive UML associations between the Component and the Communication classes as established in CAM (Figure 1). Figure 2 shows that the aspect ‘persistence’ is applied when a component ‘chat’ sends the message sendText(‘text’), making the ‘text’ value persistent. This aspect should also be applied to the creation of components with the role name ‘chat’ to restore the last state of the component. We have omitted it in our example for the sake of simplicity. The ‘userfilter’ aspect is applied before the previous message arrives at the target component and a decision is made regarding whether this message is to be sent or filtered. In addition, the ‘authentication’ aspect is applied to the creation of components with the role name ‘chat’ to check if the user is registered in the application.

Similar to other component models, we describe the interfaces of components and aspects using an IDL. A component’s IDL describes not only the services the component provides to the environment, as is usual in component platforms (the provided interface), but also those services it requires in its interaction with other components (the required interface). How to evaluate aspects is described in an evaluated interface, instead of a provided interface as for components, that includes the join points (messages, events, creation and finalization of components) that an aspect is able to intercept and evaluate. By aspect evaluation we mean the execution of the corresponding aspect advice (specific details for the aspect evaluation process are given in Section 4.2.3). If an aspect is general enough to be able to evaluate any intercepted join point, this interface can be omitted. A log aspect is a good example of this because it normally intercepts all the messages to or from a component, storing this information in a file. In addition, when an coordination aspect is being evaluated, it can also interact with the components it coordinates. This means that coordination aspects also have a required interface describing the output messages. Context dependencies are also specified as part of the component and aspect IDLs.

In CAM these IDLs are part of an XML-based architectural description language (DAOP-ADL) [26] used to describe components and aspects, together with the composition rules that govern the weaving of components and aspects. Using this language, the application’s CAM (e.g. what Figure 2 shows) can be transformed to a set of XML documents that can be easily interpreted by a runtime environment—i.e. by our DAOP, as we will show later. The complete description of the language is beyond the scope of this paper and can be found in [26].

Figure 3 shows the XML document that represents the CAM of the chat application shown in Figure 2. This XML document describes the interfaces of all components and aspects in the application (the ‘providedInterface’, ‘requiredInterface’ and ‘evaluatedInterface’ shown in Figure 3), the properties used by components and aspects (the ‘property’ XML element) and the aspect composition rules that state how to plug aspects into components (the ‘aspectCompositionRules’ XML element). In this example, no messages are sent by aspects. Had they been sent, the required interface for aspects would be the same as that for components. By showing this figure here we want to emphasize that the application’s CAM (Figure 2) is equivalent to the XML document (Figure 3) which is directly interpretable at runtime. The composition rules element of the DAOP-ADL, which defines how to compose components among them and how to compose aspects with components, will be treated in depth in Section 4.1.1.

Here we want to highlight that describing the aspect composition rules (equivalent to the aspect pointcuts in AspectJ) outside the component and aspect definition, we detach components from aspects and aspects from other aspects, avoiding that they share any kind of knowledge. Components are not aware of aspects, since they have no information about the number and type of aspects they are affected by, or even if they are affected by any aspect at all. This provides enough flexibility to apply different aspects, depending on the context in which the component is being used.

For instance, going back to our example, if later we want the chat to be non-persistent, we just need to remove from the XML document the aspect composition rule associated with the component having the role name ‘chat’ that describes the application of the persistence aspect (%BEFORE_SEND%<的消息 sendText /messages)(aspectList) persistence (%aspectList% /BEFORE_SEND)). Neither the component nor the aspect are affected at all. In addition, avoiding explicit pointcuts in the aspect definition, CAM promotes aspect reuse. Thus, aspects are also independent of the components they affect and can then be applied to different components at different times. The aspect composition rules shown in
FIGURE 3. The DAOP-ADL description of the CAM for the chat application.
Figure 3 are only an example, and more complex rules, considering wildcards and connectives to specify pointcuts, can be defined.

3.2. Component-addressing mechanisms

Although an important issue in CBSD is to make the entities of the system as independent as possible, components finally interact with other components. In this sense, a component needs to put a target component address as part of an output message, but this establishes a dependency between both of them. In our model, we try to reduce these dependencies by providing a mechanism to discover the components that play a certain role at runtime. We use the concept of role name to identify the target component(s) of an output message. We never use the component interface or implementation class names to reference components. This means that components do not maintain hard-coded references to other components.

As mentioned before, a CAM component has a role name, a role instance name and a CID. Thus, CAM offers a total of three kinds of component-addressing mechanisms, always avoiding the use of direct references among components: the role-based invocation, the role instance-based invocation and the identifier-based invocation.

The role-based invocation mechanism is the most important one. Using it a component is addressed by means of a string that identifies the role that the component plays in the system (e.g. 'components with role names ‘videoconference’, ‘chat’). However, as mentioned before, sometimes an application has more than one component playing the same role. In this case, the role instance name allows discriminating between different component instances with the same role name. The role instance-based invocation mechanism is then used to identify different instances of the same resource (e.g. among all the components playing the role name ‘chat’ only those that, additionally, have the role instance name ‘chat_with_peter’, ‘chat_with_susan’).

Finally, the identifier-based invocation addresses a component by means of its CID. As mentioned before, each component has its own CID. As the CID of a component can only be known if a message from that component has been received before, it is normally used in response messages. This kind of component-addressing mechanism is specified as part of the MessageHeader class in Figure 1.

The use of these component-addressing mechanisms provides a powerful and very flexible mechanism for late binding of components. Note that if during component communication component binding is performed using the interface name of the target component, as CORBA does, i.e. referring to the class name that contains the component interface, this forces the target component to implement the interface with this name. However, by using the role name, the role instance name or CID, we only assume that the target component is able to manage the message or event involved in a specific interaction.

These addressing mechanisms rely on a communication service’s ability to resolve for each role name which component implementation will receive a message or event. This communication service is part of the DAOP described in Section 4.

4. THE DYNAMIC ASPECT-ORIENTED PLATFORM

DAOP is a distributed component and aspect middleware platform for running applications conforming to CAM. DAOP is considered to be a global configuration entity that performs the runtime composition of components and aspects.

Figure 4 shows the architecture of the DAOP middleware platform, which contains information about the services and facilities the platform offers to components and aspects (the elements that appear above the DAOP Platform class in Figure 4), together with the information the platform stores to provide such services (classes below the DAOP Platform class in Figure 4). In the rest of this section we first describe the infrastructure of DAOP and afterwards the main services offered by the platform.

4.1. Infrastructure of DAOP

In this section we explain the internal structure of DAOP. We want to point out that DAOP was designed to be independent from any supporting language and distributed object platform, similar to Lasagne and contrary to other approaches such as JAsCo or JBoss/AOP that rely on Java and the EJB model. Basically, DAOP arranges its internal information in two objects. The ApplicationArchitecture object that stores the architectural description of the application; and the ApplicationContext object that holds the current list of component instances, aspect instances and property instances. The information contained in these objects is used to implement the services offered by DAOP.

4.1.1. The application architecture

As mentioned in the introduction, an important feature of our approach is that DAOP stores a description of the AA, which can be consulted by the platform to perform component and aspect instantiation and composition. This information was specified using the XML-based ADL (DAOP-ADL) [26] as shown in Section 3. XML documents written using DAOP-ADL represent AA instances (the example in Figure 3) describing concrete applications (the CAM of a chat application in Figure 2). During the execution, when an instance of DAOP is created, the XML document is parsed and the structure of the CAM application is stored in the ApplicationArchitecture class (see Figure 4).

In our approach, the AA is described in terms of a set of component and aspect definitions (the EntityInfo, EntitySpec, ComponentSpec, AspectSpec, CoordinationAspectSpec and EntityImpl classes) and the connections between them (the CompositionRule class). Component and aspect connections are described by means of a set of component composition rules (the CompCompositionRule class), which define how to glue software components, and a set of aspect composition
FIGURE 4. The DAOP architecture.
rules (the AspectCompositionRule class), the pointcuts in AspectJ terminology. According to our CAM, the information about the data shared among several components and aspects, i.e., properties, are also defined as part of the AA specification (in the PropertyInfo class), where properties are described by a name, a type and a value. Finally, the InitialContext class identifies, by their role names, the list of components that will be instantiated.

Observe that there is a correspondence between the XML elements in Figure 3 (component, aspect, compositionRule, compCompositionRule, aspectCompositionRule and property) and the classes mentioned above. This means that the information used at runtime by DAOP to instantiate components and aspects and to perform their composition is exactly the same architectural information generated in XML from the UML diagram of the CAM application. With this approach we close the usual ‘gap’, or loss of information, between the design and the implementation levels—the information about the software architecture of an application is usually lost at the implementation level. Furthermore, DAOP even uses this information at runtime, thereby ensuring that connections established at runtime are exactly those defined in design. As mentioned in Section 2.1, current platforms such as CCM use this kind of information only for deployment purposes.

Component and aspect descriptions. According to CAM, components and aspects are identified by their role names (the roleName attribute in class EntityInfo). Observe that the class EntityInfo and the class PropertyInfo contain information about the CAM elements shown in Figure 1 (the Entity and the Property classes respectively). A CAM entity (an aspect or a component) is described by an interface specification and a list of implementation classes that realize that interface. An entity’s interface specification (the EntitySpec class) contains information common to both components and aspects, i.e., the list of properties an entity can get or set, the list of their state information and some deployment information.

The component specification is completed with the description of the provided interface and the required interface (see ComponentSpec class). By defining not only the provided interface, but the required interface as well, DAOP encourages the (re)use of DAOP-based COTS components, which can be inserted in an application considering only the interface description in DAOP-ADL. In addition, the aspect evaluated interface in the class AspectSpec completes the aspect specification and the aspect required interface in the class CoordinationAspectSpec stores the required interface of coordination aspects.

The deployment information is also described in XML using the DAOP-ADL language. Components can be local or remote. Local components are created in the local instance of DAOP. Remote components are instantiated in a specific DAOP’s URL.

Regarding the number of aspect instances, the aspect deployment information describes whether DAOP creates: (i) only one instance of an aspect shared by all DAOP instances connected to the same application (an environment-oriented aspect), providing a centralized aspect; (ii) one instance of the aspect for each instance of a DAOP (a user-oriented aspect), providing a distributed aspect; (iii) an aspect instance that is shared by all components playing the same role (a role-oriented instance). Consider a collaborative shared environment that, in addition to the chat components with the role name ‘chat’, can create components modelling whiteboard tools, with the role name ‘whiteboard’. If a persistence aspect is defined as a role-oriented aspect, chat components may use a LDAP-based implementation of the persistence aspect while whiteboard components may use an ORACLE-based implementation. (iv) an aspect instance for all the components sharing the same role and the same role instance name (a role-instance-oriented aspect). In this case, different chats with different role instance names can be initiated as part of the collaborative shared environment mentioned above, and each of them may use a different implementation of the persistence aspect.

This variety of aspect instantiation modes provides great flexibility to final applications, and are particularly suitable in those applications that require a lot of adaptability and configurability, as in the case of the previously mentioned examples. Otherwise, the software architect can simply choose between environment-oriented and user-oriented aspects, which cover the needs of most applications.

The other deployment information for aspect is ‘criticality’. Aspects are classified as critical aspects and non-critical aspects, depending on how important the result of the evaluation of that aspect is to continue or not the application execution. For critical aspects, if the evaluation of the aspect is not successful, the execution does not continue. For instance, the authentication aspect should always be critical to ensure that if the evaluation of the aspect fails, the chat component will not be created. On the other hand, if the aspect is a non-critical aspect, the invocation will proceed although the aspect evaluation failed. An example of a non-critical aspect may be the persistence aspect. Even though making persistent the messages that are interchanged by users in a chat application may be a desired property, this is not a requirement for the application to function. Deploying the aspect as a non-critical aspect, the execution will continue even if the database shuts down, though with no persistence.

By expressing the criticality of aspects as part of the deployment information, our approach increases once again the flexibility of final applications. Thus, during the deployment of the application the software architect decides how the result of aspect evaluation affects the execution of a specific application, independently of how aspects were implemented. This information can be adapted and configured to easily generate different versions of an application with the same AA but different behaviour. Furthermore, these modifications can even be carried out at runtime.

The information about the properties used by components and aspects is also specified as part of the components and aspects descriptions in order to make their context-dependencies explicit. The listOfProperties attributes in the ComponentSpec and AspectSpec classes contain the
names of the properties used by the component or the aspect. The information about these properties is described in the PropertyInfo object as mentioned before. Property deployment information describes properties as userSite, if there is a property instance for each DAOP instance (e.g., the case of the ‘username’ property of our example), or serverSite, when all DAOP instances share the same property instance (e.g., a property that contains the URL of the user preferences store).

Finally, one or more classes providing different implementations for a component or an aspect (stored in the EntityImpl class in Figure 4) must comply with the specifications for that component or aspect stored in the ComponentSpec and AspectSpec classes. Each EntityImpl contains the name of an implementation class and whether that implementation has been selected as the current implementation.

**Component and aspect composition rules.** The last but most important subclasses of the ApplicationArchitecture object are the CompCompositionRule and the AspectCompositionRule objects. These objects store component and aspect composition rules, which describe the relationships among components and aspects, and some information for adapting mismatched interfaces. We illustrate the runtime composition mechanism of DAOP by showing how it composes two chat components instances with the ‘persistence’ and ‘userfilter’ aspects using the architectural information stored in the AA and application context objects (see Figure 5). Figure 5 also depicts some implementation code in Java, although DAOP can be implemented in other languages.

As explained above, components and aspects use role names in the implementation code to send messages (see lines 12 and 14 of code in the implementation of the chat component in Figure 5). Since component connections are established by mapping component role names, every role name assigned to components in the AA (actual role names) has to match the role names hard-coded in components (formal role names). The adaptation, if needed, will be performed at runtime by the platform using a set of component composition rules (‘CompCompositionRule’ object).

Going back to our example, in Figure 5 the programmer hard-coded the ‘collabApp’ role name to identify the target component of the message sendText(‘text’) (see line 12 in Figure 5 again). However, the software architect used the role name ‘chat’ to identify this component in the description of the AA (see Figure 3 in XML). This role mismatching is resolved at runtime using the component composition rule shown in Figure 5. Of course, during the description of the AA, the software architect will have to validate that the message ‘sendText’, which is sent to the component with the formal role name ‘collabApp’, is part of the provided interface of the component with the actual role name ‘chat’.

On the other hand, aspect composition rules define when and how to apply aspects to components. They are expressed in terms of five elements, as shown in the ‘AspectCompositionRule’ class in Figure 4. The ‘sourceCompRole’ (sRole in Figure 5) and the ‘targetCompRole’ attributes (tRole in Figure 5) identify, by their role names, the source and target components that are affected by a rule. The ‘messageName’ attribute specifies the message the rule is applied to. The ‘aspectRole’ attribute is a list that specifies the role names of the aspects that will be evaluated. This list is implemented using a bi-dimensional array of strings with the format [{A1}, {A2}, {A3, A4}] where every Ai is an aspect role name. This bi-dimensional structure allows us to specify two kinds of aspect evaluation: sequential evaluation and parallel evaluation. Aspects enclosed in the outer brackets, for instance A1 and A2, are evaluated sequentially. On the other hand, aspects in the inner brackets, for instance A3 and A4, will be evaluated concurrently.

We encourage defining composition rules for all the possible components and aspects that may be instantiated during the execution of the application, so user customization of an application may imply deleting and later adding an aspect. This is a relevant feature of our platform because the number and type of aspects applied to the running application can be adapted at runtime without code recompilation and not at ‘load’ time as usual [4, 18]. Later, in Section 4.2.6, we describe the AA configuration service, offered by DAOP, used to adapt the AA at runtime.

Finally, an aspect evaluation rule specifies the moment at which aspects are applied (the ‘when’ attribute). The available join points are enumerated in the object AspectJoinPoint, and are BEFORE_SEND, AFTER_SEND, BEFORE_RECEIVE and AFTER_RECEIVE to apply aspects before and after sending and receiving a message or an event, and BEFORE_NEW, AFTER_NEW, BEFORE_DESTROY and AFTER_DESTROY, to apply aspects before and after creating or destroying a component instance. We want to highlight that although we define ‘before’ and ‘after’ rules, CAM aspects can also be applied ‘around’ a join point as other approaches do [9, 10, 18]. Since CAM aspects can send messages to components, they can replace or modify component interactions forwarding messages to a different target and even running additional code.

Consequently, once again one of the most relevant advantages of our platform is that the moment of the evaluation (when creating/destroying components and when sending and/or receiving messages and events), the kind of evaluation (sequential or parallel), and the information about which and how components are affected by aspects is not hard-coded as part of the component or aspect implementations, increasing their reuse. Instead, as shown in this section, this information is taken out of components and aspects and stored in a third entity, i.e. the ApplicationArchitecture class inside DAOP, achieving the component and aspect independence that we claimed in previous sections.

### 4.1.2. The application context

DAOP stores the references of all the components, aspects and properties currently instantiated for a specific application in the ApplicationContext object. DAOP maintains information about component, aspect and property definitions inside the ApplicationArchitecture class, but
M. Pinto, L. Fuentes and J. M. Troya

FIGURE 5. A snapshot of a running DAOP.

references to their local or remote instances are in the application context. Thus, the ApplicationContext class is like a name service that links component and aspect instances with their role names and, likewise, property instances with their property’s name.

Before component, aspect or property creation, DAOP consults the information about these entities stored in the ApplicationArchitecture class, uses this information to instantiate the corresponding entity and, finally, stores the new instance in the ApplicationContext class. Then, the platform resolves, at runtime, which instance is associated with the specified role name, by simply consulting this object.

4.2. Services of DAOP

Similar to other component platforms, DAOP provides a set of common services to develop distributed applications, such as the instantiation of components, the communication of components, the evaluation of aspects, the storage of properties, the persistence service and the dynamic adaptation of the AA. DAOP uses the information stored in
the ApplicationArchitecture object and the ApplicationContext described above to implement all these services.

4.2.1. Component and aspect instantiation

DAOP components and aspects can create or destroy other components using the corresponding methods of the ComponentFactory interface (see Figure 4). The syntax of these methods are createComponent(String rolename, String roleinstancename) and destroyComponent(String rolename, String roleinstancename). Note that components are identified by a role and a role instance name. Thus, a software component just needs to specify strings with the role name and the instance name, and never its implementation class. Likewise, aspects are also created or destroyed, but only by the platform using a similar interface that also identifies aspects by their role name. This interface is not shown in Figure 4 because aspects in our approach are created by DAOP when needed and not directly by applications. As stated before, DAOP stores the names of the component and aspect implementation classes in the EntityImpl object as part of the application architectural information described in the ApplicationArchitecture object.

From the point of view of the software developer, the implementation of a component or an aspect in our approach is extremely simple compared with other component platforms. In DAOP, software developers do not have to manage any matter related to the implementation of remote objects, such as the definition of remote interfaces and implementations, generation and distribution of stubs and skeletons. For instance, in the current implementation of our model, DAOP components and aspects are implemented as simple Java objects.

Again refer back to Figure 5 which shows the implementation of the chat component and the persistence aspect. A DAOP component implementation must include: (i) the definition of a CID attribute to store the component identifier, (ii) a reference to the DAOP and (iii) a constructor method with two parameters that are the CID and a reference to DAOP. This constructor is not directly invoked by components. Instead, when a component uses the ComponentFactory service to instantiate a new component, it is DAOP that invokes this constructor, setting its reference and the CID inside the component.

The implementation of an aspect basically consists of providing the behaviour of the eval() method. In addition, the aspect programmer must provide the following code: (i) a variable to hold a reference to the platform and (ii) a constructor method with one parameter that is a reference to the platform. As for components, DAOP will invoke this constructor to create the aspect, setting its reference inside the aspect. Using this reference, aspects can access some of the services offered by the platform, such as the properties storage or the persistence service.

4.2.2. Component communication and coordination service

DAOP also offers a set of component communication primitives. As in other component platforms (e.g. CORBA), DAOP allows components to send synchronous and asynchronous messages, as well as to broadcast a message to several targets. DAOP also allows components to throw events to other components.

DAOP implements four primitives in order to send messages and events between components (see the CommunicationService interface and all its subinterfaces in Figure 4):

(i) execute(Message m). This method performs the delivery of asynchronous messages, addressing the target component by using any of the mechanisms described in Section 3.2.

(ii) broadcast(Message m). This method broadcasts, asynchronously, the same message to all components addressed as target components. It can be used, for instance, to send the same message to all components with the same role name (line 14 of Figure 5).

(iii) execmi(Message m). This method behaves the same way as execute, but the invocation is synchronous.

(iv) event(Event e). Using this method, components can throw events to other components. Communication by events is very useful to decouple components and is specially suited to enable (re)use. Components throw events to notify the environment that something happened in its internal state. By intercepting the throwing of events, DAOP provides a join point that occurs within the execution of a component method, similar to other approaches like AspectJ or PROSE. The difference with these white-box approaches is that DAOP can only intercept the points that the component makes visible throughout the throwing of events, considering it a black-box component.

As explained earlier, the handling of events is resolved at runtime by a coordination aspect. By applying the separation of concern principle to separate coordination we do not force the use, for instance, of a publish and subscribe mechanism, as other platforms do. Instead, the coordination aspect may implement this mechanism, but it may also contain rules such as ‘if an event e is intercepted and the source component has the role name source_crole, then a message msg is sent to the target component(s) with the role name target_crole’.

We want to highlight that this information is described in XML and is loaded into the coordination aspect when it is instantiated (more information in [25]).

Going back to our example in Figure 5, let us only consider the communication between the two chat components, postponing the aspect evaluation to the next section. This communication is performed through DAOP by invoking the broadcast communication primitive (step 1). DAOP checks the CompCompositionRule classes to see if it has to adapt the role name of the target component (step 2). Then DAOP obtains the reference of the component(s) with the role name ‘chat’ from the ApplicationContext object (step 5) and, finally, will invoke the ‘sendText’ method on these component objects (step 7).

We want to point out that using CAM/DAOP, developers can build component-based applications without considering aspects. This is possible because there are no direct
references between components and aspects, and also because the component composition mechanism presented in this section is independent of aspect evaluation.

4.2.3. Aspect evaluation
The component composition mechanism described above is extended in DAOP to incorporate dynamic evaluation of aspects. When a component creates or finalizes other components or sends a message or an event using any DAOP communication primitives, DAOP intercepts it and evaluates the corresponding aspects.

DAOP aspects should implement the AspectEvaluationService interface. This is mandatory since the platform will invoke the eval(Message m) or the eval(Event e) methods to evaluate an aspect. This is the only requirement an aspect must fulfill to be recognized by the platform as a valid DAOP aspect implementation. The eval() method contains the aspect functionality or aspect advice in AspectJ terminology.

Going back to Figure 5, take a look at the aspect evaluation rules stored in the AspectCompositionRules object (step 3). Rule number 2 states that before sending the sendText("text") message, DAOP has to evaluate the ‘persistence’ aspect (step 4). Then, once the message has reached the target DAOP(s) and before the target component receives the message, rule number 3 indicates the evaluation of the ‘userfilter’ aspect (step 6). Finally, if all aspects are evaluated successfully, the message is received by the target component(s) (step 7). If an error occurs, the other aspects are not evaluated and the message does not reach its target. DAOP throws an exception indicating that something went wrong during message delivery, although no information about the aspect that caused the problem is sent, since the component is not aware of aspects.

4.2.4. Property storage
In Section 3 we discussed the benefits of defining component and aspect properties to solve data dependencies between entities of CAM. During execution a producer entity will set the value of a property with the setProperty(String propertyName, Object value) method of the PropertyService interface in Figure 4, and later the consumer entity of that property will get its current value with the get(Property(String propertyName) method. Property instances are stored in the ApplicationContext object of DAOP.

4.2.5. Persistence
The PersistenceService interface provides the functionality required to store and retrieve the current state of components. This service may be used to implement a persistence aspect that simply has to invoke the storeComponent(CID component) and retrieveComponent(CID component) methods. The implementation of these methods serializes or de-serializes the attributes that are part of the state of a component (see the State class of CAM in Figure 1), and then stores or retrieves them in a data repository of DAOP.

Note that in Figure 5 the implementation of the persistence aspect (see line 13) uses this service to store the state of the source component, identified by its CID. This information is extracted from the class Message.

4.2.6. AA configuration
The AAConfigurationService interface provides a set of methods to modify at runtime the software architecture of the application, which is stored in the ApplicationArchitecture object as described before. It is possible to add, modify or remove the description of components, aspects, properties or even composition rules using the corresponding methods of the AAConfigurationService interface. The information provided with these methods is used to modify or adapt the architectural information contained in the EntityInfo, PropertyInfo and CompositionRule classes of the AA (lower side of Figure 4). This service is very useful to configure the application dynamically, adapting it according to the user preferences or to any necessity of the current execution environment. The correctness of the resulting AA is checked after a modification request.

A few examples of the adaptations that can be performed in our chat application, and the information that has to be added or modified in the ApplicationArchitecture object of Figure 4, are as follows: (i) replace the implementation of the chat component by an improved one. This only requires selecting the new implementation as the current implementation (see the current attribute in the EntityImpl class in Figure 4); (ii) convert the chat application into a free-access application by removing the aspect evaluation rule that applies the aspect with the role name ‘authentication’; (iii) add a new aspect and/or component into the application by adding its description to the AA and by modifying the current aspect evaluation rules to include it in the proper place.

By using this service, software developers can easily plug and unplug aspects into applications even at runtime. For instance, we may trace the connection of users into the chat application modifying the aspect composition rules for adding a trace aspect after the creation of the chat component.

5. RELATED WORK
After comparing our model with CORBA and the container programming model of EJB/J2EE and CCM/CORBA, we have found basic differences regarding: (i) the mechanisms they provide to cope with the separation of crosscutting concerns; (ii) the mechanisms they use to address components and services; (iii) the composition mechanisms between components and services; (iv) the description and use of the software architecture of applications, including component interfaces.

The number and kind of crosscutting properties that can be managed as independent services are broader in our approach. Whereas CORBA, EJB and CCM offer a concrete number of services that cannot be extended by users (e.g. transaction, security, persistence and notification), in our approach it is possible to separate any crosscutting property, for instance coordination, trace code, fault tolerance, distribution. The difference can be found in how both approaches manage these properties. While in CORBA, CCM/CORBA and EJB/J2EE
the provision of these services relies on the platform provider, components and aspects in CAM are first-class entities that coexist at the application level. Consequently, CAM/DAOP provides applications’ developers with adequate mechanisms to divide the application functionality into components and crosscutting properties modelled as aspects.

The addressing mechanisms of components and aspects (or services) also differ with respect to CORBA, CCM/CORBA and EJB/J2EE. During interactions, components need to reference other components. Using J2EE and the static invocation mechanism of CORBA, the client obtains the reference to a component and then communicates directly with it. This implies that the client code hard-codes the reference of the target component. The main problem is that this creates undesirable dependencies that reduce the (re)use of single components in different contexts. In our approach, we try to reduce the dependencies of a component with the environment by defining role names, which avoids having hard-coded references in the component code. The dynamic invocation mechanism of CORBA also decouples components from the interactions in which they participate, resolving the target component at runtime. However, unlike DAOP, with CORBA the applications themselves, and not the CORBA ORB, are the ones which have to include the mechanisms to decide at runtime the target component’s IDL. This makes the implementation of dynamic CORBA applications much more complicated than those using static invocation [27].

In addition, in our approach aspect composition rules are stored inside DAOP and are consulted during component communication, so we never include this information as part of component or aspect references. This is an important difference with other interception mechanisms, such as CORBA interceptors. CORBA hard-codes information about the interceptors that have to be applied to a specific object as part of the object’s reference, making it impossible to modify this information once the object reference has been created.

Another important difference is the way in which the information about the AA is managed. Comparing our approach with current component platforms, CCM/CORBA also provides information about the AA in XML. However, CCM/CORBA only uses that information during the application deployment and not at runtime, as DAOP does. Therefore, the kind of services that can be added to a component are established during the instantiation of the container in a static way, and they cannot be adapted at runtime as DAOP does, which we showed in the previous section. EJB/J2EE only describes components but not their composition rules, and CORBA only describes object IDLs. In DAOP the information about AA can be modified at runtime, it is not necessary to shut down and recompile the application to change its behaviour, increasing the flexibility and adaptability of final applications.

Regarding AOSD approaches, we compare them with CAM/DAOP according to the same criteria used in Section 2.2. In this sense, PROSE and CAM/DAOP differ from CAM/DAOP in that they do not apply aspects to components but to objects. The other difference between PROSE and CAM/DAOP is that PROSE does not separate advice and pointcuts in different entities, reducing the reusability of aspects in different contexts. Also, contrary to CAM/DAOP, which postpones the weaving of components and aspects until runtime, PROSE binds aspects into objects at load time, it being impossible to change the number and kind of aspects applied to an object after the object class has been loaded by the JVM Class Loader.

Although JAC aspects separate advice and pointcuts into two different entities and, additionally, JAC pointcuts are externally configured using XML files, there is an important difference between JAC and CAM/DAOP. Our approach is neither a client/server approach nor does it follow a container model as JAC. Instead DAOP is a distributed platform that does not need to define extra mechanisms, such as JAC distributed protocols, to distribute aspects in different hosts. An application in DAOP is distributed among different hosts where a local instance of DAOP is running. These DAOP instances communicate among themselves, it being possible for all the components and aspects in a DAOP application to communicate and collaborate among themselves. The aspect instantiation mode, described in Section 4.1.1 determines the number of instances that DAOP creates for each aspect and how they are distributed.

CAM/DAOP and AspectWerkz have some similarities since both approaches perform the weaving of aspects at runtime, and both separate pointcuts definition and advice into two different entities, using XML files to declare pointcuts. However, there are also important differences among them. The most important one is that AspectWerkz has been developed specially for Java, while CAM/DAOP is technology-independent. In addition, CAM/DAOP defines a distributed model for the development of component-and aspect-based application, while AspectWerkz applies only to objects. Furthermore, even if AspectWerkz offers the possibility of applying remote aspects following a client–server approach, it does not explicitly support the development of peer-to-peer distributed applications as CAM/DAOP does.

Focusing now on those approaches that provide a CAM we compare CAM/DAOP with Lasagne, JAsCo and JBoss. JAsCo and JBoss both provide a component-aspect model that is an extension of the EJB component model; Lasagne, on the other hand, defines its own architecture based on a dynamic decorator-like wrapper mechanism that is highly suitable for supporting different client-specific views on the core components, which is an important aim in Lasagne. However, regarding the reusability of aspects in different contexts, the main drawback of Lasagne is that aspects are always dependent on the components they decorate, as they have to implement the component interface, reducing aspect reusability enormously.

On the other hand, the most important difference between JBoss and CAM/DAOP is that while JBoss binds components and aspects at load time, DAOP components and aspects remain separate, even at runtime. Consequently, in CAM/DAOP Java classes do not need to be reloaded.
in case the aspect composition rules change, as in JBoss AOP. Another important difference is that JBoss AOP aspects can intercept private parts of a component, which may be considered as not being very legal.

Finally, JAsCo provides a solution that is very similar to our approach, where the JAsCo connector registry plays the same role as our DAOP platform and aspects are also attached to components at runtime. The main difference is that our approach is more flexible to changes than JAsCo for two main reasons. First, although both separate aspect advice and pointcuts into two different entities, JAsCo’s pointcuts are not described using a declarative language, so after changing a pointcut the connector needs to be (re)compiled. Second, CAM/DAOP aspects are more adaptable at runtime than JAsCo aspects. The reason is that JAsCo connectors only provide information about pointcuts, but the information about when the aspect is applied (before or after the execution of the intercepted join point) is hard-coded as part of the aspect behaviour. In CAM/DAOP both kinds of information are described externally to aspects, using DAOP-ADL, and both can be adapted at runtime.

6. CAM/DAOP PROTOTYPE

As a proof of concept, CAM/DAOP has been implemented based on Java/RMI as the base communication mechanism and the Java reflective package for dynamic composition. Users initiate CAM/DAOP applications by downloading an application applet through a DAOP Application Directory. An instance of the distributed DAOP is created at each user site during applet downloading.

We want to point out that the ApplicationContext and the ApplicationArchitecture objects in DAOP (see Figure 4) may be implemented either following a centralized or a distributed approach. In our prototype, we have implemented them as distributed entities. This means that the ApplicationContext object is present along the different instances of DAOP, each one containing the components and aspects locally instantiated. This is transparent to components and aspects, which rely on DAOP to localize the rest of the components and aspects in the application.

Regarding the ApplicationArchitecture object, the application architectural information is replicated in each instance of DAOP. During the application registration, the document describing the AA (using DAOP-ADL) is parsed and the objects that compose the ApplicationArchitecture are filled and, finally, serialized. Once a user joins an application, the AA information is downloaded as part of the application specific applet, is deserialized at the user site and the information is stored in the platform. The platform assures that this information remains consistent if changes are performed by some other user.

Our prototype also supports the use of DAOP-ADL to describe the architecture of applications. The process of describing and validating the AA in our approach is semiautomatic, as we provide a set of tools that support the software architect’s task [26]. In addition, to integrate DAOP-ADL into the platform we are currently developing tools that automatically generate DAOP-ADL descriptions of components and aspects by binary code inspection. Using this tool we simplify the software developer’s task by making it easier to plug components and aspects into an application.

Using this implementation we have mainly developed collaborative applications [28]. These applications, like group games (e.g. the Pictionary game) or collaborative spaces (e.g. a Virtual Office), are dynamic and distributed applications that are characterized by high runtime interaction among the users connected to the application and high requirements of dynamic adaptability and reusability; therefore, they are good candidates to test our approach.

Currently, we have a virtual office application (see http://150.214.108.46/CoopTEL), and after one year of evaluation we can state that the performance is satisfactory. Even when Java poses significant drawbacks related to efficiency, we find that the overload of dynamic evaluation of aspects is not so critical in distributed systems based on Java. For instance, component and aspect creation through the platform takes 30 ms, and the time to incorporate the evaluation of the aspect at runtime is insignificant (around 20 ms). Comparing this evaluation time with the time spent loading a web page from the same host, they are insignificant. Nevertheless, the advantages of CAM/DAOP do not depend on Java limitations, because the model is independent of specific programming languages or component platforms.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented CAM/DAOP, a new component and aspect model that combines CBSD and AOSD disciplines. By defining this model we have tried to overcome some restrictions of both component platforms and AOSD approaches that make it difficult to develop final applications as a set of independent and reusable software components and aspects.

The main contributions of CAM/DAOP are the following. First, it separates components and aspects into two first-order entities that are composed by DAOP at runtime. Due to the application of the separation of concerns principle and, in particular, due to the composition mechanism defined by the model, the reusability of components increases and developers are able to build complete applications in a short time.

Second, the architectural information is explicitly stored in the platform. As the connections between components and aspects are clearly specified in the AA, designers and programmers are able to comprehend the structure of any in-house or third-party application. This issue greatly facilitates the understanding and evolution of the final application.

Third, we achieve a high degree of independence between components and aspects, which makes them more reusable in different contexts: (i) CAM entities, both components and aspects, are independent regarding communication due to the role name concept, which avoids having hard-coded references among them; (ii) they are also independent regarding shared data due to the definition of properties;
iii) aspects are independent from components, as pointcuts are described in DAOP-ADL, making aspects more reusable in different contexts.

Finally, since DAOP-ADL can express the application’s CAM in XML, which later is interpreted by DAOP, we bridge the gap between design and implementation.

As might be expected, static composition is faster and therefore offers better performance than dynamic weaving. Therefore, the main limitation of CAM/DAOP may be that the use of reflection and the dynamic composition mechanism introduces some overhead at runtime.

In order to cope with the overhead introduced by dynamic composition, and taking into account that there are aspects that might not need to be adapted at runtime, we are developing a static composition mechanism using the BCEL API. Extending DAOP-ADL to specify whether an aspect must be woven into components statically or dynamically, this tool will manipulate component class files to weave static aspects at compile time. Following this approach, aspects that are composed dynamically at runtime will not invade the component code, while aspects that are composed statically will be part of the component code. Nevertheless, even with this extension we continue to claim the non-invasiveness of our model at design phase. One of the goals of this extension is to be able to study the runtime overheads of our approach to check how we are able to reduce the runtime overheads of dynamic composition in some situations. In addition, we are working on new implementations of CAM/DAOP in CORBA, CCM/CORBA and .NET.

ACKNOWLEDGEMENTS

This research was funded in part by the MCYT under grant TIC2002-04309-C02-02

REFERENCES


