A Model for Developing Component-Based and Aspect-Oriented Systems

Nicolas Pessemier[1], Lionel Seinturier[1],
Thierry Coupaye[2], and Laurence Duchien[1]

[1] INRIA Futurs, LIFL, Jacquard project/GOAL,
Bâtiment M3, 59655 Villeneuve dAscq, France
{pessemie, seinturi, duchien}@lifl.fr
[2] France Telecom R&D,
28 chemin du Vieux Chêne, BP98
38243 Meylan, France
thierry.coupaye@rd.francetelecom.com

Abstract. Aspect-Oriented Programming (AOP) and Component-Based Software Engineering (CBSE) offer solutions to improve the separation of concerns and to enhance a program structure. If the integration of AOP into CBSE has already been proposed, none of these solutions focus on the application of CBSE principles to AOP. In this paper we propose a twofold integration of AOP and CBSE. We introduce a general model for components and aspects, named Fractal Aspect Component (FAC). FAC decomposes a software system into regular components and aspect components (ACs), where an AC is a regular component that embodies a crosscutting concern. We reify the aspect domain of an AC and the relationship between an AC and a component, called an aspect binding, as first-class runtime entities. This clarifies the architecture of a system where components and aspects coexist. The system can evolve from the design to the execution by adding or removing components, aspects or bindings.

1 Introduction

Component-Based Software Engineering (CBSE) proposes to structure a program by separating concerns into clearly defined entities, called components. Reusable components with contractually specified interfaces are defined and composed together [20]. Subsequently, Architecture Description Languages [12] can be used to specify the component compositions and interactions.

Aspect-Oriented Programming (AOP) [9] identifies the code tangling and the code scattering which arise in applications. Some concerns mixed within an entity (code tangling), and some concerns scattered across several entities, are said to be crosscutting. These concerns hinders the reusability, the maintainability, and the evolvability of applications. AOP proposes artifacts (aspect, pointcut, advice) to modularize crosscutting concerns.

It has been shown that the issues of code tangling and scattering arise at the level of CBSE as well [8,11]. This is why merging AOP and CBSE makes sense.

W. Löwe and M. Südholt (Eds.): SC 2006, LNCS 4089, pp. 259–274, 2006.
© Springer-Verlag Berlin Heidelberg 2006

The integration of AOP into CBSE has already been proposed in [10,13,19], by providing a support for AOP in a component-based system. However, the application of CBSE principles to AOP is rarely proposed. In particular the implicit link between advice code and the base program where the advice applies is frequently hidden behind pointcut declarations (PcDs). Generally defined with a pattern language, PcDs select a set of joinpoints among those offered by the system. Unfortunately, once woven to the system, the implicit relationships created between a piece of advice code and the advised entities are never explicitly discernible and can surely not be individually manipulated at runtime.

In this paper, we propose a general and symmetrical model for mixing components and aspects. The approach is symmetric by considering aspects as plain components. The approach improves the component approach by giving a support for AOP, and improves the aspect approach by applying CBSE concepts to AOP. Our proposal relies on three main notions: aspect component, aspect domain, aspect binding. Aspects are contractually specified components called **aspect components** (ACs), and the relationships between ACs and regular components are reified with **aspect domains**, and **aspect bindings**. An AC embodies a crosscutting concern and can be reused in different contexts. An aspect domain is the reification of the components picked out by an AC. An aspect binding is a binding between a regular component and an AC. Thus, the model supports two levels of composition: regular components are composed together using regular bindings, and an AC is composed with regular components using aspect bindings.

We experiment this model by extending a reflective and general component model, named Fractal [4] and its ADL. In our extension, called Fractal Aspect Component (FAC for short), we introduce the notions of aspect component, aspect binding and aspect domain to the component model itself and to the Fractal ADL.

The rest of this paper is organized as follows. Section 2 introduces our general model for component and aspect. Section 3 presents the mapping of our model to the Fractal component model. Section 4 presents related work around the merging of components and aspects and some reference component models. Section 5 concludes and gives some open issues.

2 A General Model for Components and Aspects

This section describes the three main concepts we introduce to support AOP in a component model: *aspect component*, *aspect binding*, and *aspect domain*. Section 2.1 gives the motivations of our approach. The concepts are presented in the remaining three sections.

2.1 Motivations

When merging Aspect-Oriented Programming (AOP) and Component-Based Software Engineering (CBSE) two dimensions have to be considered: the

integration of aspect-oriented principles into component-based systems, and the application of component-based principles to Aspect-Oriented Programming. The integration of AOP into CBSE is motivated by the code tangling issue inherent in CBSE [8,11]. On the other hand the application of CBSE concepts to AOP is less investigated. Some approaches focus on the representation of an aspect as a component to contractually specify aspects and to increase their reusability [10,18].

In our proposal we realize a twofold integration of CBSE and AOP. We introduce three main concepts: aspect component, aspect domain, and aspect binding. These three notions are closely related to the three main concepts of the component approach: component, composite, and binding.

2.2 Aspect Component

An aspect component (AC for short) embodies a crosscutting concern. It is a regular component providing as a service a piece of around advice code. This service represents the behavior which will be woven around a set of regular components. This notion is similar to the notion of Aspectual Component proposed in 1999 by Karl Lieberherr et al [11] to express each aspect separately in a modular structure.

Our approach is symmetric by making no differences between an aspect and a component. Thus, an aspect which is represented as a component, becomes a reusable contractually specified entity. Another consequence of the symmetric approach is that regular and aspect components are composed the same way using the same rules. This facilitates the adaptation to new requirements when the system evolves.

2.3 Aspect Binding

Two kinds of binding exist within our model: regular and aspect binding. A regular binding expresses that a component is using a service provided by an other component. An aspect binding expresses that a component is aspectized by an aspect component. It is the reification of the individual relationship between an aspect component (AC) and a regular component where the AC applies.

In most existing AOP languages, the relationship between an aspect and the objects containing joinpoints picked out by the aspects is explicit in the source code but is implicit at runtime. Indeed, this relationship is structurally defined by a pointcut in the source code, but is lost when the woven code is executed. By introducing the notion of an aspect binding, we reify at runtime this relationship.

Because our approach is symmetric, all possible interactions between regular components and aspect components require full consideration. In a system using components and aspects, the possible interactions are described below.

– The *component to component* interaction is the classical *client-server* interaction. The client component uses a service provided by a server component

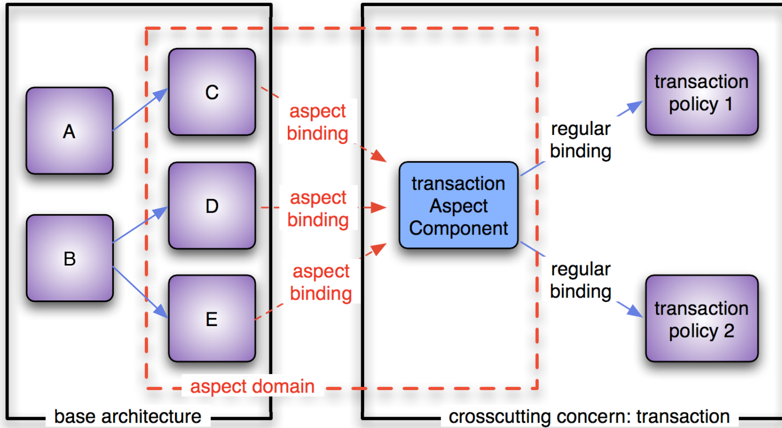

Fig. 1. Aspect binding best practice

interface. This kind of interaction exists in every component model using the notion of binding.

– The *component to AC* interaction is our notion of an aspect binding. It expresses the fact that an aspect component is woven on a component. In Figure 1 we can see this type of interaction between `C`, `D`, `E` and the transaction aspect component.
– The *AC to component* interaction, using a regular binding, is used as an AOP best practice. In Figure 1 we can see this kind of interaction between the transaction aspect component and various transaction policy components. In this example, changing a transaction policy is performed through a reconfiguration between the transaction AC and the components providing transaction policies.
– The *AC to AC* interaction can express a collaboration between two aspects using regular bindings, or the fact that the second aspect is woven on the first one. In asymmetric approaches this type of relationship is frequently unconsidered. Few techniques are given to make two aspects collaborate such as the use of context passing. The possibility of weaving aspects on other aspects is also uncommon.

2.4 Aspect Domain

An aspect domain is the reification of the components picked out by an AC. The goal of an aspect domain is to keep an overview on all the components affected by an aspect. It offers an abstraction on each AC woven on a set of components. A benefit that can be derived from the aspect domain notion is that the crosscutting interactions of a component-based system are clearly specified and are easily manipulable as regular interactions.

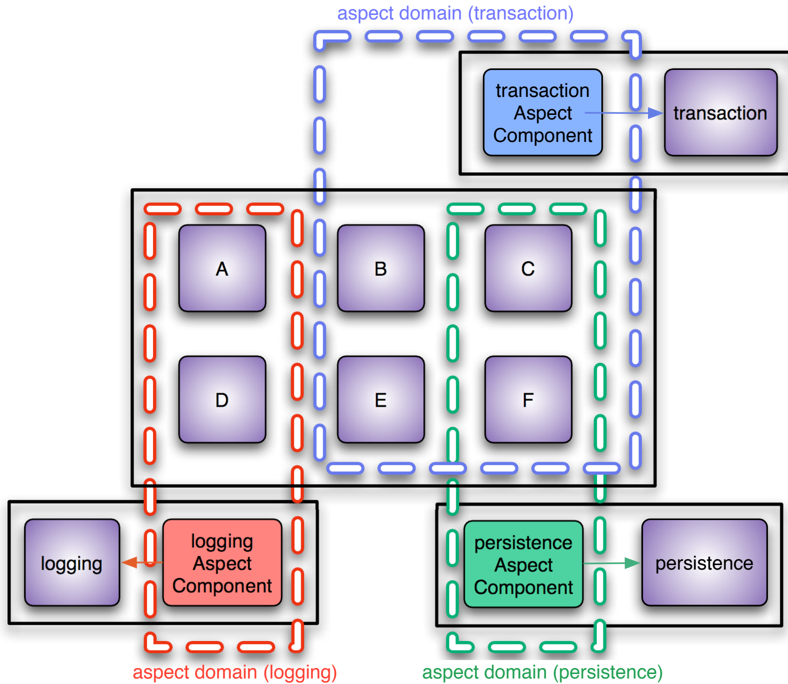

Fig. 2. FAC overview: weaving of three crosscutting concerns

Figure 2 illustrates the notion of aspect domain on a generic component-based application (components `A` to `F`). The aspect domains are represented as dotted rectangles, aspect bindings have been omitted for clarity sake. This application contains several crosscutting concerns: a logging, a persistence, and a transaction concern. These concerns are well known to be scattered and not cleanly modularized into one specific module. Their integration into an application is a hard task. In a full-fledged component, obtaining the same result requires numerous and tricky modifications. Moreover, once integrated to a system, it is difficult to remove one of these concerns in an easy and proper way. Once woven to a set of components the aspect domains of the ACs appear, offering reification on crosscutting relationships over the system.

3 Mapping onto the Fractal Component Model

This section presents the mapping of the main notions presented in the previous section onto the Fractal component model, which is a general and extensible component model supporting dynamic (regular) bindings. Our extension of Fractal is called FAC for Fractal Aspect Component. Section 3.1 presents the Fractal component model, and Section 3.2 proposes our extension FAC.

3.1 Fractal: A General and Reflective Component Model Supporting Dynamic Bindings

Fractal is an ObjectWeb consortium[1] project that proposes an extensible and modular component model [4]. This Section describes Fractal main features. Note that Fractal is independent of any programming language. Several implementations exist in different languages such as Java, SmallTalk, C, C++, and the languages supported by the .NET platform.

Contrary to component models for application servers such as EJB or .NET, Fractal is a general and reflective component model for developing complex software systems, such as operating systems and middleware. Besides the notion of a component, Fractal offers the notion of *composite-component* (allowing different views and abstractions on a system), *shared component* (a component nested by several composite components), *dynamic binding* (between components). Fractal is a reflective component model and offers introspection (system monitoring), and reconfiguration capabilities (modification of the system architecture).

A Fractal component has two parts: a *content* and a *membrane*. The *content* of a composite component is built as a set of sub-components, and the *content* of a primitive component (black box component) implements its provided services.

A component *membrane* offers a level of control and a level of interception. The control can be accessed through a set of so-called control interfaces which manage the non-functional properties of a component such as its life cycle, bindings, content, name, or attributes. This set of control interfaces can be extended with new control interfaces that can be added to a component membrane. The interception mechanism reifies messages sent by and received on component interfaces. These messages can be modified, discarded or delivered to the component.

An interface is an access point to a component comparable to the notion of a *port* in several component models such as ArchJava [2] or CCM [14]. A Fractal component offers external and internal interfaces. External interfaces are accessed from the outside of the component, while internal interfaces are only accessible from the composite's sub-components.

A binding is a communication channel between a client interface and a server interface. A client interface uses operations provided by a server interface. Fractal architectures can be described with Fractal ADL, which is an XML language to describe and to instantiate a Fractal component assembly.

Figure 4 presents the Fractal ADL syntax defining the architecture of Figure 3. Lines 2–3, 4, and 9 show the definition of server interfaces (`role="server"`). Lines 3–7 define the component `A` and Lines 8–11 the component `B`. Lines 12–13 are binding declarations of the binding between the server interface `r` of the composite and the server interface `r` of component `A`, and the binding between the client interface `s` of the component `A` and the server interface `s` of the component `B`.

Although component approaches such as Fractal offer several artifacts for the strong encapsulation of entities, the reification of dependencies, and the building of architecture from high level point of view, these approaches suffer from code

[1] `http://objectweb.org`

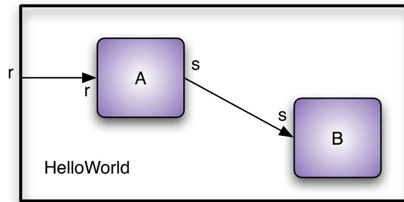

Fig. 3. Fractal-ADL: helloworld application

```
01 <definition name="HelloWorld">
02   <interface name="r" role="server" signature="java.lang.Runnable"/>
03   <component name="A">
04      <interface name="r" role="server" signature="java.lang.Runnable"/>
05      <interface name="s" role="client" signature="Service"/>
06      <content class="AImpl"/>
07   </component>
08   <component name="B">
09      <interface name="s" role="server" signature="Service"/>
10      <content class="BImpl"/>
11   </component>
12   <binding client="this.r" server="A.r"/>
13   <binding client="A.s" server="B.s"/>
14 </definition>
```

Fig. 4. Fractal-ADL: XML description of the helloworld application

tangling and code scattering. These two issues seriously limit the evolution of a system. Thus, when a crosscutting concern has to be plugged to a Fractal component assembly, the amount of reconfigurations that must be performed may become quite heavy. The next section details the mapping of our concepts of *aspect component, binding*, and *domain* onto the Fractal component model.

3.2 Fractal Aspect Component (FAC)

FAC is our mapping of the general model exposed in Section 2 onto the Fractal component model. It uses existing notions of the Fractal model and introduces new ones.

Figure 5 presents the FAC metamodel. It is based on the Fractal metamodel. The mapping of the three main notions (*aspect component, aspect domain*, and *aspect binding*) is straightforward. An aspect component is defined as a regular component; it provides as a service a piece of advice code (see the AspectComponent Interface). An aspect domain is a composite component that contains a set of ACs representing a crosscutting concern, and the components impacted by the ACs. Within the context of an aspect domain, aspect bindings can be

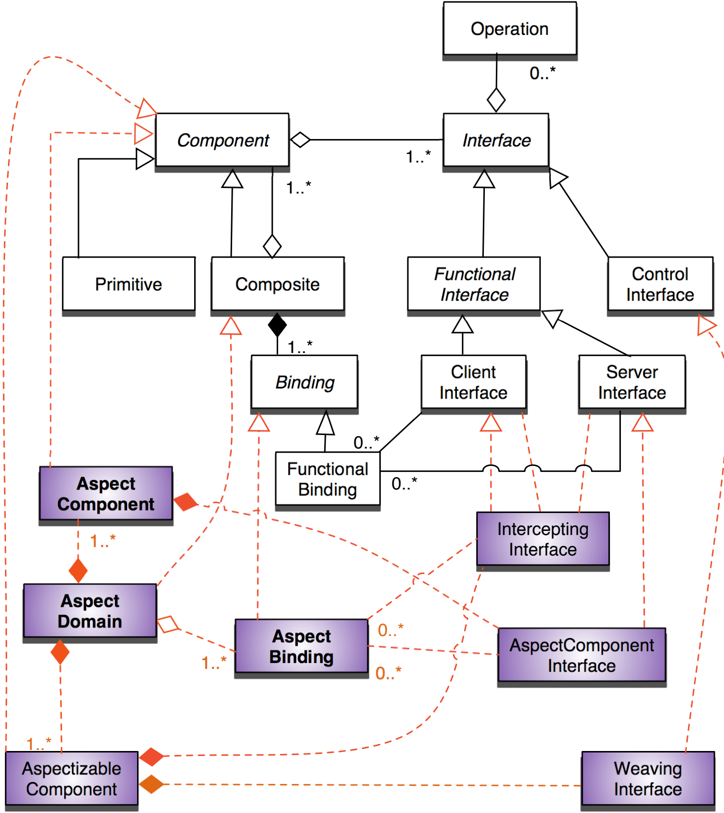

Fig. 5. FAC metamodel

defined between components and the AC. The following sub-sections describe the concepts of *aspect component interface*, and *weaving interface*. We then detail the FAC pointcut language. Finally, we discuss the two implementations of FAC.

3.3 FAC Join Point Model

Two different types of join points are supported by FAC: incoming calls on server interface operations, and outgoing calls on client interface operations. This choice is motivated by the fact that we consider AOSD in a component world.

As components are black boxes, it is rather natural to consider only join points on externally visible elements, i.e., exported and imported interfaces. Taking into account other kinds of join points, such as the ones on implementations, would break component encapsulation. Yet, for cases where this would be necessary, we believe that a best practice is to use a combination of component-based and implementation (e.g. object) based aspect-oriented tools.

The level of interception defined by FAC is very similar, at the component level, to the composition filters approach ([1]), which defines IN and OUT filters on objects to intercept messages.

3.4 FAC Pointcut Language

The FAC pointcut language is used to select join points. A pointcut expression is divided in two parts:

– A keyword that specifies if the incoming calls (keyword `SERVER`) or outgoing calls (keyword `CLIENT`) or both of them (no keyword) must be selected,
– Three regular expressions separated by semicolons that specify which components, interfaces, and operations must be selected.

Figure 6 gives some examples of PcDs. The regular expressions relies on the *java.util.regexp* package.

Pointcut Expressions	**Captured Elements**
`*;*;deposit*:void`	Every incoming and outgoing method returning void that start with deposit in any component and interface
`CLIENT B;*;deposit*`	Every outgoing method named deposit in any interface of a component named B
`SERVER B;ITransfert;*`	Every incoming method in ITransfert interface of a component B

Fig. 6. FAC pointcut language: Examples

3.5 The Aspect Component Interface (ACI)

The Aspect Component Interface (ACI) follows the AOP Alliance API[2], which is an open source initiative to define a common API for AOP frameworks. Figure 7 presents the *AspectComponent* Java interface and an example of an *AC*.

ACs apply on component methods exposed by client and server interfaces. The parameter of the *invoke* method is a reification of a Fractal interface invocation. It provides a set of methods to introspect a join point. The argument of the invocation can be modified, the intercepted method can be called (*proceed*), and the reference of the intercepted component can be retrieved.

Writing an *AC* requires implementing the *AspectComponent* interface. The *invoke* method describes the behaviour of the aspect. The code written in this method will be executed around the join point, i.e., a method call or execution on a component interface.

[2] `http://aopalliance.sourceforge.net/`

```
/**
 * Interface provided by an Aspect Component
 * to define an advice.
 */
public interface AspectComponent extends
            org.aopalliance.intercept.Interceptor {
  /**
   * Define an advice executed around incoming
   * and/or outgoing method invocations reified by m
   * @param m the reification of the method invocation
   * @return the result of the advice
   */
   Object invoke(FcMethodInvocation m) throws Throwable;
}

/**
 * An example of an AC with before and after code.
 */
public class GenericAC implements AspectComponent {
   Object invoke(FcMethodInvocation m) throws Throwable {
      System.out.println("before "+m.getMethod());
      Object ret = m.proceed();
      System.out.println("after "+m.getMethod()+" invoked");
      return ret;
   }
}
```

Fig. 7. The *AspectComponent interface*

The *proceed* call denotes the original method call. The code written before and after *proceed()* represents the before and after advices of AOSD. If more than one aspect applies on a given join point, the *proceed* call will trigger the next aspect, till the original method code is reached. If *proceed* is omitted the original method call will not apply. This can be useful to prevent, for example, the execution of the intercepted method.

3.6 Weaving Interfaces

The *Weaving Interface (WI)* of a component plays a key role in FAC. It manages the weaving of ACs around the interfaces of the component it controls. In the context of Fractal, we chose to represent the *WI* as a Fractal control interface in the component membrane. The *WI* uses the interception mechanism, which is provided by the membrane of components to intercept incoming and outgoing calls on its functional interfaces, and then, delegates the calls to the aspect components bound to (with an aspect binding) these operations. The *weaving interface* in FAC has three main objectives:

– Set/unset *aspect bindings* to *aspect components*,

```
void setAspectBinding(Component comp, ItfPointcutExp regExp,
                                              AspectComponent ac);
void unsetAspectBinding(AspectComponent ac);
```

– Automatically weave an *AC* around a set of components following a pointcut declaration (this weaving task will automatically create an *aspect domain*, add the components which match the pointcut declaration into this *aspect domain*, and bind with aspect bindings the *AC* and the impacted components),

```
void weave(Component rootComp, AspectComponent ac,
                                              ItfPointcutExp pExp,
                                              String aspectDomain);
void unweave(Component rootComp, Component ac);
```

– Provide a set of operations to order/re-order *ACs* which apply on an interface operation.

```
String[] changeACorder(String acName, int newPosition);
```

In FAC, a component supporting the *weaving interface* is called an aspectizable component. Otherwise, no aspects can be woven to this component. Since the weaving of an *AC* using the *weaving interface* is recursive and traverse the component hierarchy, if the component controlled by the *WI* is a composite component the weaving is also performed by its sub-components. A weaving operation can be initiated on the system as a whole (top-level composite) or on any sub system (intermediate composite).

All the operations provided by the interface can be invoked either with the Fractal ADL (extended with FAC notions) or directly at runtime.

The following piece of XML code presents the architecture of a Fractal assembly where a directive (tag <weave>) weaves a *traceAC* component (defined lines 2–4) to each component of the composite C (`rootComp="this"` line 12), which has an interface operation starting with "s" and returning "void". The aspect domain of this weaving will be automatically created and the composite representing this domain will be named "D" (`adomain="D" line 12`).

```
01 <definition name="C">
02 <component name="traceAC"/>
03    <interface name="ACI" role="server"  signature="AspectComponent"/>
04 </component>
05 <component name="A"/>
06    <interface name="itf1" role="client"  signature="Itf1"/>
07 </component>
08 <component name="B"/>
09    <interface name="itf1" role="server"  signature="Itf1"/>
10 </component>
11 <binding client="A.itf1" server="B.itf1"/>
12 <weaving ac="traceAC" pcd="*;*;s*:void"  rootComp="this" adomain="D"/>
13 </definition>
```

Every reconfiguration operation including the ones of our extension (setting/unsetting of aspect binding, weaving of an AC are dynamic operations).

3.7 Implementation Issues

The mapping of our general model for component and aspect on the Fractal component model has been validated with two different implementations in Java. Our first implementation extends the reference implementation of the Fractal component model in Java called Julia [4]. Julia uses a mixin [3] mechanism to program the level of control of components. The second implementation extends another implementation of the Fractal component model in Java, called AOKell [17], which uses AspectJ [9] aspects to implement control membranes.

4 Related Work

In this section, we compare FAC with different kinds of approaches. Firstly, we focus on approaches using the notions of component and aspect at a programming language level. Secondly, we investigate approaches using a symmetric representation of components and aspects. Thirdly, we study others component models.

4.1 Component and Aspect at the Programming Language Level

CaesarJ [13] is a Java based programming language, defined as an extension of the AspectJ language. The components are implemented as collaborations of classes. A collaboration defines a provided part and an expected part. The provided part is implemented with reusable CaesarJ components that are a set of virtual classes. A virtual class in CaesarJ is a kind of inner class, which can be overridden in the subclasses of the enclosing class. When overriding an inner class the new functionalities are directly usable by the parent class. With this mechanism the provided operation of a collaboration interface can be delayed to virtual sub-classes. On the other hand, the expected part is achieved by CaesarJ bindings, which are aspects woven afterward during a deployment phase. The main advantage of CaesarJ is its ability to stay close to the programming language and to be a superset of the AspectJ language.

JAsCo [19] is an AOP language originally designed for component-based systems. It introduces two main notions: aspect bean and connector. Aspect beans are reusable Java beans describing the extra behavior to apply to components. Aspect beans uses hooks that are similar to inner classes describing the advice code and the pointcut declaration. Connectors are used to deploy hooks (some kind of access points to join points) within a specific context.

Contrary to FAC, CaesarJ and JAsCo are programmatic approaches. In the case of CaesarJ, aspects are dedicated to the expression of the relationships between components. The problem is that these aspects manage the bindings of all the components of the system. In the case of JAsCo, only crosscutting relationships are expressed thanks to connectors, and the management of dependencies between base entities is missing.

4.2 Symmetric and Unified Approaches: Aspects Conceived as Components

FuseJ [18], which is the follow-up project by the JAsCo team, mainly focuses on the nature of an aspect that is represented as a regular bean. The approach is symmetric: all the concerns are implemented as plain components. Components in FuseJ are equiped with gates. A gate is a kind of interface to specify component services: aspect-oriented and regular. The connectors (extension of JAsCo connectors) specify the types of interaction between gates. FuseJ defines regular and aspect-oriented connectors. Regular connectors are in charge of functional connections between gates, and aspect-oriented connectors are in charge of weaving a component behavior to another component. All the connections defined by a component can be locally consulted. FuseJ does not yet propose a global description of a component architecture with its connections. The model does not support the managing of aspect domain and aspect binding as FAC does.

DyMAC [10] is a component and aspect-based middleware framework. It uses aspect-oriented composition to connect the application logic to the middleware services. Similarly to FAC, an aspect component in DyMAC encapsulates an advice. However around advices are not supported. Special kinds of connectors are statically described in XML files to write technical services of the middleware layer. Connectors in DyMAC looks like aspect bindings in our approach. Nevertheless, aspect ordering is static in DyMAC whereas FAC provides an API for aspect component ordering.

4.3 Other Component Models

OpenCOM [6] is a lightweight reflective component model and as such close to the Fractal component model. The key concepts of the model are interfaces, receptacles and connections. A component has a set of receptacles and interfaces. Interfaces are used to express provided services and receptacles to express required services (comparable to Fractal client and server interfaces). Unlike Fractal, OpenCOM defines a fixed meta-object protocol for components. The meta objects in OpenCOM can be compared to aspect components (ACs) in FAC. However, this meta level is fixed and thus does not support the dynamic adding and removing of meta objects.

K-Component [7] is a component model for building context-adaptive applications. Instead of using an Architecture Description Language to statically describe a component architecture, the model reifies the structure of the application and describes adaptation contracts written with an Adaptation Contract Description Language (ACDL) to dynamically reconfigure the application. The representation of the architecture is defined with a typed graph. Thus, the reconfiguration of the architecture is performed through a graph transformation. The K-Components are defined using the OMG-IDL3 language and C++ idioms. The main drawback of this approach is that adaptation is always realized through reconfiguration of the component architecture. An interception mechanism is missing to add an AOP support to the approach.

JBoss AOP [5] is a Java framework for AOP. It can be used in the context of the JBoss application server or standalone. As JAC [15], JBoss-AOP offers a set of pre-programmed aspects that can be used directly. JBoss AOP aspects can be woven with annotations, classic pointcut declarations or in a dynamic way at system runtime. When applied to the JBoss application server, aspects are woven to components. Similarly to FAC, dedicated XML fragments are used to deploy aspects. However, in FAC, XML files are used to describe a component assembly, with bindings. The notion of binding in EJB component model is missing. Components are coarse-grained components, encapsulated by containers, which do not express relationships between each other.

5 Conclusion

In this paper we have presented a general model for components and aspects called FAC and its mapping onto the Fractal component model. This model introduces three main notions: aspect component, aspect domain, and aspect binding. A crosscutting concern is embodied by a regular Fractal component called an aspect component. We have shown that an aspect component is an encapsulation of advice code. An aspect domain is the reification of the notion of a pointcut: the components picked out by an aspect component. The implicit relationship between a woven aspect component and the component in which the aspect component applies is a first-class entity called an aspect binding.

The main contribution of our approach is to bring aspect-oriented concepts to the component world, and conversely, to improve aspect-oriented approach with component notions. Thus, our three main notions (aspect component, aspect binding, aspect domain) are mapped onto the Fractal component model using existing notions of component, binding, and composite component.

We also provide a runtime support for crosscutting relationship reflection, which is an open issue in the aspect-oriented community. Moreover, we offer various abstraction views on aspect components woven on components in order to help the evolution of the modular and crosscutting concerns of a component and aspect system.

The long term objective of FAC is to work with aspects at three different levels [16]. The first level is the use of AOP at the program level, namely the level of objects that are encapsulated by components. Current AOP approaches fulfill this need. The second level is FAC itself with the notions of aspect component, aspect binding, and aspect domain. Joinpoints at this level are invocations on component interfaces. And finally, we plan to consider a third level, an architectural level, where joinpoints are architectural operations and transformations.

Acknowledgments

This work was partially funded by France Telecom under the external research contract number 46 131 097.

References

1. M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In O. Lehrmann Madsen, editor, *ECOOP'92: Proc. of the European Conference on Object-Oriented Programming*, pages 372–395. Springer, Berlin, Heidelberg, 1992.
2. J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *ICSE'02: Proc. of the International Conference on Software Engineering*, Orlando, FL, USA, May 2002.
3. G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.
4. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An open component model and its support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering*, Edinburgh, Scotland, May 2004.
5. B. Burke and al. JBoss-AOP. *www.jboss.org/developers/projects/jboss/aop*.
6. M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Proceedings of Middleware'01*, 2001.
7. J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. , LNCS 2192*, pages 81–88. Springer-Verlag, Sept. 2001.
8. F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 65–75, New York, NY, USA, 2002. ACM Press.
9. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.
10. B. Lagaisse and W. Joosen. Component-based open middleware supporting aspect-oriented software composition. In *CBSE*, pages 139–154, 2005.
11. K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.
12. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transaction on Software Engineering*, 26(1):70–93, January 2000.
13. M. Mezini and K.Ostermann. Conquering Aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 90–100. ACM Press, March 2003.
14. OMG. CORBA Components, v3.0 (full specification), Document formal/02-06-65, june 2002.
15. R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC : An aspect-based distributed dynamic framework. *Software Practise and Experience (SPE)*, 34(12):1119–1148, Oct. 2004.
16. N. Pessemier, O. Barais, L. Seinturier, T. Coupaye, and L. Duchien. A three level framework for adapting component-based systems. In *Second International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT05)*, Glasgow, Scotland, July 2005.

17. L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with components and aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE06)*, Lecture Notes in Computer Science, Stockholm, Sweden, jun 2006. Springer. To appear.
18. D. Suve. FuseJ web site. `http://ssel.vub.ac.be/fusej/`.
19. D. Suve, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 21–29. ACM Press, 2003.
20. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., 2002.