A seamless extension of components with aspects using protocols

Angel Núñez and Jacques Noyé

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 defined 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 developed independently from the aspects, which can be developed 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 [2], [3], [5], 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 transform 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 [5], a language for programming concurrent stateful aspects in Java. This language is based on the Concurrent Event-based AOP (CEAOP) [3] approach 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 in which a single aspect is weaved 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 named interfaces and
a protocol. An interface declares provided and required services. The protocol defines the behavioral interface of a
component using an FSP. We assume a protocol with the following syntax:

\[
\begin{align*}
\text{Behavior} & ::= \text{ProcDef} ( , \text{ProcDef} )^* . \\
\text{ProcDef} & ::= \text{ProcId} = \text{Body} \\
\text{Body} & ::= ( \text{Prefix} ( | \text{Prefix} )^* ) \\
\text{Prefix} & ::= ( \text{Label} ( \text{Params} ? ) \rightarrow )^* \text{ProcId}
\end{align*}
\]

The label of each transition (Label) corresponds to the name of a service. We say that the transition refers to the
service. The semantics of a transition named with a label label depends on the type of service associated to the label.
If label refers to a provided service, then the semantics of the transition is that the component receives a request of
the service. If label refers to a required service, then the semantics of the transition is that the component requests
such a service.

Components are connected through their interfaces. We just consider binary communication (one sender, one re-
ceiver). 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.

The model permits the definition of primitive and compound components. A primitive component declares its
interfaces, its protocol and defines an implementation for the provided services. A compound component is an as-
sembly of subcomponents. Its interfaces are formed by interfaces exported from subcomponents that have not been
bound. Its protocol is obtained from the protocols of its subcomponents by performing some FSP operations such as
composition.

A. Action interfaces

A recent approach introduces the notion of open modules [1], 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. The action interface declares abstract internal actions that are observable from outside the component and
are included in the component protocol together with provided and required services.

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 component flavor. Like any component, it is
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. The concrete syntax of an
aspect component is as follows:

\[
\begin{align*}
\text{AspDef} & ::= \text{aspect component ID} \{ \text{AspBody} \} \\
\text{AspBody} & ::= \text{IntDec}^* \text{BehaviorDef} \\
\text{IntDec} & ::= \text{interface ID} ;
\end{align*}
\]

An aspect component declares a set of interfaces (IntDec) and defines a protocol (BehaviorDef). The definition
of an interface is done outside the aspect component with a syntax as follows:

\[
\begin{align*}
\text{IntDef} & ::= \text{interface ID} \{ \text{IntBody} \} \\
\text{IntBody} & ::= ( \text{Action } ; )^* \\
\text{Action} & ::= \text{Mod Label} ( \text{Params} ) \\
\text{Mod} & ::= \text{event} \\
\text{Mod} & ::= \text{skippable event} \\
\text{Mod} & ::= \text{action}
\end{align*}
\]

An interface consists in a list of abstract actions (Action). Actions denoted with the keyword event are actions
the aspect component observes in the base program. Actions denoted with the keyword skippable event are actions
the aspect component can make the base program skip. Actions denoted with the keyword action are actions
the aspect component requires to implement its advices.

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

\[
\begin{align*}
\text{BehaviorDef} & ::= \text{ProcDef} ( , \text{ProcDef} )^* . \\
\text{ProcDef} & ::= \text{ProcId} = \text{Body} \\
\text{Body} & ::= ( \text{Prefix} ( | \text{Prefix} )^* ) \\
\text{Prefix} & ::= ( \text{ActLabel} | \text{AspLabel} ) \rightarrow \text{ProcId} \\
\text{ActLabel} & ::= \text{Label} ( \text{Params} ) \\
\text{AspLabel} & ::= \text{ActLabel} > \text{Before PS After} \\
\text{Before} & ::= ( \text{ActLabel } ; )^* \\
\text{After} & ::= ( ; \text{ActLabel} )^* \\
\text{PS} & ::= \text{skip } | \text{proceed}
\end{align*}
\]

The protocol is declared as an automaton such that the action labels (ActLabel) correspond to actions declared in
the interface of the aspect. A normal transition is a transition labeled with an action label that is triggered by an
action that occurs in the base program. The protocol can
also include aspect transitions, which are transitions labeled with aspect labels (AspLabel). An aspect transition denotes the fact that for a given action in the base program the aspect may execute advices and can make the base program skip or proceed the action. For the time being, we do not explain the semantics of parameters (Params) of actions.

B. Connectors

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

\[
\text{Connector} \quad ::= \quad \text{connector} \{ \text{Connection}^* \}
\]

\[
\text{Connection} \quad ::= \quad \text{connects} \ \text{Action} \ \text{to} \ \text{Pattern} \ ;
\]

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 generate a system with plain components. This is done by transforming the aspect component into a \textit{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.

A.1 Generation of the protocol

The protocol of the PAC is the result of transforming the protocol of the aspect component. The aspect component observes actions of an event-based application, this is implemented in the PAC has 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 applying the following transformations:

\[
T("\text{label}(\text{params}) \to \text{P}^n") =
"\text{eventB\_label}(\text{params}) \to \text{eventE\_label}() \to \text{P}^n"
\]

\[
T("\text{label}(\text{params}) \to \text{before} \ ; \ \text{ps} ; \ \text{after} \to \text{P}^n") =
"\text{eventB\_label}(\text{params}) \to \text{before} \to
\text{psE\_label}() \to \text{psE\_label}() \to \text{after} \to
\text{eventE\_label}() \to \text{P}^n"
\]

The first transformation describes that a transition when a base-program action \text{label}(\text{params}) occurs, is programmed as the reception of an event \text{eventB\_label}(\text{params}) indicating the action is about to be executed, and an event \text{eventE\_label}() indicating the action has been executed.

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

1. The PAC receives the event \text{eventB\_label}(\text{params}) from a base component when the action is about to be executed.
2. Then, it executes the sequence of actions denoted by \text{before} and emits either the event \text{skipB\_label}() or the event \text{proceedB\_label}() to indicate to the base component if the action have to be skip or proceeded.
3. The base component receives the last event, skip or proceed the action, and emits either the event \text{skipB\_label}() or the event \text{proceedB\_label}() indicating that the action has just been skip or proceeded, respectively.
4. The PAC receives the last event, executes the sequence of actions denoted by \text{after} and emits the event \text{eventE\_label}() to indicate to the base component that it can continue its computation.
5. The base component receives this event and continues.

A.2 Generation of the interfaces

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

An action declared as \text{event \ label}(\text{params}) in the aspect-component interface is used in normal transitions of the aspect-component protocol. It generates the following interface in the PAC:

\[
\text{interface sync\_label} \\
\text{provides void eventB\_label}(\text{params}); \\
\text{provides void eventE\_label}(); \\
\}
\]

In an analogous way, an action declared as \text{skippable \ event \ label}(\text{params}) in the aspect-component interface is used in aspect transitions of the aspect-component protocol. It generates the following interface in the PAC:

\[
\text{interface sync\_label} \\
\text{provides void eventB\_label}(\text{params}); \\
\text{requires void eventE\_label}(); \\
\text{requires void proceedB\_label}(); \\
\text{provides void proceedE\_label}(); \\
\text{requires void skipB\_label}(); \\
\text{provides void skipE\_label}(); \\
\}
\]

Finally, an action declared \text{action \ label}(\text{params}) in the aspect-component interface generates the following inter-
face in the PAC:

```java
interface a_label {
  requires void label(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. We use a Baton connector to determine whether it is possible or not to connect a base component to the PAC.

A Baton connector matches services and internal actions declared in the interface of a base component. If a service or internal action \( s() \) 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 \( \text{event label}(\text{params}) \) or \( \text{skippable event label}(\text{params}) \), then there is an interface called \( \text{sync}_n \) in the boundaries of the PAC. A complementary interface 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() \rightarrow P\) = "eventB_label() \rightarrow s() \rightarrow \text{eventE_label()} \rightarrow P"
\]

\[
T("s() \rightarrow P\) = "eventB_label() \rightarrow \text{proceedB_label()} \rightarrow s() \rightarrow \text{proceedE_label()} \rightarrow P
| eventB_label() \rightarrow \text{skipB_label()} \rightarrow
| \text{skipE_label()} \rightarrow \text{eventE_label()} \rightarrow P^n"
\]

The first transformation applies if the abstract action \( \text{label}(\text{params}) \) has been declared \( \text{event} \). Then the component has to generate one event before the execution of the concrete action and other event after. The second transformation applies if the abstract action has been declared \( \text{skippable event} \). Then the component has to generate events that introduce the possibility of skip the action (more details in Section V).

If the abstract action has been declared as \( \text{action label}(\text{params}) \), then there is an interface \( \text{a_label}(\text{params}) \) that is connected to the interface of \( s() \).

We have introduced a language for programming aspect components and its implementation as a plain component. Afterwards, this section 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 web-site 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.

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 \( \text{Consistency} \), to solve this problem. The aspect component programmed in Baton is as follows:

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

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

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

Now consider the application has been programmed in the component-based system of the following figure.
The application is formed by five components: client represent a client, admin an administrator, auth an authorization entity, catalog the on-line catalog and logger the component that implements the advice of the aspect. The interfaces client.i2 and catalog.i1 declare a service getItems() and a service pay(). The interfaces client.i1 and auth.i1 declare a service login(String user,String pass). The interfaces admin.i1 and catalog.i2 declare a service addItem(Item item).

In order to weave the Consistency aspect, we define the following Baton connector:

\[
\text{connector Connector} = \\
\begin{align*}
\text{connects login() to } & \ast.\ast.\text{login(..);} \\
\text{connects checkout() to } & \ast.\ast.\text{pay(..);} \\
\text{connects update() to } & \ast.\ast.\text{addItem(..);} \\
\text{connects log() to } & \ast.\ast.\text{log();}
\end{align*}
\]

The application of this aspect generates a PAC that is connected to the corresponding components of the system. The following figure illustrates the PAC with the corresponding interfaces (in the figure we have hidden the component catalog).

Finally, the code below shows the protocol of the resulting PAC:

\[
\begin{align*}
\text{OutSession} &= \\
( & \text{eventB_login()} \rightarrow \text{eventE_login()} \rightarrow \\
& \text{InSession},) \\
\text{InSession} &= \\
( & \text{eventB_update()} \rightarrow \text{skipB_update()} \rightarrow \\
& \text{skipE_update()} \rightarrow \text{log()} \rightarrow \text{eventE_update()} \rightarrow \\
& \text{InSession} \\
| & \text{eventB_checkout()} \rightarrow \text{eventE_checkout()} \rightarrow \\
& \text{OutSession} \\
).
\end{align*}
\]

VII. Related Work

The work on open modules [1] 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 [8] aims at achieving a symmetric, unified component architecture that treats aspects and components as uniform entities. Then, it addresses the problem of properly configure 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 component that depend on a global shared state. Baton permits to program these 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 aspect components and plain components can produce a system with only plain components. As future work, we plan to extend these ideas to a more realistic component model including, for instance, multi-ary communication and integrate support for distributed aspects in the line of AWED [4].

References