A Safe Aspect-Oriented Programming Support for Component-Oriented Programming

Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, Laurence Duchien

A Safe Aspect-Oriented Programming Support for Component-Oriented Programming

Nicolas Pessemier[(1)], Lionel Seinturier[(1)],
Thierry Coupaye[(2)], Laurence Duchien[(1)]
[(1)] INRIA Futurs - LIFL, Project Jacquard/GOAL
59655 Villeneuve d'Ascq, France
[(2)] France Telecom R&D, 28 chemin du Vieux Chêne,
38243 Meylan, France
{pessemie,seinturi,duchien}@lifl.fr
thierry.coupaye@rd.francetelecom.com

***Abstract*— In this paper we show that Aspect-Oriented Programming (AOP) can be safely supported by Component-Oriented Programming (COP) by providing a way to control the openness of a component with regards to AOP techniques. Our proposal reconciles the intrusive nature of AOP with the "black box property" of components in COP. We build a compromise between modularity and openness applying the open modules approach to components. The experiment has been achieved on FAC, our model that unifies the notions of component and aspect. We show that most of open modules principles are directly available within our approach, we then study requirements for others. Once all these principles integrated, we are able to tune the accessibility of the content of a component to AOP during system runtime. Thus, components become grey boxes with dynamic variation points accessible to AOP techniques.**

I. Introduction

Component-Oriented Programming (COP) proposes to enhance object-oriented programming by separating concerns into clearly defined entities, called components. Reusable components with contractually specified interfaces are defined and composed together [13]. Nevertheless, whatever the decomposition adopted to represent a system, it has been shown that some concerns are mixed within a same component (code tangling), and that some concerns are scattered across several components [3], [6]. These concerns which are called crosscutting concerns hinder the reusability, the maintainability, and the evolvability of applications.

To tackle these issues, some approaches have proposed a support for Aspect-Oriented Programming (AOP) in component-based systems [5], [7], [12]. AOP is a well-known paradigm to overcome this issue by modularizing crosscutting concerns using aspects [4]. The main issue of these approaches is that AOP is applied regardless of the components themselves, the aspects are woven on the objects which implement the components. This intrusiveness breaks the encapsulation property of components and consequently their implicit contracts. This appears to be a major issue in COP where contracts and encapsulation are fundamentals.

At the object level, solutions have been proposed to overcome the issue of intrusiveness of AOP. For example, Aldrich introduced the notion of open modules, a new module system to open a program to AOP while keeping modularity by hiding implementation details of the module [1]. A module is defined as a set of entities which share a set of access points for the join points (points in the program execution flow where aspects will apply) exported by the module. Using this module system, the content of a module can be preserved by designating only the variation points where aspects can act.

In this paper we propose to push the open modules approach a step further by applying it to COP. The objective is to provide a safe way to support AOP in COP by controlling the openness of a component with regards to AOP. This control over variation points of a component can be seen as a compromise between modularity and openness. Our study focus on an extended component model for components and aspects which is presented in [10]. Our model, named FAC, unifies COP and AOP notions together by representing AOP notions as component ones. We show that when components and aspects are unified, some open modules properties are directly handled as first-class entities in our model. We have then extended our model to handle all open modules properties. A component can declare its variation points. Since our model is dynamic, this declaration can evolve at runtime.

The remainder of this paper is organized as follows. Section II provides some background on our unified model for aspects and components and on the open modules approach. Section III shows how we have applied open modules to FAC. Finally, Section IV concludes.

II. Background

This section provides some background on our previous work [10] on the unification of AOP and COP, and introduces the principles of the open modules approach.

A. FAC: An unification of AOP and COP towards COP

Since COP fails in supporting crosscutting concerns [3], [6], our motivation was to give a support to AOP in COP but also to take advantage from the strong encapsulation property of COP in AOP. Therefore, our proposal is built as a twofold integration of AOP and COP which has the benefit of representing AOP notions using COP ones. Thus, we introduce

three main concepts which are related to the general notions of component, binding and composite-component that generally appear in COP [13].

- An **Aspect component** is the representation of an aspect as a component. It offers as a provided interface a piece of advice code (the additional behavior to weave on other components). Basically, an aspect component applies around incoming and outgoing calls on component interfaces. Because we represent an aspect as a component we call our approach symmetric. Aspects and components are components, and can interact together using bindings (Figure 1 represents an aspect component connected to other components using various types of bindings). Traditionally in AOP two dimensions are considered: the base and the aspect dimension. Aspects are woven on the base and the base if oblivious of the aspect dimension. In our approach these two dimensions are unified to facilitate the interactions between components and aspects and their evolution.
- An **Aspect domain** is the representation of the domain of action of an aspect. It is represented as a composite component which contains the aspect component and the components on which it is woven. The notion of aspect domain can be seen as a reification of the notion of pointcut in AOP. Usually a pointcut is a description of a set of join points on which an advice code is woven. In our model we reify this notion as a first class entity (a composite-component) which contains the aspect component woven, and all the components affected by the aspect component. This clarifies the domain of impact of an aspect component in a system. This explicit relationship between advised code and aspects is a notion currently missing in AOP. Figure 1 gives an example of the aspect domain of the `transaction Aspect Component` which is woven on components `C`, `D` and `E`.
- An **Aspect binding** is the representation of the implicit link which exists between an aspect and a component on which the aspect is woven. The aspect binding notion can be seen as a more fine-grained notion than the aspect domain to capture the interaction between a component and a particular aspect component. Our philosophy is to consider only one dimension (aspects are components) and two types of interactions (regular bindings and aspect bindings). Aspect bindings are used to connect an aspect component with a component. By this way, each component can locally manage the aspects applied on its incoming and outgoing interfaces.

We have successfully mapped this general model with its three notions to the Fractal component model. Fractal [2] is a reflective and extensible component model, where bindings can be set and unset dynamically (at runtime); reflection is available through the use of special kinds of meta-interfaces called control interfaces.

We have extended this model by introducing our three main notions. This extension is called Fractal Aspect Component (FAC for short). We have used the provided notions of component and composite-component to represent our notions of aspect component and aspect domain. We have introduced a new control interface called the *weaving interface*. This interface, which appears on each component of the system, is in charge of setting/unsetting aspect bindings and of weaving aspect components. It has the benefit of locally managing the ordering of aspects for a component.

In addition to the advantages mentioned in the description of our three notions, the mapping onto the Fractal component model allows setting/unsetting aspect bindings dynamically. This makes our weaver dynamic and this opens the way to dynamic adaptation [9].

The complete description of our general model for component and aspect and its mapping to the Fractal component model [2], named FAC is beyond the scope of this paper and can be found in [10]. The next sub-section introduces the open modules approach which allows controlling the degree of openness of a module to AOP.

B. Open modules approach

The concept of *open modules* has been introduced by Aldrich to limit the access to join points of a system, which are accessed intrusively in AOP. With this approach any exposed join points has to be declared within a module, a special set of classes, to be accessed by aspects. The *open modules* approach is defined by Aldrich as follows: *"Open Modules describes a module system that:*

- ***Rule 1** allows external advice to interactions between a module and the outside world (including external calls to functions in the interface of a module)*
- ***Rule 2** allows external advice to pointcuts in the interface of a module*
- ***Rule 3** does not allow external modules to directly advise internal events within the module, such as calls from within a module to other functions within the module (including calls to exported functions).*"

We intentionally add rule numbers on items to facilitate the discussion in the following sections. The complete description of this module system can be found in [1].

More recently, the concept of open modules has been applied to AspectJ [8]. In this study, authors have extended the concept with new interesting features. Most of them are specifically related to AspectJ in order to support AspectJ pointcuts. Nevertheless it seems to us that some can be generalized out of the context of AspectJ. Thus, the most important feature with regards to our application of open modules to COP is the ability to open and not only to reduce the visibility on join points of a module. This can become extremely useful when using for instance debug aspects, or when considering dynamic adaptation using AOP. Among new features provided by this study, an interesting one is the ability to designate to which a pointcut is exposed to using a pattern language based on package hierarchy.

To help the discussion of the next section we will call the ability to designate which aspect has access to a module ***Rule 4***, and the ability to open a module by exposing join points ***Rule 5***.

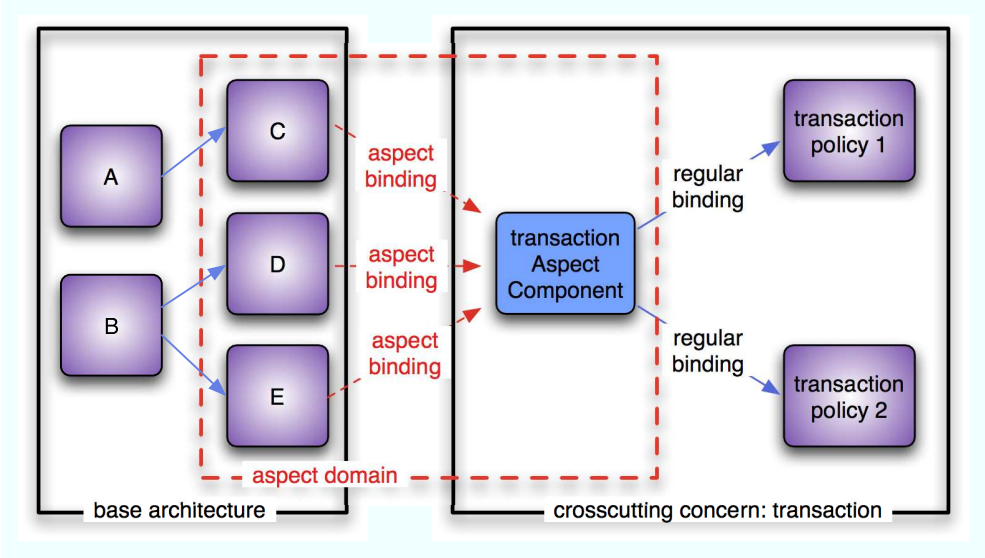

Fig. 1. Aspect binding best practice

III. Applying Open Modules to FAC

In this section we detail how we have applied the open modules approach to FAC. Some of the rules defined in open modules are directly mapped to existing notions of our model (Section III-A), some others require the introduction of new features to be correctly managed (Section III-B).

A. Similarities

The first obvious similarity is related to the notion of a module and a component. A module in the open modules approach is a collection of classes which share a set of access points to AOP. A component in COP is a contractually specified entity which provides and requires services by means of interfaces. A component is a black box which naturally hides its implementation details as required by *Rule 3* of open modules. Given that in FAC, join points are incoming and outgoing calls on component interfaces, the definition remains correct with regards to *Rule 1*, *2* and *3*. *Rule 1* and *2* are understood in FAC by the fact that an aspect component only applies to client and server interfaces (*Rule 2*). Following the definition of *Rule 1*, an aspect component interacts between a module (component) and the outside world (other components). *Rule 3* is preserved as soon as we do not want to break encapsulation in FAC. Join points are not points inside a component. However, when applying aspect component behavior on component external interfaces, we may consider that the original behavior of the component is altered by the aspect component. Thus, the behavior expected from a given component may be different. It seems important that the weaving of aspect component on provided and required interfaces of a component should be better controlled in order to provide a safer integration of crosscutting concerns. We elaborate more on that particular point in Section III-B.

The second similarity is related to *Rule 4* which has been defined to the particular use of AspectJ but can be also used within our system. This rule allows to clearly designate which aspect can apply on a given exported join point of a module. A regular expression is given as a parameter of the `expose to` declaration which designates a set of packages that are authorized to access the module. In FAC we have a very similar notion: the aspect binding. An aspect binding is set between a component and an aspect component using the weaving interface of the component. Because the weaving interface is a kind of meta interface, we can consider that the access policies defined on components are of the same type of meta-informations than the ones corresponding to the `expose to` definitions used in the extension of AspectJ supporting open modules. This means that we can consider *Rule 4* as naturally handled by each individual component which are able to choose the aspect to be impacted by.

At this point we have seen that *Rule 1*, *2*, *3*, and *4* are naturally handled by our model and its mapping to FAC. In the next section we study the requirements to manage remaining rule, Rule *5*.

B. Dissimilarities

We have seen that in our model and in its mapping to Fractal, FAC, the considered join points are incoming and outgoing calls on component interfaces. Thus, *Rule 3* is implicitely preserved when considering components as modules. However, this also means that the join points inside the component are not exposed. The original idea of open modules is to define some pointcuts and join points and make them available by means of interfaces of a module. In our case, the content of a component is implicitly protected, but we still need a support to give an access to other join points, *i.e.*, join points inside a component. Nevertheless, associating the level of implementation of components and the level of interaction between components (more architectural) takes part in our global vision of what really means applying AOP to COP that we have exposed in [9]. The unification of these two levels will allow us to look inside components and to externalize some

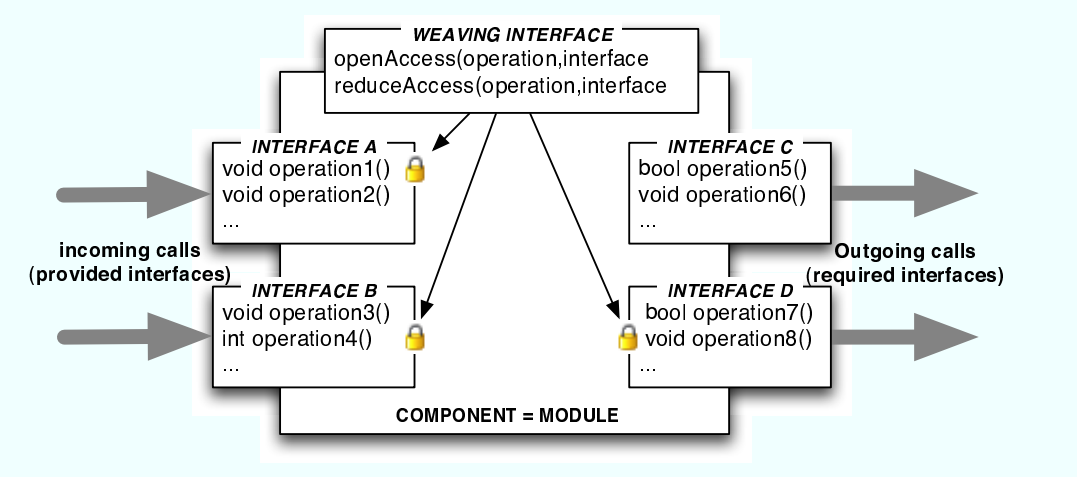

Fig. 2. A conceptual view of Open Modules applied to FAC. A component is a module which controls the access to its provided and required interfaces by means of the weaving interface. The content of the component is not exposed to aspects.

join points. We believe that these internal join points must also be controlled by the weaving interface. Thus, we would be able to fulfill needs for accessing internal join points, while preserving *Rule 3* by means of the weaving interface to control what is accessed or not. This approach has limitations with regards to legacy components that would not have be able to be instrumented to support AOP techniques. In our approach, we only consider a full-fledged component and aspect approach, where the design of the system follows the same formalism, the same design model.

The remaining rule (*Rule 5*) states that the open modules should not be limited to reducing the access to aspects but also to opening it. In Section II-A we have defined the role of the weaving interface as an interface to manage the setting/unsetting of aspect bindings, the weaving and the ordering/re-ordering of aspect components. We have extended the role of this interface to also manage the openness of a component to AOP. A conceptual view of the role of the weaving interface is presented in Figure 2. The weaving interface is able to prevent the weaving of aspects on a particular operation of an interface. Because FAC is a fully dynamic framework, these policies can be changed during runtime. A component can then be adapted to open or reduce the access to its join points. Moreover, since we have discussed it in Section III-A, weaving an aspect component on component external interfaces may change the expected behavior from other components as soon as it intercepts communications between interfaces. The idea of controlling the external join points which are accessible or not by other components seems interesting even if it was not originally considered by Aldrich in the first version of open modules.

IV. Concluding remarks

We have seen that AOP and COP can be reconciled on the particular issue of the intrusive property of AOP versus the strong encapsulation property of COP. To do so, we have applied the open modules approach to FAC, a unified model for components and aspects. Following the open modules philosophy our approach is able to open a component to AOP while keeping its content hidden from the outside. This compromise opens the way to a safe integration of AOP in COP. It is safe in the sense that the intrusiveness of AOP is finely managed on each individual component. Moreover in the case of FAC, we have seen that the openness of a component to AOP can be managed at runtime, allowing a component to adapt to unanticipated requirements. Black box components become grey box components providing variability points where AOP can access.

This integration of open modules approach to FAC takes part of our overall vision of applying AOP to COP presented in [9]. This vision is based on three levels: (1) An architectural level which is achieved with FAC where aspects notions are mapped on component ones; (2) A control level where aspects can be used to inject the control level of components into the remaining level, (3) the level of implementation of components. In [11] we show how AOP can be used for (2). In this paper we present a link between the architectural and the implementation level: Opening a component to expose internal join points and then, weaving architectural aspects to this internal join points (link between (1) and (3)). Our next step is to make all this three levels work together, allowing a coherent weaving of aspects whatever the chosen level.

Acknowledgments

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

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586, pages 144–168. Springer, 2005.

[2] 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.

[3] 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.

[4] 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.

[5] B. Lagaisse and W. Joosen. Component-based open middleware supporting aspect-oriented software composition. In *CBSE*, pages 139–154, 2005.

[6] 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.

[7] 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.

[8] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding Open Modules to Aspectj. In *Proceedings of the 5nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. ACM Press, March 2006.

[9] 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.

[10] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, Lecture Notes in Computer Science. Springer, Mar. 2006.

[11] 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.

[12] 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.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2002.