A Seamless Extension of Components with Aspects using Protocols

Angel Núñez, Jacques Noyé

A Seamless Extension of Components with Aspects using Protocols

Angel Núñez, Jacques Noyé
Project OBASCO, EMN-INRIA, LINA
4 rue Alfred Kastler, 44307, Nantes, France
{angel.nunez,jacques.noye}@emn.fr

***Abstract*—This paper shows how components and aspects can be seamlessly integrated using protocols. A simple component model equipped with protocols is extended with *aspect components*. The protocol of an aspect component observes the service requests and replies of plain components, and possibly internal component actions, and react to these actions (possibly preventing some base actions to happen as is standard with AOP). A nice feature of the model is that an assembly of plain and aspect components can be transformed back into an assembly of components. All this is done without breaking the black-box nature of the components (dealing with internal actions requires to extend the component interface with an *action interface*).**

I. Introduction

Aspect-Oriented Programming (AOP), initially developed in the context of Object-Oriented Programming (OOP), has shown that classes are not enough to properly modularize all the concerns of an application. The use of classes alone leads to so-called *crosscutting concerns*, scattered in the various classes that build the application. AOP makes it possible to collect these scattered parts of the concern in a new modularization construct: an *aspect*, and leave the set of classes to which the aspect applies, the *base program*, free from any code for the concern. It is then the job of the compiler to *weave* the aspect and the base program, *i.e.*, to introduce concrete connections between the aspect and the classes, using the aspect *pointcut* and *advice*. The *pointcut* is a predicate defining the *join points*, *i.e.*, the execution points in the base program which should be affected by the aspect. The *advice* defines the new behavior to be inserted at the join points, including calls to base program methods. An abstract way of considering *weaving* is to see it as a transformation back to the scattered and tangled code that would have been written by hand using plain OOP (in practice, however, weaving is not a source-to-source transformation, but a direct transformation to lower-level code, typically bytecode). Apart from improving the modularity of the application, AOP also allows incremental programming: the base program can be developped independently from the aspects, which can be developped at a later stage.

The situation is not really different when moving from OOP to plain Component-Oriented Programming (COP). Crosscutting concerns have to be dealt with. In a strict black-box model, incremental programming is not possible. The crosscutting concern has to be implemented as a (collection of) component(s). Connection code has to be introduced in the implementation of the base components, which must also be equipped with the proper interfaces.

This paper deals with improving on this situation by showing how AOP and COP can be seamlessly integrated. We start with a simple component model where components are defined as a set of (structural) interfaces describing their provided and required services and a protocol, describing the behavior of the components in terms of service requests and replies as well as internal actions. We then extend this model by adding *aspect components*, which are also defined as a set of interfaces and a protocol. This protocol has however a slightly different meaning than a standard component protocol. It corresponds to the definition of a *stateful* concurrent aspect [1], [2], [3], which can observe various base actions (service requests and replies, internal actions) and react accordingly. This includes the possibility of preventing a base action from happening, a standard feature of AOP. In this model, weaving can be seen as a transformation of the initial system of plain components and aspect components into a system of plain components.

Section II gives more details on our approach. Section III describes our simple reference model. Section IV extends this model with aspect components. Section V shows how weaving transforms an initial system with aspect components into a system of plain components. Section VI illustrates the approach with a small example. Section VII discusses related work. Finally, Section VIII concludes.

II. The approach

As explained in the introduction we integrate the notion *(class, aspect)* of AOP with the notion *(component, aspect component)* in a seamless way. For doing that, we use Baton [3], a language for programming concurrent stateful aspects in Java. This language is based on the Concurrent Event-based AOP (CEAOP) approach [2] that models concurrent base programs and concurrent aspects as Finite State Processes (FSP). CEAOP models the weaving of aspects into the base program as FSP composition of the corresponding FSPs. Baton implements these ideas in the OOP world.

In order to implement the integration of AOP and COP, we evolve Baton into a language for programming aspect components that applies to component-based applications. The weaving of aspect components written in Baton into an application is implemented as the generation of a plain component

representing the aspect component, which is connected to the rest of the components of the system.

For the time being, this paper just considers the case of weaving a single aspect into a component-based system. However, we lay the foundations for the full support of the concurrent aspects modeled by CEAOP.

In the following sections, we describe a simple component model used as a reference model. Then, we present the syntax of Baton and we describe the weaving of aspect components.

III. A SIMPLE COMPONENT MODEL

This section describes a very simple component model with the basic features assumed by our aspect-component language.

We consider a minimal component model, whose components are *black boxes* equipped with interfaces and a protocol. Furthermore, the model allows the definition of *primitive* and *compound* components.

A primitive component declares its interfaces and its protocol with the following syntax:

```
Component   ::= component Id implements
                Interfaces { Behavior }
Interfaces  ::= Id ( , Id )*
```

The definition of the interfaces is done outside the component. Each interface declares provided and required services with the following syntax:

```
Interface   ::= interface Id { IntBody }
IntBody     ::= ( Action ; )*
Action      ::= Mod Name ( Params? )
Mod         ::= provides
Mod         ::= requires
```

Action represents a service that can be provided or required, *Name* is the name of the service, and *Params* are optional parameters declared by the service (parameters are used for message passing purposes).

The protocol defines the behavioral interface of the component using an FSP. We assume a protocol with the following syntax:

```
Behavior    ::= ProcDef ( , ProcDef )* .
ProcDef     ::= ProcId = Body
Body        ::= ( Prefix ( | Prefix )* )
Prefix      ::= ( ActLabel -> )* ProcId
ActLabel    ::= Name ( Params? )
```

The label of each transition (*ActLabel*) consists of the name of a service declared in some interface (*Name*) and its optional parameters (*Params*). We say that the transition *refers* to the service. The semantics of each transition depends on the type of service the transition refers. If a transition refers to a provided service, then the semantics of the transition is that the component receives a request for the service. If a transition refers to a required service, then the semantics of the transition is that the component sends a request for the service.

A compound component is an assembly of subcomponents. Its interfaces are formed by interfaces *exported* from subcomponents. An exported interface is such that it has not been bound or it only defines provided services. The protocol of a compound component is obtained from the protocols of its subcomponents by performing FSP composition.

Components are connected through their interfaces. We just consider binary communication (one sender, one receiver). When connecting two interfaces, services are bound through name matching. The condition is that each required service of one interface is provided by a service of the second interface.

A recent approach introduces the notion of *open modules* [4], which can be used to expose internal actions of a black-box component. We extend the component interface with an *action interface* in order to include this notion. Then a primitive component may not only declare the standard interface of provided and required services, but also action interfaces.

The action interface defines abstract internal actions that are made observable from outside the component and are included in the component protocol together with provided and required services. The syntax of an action interface is as follows:

```
Interface   ::= interface Id { ActIntBody }
ActIntBody  ::= ( ExpAction ; )*
ExpAction   ::= exposes Name ( Params? )
```

IV. A LANGUAGE FOR PROGRAMMING ASPECT COMPONENTS

We seamlessly integrate the notion of aspect in AOP into the notion of aspect component. For doing this, we present Baton as a language for programming aspect components. This section describes the syntax of the language.

A. Aspect components

An aspect component, as the name implies, is an aspect with a component flavor. Like a component, it is defined using a set of interfaces and a protocol. Its protocol has however a slightly different meaning than a standard component protocol. It corresponds to the definition of a stateful concurrent aspect. The concrete syntax of an aspect component (see below) is very similar to the syntax of a plain component, the differences are in the definition of the interfaces and the protocol.

```
Component   ::= aspect Id implements
                Interfaces { Behavior }
Interfaces  ::= Id ( , Id )*
```

An interface is defined by the following syntax, which is very similar to the syntax of a plain-component interface:

```
Interface   ::= interface Id { IntBody }
IntBody     ::= ( Action ; )*
Action      ::= Mod Name ( Params? )
Mod         ::= event
Mod         ::= skippable event
Mod         ::= action
```

Whereas a plain-component interface declares required and provided services, an aspect-component interface declares abstract actions representing base-program actions. Actions denoted with the keyword `event` represent actions that the aspect component observes in the base program. Actions denoted with the keyword `skippable event` represent actions that the aspect component observes and can make the base program skip. Actions denoted with the keyword `action` represent actions that the aspect component requires to implement its advices.

The syntax of the aspect-component protocol is as follows:

Behavior ::= *ProcDef* (, *ProcDef*)* .
ProcDef ::= *ProcId* = *Body*
Body ::= (*Prefix* (| *Prefix*)*)
Prefix ::= (*ActLabel Advice?* ->)* *ProcId*
ActLabel ::= *Name* (*Params?*)

As we can see, the syntax of the aspect-component protocol is almost the same as the syntax of the plain-component protocol, except that the former allows the definition of *advised* transitions (i.e. transitions including an aspect advice, represented by the non-terminal *Advice*). For both, advised and non-advised transitions, *ActLabel* corresponds to an abstract action declared in the interface, more precisely, it corresponds to an action declared with the keyword `skippable event` for advised transitions, and to an action declared with the keyword `event` for non-advised ones. The semantics of a non-advised transition is that the aspect changes its state with the occurrence of the corresponding base action. The semantics of an *advised* transition is that, in the context of the corresponding base action, the aspect may execute advices and may prevent the action from happening. The syntax of *Advice* is as follows:

Advice ::= > *Before PS After*
Before ::= (*ActLabel* ;)*
After ::= (; *ActLabel*)*
PS ::= skip | proceed

Advices (*Before*, *After*) are sequences of abstract actions. Each of these abstract actions is an action declared with the keyword `action` in the aspect-component interface. The semantics of each action is that the aspect component sends a request for the corresponding service in some component of the system.

The parameters (*Params*) declared in the syntax of the aspect component are used for passing information from the base program to the aspect component. These parameters are available in the scope of each transition (*Prefix*).

B. Connectors

A connector binds abstract actions declared in the interface of an aspect component with concrete actions declared in the interface of the base components. The syntax of a connector is as follows:

Connector ::= connector { *Connection** }
Connection ::= connects *Action* to *Pattern* ;

Action is an action declared in the interface of an aspect component. *Pattern* corresponds to a pattern that permits to match actions declared in the interface of a component.

V. Weaving

Weaving an aspect component into a component-based system corresponds to generating a system with plain components. This is done by transforming the aspect component into a *plain aspect component* (PAC) and connecting it to the rest of the components of the system.

A. The aspect component as a plain component

This section describes how an aspect component is implemented as a plain component. In the remainder we describe the generation of the interfaces and the protocol of this component.

1) Generation of the protocol: The protocol of the PAC is the result of transforming the protocol of the aspect component. As previously explained, the aspect component observes actions of the component-based application, this is implemented in the PAC as the reception and sending of synchronization events (equivalent to the events introduced by the CEAOP model). These events are implemented as component services. We obtain the protocol of the PAC by applying the following transformations:

T(*name(params)* -> *P*) =
 eventB_*name(params)* -> eventE_*name()* -> *P*

T(*name(params)* > *before*; *ps*; *after* -> *P*) =
  eventB_*name(params)* -> *before* ->
   *ps*B_*name()* -> *ps*E*name()* -> *after* ->
   eventE_*name()* -> *P*

The first transformation describes that taking into account a base program action *name(params)* is implemented as the reception of an event `eventB`_*name(params)* indicating that the action is about to be executed (the `B` in `eventB` is for begin) followed by the reception of an event `eventE`_*name()* indicating that the action has been executed (the `E` in `eventE` is for end).

The second transformation describes that a transition that introduces advices, and can make the base program skip an action *name(params)*, is programmed through the following communication between the PAC and a base component:

i. The PAC receives the event `eventB`_*name(params)* from a base component when the action is about to be executed.

ii. Then, it executes the sequence of actions denoted by *before* and emits either the event `skipB_`*name()* or the event `proceedB_`*name()* to indicate to the base component whether the action has to be skipped or not.
iii. The base component receives the last event, skip the action or proceed, and emits either the event `skipB_`*name()* or the event `proceedB_`*name()* indicating whether the action has just been skipped or not.
iv. The PAC receives the last event, executes the sequence of actions denoted by *after* and emits the event `eventE_`*name()* to indicate to the base component that this base component can continue with its computation.
v. The base component receives this event and continues.

2) Generation of the interfaces: The interfaces of the PAC are derived from the interfaces of the aspect component. They basically consist of the declaration of the synchronization events used in the PAC protocol.

An action declared as `event` *name(params)* in the aspect-component interface is used in non-advised transitions of the aspect-component protocol. It generates the following interface in the PAC:

```
interface SyncA_name {
  provides eventB_name(params);
  provides eventE_name();
}
```

In an analogous way, an action declared as `skippable event` *name(params)* in the aspect-component interface is used in advised transitions of the aspect-component protocol. It generates the following interface in the PAC:

```
interface SyncA_name {
  provides eventB_name(params);
  requires eventE_name();
  requires proceedB_name();
  provides proceedE_name();
  requires skipB_name();
  provides skipE_name();
}
```

Finally, an action declared as `action` *name(params)* in the aspect-component interface is used in the advice of advised transitions and generates the following interface in the PAC:

```
interface A_name {
  requires name(params);
}
```

B. Connecting plain components

Once the PAC has been generated, the second part of the weaving process is to connect the PAC to the rest of the components of the system. The Baton connector tells us which base component should be connected to the PAC, more precisely, which concrete actions from the base components should be connected to which abstract actions of the aspect component.

A Baton connector matches services and internal actions declared in the interface of a base component. If a service or internal action *s(params)* is matched, then there is an association with an abstract action used by the aspect component (to simplify things, we suppose that for each abstract action only one service or internal action is matched in all the component hierarchy). If the abstract action has been declared as `event` *name(params)* or `skippable event` *name(params)*, then the PAC implements an interface `SyncA_`*name*. A complementary interface, namely `SyncB_`*name*, is introduced in the component to make the connection. Furthermore, the necessary modifications in the protocol of the base component are performed.

We define two transformations to the base-component protocol:

```
T(s(params) -> P) =
 eventB_name(params) ->
  s(params) ->
  eventE_name() -> P


T(s(params) -> P) =
 eventB_name(params) -> proceedB_name() ->
  s(params) ->
  proceedE_name() -> eventE_name() -> P
 |eventB_name(params) -> skipB_name() ->
  skipE_name() -> eventE_name() -> P
```

The first transformation applies if the abstract action *name(params)* has been declared as `event`. Then the component has to generate one event before the execution of the concrete action and another event after. The second transformation applies if the abstract action has been declared as `skippable event`. Then the component has to generate events that introduce the possibility of skipping the action (as seen in the generation of the PAC protocol).

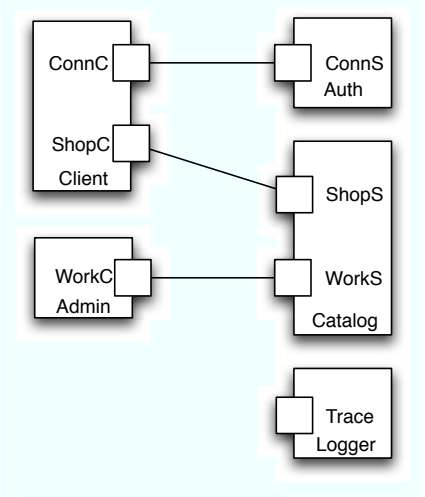
If the abstract action has been declared as `action` *name(params)*, then there is an interface `A_`*name* that is connected to the interface of *s(params)*.

We have introduced a language for programming aspect components and shown how these components can be implemented as plain components. Afterwards, this section has described the process of weaving. The next section presents an example to illustrate the approach.

VI. Example

To illustrate the approach we use a simple example based on e-commerce applications. Clients connect to a website and must login to identify themselves, then they may browse an on-line catalog. The session ends at checkout, that is, as soon as the client has paid. In addition, an administrator of the shop

can update the website at any time by publishing a working version. Consider the application has been programmed as the component-based system of the following figure:



The application consists of five components: `Client` represent a client, `Admin` an administrator, `Auth` an authorization entity, `Catalog` the on-line catalog, and `Logger` a component that implements a logging functionality. The interfaces `Client.ShopC` and `Catalog.ShopS` declare a service `getItems()`, which is used by the client to browse the catalog, and a service `pay(List items)`, used to make a payment. The interfaces `Client.ConnC` and `Auth.ConnS` declare a service `login(Credential credential)`, which is used by the client to identify itself. The interfaces `Admin.WorkC` and `Catalog.WorkS` declare a service `addItem(Item item)`, which is used by the administrator to update the catalog.

As an example we show the definition of the component `Admin`:

```
component Admin implements WorkC {
  Begin = ( addItem(Item item) -> Begin).
}


interface WorkC {
  requires addItem(Item item);
}
```

Let us now consider the problem of canceling updates to the client-specific view of the e-commerce shop during session, e.g. to ensure consistent pricing to the client. We can define a suitable aspect component, which we call `Consistency`, to solve this problem. The aspect component programmed in Baton is as follows:

```
aspect Consistency implements ConsistencyI
{
  OutSession =
    ( login() -> InSession ),
  InSession =
    ( update() > skip; log() -> InSession
    | checkout() -> OutSession ).
}

interface ConsistencyI {
  event login();
  event checkout();
  skippable event update();
  action log();
}
```

This aspect initially starts in state `OutSession` and waits for a `login()` action from the base program (other actions are just ignored). When the `login()` action occurs, the base program resumes by performing the `login()`, and the aspect proceeds to state `InSession` in which it waits for either an `update()` or a `checkout()` action (other actions being ignored). If `update()` occurs first, the associated advice `skip; log()` causes the base program to skip the `update()` action and the `log()` action is performed. Then the base program resumes and the aspect returns to state `InSession`. If `checkout()` occurs first, the aspect returns to state `OutSession`. Since `update()` actions are ignored in state `OutSession`, updates occurring out of a session are performed, while those occurring within sessions (state `InSession`) are skipped.

In order to weave the `Consistency` aspect component, we define the following Baton connector, which binds the abstract actions declared in the interface of the `Consistency` aspect component with concrete actions declared in the system:

```
connector Connector {
 connects login() to *.*.login(..);
 connects checkout() to *.*.pay(..);
 connects update() to *.*.addItem(..);
 connects log() to Logger.Trace.log();
}
```

In the weaving of the aspect component into the application, the PAC is generated and connected to the corresponding components of the system. The code below shows the definition of the resulting PAC:

```
component PAC implements SyncA_login,
         SyncA_checkout, SyncA_update,
         A_log
{
 OutSession =
 ( eventB_login() -> eventE_login() ->
   InSession),
 InSession =
 ( eventB_update() -> skipB_update() ->
   skipE_update() -> log() ->
   eventE_update() -> InSession
 | eventB_checkout() -> eventE_checkout()
   -> OutSession ).
}
```

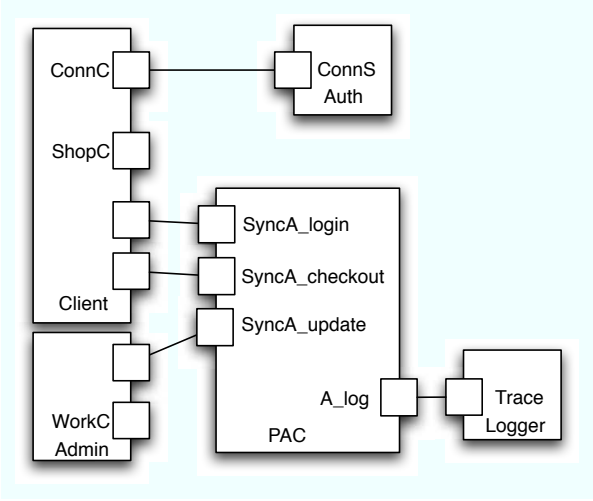
```
interface SyncA_login {
  provides eventB_login();
  provides eventE_login();
}


interface SyncA_checkout {
  provides eventB_checkout();
  provides eventE_checkout();
}


interface SyncA_update {
  provides eventB_update();
  requires eventE_update();
  requires proceedB_update();
  provides proceedE_update();
  requires skipB_update();
  provides skipE_update();
}


interface A_log {
  requires log();
}
```

The following figure illustrates the PAC with its corresponding interfaces connected to the rest of the system (for the sake of space, we have hidden the component `Catalog`).



The weaving also produces the instrumentation of some base components.As an example the component `Admin` becomes equivalent to:

```
component Admin implements
                    WorkC, SyncB_update
{
 Begin =
 ( eventB_update() -> proceedB_update() ->
   addItem(Item item) -> proceedE_update()
   -> eventE_update() -> Begin
 | eventB_update() -> skipB_update() ->
   skipE_update() -> eventE_update() ->
   Begin ).
}


interface WorkC {
  requires addItem(Item item);
}


interface SyncB_update {
  requires eventB_update();
  provides eventE_update();
  provides proceedB_update();
  requires proceedE_update();
  provides skipB_update();
  requires skipE_update();
}
```

VII. Related Work

The work on open modules [4] suggests that module interfaces should be extended with pointcut names to be used by aspect implementors in order to advise the aspect as well as by the module implementor who, in case of an evolution of the module, may have to update the definition of the pointcut. We do something very similar with action interfaces, which, together with the component protocol, is an abstract description of the execution points within the component that an aspect may affect.

FuseJ [5] aims at achieving a symmetric, unified component architecture that treats aspects and components as uniform entities. Then, it addresses the problem of properly configuring connections between components implementing a concern and the rest of the system. FuseJ proposes a powerful *configuration language* to program component connections that support crosscutting connections. This is conceptually similar to a Baton connector.

Fractal Aspect Component (FAC) [6] introduces a general model for components and aspects. FAC decomposes a software system into regular components and *aspect components* (ACs), where an AC is a regular component that embodies a crosscutting concern. An *aspect domain* is the reification of the notion of a pointcut: the components picked out by an AC. Furthermore, the implicit relationship between a woven AC and the component in which the aspect component applies is a first-class entity called an *aspect binding*. A posterior work [7] introduces the notion of open modules to FAC.

None of these approaches support the definition of connections between components implementing advices and base components that depend on a global shared state. Baton permits to program this kind of smart connections that corresponds to stateful aspects in the AOP terminology. Furthermore, none of these approaches seamlessly integrate AOP into COP as Baton does.

VIII. Conclusion

This paper proposed a solution of the problem of modularizing crosscutting concerns in component-based system. Our main contribution is to show how AOP and COP can be seamlessly integrated. The tuple *(class,aspect)* of AOP has been introduced into COP as the tuple *(component, aspect components)*. In concrete terms, Baton, a language for programming concurrent aspects in Java, has been evolved into a language for programming aspect components that are applied to a component-based system.

We have shown how weaving an aspect component and plain components can produce a system with only plain components. This can actually be extended to the weaving of many aspects composed together using composition operators [2]. The operators are then also translated into plain components. The action interface makes it possible to deal with aspects, including some form of incremental development, without breaking the black-box property of components. Indeed, the action interfaces have to be anticipated and made part of the component interface at design time but aspect weaving can still take place at deployment time (this implies that component implementations can be instrumented at deployment time, which is for instance the case when these implementations are provided as Java bytecode).

As future work, we plan to extend these ideas to a more realistic component model including, for instance, multi-ary communication. In this regard, we could combine efforts with works on component models with explicit protocols such as [8]. We also plan to integrate support for distributed aspects in the line of AWED [9].

Acknowledgment

We would like to thank the anonymous reviewers for their helpful comments and mention that this work has been partly supported by the project AMPLE: Aspect- Oriented, Model-Driven, Product Line Engineering (STREP IST-033710).

References

[1] R. Douence, P. Fradet, and M. Südholt, "Composition, Reuse and Interaction Analysis of Stateful Aspects," in *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, K. Lieberherr, Ed. Lancaster, UK: ACM Press, Mar. 2004, pp. 141–150.

[2] R. Douence, D. Le Botlan, J. Noyé, and M. Südholt, "Concurrent Aspects," in *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'06)*. Portland, USA: ACM Press, Oct. 2006, pp. 79–88.

[3] A. Núñez and J. Noyé, "Baton: A Domain-Specific Language for Coordinating Concurrent Aspects in Java," in *Proceedings of the 3ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA07)*, Toulouse, France, Mar. 2007.

[4] J. Aldrich, "Open modules: Modular reasoning about advice," in *ECOOP 2005 - Object-Oriented Programming, 19th European Conference*, ser. Lecture Notes in Computer Science, M. Odersky, Ed., vol. 3586. Glasgow, UK: Springer-Verlag, Jul. 2005, pp. 144–168.

[5] D. Suvée, B. D. Fraine, and W. Vanderperren, "A symmetric and unified approach towards combining aspect-oriented and component-based software development." in *CBSE*, ser. Lecture Notes in Computer Science, I. Gorton, G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, Eds., vol. 4063. Springer, 2006, pp. 114–122.

[6] N. Pessemier, L. Seinturier, T. Coupaye, and L. Duchien, "A Model for Developing Component-based and Aspect-oriented Systems," in *Proceedings of the 5th International Symposium on Software Composition (SC06)*, ser. Lecture Notes in Computer Science, vol. 4089. Vienna, Austria: Springer-Verlag, Mar. 2006, pp. 259–273.

[7] ——, "A Safe Aspect-Oriented Programming Support for Component-Oriented Programming," in *Proceedings of the 11th International ECOOP Workshop on Component-Oriented Programming (WCOP06)*, R. Reussner, C. Szyperski, and W. Weck, Eds., Nantes, France, Jul. 2006, technical Report 2006-11, Universität Karlsruhe, Fakultät für Informatik.

[8] F. Fernandes, R. Passama, and J. C. Royer, "Components with Symbolic Transition Systems: A Java Implementation of Rendez-Vous," in *Proceedings of the Communicating Process Architecture Conference*, 2007.

[9] L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée, "Explicitly distributed AOP using AWED," in *OOPSLA 2006, Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, P. L. Tarr and W. R. Cook, Eds. Portland, Oregon, USA: ACM Press, Oct. 2006.