Baton

A Domain-Specific Language for Coordinating Concurrent Aspects in Java

Angel Núñez — Jacques Noyé

Project OBASCO, EMN-INRIA, LINA
4, rue Alfred Kastler
44307 Nantes cedex 3, France
{Angel.Nunez, Jacques.Noye}@emn.fr

ABSTRACT. *Aspect-Oriented Programming (AOP) promises the modularisation of so-called cross-cutting functionality in large applications. Currently, almost all approaches to AOP provide means for the description of sequential aspects that are to be applied to a sequential base program. A recent approach, Concurrent Event-based AOP (CEAOP), has been introduced, which models the concurrent application of aspects to concurrent base programs. CEAOP uses Finite State Processes (FSP) and their representation as Labeled Transition Systems (LTS) for modeling aspects, base programs and their concurrent composition, thus enabling the use of the Labeled Transition System Analyzer (LTSA) for formal property verification. The initial work on CEAOP does not provide an implementation of its concepts, restricting the study of concurrent aspects to the study of a model. The contribution of this paper is the provision of an implementation of CEAOP as a small DSAL (Domain-Specific Aspect Language), Baton, which is very close to FSP, and can be compiled into Java. As an intermediate layer, we have developed a Java library which makes it possible to associate a Java implementation to a finite state process. The compilation process consists of translating both the Baton aspects and the Java base program into Java finite state processes. This translation relies on Metaborg/SDF to extend Java with Baton and Reflex to instrument the base program.*

RÉSUMÉ. *La programmation par aspects (AOP) promet une meilleure modularité des préoccupations fonctionnelles dispersées dans les applications de grande taille. Aujourd'hui, presque toutes les approches de la programmation par aspects fournissent les moyens nécessaires à la description d'aspects séquentiels appliqués à des programmes de base qui sont aussi séquentiels. Une approche récente, Concurrent Event-based AOP (CEAOP), permet la modélisation d'aspects concurrents. CEAOP utilise des notions issues des* Finite State Processes *(FSP) et de leur représentation sous la forme de systèmes de transitions étiquetées pour la définition des aspects et des programmes de base. Il est alors possible d'utiliser l'outil LTSA à des fins de vérification. Les travaux initiaux sur CEAOP ne fournissent pas d'implémentation de ses concepts, limitant l'étude des aspects concurrents à une analyse des systèmes de transitions. Le contribution de ce papier est la production d'une implémentation de CEAOP sous la forme*

d'un langage dédié aux aspects concurrents. Ce langage, appelé Baton, est très proche de FSP, et peut être compilé en Java en utilisant une bibliothèque Java qui rend possible l'association d'une implémentation Java à un processus à états finis. La procédure de compilation consiste à traduire les aspects écrits en Baton et le programme de base écrit en Java en des processus à états finis implémentés en Java. Cette traduction est réalisée grâce à l'outil Metaborg/SDF, qui permet la combinaison de Java et des constructions propres à Baton, et à Reflex, qui réalise l'instrumentation du programme de base.

KEYWORDS: *AOP, Concurrency, FSP, LTS, Java, Reflex*

MOTS-CLÉS : *AOP, Concurrence, FSP, LTS, Java, Reflex*

1. Introduction

A recent approach, Concurrent Event-based AOP (CEAOP) [DOU 06b, DOU 06a], has been introduced, which models the concurrent application of aspects to concurrent base programs. CEAOP uses Finite State Processes (FSP) and their representation as Labeled Transition Systems (LTS) for modeling aspects, base programs and their concurrent composition. The use of FSP provides a straightforward model for the composition of concurrent processes, and enables the use of the tool Labeled Transition System Analyser (LTSA) [MAG 06] for formal property verification.

CEAOP translates aspects and base program into FSPs by applying proper transformation rules. Those FSPs are composed in terms of FSP composition, which is customized by the use of high-level operators defined in the model. These operators introduce more or less synchronization when required. This schema is interesting since it permits aspects to execute their advices (modeled as FSP actions), in coordination with the base program using the CEAOP semantics, without requiring the need of a different mechanism (aspect-oriented or otherwise) whose only purpose is to introduce synchronization-related code.

CEAOP has been proposed and developed as a model. Not much has been done in terms of a concrete implementation. The importance of an implementation is that it would permit to test and experiment with concrete scenarios, to study the real applicability of the concepts introduced. This paper is a step towards this aim by providing an implementation of CEAOP as a small Domain-Specific Aspect Language (DSAL) named Baton. We consider Baton as a DSAL because it is not a full-fledged Aspect Language. It concentrates on some issues only of Aspect-Oriented Programming such as stateful aspects and concurrency. Actually, we believe that, instead of building new AOP languages from scratch, it would be nice to build them by composing DSALs.

Baton is very close to FSP, and can be compiled into Java. The compilation process consists of translating both the Baton aspects and the Java base program into Java finite state processes (Java FSPs). This translation relies on Metaborg/SDF [BRA 05] to extend Java with Baton, and Reflex [TAN 05] to instrument the base program. The overall translation scheme of Baton is illustrated in Fig. 1. Baton aspects are translated

into Java FSPs. They are also modeled as CEAOP aspects, which can be translated into FSPs. Something analogous occurs with the base program. The most important property of the implementation is that the translation schemes at both levels are the same and that each individual Java FSP can be modeled as the corresponding FSP, so that the composition at the implementation level behaves as the composition of the models by construction.

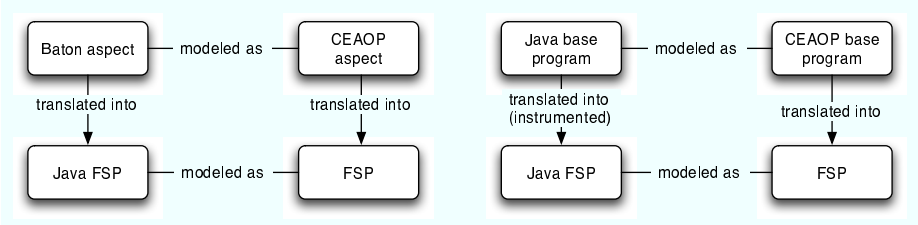

Figure 1. *Overall translation scheme of Baton: aspects and base program.*

As an intermediate layer, we have developed a Java library that makes it possible to compose Java FSPs in terms of FSP composition. Java FSPs, which are active objects, are coordinated using a centralized monitor.

This paper is structured as follows. Section 2 reviews the basic ideas behind CEAOP as a model. Section 3 is a first refinement of this model toward a language for concurrent aspects. Section 4 introduces Baton, our DSAL implementing CEAOP, which permits to define aspects that are translated into Java FSPs behaving as specified in the model. Section 5 describes how those Java FSPs work at runtime. Section 6 shows how aspects written in Baton as well as the base program written in plain Java are translated to Java FSPs. Section 7 is about related work and Section 8 concludes.

2. CEAOP overview

CEAOP is an approach for AOP that models the concurrent execution of aspects and the base program computation. CEAOP addresses the three major issues concerning the coordination of concurrent aspects: (i) aspects modularize functionalities that typically modify base executions at a large number of execution points, (ii) modifications ("advices") can be divided into pieces that can be coordinated differently with the base execution, and (iii) multiple advices may apply at an execution point, in which case different coordination strategies may be applied in the concurrent setting (in contrast to the standard "advice chaining" strategy used in sequential AOP).

In order to explain CEAOP we use an example [DOU 06a, DOU 06b] inspired by typical 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. We model this using the following sequential FSP:

```
Server    = ( login -> InSession  | update -> Server ),
InSession = ( checkout -> Server  | update -> InSession
            | browse -> InSession ).
```

A sequential FSP is defined by a series of (sub)process definitions (here `Server` and `InSession`), whose bodies are in turn defined using (atomic) actions (`login`, `update`, `checkout`, and `browse`), the sequence operator `->`, the choice operator `|`, and the names of the subprocesses being defined, which creates recursive definitions. The subprocesses directly correspond to significant states in the LTS associated to the FSP. We will use the subprocess names to denote these states.

Let us now consider the problem of canceling updates to the client-specific view of the e-commerce shop during sessions, *e.g.*, to ensure consistent pricing to the client. We can define a suitable aspect, which we call `Consistency`, to solve this problem. This aspect can be defined using a *pseudo-FSP*: an FSP in which some actions have been equipped with expressions of the form `>` *b ps a* to denote advices, where *ps* is one of the keywords `proceed` or `skip`, and *b*, *a* denote sequences of *advice actions* that are executed respectively before and after *ps*. At the level of pseudo-FSP, all the actions, except the advice actions, are interpreted as *events* emitted by the base program. `Consistency` can be defined as follows:

```
Consistency = ( login -> InSession ),
InSession   = ( update > skip, log -> InSession
              | checkout -> Consistency ).
```

This aspect initially starts in state `Consistency` and waits for a `login` event from the base program (other events are just ignored). When the `login` event 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` event (other events being ignored). If `update` occurs first, the associated advice `skip, log` causes the base program to skip the `update` action and the aspect performs the `log` action. Then the base program resumes and the aspect returns to state `InSession`. If `checkout` occurs first, the aspect returns to state `Consistency` and the base program execution resumes. Since `update` events are ignored in state `Consistency`, updates occurring out of a session are performed, while those occurring within sessions (state `InSession`) are skipped.

Moreover, each time the website is updated (*i.e.*, the administrator publishes an internal working version), it is desirable that a second aspect rehashes a database of links before the publication, and backups the database afterward. The second aspect, called `Safety`, can be defined as follows:

```
Safety = ( update > rehash, proceed, backup  -> Safety ).
```

Both aspects interact through the event `update`. Plain EAOP [DOU 02, DOU 04] can deal with the composition of such aspects, but in a sequential setting. However, the concurrent execution of the aspects together with the base program may be desirable (*e.g.*, in the case of the rehashing and backup actions of the `Safety` aspect, which are rather time-consuming operations). CEAOP explains the semantics of weaving the aspects into the base program in two steps: 1) each aspect and the base program are translated into FSPs; 2) the concurrent behavior of the aspects applied to the base program is modeled as the parallel composition of those FSPs.

2.1. *Translation*

The aim of the translation is to express both the base program and the aspects as appropriate FSPs. The informal translation of a pseudo-FSP into an FSP is done in two steps. First, for each pseudo-FSP subprocess we include *waiting loops* on the events of the pseudo-FSP not included in the subprocess. For instance, the events dealt with by the `Consistency` aspect are {`login`, `update`, and `checkout`}. Therefore, we include waiting loops on both `update` and `checkout` in the `Consistency` subprocess and on `login` in the `InSession` subprocess. Second, we introduce synchronization events that are used to coordinate the aspects and the base program. Aspect expressions *e* `>` *b ps a* are translated into `(eventB_`*e* `->` *b* `->` *ps*`B_`*e* `->` *ps*`E_`*e* `->` *a* `->` `eventE_`*e*`)`. An event *e*, which can trigger an advice, is called a *skippable* action (the advice may decide to skip the action or not). Each skippable action that still appears in the pseudo-FSP (*e.g.*, as a waiting loop) is translated into `(eventB_`*e* `-> proceedB_`*e* `-> proceedE_`*e* `-> eventE_`*e*`)`. The following FSP is the result of the translation of the `Consistency` aspect of our example:

```
Consistency = ( login -> InSession | checkout -> Consistency
              | eventB_update -> proceedB_update -> proceedE_update
                  -> eventE_update -> Consistency ),
InSession   = ( login -> InSession | checkout -> Consistency
              | eventB_update -> skipB_update -> skipE_update -> log
                  -> eventE_update -> InSession ).
```

In an analogous way, the FSP for the Safety aspect is as follows:

```
Safety = ( eventB_update -> rehash -> proceedB_update
             -> proceedE_update-> backup  -> eventE_update -> Safety ).
```

In addition, the base program is directly modeled by an FSP (*e.g.*, the FSP of the e-commerce application shown previously). But this is not quite the FSP that we can use for composition. Indeed, in order to allow control of the skippable actions by the aspects, this original FSP has to be “instrumented”. This is implemented by replacing each sequence expression starting with a skippable action *e* `->` *P* by the expression:

```
eventB_e -> (proceedB_e -> e -> proceedE_e -> eventE_e -> P
           | skipB_e -> skipE_e -> eventE_e -> P )
```

2.2. *Composition*

Once the aspects and the base program have been translated to FSP, their composition is based on parallel composition, a form of synchronized product, where interactions are modeled by shared actions. When an action is shared among several processes, the shared action must be executed at the same time by these processes. As an example, Fig. 2 shows the output of the parallel composition of the Consistency aspect and the base program. The left-hand side cycle performs `update`s outside of sessions. The right-hand side cycle skips `update` commands during sessions and does some logging. The middle cycle starts and ends sessions.

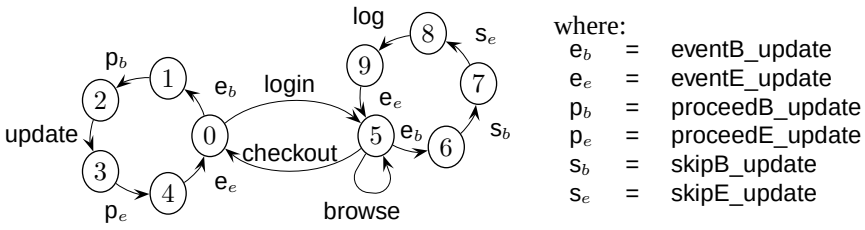

Figure 2. *Woven example*

Composition operators can be designed to compose the aspects in different ways. For instance, let us consider the `ParAnd` operator. The semantics of this binary operator is the following. When two advices can be applied at the same joinpoint, their before action sequences are executed in parallel, but there is a rendez-vous on `proceed` and `skip`. If both of them wish to proceed, they will proceed in parallel. If (at least) one of them wishes to skip, both will skip in parallel. In our example, `ParAnd(Consistency,Safety)` composes both advices during sessions to get, using informal syntax, `backup skip (log || rehash)`, which ensures that all database management actions are performed, if reasonable, in parallel. This composition is modeled in FSP by renaming some synchronization events in the aspect definitions and by defining a process `ParAnd` that dynamically renames `skip` and `proceed` messages. Both aspects share the events `eventB_`*e* and `eventE_`*e* so that the beginning and the end of advices are synchronized. Before (and after) `skip` or `proceed`, advices of the aspects are executed in parallel. The woven program is represented by the automaton of Fig. 3.

It makes clear that the advices are executed concurrently: both sequences `log`→`backup` and `backup`→`log` are valid. Furthermore, the user can still browse in parallel with the advice. More concurrency can be introduced by hiding the event `eventE_`*e* before the parallel composition.

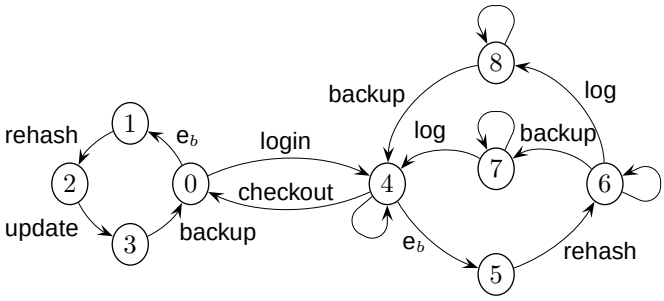

Figure 3. *Woven program using* `ParAnd` *(unlabeled loops are* `browse` *events)*

Other operators can be defined similarly. For instance, the advices composed with `ParOr` proceed when at least one of them proceeds.

This concludes our presentation of the CEAOP model. More details are available in [DOU 06b, DOU 06a], using instead of pseudo-FSP an equivalent but different way of expressing aspects. We have chosen here a syntax close to FSP in order to simplify the presentation. The following section introduces Baton, our implementation of this model.

3. Toward a language for concurrent aspects

At the model level, pseudo-FSP syntax provides a good language for modeling concurrent aspects on top of CEAOP. Aspects can be written using a syntax very close to FSP that makes it easy to understand the underlying concepts. However, this syntax lacks expressiveness. For instance, information captured on events cannot be passed to advices. We improve the expressiveness of pseudo-FSP syntax by allowing subprocesses and actions to declare parameters. The use of parameters permits to model more realistic applications where actions may interchange information. As an illustration, the code below is a version of the pseudo-FSP of the `Consistency` aspect that uses parameters:

```
Consistency       = ( login(client) -> InSession(client)),
InSession(client) = ( update(admin) > skip, log(client,admin)
                        -> InSession(client)
                    | checkout(client) -> Consistency).
```

In this example, the `Consistency` aspect uses the variables `client` and `admin`, which represent a client that logs in and an administrator doing an update, respectively. Parameters of events represent information that is bound or input, whereas parameters of advice actions represent information that has been previously bound and that is now output.

Parameters are not relevant for modeling concurrency. However, we use the parameters to define the information passing policy when coordinating shared actions: 1) The parameters of a skippable action *e* of a pseudo-FSP are associated to the respective synchronization events `eventB_`*e* of the corresponding aspect FSP and base program FSP. The event is classified as a receiving action for the aspect and as a sending action for the base program. 2) The parameters associated to the non-skippable actions of a pseudo-FSP are associated to the respective actions of the corresponding aspect FSP and base program FSP. These actions are classified as receiving actions for the aspect and as sending actions for the base program. 3) The parameters associated to advice actions of a pseudo-FSP are associated to the respective actions of the corresponding aspect FSP, which are classified as sending actions. 4) In the coordination of a shared action, the sending instance of the shared action provides the receiving instances with values for the associated parameters.

Adding parameters and the notion of sending and receiving actions makes the model syntactically closer to the implementation without changing its semantics, except that we only consider models with a single sending action instance per shared action. In particular, the semantics of the composition is not changed even if it can now be interpreted as resulting in parameter passing. In the following section we present our DSAL, which uses the pseudo-FSP syntax with parameters presented in this section.

4. Baton

Baton is a small DSAL that provides us with a first implementation of CEAOP. The architecture of a program written using Baton is based on three concepts: *aspects*, *connectors*, and *base objects* (see Fig. 4). Aspects and base objects are considered to be *components*, whose behavior is described through finite state processes. By components, we mean that the corresponding Baton and Java classes can be compiled independently and reused as black boxes. Using these components consists of first instantiating them and second connecting them through *connectors*. Aspects and connectors are defined using the syntax provided by Baton. The base program is defined in plain Java (source code or bytecode), more specifically as active objects.

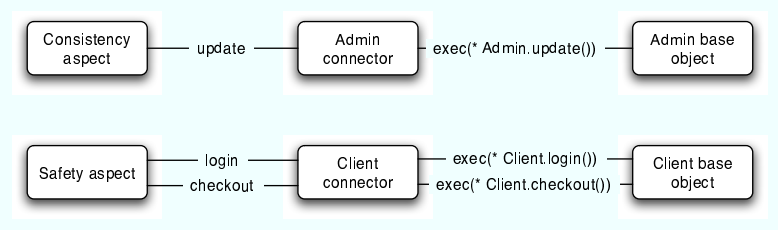

Figure 4. *Architecture of the e-commerce application written in Baton. Admin and Client are the classes representing the client and the administrator, respectively.*

The behavior of an aspect is described in pseudo-FSP (see Section 3), using a Java-like syntax. On top of its behavior, an aspect may also include method definitions, used to locally define actions, as well as instance variables. For instance, the `Consistency` aspect can be written using a local definition of the action `log` as follows:

```
aspect Consistency {
  public void log() {
    System.out.println("Update skipped.");
  }
  behaviour {
    Server    = ( login -> InSession ),
    InSession = ( update > skip,log -> InSession | checkout -> Server ).
  }
}
```

In the code above the action label `log` denotes the execution of the method `log`, which just writes the message "Update skipped" after skipping the action `update`. Baton allows us to improve this aspect by using parameters. The `Consistency` aspect of the code below improves the message displayed by the method `log` by receiving both the client and the administrator involved in the skipped `update` action.

```
aspect Consistency {
  public void log(Client client, Admin admin) {
    System.out.println(admin + " skipped:" + client + " is connected.");
  }
  behaviour {
    Server  = ( login(Client client) -> InSession(client) ),
    InSession(client) =
      ( update(Admin admin) > skip, log(client,admin)
          -> InSession(client)
      | checkout(client) -> Server ).
  }
}
```

Aspects defined using the Baton syntax are *primitive aspects*. More complex aspects can be defined by composing them using built-in operators (*e.g.*,`ParAnd` and `Fun`), resulting in *composite aspects*.

The base program consists of one or more *base objects*. These objects are *active*, *i.e.*, they are equipped with their own thread. The base objects can also be composed together via connectors to build up the base program. At the model level, each base program object is abstracted into an FSP. At the implementation level, we do not need the full FSPs in order to connect the base objects. We just need to know which are their shared actions in order to instrument them, using at the implementation level the ideas described at the model level. These shared actions are expressed as AspectJ pointcuts and are associated a possibly parameterized label in connectors. For instance, the code below:

```
connector ClientConnector{
  connect login : execution(* Client.login(..));
  connect checkout : execution(* Client.checkout(..));
}
```

A structure, namely `main`, represents a main program. It permits to instantiate several aspects, compose the aspects using operators, instantiate several base objects, and connect these objects with the aspects. For example, the code below composes an instance of `Consistency` and an instance of `Safety` using the `ParAnd` operator. Afterwards, the instruction `Baton.connect` makes a first connection between the composite aspect to an object representing a client and a second connection to an object representing an administrator.

```
main Ecommerce{
  Aspect aspect = new ParAnd(new Consistency(), new Safety());
  Client client = new Client();
  Admin admin = new Admin();
  Connector clientCon = new ClientConnector();
  Connector adminCon = new AdminConnector();
  Baton.connect(aspect,clientCon,client);
  Baton.connect(aspect,adminCon,admin);
  Baton.start();
}
```

The compilation of the base objects and the aspects into Java FSPs, as well as their composition is explained in Section 6. But let us see first how Java FSPs work at runtime so that their composition produces a program whose model is the composition of their models.

5. A Java library for FSP composition

This section presents our Java implementation of finite state processes. We mainly discuss how Java finite state processes work and interact at runtime. We work with the graphical form of an FSP, *i.e.*, its associated LTS, whose correspondence with FSP is described in [MAG 06]. Section 5.1 explains the principles behind the composition of finite state processes. Section 5.2 describes the implementation of these processes and their coordination in Java. Section 5.3 studies the semantics of the coordination in Java.

5.1. *Principles of LTS composition using a centralized monitor*

LTS composition defines a single LTS describing the composition of several LTSs. This single LTS is used to coordinate the processes implementing the composed LTSs.

In a scheme without any coordination, each single process runs on its own, following its LTS description and taking its own decisions. The LTS describing the composition coordinates the single processes, by telling them which transitions to follow at each instant.

The semantics of LTS composition is presented in [MAG 06]. We can explain the same ideas in a way more useful for the purposes of this paper. Let us consider n LTSs L_i (with $1 \leq i \leq n$), with their shared alphabets αL_i and their sets of transitions Δ_i. A state s_c of the composition corresponds to a state tuple $(s_1, s_2, ..., s_n)$, where s_i is a state of L_i. The transition $((s_1, s_2, ..., s_n), a_j, (s'_1, s'_2, ..., s'_n))$ is a transition in the composition, if $\forall L_i$ such that $a_j \in \alpha Li$, $(s_i, a_j, s'_i) \in \Delta_i$. Then, $\forall i$ such that $a_j \notin \alpha L_i$, $s'_i = s_i$. The first state of the composition corresponds to the state tuple $(s^0_1, s^0_2, ..., s^0_n)$, where s^0_i corresponds to the initial state of L_i. In other words, in a state $(s_1, s_2, ..., s_n)$ of the composition, a_j corresponds to a transition if $\forall L_i$ such that $a_j \in \alpha L_i$, a_j is a transition in the state s_i. Then the semantics of the composition tells us that all these LTSs transit together through a_j.

In order to coordinate several processes in terms of LTS composition, we have designated a central entity, namely the *monitor*. It has the global view of the execution of the processes so that it conducts their coordination. Since each process cannot take the decision about which transition to follow at the right time on its own, the decision is taken by the monitor, based on the information obtained from the other processes. We can define the monitor as follows. Let us consider n processes P_i (with $1 \leq i \leq n$) behaving as their respective LTSs L_i, and m shared actions a_j (with $1 \leq j \leq m$). At a given instant, we say that P_i is in state s^w_i if it is *waiting* to pass to a next state and its LTS L_i is in a state s_i, or it is in state s^b_i if it is *busy* and its LTS L_i is in a state s_i. We define b_j a bound indicating the amount of i such that $a_j \in \alpha L_i$, and c_j a counter indicating the amount of i such that a_j is a transition from s_i and P_i is in state s^w_i. At each instant, c_j tells us the number of processes waiting to pass to a next state (possibly) using the action a_j. The monitor is an entity in charge of checking at each instant for all the actions a_j whether $c_j = b_j$. When in an arbitrary instant, for an action a_k, $c_k = b_k$, then the action a_k may be chosen as the action to proceed. If a_k is chosen, then $\forall i$, such that $a_k \in \alpha L_i$, P_i may transit to a next state using a_k. Moreover, all the P_i must transit together.

The election of an action a_j such that $c_j = b_j$ in a certain instant is correct with respect to LTS composition because in the respective state $(s_1, s_2, ..., s_n)$ of the LTS composition, the transition a_j effectively corresponds to a transition in the composition. This is because $\forall i$ such that $a_j \in \alpha L_i$, its process P_i is waiting to pass to the next state using a_j, and therefore a_j is a transition in the state s_i of L_i.

Figure 5 shows an example of the synchronization scheme. There are three LTSs P_1, P_2 and P_3 with the alphabets $\alpha P_1 = \{a_1, a_2\}$, $\alpha P_2 = \{a_2, a_3\}$ and $\alpha P_3 = \{a_2\}$. At a certain instant, P_1 and P_3 are waiting to pass to a next state, whereas P_2 is still busy. The monitor keeps the counters updated and checks that the action a_1 has reached its bound.

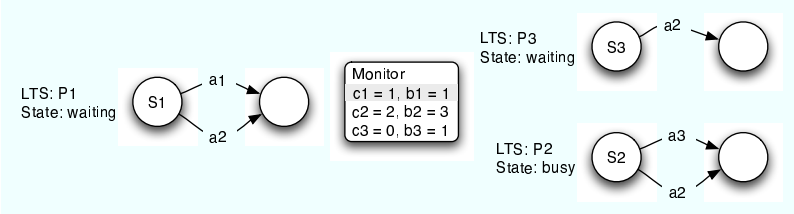

Figure 5. *Example of synchronization.*

5.2. *Implementation of the LTSs and the monitor*

The process that follows a given LTS is implemented as an object of the abstract class `LTS`. This object is supposed to be an active object mapping its current state to the state of its LTS and moving along its transitions. Furthermore, it provides the functionality to be coordinated with the rest of the system.

The monitor is designed as an object that keeps the variables c_j and b_j described before, and that is able to choose the actions followed by each process in the composition. It keeps watch over the execution of the processes and indicates the transition to follow at the right instant to the waiting processes. In order to calculate the bounds of each action, the initialization of the monitor requires the complete alphabet of each LTS participating in the composition.

The mechanism of synchronization can be explained as follows. When a process is in some state, ready to pass to a next one, it notifies the different actions available in the current state of its LTS to the monitor and waits for a decision. The monitor increments the counters c_j associated to the actions passed by the process, and checks whether some action has reached its bound b_j. If no action has reached its bound, it does nothing. Otherwise, if an action reaches its bound, the monitor chooses the action to proceed with. Then, it wakes up and notifies the decision to all the processes having the chosen action in their LTS alphabets. When it notifies the decision to a process, it also decrements the counters of the actions not chosen that were previously notified by the process. When the processes wake up and receive the decision, they transit to the next state. Then, some extra synchronization is necessary to ensure that these LTSs transit together.

5.3. *Semantics of the coordination*

Suppose the monitor has chosen an action *a*, and the processes P_1, P_2,..., P_m are ready to pass to their next states using this action. The LTS semantics says that the action *a* has to be executed at the same time by all these LTSs. In order to give a general meaning to this definition, the action *a* can be refined into the *subactions* a_1, a_2,..., a_m, where a_i corresponds to the action *a* in P_i. The execution (or interpretation)

of the action a corresponds to the execution (or interpretation) of each sub-action a_i by the process P_i. The action *a* is executed at the same time by all P_i if their subactions a_i are executed together, *i.e.*, any possible interleaving $b_1 \rightarrow b_2 \rightarrow ... \rightarrow b_m$ is allowed such that $\forall\ b_i\ \exists\ a_i$ such that $b_i = a_i$ (no action is executed in the middle of two subactions). A *synchronization on the entry* of a makes all P_i meet at the same point. From this point they begin the execution of their subactions a_i, generating some interleaving. A *synchronization on the exit* imposes that all processes meet at the end of the execution of their subactions. This impedes that some P_i executes some other action c (such that $\forall\ a_i,\ a_i \neq\ c_i$) when it has finished the execution of its subaction but some process has not already finished. In this way, an interleaving $b_1 \rightarrow b_2 \rightarrow ... \rightarrow c \rightarrow ... \rightarrow b_m$ is avoided.

The synchronization on entry is implemented as the notification that each process does when it is ready to pass to a next state. Then, each process passes to the monitor the different actions that it could follow. In an ideal scenario, if the next action to be chosen is known a priori, all the processes having this action in their alphabets should wait in a common *waiting set*. However, the action to be chosen is not known a priori, so it is not possible to group the processes by any criterion. As a result, we chose to make the processes wait in their own waiting set. Afterwards, the monitor has to wake up these processes one by one. Finally, the synchronization on exit is implemented as a notification to the monitor that every process has to perform after a shared action. Following this notification, the process waits in a common waiting set until the last process exits, which can be detected thanks to the notifications.

6. Aspect compilation and base instrumentation

This section describes the Java finite state processes library, which implements finite state processes and their coordination in terms of FSP composition.

Java FSPs define an abstract class, namely `LTS`, representing an LTS process. An `LTS` object is an active object executing LTS actions and switching from one LTS state to the other. This object is equipped with the mechanisms previously described, so that it can be properly coordinated with other `LTS` objects. An aspect is implemented as an instance of the class `InterpretedLTS`, which is a subclass of `LTS`. An `InterpretedLTS` object is equipped with the definition of the LTS that describes the aspect behavior. This object interprets each transition label as the execution of an instance method named with the same label. In order for a base program object to behave as an LTS, we instrument the code of the object. The methods of interest are intercepted, so that their execution is delegated to an instance of an `ObservedLTS` class. This class is another specialization of `LTS` that represents a slice of the process of a base program object. The instrumentation is done using Reflex [TAN 05].

Operators are defined by implementing the interface `Operator`. Our library provides an implementation of some basic operators such as `ParAnd` and `Fun`. These

operators work by relabeling the respective `InterpretedLTS`s and by possibly creating other `LTS` objects.

Baton syntax is translated into Java code in a process of *assimilation* using the tool Metaborg/SDF [BRA 05]. The assimilation consists of the following steps: (1) The assimilation of a Baton aspect generates a subclass of `InterpretedLTS` with the LTS describing the aspect behavior and with the instance variables and Java methods defined in the Baton aspect. (2) The assimilation of a Baton connector generates a Java class providing a method to obtain part of the Reflex configuration used for instrumenting the base program. (3) The assimilation of a Baton main permits to obtain the final Java FSPs. The instantiation of a Baton aspect, a Baton connector and an operator are translated into the instantiation of the respective classes, which were generated in the previous steps (except the class of the connector that is a built-in class). The connection between a Baton aspect and a base object using a Baton connector creates an instance of `ObservedLTS` for the base objects and generates the Reflex configuration that carries out a connection between the base object and the `ObservedLTS` object. Finally, the execution of the resulting `LTS` objects starts the FSP coordination at runtime explained in the previous section.

7. Related Work

EAOP is a general framework for AOP, in which aspects are defined in terms of *events* emitted during program execution and *crosscuts* relate sequences of events. Its formal model [DOU 02, DOU 04] introduces *stateful aspects* and makes it possible to automatically deduce possible malicious interactions between aspects. CEAOP is a stateful aspect model that extends EAOP in the concurrent case. Therefore, Baton behaves, in the sequential case, as the EAOP prototypes. JAsCo [SUV 03, VAN 05] is an implementations of EAOP. It introduces the concepts of *aspect beans*, which define reusable abstract aspects, and *connectors*, which makes it possible to dynamically deploy aspect beans in the context of concrete components. Baton is based on the same basic notions of aspects and connectors. It does not provide dynamic deployment but unlike JasCo provides explicit support for concurrency.

In the area of Component-Oriented Programming, SYNTHESIS [TIV 06] introduces a tool for the automatic correct assembly of components. Components, defining their individual behaviors through an LTS, are correctly connected together in order to avoid deadlocks. The assembly is performed by a central adapter which acts as glue code between the components. The implementation of Baton is related to this schema. Aspects and base program objects in Baton are designed as components using LTSs to define their behavior. We also detected the necessity of a central entity. However, SYNTHESIS considers the case of one-to-one connections between clients and servers, whereas in our case several aspects can be coordinated with a base program object through a shared action.

ProActive [BAD 06] implements in Java a parallel and distributed conceptual programming model for the GRID. The basic model of ProActive deals with active objects. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are asynchronous with transparent future objects and synchronization is handled by a mechanism known as *wait-by-necessity*. We considered using ProActive to implement Baton. Unfortunately, ProActive does not provide a mechanism to synchronize active objects on common actions, which is the main requirement of Baton. The nearest feature of ProActive is *group communications* [BAD 02], which allows several active objects to serve a request in parallel.

8. Conclusions

This paper has presented a preliminary implementation of CEAOP as a small DSAL, Baton. This experiment is a first step towards a concrete language that would permit to test and experiment with concrete scenarios, and to study the real applicability of the concepts introduced by CEAOP. This implementation is based on an implementation-level version of Finite State Processes: active objects coordinated by a centralized monitor.

This direct mapping between FSPs at the model level and Java FSPs at the implementation level has a number of advantages. In particular, it makes it easier to guarantee the correctness of the implementation. It also makes it possible to provide a quite expressive language with a form of group communication. It is however bound not to be satisfactory when considering distributed systems, which we contemplate as future work. One possibility may be to distribute the monitor along the lines of [AUT 06], possibly restricting action sharing to binary sharing (that is, one sender, one receiver). In order to deal with more realistic applications, we also need to improve the advice language, in particular by allowing to conditionally decide whether a base action should be skipped or not.

Finally, we are interested in investigating a stronger integration between aspect-oriented and component-oriented programming as it seems that CEAOP makes this integration quite natural.

Acknowledgments

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).

9. References

[AUT 06] AUTILI M., FLAMMINI M., INVERARDI P., NAVARRA A., TIVOLI M., "Synthesis of Concurrent and Distributed Adaptors for Component-Based Systems.", GRUHN V., OQUENDO F., Eds., *European Workshop on Software Architecture (EWSA 2006)*, vol. 4344 of *lncs*, Nantes, France, 2006, Springer, p. 17-32.

[BAD 02] BADUEL L., BAUDE F., CAROMEL D., "Efficient, Flexible, and Typed Group Communications in Java", *Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, 2002, ACM Press, p. 28–36.

[BAD 06] BADUEL L., BAUDE F., CAROMEL D., CONTES A., HUET F., MOREL M., QUILICI R., "*Grid Computing: Software Environments and Tools*", chapter Programming, Deploying, Composing, for the Grid, Springer-Verlag, January 2006.

[BRA 05] BRAVENBOER M., DE GROOT R., VISSER E., "Metaborg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT", *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, vol. 4143 of *lncs*, Braga, Portugal, Jul. 2005, sv.

[DOU 02] DOUENCE R., FRADET P., SÜDHOLT M., "A Framework for the Detection and Resolution of Aspect Interactions", *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, vol. 2487 of *Lecture Notes in Computer Science*, Pittsburgh, PA, USA, Oct. 2002, Springer-Verlag, p. 173–188.

[DOU 04] DOUENCE R., FRADET P., SÜDHOLT M., "Composition, reuse and interaction analysis of stateful aspects", *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, Lancaster, UK, March 2004, ACM Press, p. 141–150.

[DOU 06a] DOUENCE R., LE BOTLAN D., NOYÉ J., SÜDHOLT M., "Concurrent Aspects", *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'06)*, Portland, USA, Oct. 2006, ACM Press, p. 79-88.

[DOU 06b] DOUENCE R., LE BOTLAN D., NOYÉ J., SÜDHOLT M., "Towards a model of concurrent AOP", SPLAT'06, March 2006.

[MAG 06] MAGEE J., KRAMER J., *Concurrency: State Models and Java Programming*, Wiley, 2nd edition, 2006.

[SUV 03] SUVEE D., VANDERPERREN W., JONCKERS V., "JAsCo: An Aspect-Oriented approach tailored for Component Based Software Development", *Proceedings of the 2nd international Conference on Aspect-Oriented Software Development*, Boston, Massachusetts,USA, March 2003, ACM Press, p. 21–29.

[TAN 05] TANTER É., NOYÉ J., "A Versatile Kernel for Multi-Language AOP", *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, vol. 3676 of *Lecture Notes in Computer Science*, Tallin, Estonia, Sep. 2005, Springer-Verlag, p. 173-188.

[TIV 06] TIVOLI M., AUTILI M., "SYNTHESIS, a Tool for Synthesizing Correct and Protocol-Enhanced Adaptors", *RSTI L'Objet journal*, vol. 12, num. 1, 2006, p. 77-103.

[VAN 05] VANDERPERREN W., SUVÉE D., CIBRÁN M. A., FRAINE B. D., "Stateful Aspects in JAsCo.", GSCHWIND T., ASSMANN U., NIERSTRASZ O., Eds., *Software Composition*, vol. 3628 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005, p. 167-181.