# Extending Component Composition Using Model Driven and Aspect-Oriented Techniques

Pedro J. Clemente, Juan Herández and Fernando Sánchez
Department of Computer Science, University of Extremadura, Spain
Email: {pjclemente, juanher, fernando}@unex.es

*Abstract*— **Component-based software engineering is an interesting and emerging discipline focused on reuse plug&play pieces of software. However, these pieces of software are distributed by the companies as binary units of composition. So, nowadays the system requirements are continuously evolving, as a consequence the component acquired normally need to require be adapted to these new requirements. However, once a binary component is ready to market, its contract cannot be changed in order to be adapted to new context or new requirements. In this sense, new modularization mechanisms like aspect-orientation can facilitate the software systems adaptation and evolution. Besides, this kind of modularization joined to model driven techniques can help to obviate contracts and weave new behavior to an already developed system. In this paper, a component composition mechanism based on aspect-oriented and model-driven techniques is presented, extending the typical composition based on interfaces and events. To do this, the components and their relations (basic relations –provides, uses– and extended relations described using aspect-oriented techniques) are modeled using UML. Then, using our approach based on model-driven development (MDD) the system modeled is transformed to obtain a component based system based on a specific component model. This work has been developed using Corba Component Model (CCM) as the component model[1].**

**Index Terms— Model Driven Development (MDD), Aspect-Oriented Software Development (AOSD), CBSE, MDA, CCM.**

## I. INTRODUCTION

Component-based software development (CBSD) has been recognized as one of the key technologies for the construction of high-quality, evolvable, large software systems in timely and affordable manners. Constructing an application under this setting involves the use of prefabricated pieces, perhaps developed at different times, by different people, and possibly with different uses in mind. The ultimate goal, once again, is to be able to reduce development costs and effort, while improving the flexibility, reliability, and reusability of the final application due to the (re)use of software components already tested and validated. This approach shifts organizations from application development to application assembly.

However, the ideal panorama that advocates component based development (as a fast assembly mechanism for building final enterprise applications) is based on a false comparison with electrical and electronic devices: suitable tension, suitable voltage, suitable connectors, plug-and-play, etc. CBSD still has to deal with many challenges before achieving this ideal goal, from the most obvious disagreements/discrepancies (different definitions of what a software component is, market competition because of distinct companies trying to make their products industry standards) to more technical problems (interoperability issues, modular adaptation and evolution to new requirements, component composition from early phases of the software life cycle, just to mention a few).

This article focuses on the composition and later assembly of black-box components at the different stages of the component life cycle. The composition mechanism is driven from the modeling phase, allowing software components both to evolve and to be adapted to changes and new requirements. In the scope of this work, we follow the notion of component given by C. Szyperski [1]: A software component is a unit of composition with contractually-specified interfaces and explicit context dependencies only.

The CORBA Component Model (CCM) [2] is one of the industry-standard component models that, from our point of view, closest follows the Szyperski's definition. A CCM component offers a contract to other components by means of its interface, which may describe facets (provided interfaces), receptacles (required operation interfaces), event sources (the produced events), event sinks (the consumed events) and configurable properties through attributes. Nevertheless, the simple task of adding new requirements to an already functioning system composed of CCM black-box components becomes a difficult task because of the binary nature of the existing components. Although the general approach of the technical solution could be the insertion of proxies or wrapper components into assemblies, this may become a tedious task because they must be hand-written, being thus error prone.

On the other hand, aspect-orientation provides transparent and flexible composition mechanisms (mainly at programming level), allowing crosscutting concerns (or new behaviour) to be automatically woven with other sys-

This paper is based on "Driving component composition from early stages using aspect-oriented techniques," by Pedro J. Clemente, Juan Hernández and Fernando Sánchez, which appeared in the Proceedings of the 40th Hawaii International International Conference on Systems Science. Adaptive and Evolvable Software Systems: Techniques, Tools and Applications Track, Hawaii, USA, January 2007. © 2007 IEEE.

[1]This project has been financed by CICYT project number TIN2005-09405-C02-02

tem concerns. In this sense, we claim that aspect-oriented composition mechanisms may be applied to black-box component composition, thus allowing components to be adapted and reused.

In this paper we provide an approach based on model-driven development (MDD) [3] that allows component composition to be driven from the early stages of the component life cycle, by applying aspect-oriented weaving techniques at design level and wrapping concepts to CBSD. We extend wrapping beyond its current use as an implementation artifact, applying it to all stages of CBSD in order to automatically draw up black-box component composition from analysis and system design until deployment. Concretely, this paper present the following contributions:

- A new UML2.0-based profile for modeling component and aspect based systems is proposed. The models obtained through the use of this profile guide the system development. This profile includes several new stereotypes to model concerns and their relationships with the base component based system. This UML profile can be used to model both static and behavior system view.
- A model-to-model transformation is presented. The model obtained using our profile is transformed from a model based on components and aspects to a model based only on components. In this sense, an extended composition mechanism based in the main aspect-oriented principles is used to allow component composition to be performed during the design phase.
- Software artifacts such as XML assembly descriptors and wrapping code are automatically generated from the previously transformed models. These artefact's are required to reuse third-party binary components and to compose the final system using an specific component model like CORBA Component Model.

The rest of the article is structured as follows. In Section 2, the component composition problem in the CCM framework is presented. Section 3 shows an approach based on aspect oriented techniques to mitigate the problems in CCM component composition. Our proposal guides the component based system development throughout all phases of component life cycle, namely design, code, assembly, packaging and deployment. Finally in Sections 4 and 5, we present related work and conclusions of the paper.

## II. The Key: Changing the System's Requirements

The goal of this section is to illustrate the main challenges arising in an already existing CCM system when new requirements need to be fulfilled. We present a generic case study in order to focus the reader on the principal problems. Concrete CCM examples matching our generic case study may be found in the Douglas Schmidt's CCM-tutorial [4].

Let us assume that two components, namely *Component1* and *Component2* conform to our initial system, and

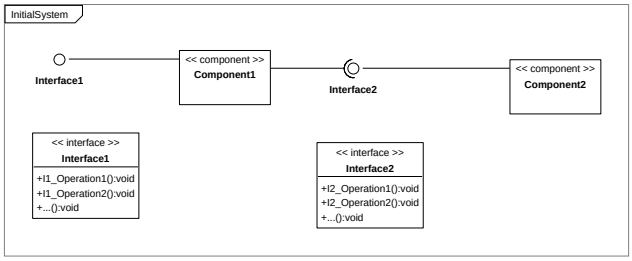
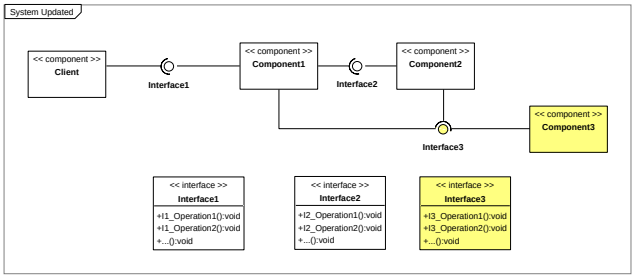



Figure 1. Initial component based system



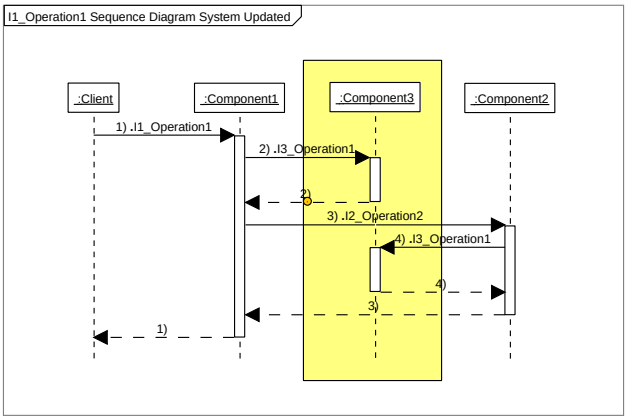




Figure 2. Class and sequence diagrams of the new component based system

they are related to each other through the uses/provides protocol as it is depicted in Figure 1. This figure represents a typical UML component based representation.

Assume that our initial system must be adapted with new functionalities fulfilling unanticipated requirements. The new behaviour to be used in the initial system is modularly encapsulated in *Component3* which offers its services to *Component1* and *Component2* . Accordingly, these two components should update their dependencies to use the services provided by *Component3* (see Figure 2).

To update the system previously described, we may either re-implement *Component1* and *Component2* (requiring them to be white-box components[2]), or wrap base components (in case Component1 and Component2 have black-box nature). We are interested in the later scenario

[2]Although there are aspect-oriented languages that allow aspects to be injected in bytecodes, they are language-implementation specific, which is not the case of CCM where components may be implemented in different programming languages.

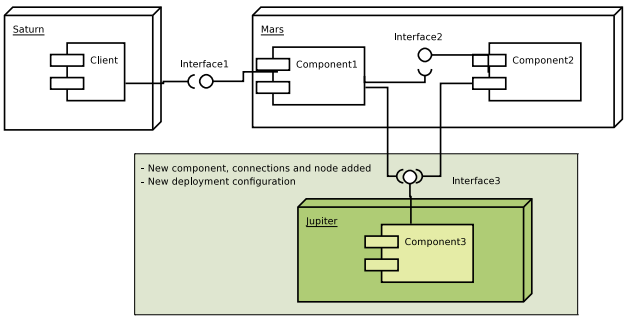



Figure 3. Deployment of the updated system

because component's source code is not always available (such is the case of components acquired to third-parties).

Wrappers intercept and forward messages to wrapped components. However, wrapper implementation should be done manually; that is to say, developers must implement all wrappers in the system writing the appropriate code, which is error prone. Besides, the system design does not reflect the changes carried out at implementation level as a result of wrapping. The consequence is a lack of coherence between implementation and design levels, making later maintenance and evolution of the system difficult. We claim that the new functionalities can be managed from the design phase, thus filling the gap between design and wrapper implementation.

Going step further, what happens if, in addition, the new functionality provided by *Component3* cuts-across the initial components? (This would be the case, for example, where *Component3* encapsulates the functionality of the Java Authentication and Authorization Service (JAAS)).

Not only should the component implementation be changed, but the component deployment should be reconfigured. In figure 3, *Component1* and *Component2* instances should use an instance of *Component3,* as a consequence they should know where this instance (*Component3*) will be deployed. The system configuration is an important issue in distributed systems like this, and this configuration should be updated when the system is designed and evolved. Again, the developer should change the specific system configuration and these tasks are tedious and error prone.

So, on the one hand, the component models are complex and specific characteristics to manage the component model complexity are required. On the other hand, the development processes to build very large systems require a set of basic characteristics such as adaptability, flexibility and reusability, which can be obtained using new kinds of modularizations.

Aspect Oriented Software Development (AOSD) is a set of emerging technologies that seeks new modularizations of software systems. AOSD allows multiple concerns to be separately expressed but nevertheless be automatically unified into working systems [5].

In the last few years, several aspect-oriented design approaches have emerged [3]. Aspect-oriented design provides mechanisms for the designer to reason about concerns separately (whether they are crosscutting or not), and to capture concern design specifications modularly [6]. Consequently, a way to specify how these concern modules should be composed in the full system design is required. This includes both a means to specify how to compose concerns at a later stage of the development cycle, and also a means to compose concern design artifacts. The aforementioned aspect-oriented design approaches (and others described in [6]) have notably contributed to provide the AOSD community with specific and explicit means to model aspect-oriented systems, deriving software engineering quality properties as a result. However, they have been thought for developing new systems from scratch, without black-box component reuse in mind. Our intention is not to compete with aspect-oriented design approaches but to apply aspect-oriented composition techniques at design level for making component-based systems to be automatically adapted from design, enhancing thus both component reuse and evolution of the systems. The details of our approach are explained in the next section.

## III. An approach for driving component-based system development

This section presents our approach based on model-driven development (MDD) [3] which allows component composition to be driven from early stages of the component life cycle, by applying aspect-oriented composition techniques at design level and wrapping concepts to CBSD. This approach integrates CBSD, AOSD and MDD to build component based systems using the best characteristics of each tendency.

This section is structured following the schema depicted in Figure 4, which shows a general overview of our approach:

- Firstly, the software architect uses a new UML2.0-based profile called *AspectComponent* (figure 4-(1)) for describing component and aspects. This profile is used in component, sequence and deployment diagrams, allowing the static and behaviour structure of the system to be described. Section III-A describes the use of the *AspectComponent* profile.
- Once the system including new requirements has been modeled, a model-to-model transformation is performed (Figure 4-(2)). The aim of this transformation is twofold: first, to provide a symmetric separation of concerns at design level, because once transformation is carried out, components are the only entities of the models; second, to compose concerns at design level using wrapping concepts. Section III-B outlines the transformation process.
- Next, the model based on components previously obtained is processed in order to obtain an XML

[3] An exhaustive survey on aspect-oriented design approaches may be found in [6].

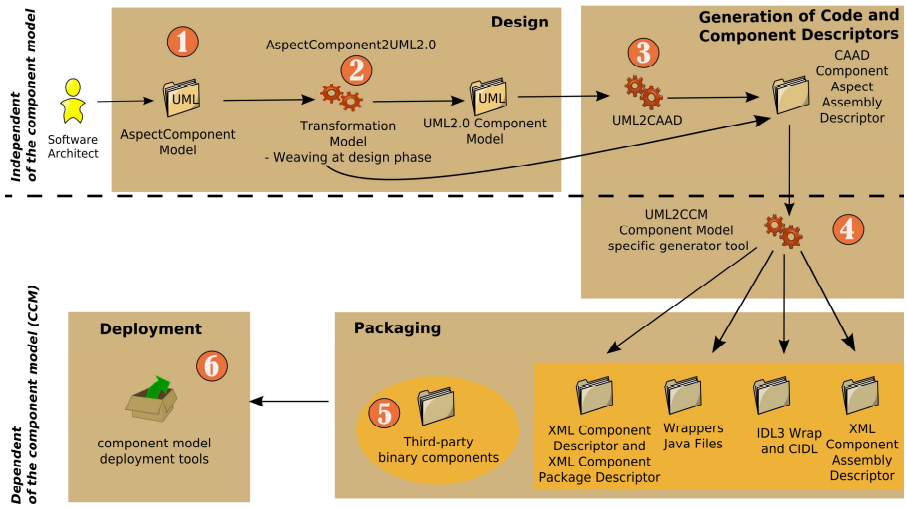



Figure 4. Aspect Component Based System Development (ACBSD)

descriptor called CAAD (*Component Aspect Assembly Descriptor*), which describes all the connections among components, aspects, interfaces, provides, uses, etc. (Figure 4-(3)). The XML CAAD file now becomes the input of our tool UML2CCM (Figure 4-(4)) to generate the appropriate code automatically: component definitions in IDL3, wrapper code and CCM descriptors (CCM components, package and assembly descriptors) . Section III-C describes specific code generation to obtain a CCM system.

- Finally, the code previously generated may be packed with third-party binary components (Figure 4-(5)), and then, the final system can be deployed using (Figure 4-(6)). Component packaging and deployment are detailed in Section III-D.

### A. AspectComponent: *UML profile to model systems based on components and aspects*

The description of a component based system requires all phases of component based development to be modeled defining the interfaces, the required and provided services, the components, the assembly, the deployment, etc. [7]. In this sense, we may guide the complete software development process from the models obtained during the design phase [8], from models to code.

A new UML2.0-based profile called *AspectComponent* has been specified with the aim of modeling adequately the relationships among components and aspects in the system [9]. The UML2.0 component model [10] already supports the description of *component, interface, provides, uses,* and so on. We have extended UML2.0 with new stereotypes (*concern, config_concern, uses_concern, hooks* and *actions*) for aspect-oriented modeling, facilitating the identification of all elements involved in the system. The use of the AspectComponent UML profile involves the following points: the basic use of the defined stereotypes, the composition configuration, the hooks and the actions to be performed among components and aspects.

*a) Description of the defined stereotypes.:* Components may be extended with the new functionality provided by a given concern through the use of the *uses_concern* dependency. These kind of dependencies allow components to be adapted to new requirements without change the initial component design. The use of the *AspectComponent* Profile is shown in Figure 5, where *Component1* and *Component2* are extended to use the functionalities provided by *Component3* and *Component4* in a transparent way by means of the *uses_concern* dependency. To do that, the access from Component1 and Component 2 to Component3 and Component4 must be configured, through the definition of composition rules, that are explained below.

The *uses_concern* dependency could define two tag values: *type* and *order_concern*. On the one hand, *type* tag value defines whether the new functionalities should be applied on the *provide* interfaces (*in*) or should be applied on the *uses* interfaces (*out*). On the other hand, *order_concern* tag value defines the priority of a *concern* when is applied on a specific component. This allows us to configure the aspect execution order when a component uses several concerns. For example in figure 5, *Component2* uses two concerns (*concern Component4* and *concern Component3*) where the actions associated to *concern Component3 (order_concern=1)* will be executed before the actions associated with *concern Component4 (order_concern=2)*.

*b) How configure the component composition and aspects:* In the same way that ApectJ-like languages allows abstract and concrete pointcuts to be defined, our profile provides two different composition configuration strategies for modeling the relationships among component and aspects: *generic composition configuration* and *concrete composition configuration*.

- *generic composition configuration***.** Allows the basic configuration composition of concern to be expressed in terms of *set_hooks* and *actions*. This configuration can be reused by all components that use the concern

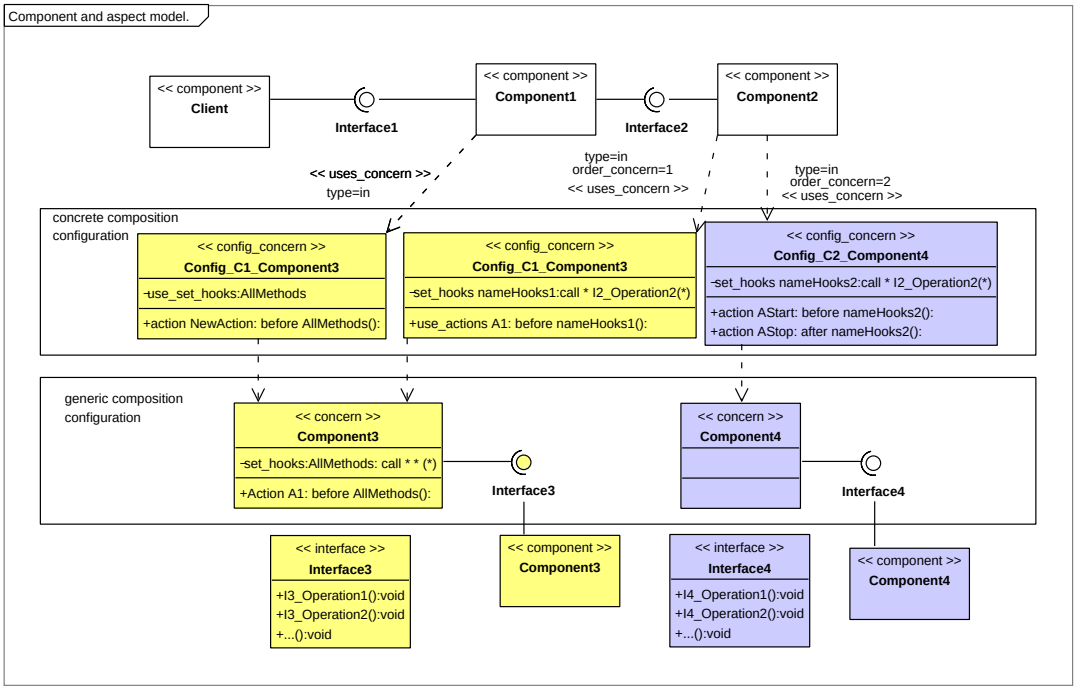



Figure 5. Analysis of the proposed system

being modeled. This kind of configuration is defined by *concern* stereotype.

- *concrete composition configuration.* Te aim of this configuration is to provide a concrete composition among components and concerns. In this kind of configuration, the *actions* and *set_hooks* established in *generic composition configuration* can be reused by *use_set_hooks* and *use_actions.* This configuration is defined by *config_concern* stereotypes.

*c) The hooks.:* Hooks are join points at the design level. They correspond to any location in a component assembly where a plug may be performed. The hooks in component based system must be described in terms of provide/use interfaces. From a theoretical point of view, this situation is due to the fact that components cannot be explicitly manipulated but through the provided/required interfaces. Consequently, the *hooks (join points)* described in other aspect oriented approaches such as AspectJ, HyperJ, JAC, etc. are reduced because we only have access to component interfaces.

*Hooks* can be defined using wild-cards (*e.g. call int * (*)*) and *set_hooks* is a set of *hooks* composed by AND, OR, NOT operators. The *actions* to carry out at the *set_hooks* can take place before, after or around (before, after, around) the original invocations in the system. For example, in Figure 5 the *set_hook* named *AllMethod* is defined on all invocations *(e.g. call * * (*))* carried out on the component which use the *concern* called Component3.

*d) Actions among concerns and components.:* The new functionalities (new concerns) are implemented by means of components and these components offer their services by interfaces. The *hooks* should be focused on the methods of the components interface. The *actions* make

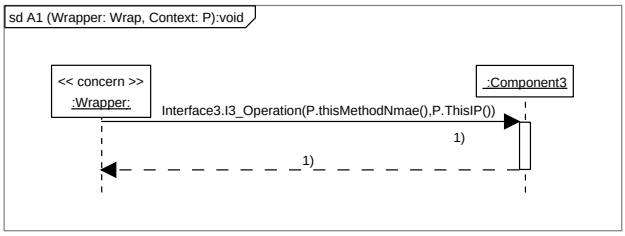



Figure 6. Sequence of A1 action (UML2.0)

reference to an specific sequence diagram which models how the new functionalities should be used. During the model of the configuration of the aspect (*config_concern* stereotype) each set of hooks (called *set_hooks*) is described, and subsequently the actions to take on these set of hooks are described in a sequence diagram (see simple example in Figure 6). The sequence diagrams in UML 2.0 [10] allow the use of sequences, alternatives, loops, references to other sequence diagrams, etc. to describe the interactions among components or objects, consequently, the same elements allow the definition of how the composition with the new functionalities should be carried out.

The definition of the hooks as well as the definition of the actions to be carried out is external to the definition of the component that provides the concern functionality, and external to the component that receives the application of the aspect. These definitions can be reused by other components or be specialized, so that each component manages the interaction with the aspects in a suitable way.

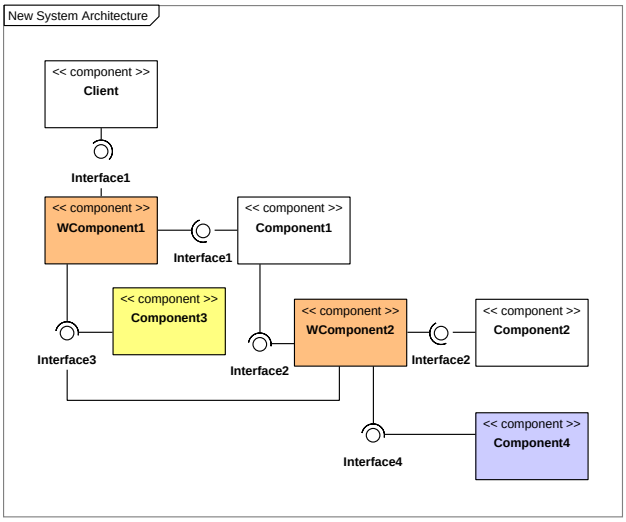



Figure 8. Description of the new system architecture (diagram generated)

### B. Model to model transformation

During the following development phase, the model defined by AspectComponent UML Profile is automatically transformed to a model defined only by UML 2.0 component model. Including in this transformation the component diagrams, sequence diagrams and deployment diagrams. This is kind of transformation called model-to-model transformation which translate between source and target models.

The UML model is processed and merged to obtain the new system model, using at transformation tool called *AspectComponent2UML2.0* (point 2 at Figure 4). In order to this, the UML diagrams are exported to XMI files, which are loaded in a MetaData Repository (MDR). MDR implements MOF (Meta Object Facilities) and allows one to recover the information about the UML diagrams in an objects structure. The model loaded at MDR is processed using Model Transformation Language (MTL) (which is a Query/View/Transformation (QVT) implementation).

In figure 7 an imperative algorithm coded by MTL can be observed. This MTL short algorithm shows how search for an interface in the model. So, MTL allow us to process the UML model using mainly imperative structures.

The output of this process is a model defined by the UML2.0 component model, and in order for, the component diagrams, the sequence diagrams and the deployment diagrams must be transformed.

*1) Model to model transformation: component diagrams.:* The component diagrams are updated during the transformation process, applying design patterns. Concretely, the components which had in the base model an *uses_concern* has been extended through wrappers. In this sense, by means of the application of design patterns we were able to generate automatically the system architecture. In this case, the system architecture is extended using a non-intrusive way. An example of component diagram transformed can be observed in Figure 8.

The wrappers modularize the architectural dependencies described previously using the *uses_concern* stereotype. For example, the component called *WComponent1* use the interfaces *Interfaces1* and *Interface3*, and provides the interface *Interface1*. The output of the component diagrams transformation is a new system architecture based only on components.

The transformation rules of the system architecture are summarized on the algorithm 1:

**Algorithm 1** Transformation rules of the system architecture. From AspectComponent *model* to *UML2.0*

1) **For each** *component* in *AspectComponent* component diagram, a new *component* with the same characteristics is created in the *UML2.0ComponentModel* component diagram.
2) **For each** *interface* in the *AspectComponent* component diagram, a new *interface* with the same characteristics is created in the *UML2.0ComponentModel* component diagram.
3) **For each** *provides* in *AspectComponent* component diagram, a new *provides* with the same characteristics is created in the *UML2.0ComponentModel* component diagram.
4) **For each** *uses* in *AspectComponent* component diagram, a new *uses* with the same characteristics is created in the *UML2.0ComponentModel* component diagram.
5) **For each** *config_concern* in the *AspectComponent* component diagram.
   a) The component related by *uses_concern* at *AspectComponent* component diagram is identified.
   b) **If** the component does not have a component wrapper linked at *UML2.0ComponentModel* **then**
      i) A new wrapper component is created and linked to the component. The component wrapper name is *W+ComponentName* if the *uses_concerns type* is labeled using *in* or *inout.*
      ii) A new wrapper component is created and linked to the component. The component wrapper name is *W2+ComponentName* if the *uses_concerns type* is labeled using *out* or *inout.*
   c) Then, the aspect used by the config_concern description is found. Next, the provides and uses required by the wrapper components (created wrappers) and based on the *concern* configured are defined.
   d) Then, the connections among the components are reconfigured, attending to the new wrapper components.
   e) Finally, the attributes and operations are defined at the wrapper, using the *config_concern* definition.

```
1  //Return an interface called "name"
2  getInterface (name : Standard::String): source_model::Core::Interface{
3     InterfaceIterator : Standard::Iterator;
4     anInterface : source_model::Core::Interface;
5     finalInterface : source_model::Core::Interface;
6
7     InterfaceIterator :=!source_model::Core::Interface!
8                          .allInstances().getNewIterator();
9     InterfaceIterator.start();
10    while InterfaceIterator.isOn()
11    {
12       anInterface := InterfaceIterator.item()
13                      .oclAsType(!source_model::Core::Interface!);
14       if anInterface.name.[=](name){
15          finalInterface := anInterface;
16       }
17       InterfaceIterator.next();
18    }
19    return finalInterface;
20 }
```

Figure 7. Example of MTL code for model transformation

*2) Model to model transformation: Sequence diagrams.:* The sequence diagrams allow the description of the interaction among components of the system and the relations with the new concerns. The interactions among components defined and the wrapper components generated to compose the system can be observed in the sequence diagrams.

Figure 9 represents the interaction among the base components and the new functionalities added. In this figure, the wrappers required have been added to the diagram and then the interconnections among components have been updated. To obtain this sequence diagram the initial (sequence diagram) and the actions sequence diagram (Figure 6) are merged where the hooks are active. This means carring out the weaving at design phase.

Summarizing the process to carry out the weaving process, each component instance at the sequence diagrams is checked to determine if a wrapper should be used[4]. Next, each invocation is checked, and when a *hook* is active on a specific invocation, the right *action* sequence diagram is included (see algorithm 2).

Using UML2.0 sequence diagrams, the references to the action required (Interaction Fragment [10] sequence diagram defined during AspectComponent model) are included in the sequence diagram. For example, in figure 9 at the *WComponent1* a reference to *A1* sequence diagram (Figure 6) has been included.

Later, the sequence diagrams, along with the class diagrams defined previously in this design phase, are the basis for: system code, the search of binary components which satisfy the requirements specified and the generation of the corresponding component composition code.

[4]For performance reasons, only a *wrapper* component is used to compose several concerns on the same component.

**Algorithm 2** Sequence diagrams. Transformation rules

1) **For each** *component instance* in sequence diagram based on *AspectComponent*
2) **If** this *component instance* requires an *instance of wrapper component* **then**
   a) Add an *instance of wrapper component* required
   b) **For each** *invocation* on the services offered by the original *component instance*
      i) *Active_hooks* ← false
      ii) **If** this *invocation* is an active hook based on any set of *set_hooks* defined at *config_concern* or *concern* for this kind of instance of component, **then**
         A) *Active_hooks*←true
      iii) **If** *Active_hooks* = true **then**
         A) Add the *actions* (after, before and around) related with the *Active_hooks* following the order of aspect to be applied, including the invocations forward the original *component instance*
         B) **else** forward the *original invocation* to the original *component instance*

As can be observed, component diagrams and sequence diagrams describe the new system architecture and their behavior, where the wrapper components allows the addition of new functionalities, which have been defined using aspect-oriented techniques.

*3) Model to model transformation: deployment diagrams:* Following the model transformation, the deployment configuration should be transformed too. In this

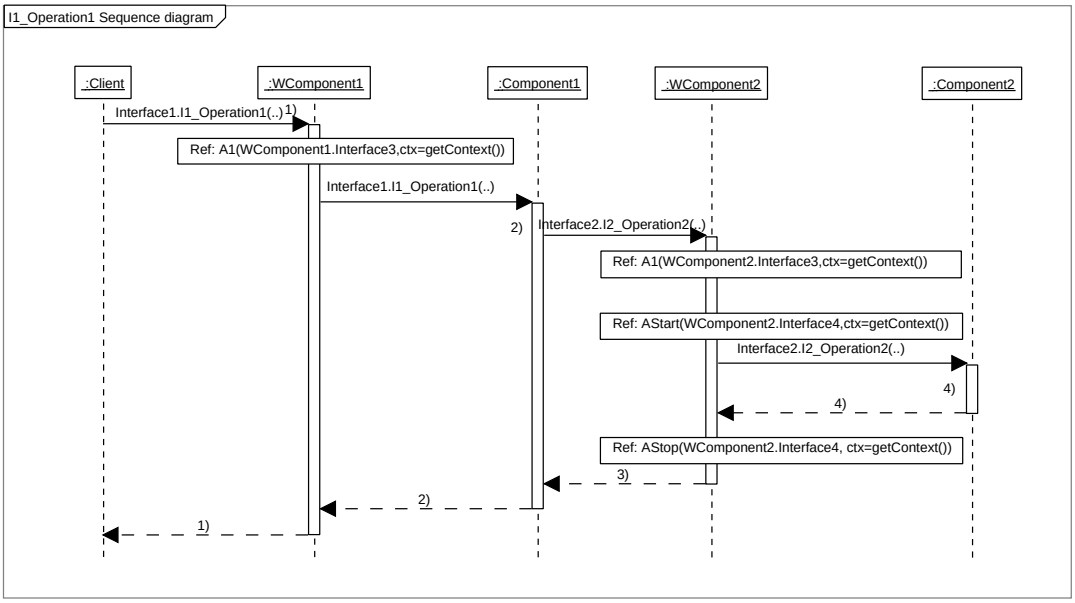



Figure 9. Example of sequence diagram to compose new concerns in the system (UML2.0) (diagram generated)

sense, the deployment diagrams represent when each component instance will be deployed, including the host or nodes and the relationships among the component instances. To manage the new component relationships (provides, uses and new functionalities) a set of new component wrapper instances should be added and be configured to the system. These new component wrapper instances are generated in a transparent way in deployment diagrams, making it easy for the developers to understand the execution architecture.

Figure 10 shows a deployment diagram generated where *WComponent1* and *WComponent2* instances have been generated and configured to use and provide the rights services. On the one hand, these component instances (*WComponent1* and *WComponent2*) are instanciated at the same node as the component wrapped. This decision allows the minimization of the performance cost facilitating local communication among components and their wrappers.

### *C. Model to text transformations: Generation of code and component descriptors*

Once the system has been designed, the generation of artifacts to compose the final system at implementation level is carried out by an automatic way. For this, the code for wrappers designed and the component descriptors are generated following two phased: the first of them is independent of the component model used (Figure 4-(3)), an the second depends of the final component model used to develop the system (Figure 4-(4)), for us Corba Component Model.

*1) Component Aspect Assembly Descriptor (CAAD): Independent descriptor of the component model:* With the aim of capturing each of the descriptions added at the design phase a new XML schema specification called *Component Aspect Assembly Descriptor (CAAD)* has been developed. This schema allows us to identify all the characteristics added to the software system including components, wrappers, concerns, hooks and actions. The XML files which are validated with this XML schema describe the new concerns added to the system, indicating the wrappers needed, and how the components should be composed and be deployed. The generation of XML CAAD is independent of the component model used to develop the system, and can be used as basis to generate code for several component model platforms (CCM, Fractal [11], etc.). Figure 11 shows an overview of *Component Aspect Assembly Descriptor (CAAD) Schema.*

*2) Specific CCM generation code and descriptors:* Based on XML CAAD files, the code for a specific component model can be generated using the appropriate tool. We are developing a tool called CAAD2CCM which generates CCM code and descriptors from CAAD files. Another approach could be the use of a tool to directly process the design model to generate the same files. However, based in CAAD the generation process to specific component model (CCM, Fractal, etc.) is more simple than to directly process the design model.

Bellow, we present the main artifacts which are required to implement CCM components, and then we present the process to generate code and descriptors for CCM.

*a) Summarizing the CCM Component code and descriptors required.:* Each one of the components described in the design phase which have not been located in the components repository, or have not been acquired from third parties, must be implemented. We are now going to summarize the main steps in developing CCM components.

Firstly, in CCM, the components should be defined using the *Interface Definition Language (IDL3)*. Also, the component implementation structure should be described

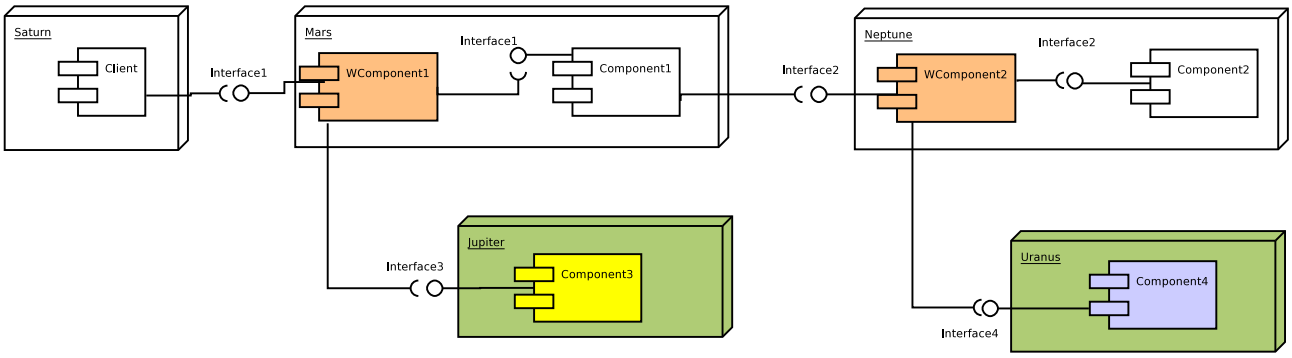



Figure 10. Example of deployment diagram generated

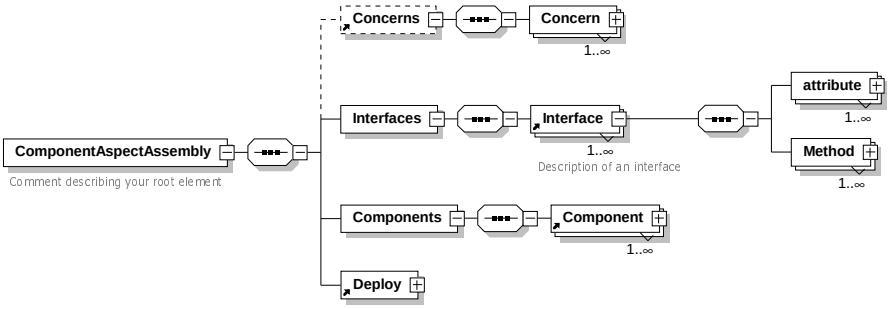



Figure 11. XML Component Aspect Assembly Descriptor schema overview

using *Component Implementation Definition Language (CIDL)* [2].

Then the component implementation is carried out using the high level programming languages (Java, C/C++, etc.), in this sense Java is used as the programming language. However, other languages can be used to develop CCM components, due to, the CCM specification being independent of the language used to develop the components. That is to say, at the end of the process the components written using several languages can be deployed successfully.

Finally, several XML files describe the assembly among components in CCM. These files are the following: *Corba Component Descriptor (.ccd), Component Package Descriptor (.csd), Corba Assembly Descriptor (.cad)* and *Corba Properties Descriptor (.cpf)*.

The first of these descriptors (*Corba Component Descriptor*) allows us to configure the CCM container properties like transactions, life cycle, security, etc. Also, this XML descriptor describes the services provided and used by a component. *XML Component Package Descriptor* is a specialization of a general software package based on Open Software Description (OSD). The software packaging scheme, could be used to package arbitrary software entities. Therefore, the XML file named *Corba Assembly Descriptor* describes the components in the system (based on references to XML Component Package Descriptors), the component instances in the systems, and the connections among them, that is to say, the links among components which require a service and provide this service. In this file, the location of each component is described (this characteristic is extended in the next section). Finally, *Corba Properties Descriptor* allows us to describe the specific configuration properties for the components in the system.

*b) Wrappers code generation.:* From *Component Aspect Assembly Descriptor (CAAD)* and using *CAAD2CCM* tool, the wrappers IDL3 definition (necessary specification to generate stubs and skeletons in CCM, see Figure 12-a) and the *wrappers* implementation (java files) are generated. A *wrapper* implementation (java files) comprises: the wrapper skeleton and the wrapper code for this skeleton.

On the one hand, the wrapper skeleton describes the interfaces which this component wrapper provides and uses. On the other hand, the code for these wrappers is added.

In the example (see Figure 12-b), the implementation of the method named *I1_Operation1* in the *WComponent1* wrapper responds to the sequence diagram described previously (see Figure 9), that is to say, this method should invoke the *I3_Operation1* operation in the component *Component3* which provides the *Interface3* interface. Finally, for each component generated, a XML component descriptor is generated.

*c) Generation of CCM descriptors.:* From CAAD, other XML files are generated: *XML Component Descriptor, XML Component Package Descriptor* and *XML*

```
1 Component WComponent1{     // (A) IDL3 for WComponent1
2     uses Interface1 cli_Interface1;
3     uses Interface3 cli_Interface3;
4     provides Interface1 svr_Interface1;
5 };
6                             // (B) Java code for I1_Operation1 method of WComponent1Impl
7 public class WComponent1Impl extends org.objectweb.ccm.mySystem.WComponent1SessionComposition.ComponentImpl {
8     public void I1_Operation1(String text, String textout) {
9         Context P = getContext();   // Obtain the actual wrapper context. Then
10        // Obtain the object reference associated to the 'cliILog' receptacle.
11        Interface3 cli_Interface3 = get_context().get_connection_cli_Interface3();
12        if(cli_Interface3 == null)// Check if the connection is available.
13            return;   // then Calls the I3_Operation1 service in Component3
14        cli_Interface3.I3_Operation1(P.thisMethodName(), P.thisIP());
15        // Obtain the object reference associated to the 'cliIPDA' receptacle.
16        Interface1 cli_Interface1 = get_context().get_connection_cli_Interface1;
17        if(cli_Interface1 == null) // Check if the connection is available.
18            return;   // then Calls the I1_Operation1 service in Component1
19        cli_Interaface1.I1_Operation1(text, textout);
20    } // ...
21 }
```

Figure 12. IDL3 definition and Java code generated for *WComponent1* wrapper

*Properties Descriptor* for each component and *XML Corba Assembly Descriptor.* The *XML Corba Assembly Descriptor* is based on the deployment diagrams where the kinds of components in the system are described, the instances in the system and the interactions among them.

Consequently, the system architecture has been developed as described in the previous phases, in order to do this, several artifacts (wrapper definition, their implementation, composition files, etc.) have been generated from the artifact defined in the design phase. *The system has been adapted to new requirements*, composing the components like COTS or black-box. .

### D. Packaging and deployment

Finally, the system artifacts (third-party binary components, wrappers code, IDL3 component definitions, and XML descriptors) should be packaged. A software package is represented by a XML Descriptor and a set of files. The descriptor and associated files are grouped together in a ZIP archive file.

A Component Assembly Package consists of a set of component packages and an XML Corba Assembly Descriptor (which have been generated during the phase of code and descriptors generation). The XML Corba Assembly Descriptor specifies the components that make up the assembly, component instances, and connections. Connections are between interface ports, represented by provides and uses features.

XML Component and Assembly Packages are provided as input to a deployment tool. A deployment tool [12] deploys individual components and assemblies of components to an installation site, usually a set of hosts on a network. Components within an assembly may be installed on a single machine or scattered across a network. Based on an assembly descriptor, the deployment tool installs and activates component homes and instances; it configures component properties and connects components together via interface as indicated in the assembly descriptor [2].

## IV. Related works

The work presented includes two areas of interest in software engineering such us aspect oriented software development (AOSD) and component based software engineering (CBSE). Both paradigms are combined in an adequate way to obtain more adaptable and reusable software. Next, the related works are presented taking into account proposals for *Analysis and design* and *components and aspects.*

*d) Analysis and design.:* Currently, UML is a standard graphical language of modeled software. In this sense, we can find several approaches for the analysis and design both aspect-oriented software development (AOSD) and component based software (CBS).

The component software modeled [7] presents new challenges for software engineering because several phases of component based software development should be modeled, from requirements phase to the deployment of the system, as has been presented in this proposal.

Theme/UML [13] is an extension to standard UML to support the modularization of designs into *themes.* A *theme* is any feature, concern, or requirement of interest that must be handled in the system. A theme is represented by a design element which encapsulates a collection of structure and behaviors. There are two kinds of themes: crosscutting themes and base themes. Theme/UML allow

the crosscutting themes to merge with the base themes to obtain a specific output. The use of Theme/UML to model component based systems and crosscutting concerns makes the components reuse difficult because their static and dynamic view can change as a result of the multiple perspective merge. That is to say, the component should be composed by means of the union of each perspective (each *theme*), consequently, the class diagrams and collaboration diagrams are updated. Finally, the component implementation will also be updated. However, the components are binary composition unit and we can not change their interfaces or their implementation. This is a one of the main properties to favour component reuse.

Aspect Oriented Design Modelling (AODM) [14] provides a profile to model crosscutting concerns and then merge them with the functional concerns. Crosscutting concerns are encapsulated in specific entities called aspects where are typical elements described in AOP like pointcuts and advice. AODM involve both static and dynamic views (class diagrams and collaboration diagrams). In the same sense of Theme, the merge process requires changing the behavior of the components affected by crosscutting concerns. AODM can not be used to model a component based system due to the AODM proposal hiding several phases on component based development (CBD) as the deployment phase and AODM does not handle in a transparent way the crosscutting concerns in the system.

*e) Components and aspects.:* The application of new concerns to a component based system presents several alternatives focused mainly on the description of proprietary component platforms. The description of specific platforms makes it easier to apply aspect to a component based system.

Pinto [15] presents a framework to include aspect into component based systems in a dynamic way. In this framework the aspects and components are different first class entities. These entities can be composed in a dynamic way, that is to say, during the execution of the application through the information stored in the middleware layer. This framework is based on a specific component model designed to support the composition among components and aspects. As a consequence, a specific component model is needed in order to apply this proposal.

Grundy [16] presents a new component model called Jview defined to add new services using the concept of aspect, which can be identified and incorporated in the system. Again, in this model the implementation of the component is required to obtain the new system. A specific component model is needed to apply this proposal.

Netinat [17] proposes an approach based on a stratified framework for the dynamic and automated composition of aspect in a component based system. This work presents a structure to describe the coordination among components and aspects. Each component intercepts its invocations and sends it to an Aspect Moderator which re-sends the invocations to the adequate aspect. This model supposes the existence of one or several central elements called Aspect Moderator, and as a consequence distributed system architecture can not be easily implemented.

AspectJ2EE [18] applies aspects to EJB components. In this sense the EJBs deployment descriptor is extended to define aspects and pointcuts. The aspects are implemented like templates (called parametrized aspects) which can be instantiated by specific components (bean components). The decorator pattern (also wrapper pattern) is used to generate the code for the new functionalities based in the parametrized aspects. The mechanism used to compose the base components and the aspects is based on inheritance (the classes generated inheritance of the base component). The inheritance is not a good mechanism to compose components because the components are not objects and should be replaced by other compositional mechanisms like the provide/use protocol [1].

Truyen [19] presents in Lasagne a view based on pattern design used like wrapper elements. However, the wrappers should be implemented by the developer, and they are not generated as they are in our proposal. Besides, the wrapper implementation is based on inheritance too.

Finally, concerning the containers, Duclos [20] extends the capabilities of CCM containers by the aspect descriptions. However, the developer should know, during the phase of the component implementation, which services or aspect are added to the container and use them.

## V. Conclusions

Component based systems should normally be adapted to new requirements. For us, Aspect Oriented Software Development can help to mitigate this problem. In this sense, an approach has been presented for the reutilization and adaptation of components which are part of an initial system. On the one hand, the components can be reused during the software development process and, on the other hand, the components can be adapted during future evolutions of the system.

The proposal presented integrates aspects and components to decrease the code tangling phenomenon, which is present in component based software development. The building of flexible and adaptable software has been favoured by means of a new component composition mechanism based on aspect-oriented techniques. This proposal, based on model-driven development, allows the addition of new functionalities to the component based systems, begins from the model phase, passing through all phases of the component based development (design, code, assembly, packaging and deployment). The system is modeled using components, aspects and their relationships, and then a model to model transformation is carried out to obtain a new model based only on components. From the model obtained, a set of XML descriptors and specific code are generated. In our proposal, the system is finally updated through wrappers, which are generated and handled automatically.

The adequate use of this proposal makes it easier to reuse, adapt and evolve component based systems through the transparent composition of new components in the system. Finally, the joint use of aspect-oriented and model-driven techniques in the component based software engineering domain opens up a new dimension of software development.

## REFERENCES


[1] C. Szyperski, *Component Software. Beyond Object-Oriented Programming*, 2nd ed. Cambridge, MA: Addison-Wesley, 2002.

[2] OMG, *The CORBA Component Model*, Object Management Group, June 2002, OMG document formal/2002-06-65.

[3] B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Software.*, vol. 20, no. 5, pp. 19–25, 2003.

[4] D. Schmidt, "Tutorial on the Lightweight CORBA Component Model (CCM)." OMG Workshop on Distributed Object Computing for Real-time and Embedded Systems. Arlington, VA (USA), 2003.

[5] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, Eds., *Aspect-Oriented Software Development*. Boston: Addison-Wesley, 2005.

[6] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdogan, and A. J. Siobhán Clarke and, "Survey of Aspect-Oriented Analysis and Design Approaches," AOSD-Europe, Tech. Rep. AOSD-Europe-ULANC-9, May 2005. [Online]. Available: http://www.aosd-europe.net/documents/index.htm/analys.pdf

[7] J. Chessman and J. Daniels, *UML Components: A Simple Process for Specifying Component-Based Software*. ISBN:0-201-70851-5: Addison-Wesley Longman Publishing Co., Inc., 2001.

[8] P. J. Clemente, J. Hernández, J. L. Herrero, J. M. Murillo, and F. Sánchez, "Aspect-Orientation in the Software Lifecycle: Fact and Fiction," R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, Eds. Boston: Addison-Wesley, 2005, pp. 407–423.

[9] P. J. Clemente, J. Hernández, and F. Sánchez, "Driving component composition from early stages using aspect-oriented techniques," in *Software Technology. Adaptive and Evolvable Software Systems: Techniques, Tools, and Applications Track at HICSS07.*, ser. IEEE Computer Society Press. ISSN: 1530-1605, Hawaii, United Stated., January 2007.

[10] OMG, *UML 2.0 Superstructure and Infrastructure Specification*, Object Management Group. Document formal/05-07-04, July 2004.

[11] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "An Open Component Model and Its Support in Java." in *CBSE*, ser. Lecture Notes in Computer Science, I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, Eds. Springer, 2004, vol. 3054, pp. 7–22.

[12] P. Merle, "OpenCCM - The Open CORBA Component Model Platform," Web site: http://openccm.objectweb.org/, 2006.

[13] E. Baniassad and S. Clarke, "Theme: An Approach for Aspect-Oriented Analysis and Design," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 158–167.

[14] D. Stein, S. Hanenberg, and R. Unland, "An UML-based Aspect-Oriented Design Notation," in *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, G. Kiczales, Ed. ACM Press, Apr. 2002, pp. 106–112.

[15] M. Pinto, L. Fuentes, M. Fayad, and J. M. Troya, "Separation of coordination in a Dynamic Aspect-Oriented Framework," in *Proc. of the First International Conference on AOSD*. The Netherlands: ACM, Apr. 2002, pp. 134–140.

[16] J. Grundy, "Multi-Perspective Specification, Design and Implementation of Components Using Aspects," *International Journal of Software Engineering and Knowledge Engineering.World Scientific,*, vol. 10, no. 6, December 2000.

[17] P. Netinant, T. Elrad, and M. E. Fayad, "A Layered Approach to Building Open Aspect-Oriented Systems: A Framework for the Design of On-Demand System Demodularization," *Comm. ACM*, vol. 44, no. 10, pp. 83–85, oct 2001.

[18] T. Cohen and J. Gil, "AspectJ2EE = AOP + J2EE." in *ECOOP*, ser. Lecture Notes in Computer Science, M. Odersky, Ed., vol. 3086. Springer, 2004, pp. 219–243.

[19] E. Truyen, "Dynamic and Context-Sensitive Composition in Distributed Systems," Ph.D. dissertation, K.U. Leuven, Belgium, November 2004.

[20] F. Duclos, J. Estublier, and P. Morat, "Describing and using non functional aspects in component based applications," in *Proceedings of the 1st international conference on Aspect-oriented software development*. ACM Press, 2002, pp. 65–75.



**Pedro J. Clemente** is currently an associate professor at the Computer Science Department of Extremadura University (Spain). He belongs to the Quercus Software Engineering Group. He received his B.S. in Computer Science from the University of Extremadura in 1998 and Ph.D. in Computer Science form the Extremadura University in 2007. His research interest include aspect orientation, model driven development and component based software engineering.

**Juan Hernández** is an Associate Professor of Languages and Systems and the Head of the Quercus Software Engineering Group of the Extremadura University (Spain), where he also currently leads the Computer Science Department. He received the BSc in Mathematics from the University of Extremadura and the PhD degree in computer science from the Technical University of Madrid. His research interests include component-based software development, aspect orientation and distributed systems. He is involved in several research projects as responsible and senior researcher related to these subjects. He has participated in many workshops and conferences as speaker and member of the program committee. He is currently member of the Spanish steering committee on Software Engineering, and organized several workshops and international conferences. He is a member of both the ACM and the IEEE Computer Society.

**Fernando Sánchez-Figueroa** has a PhD in Computer Science. He is currently Head of the Vitual Campus at the University of Extremadura (Spain). He is involved in several projects related to Web accessibility for visually impaired people using mobile devices. He holds the Telefónica grant for the "Application of Information and Communication Technologies to the University Environment".

## APPENDIX I. COMPONENT ASPECT ASSEMBLY DESCRIPTOR (CAAD) SCHEMA