INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Towards a Concurrent Model of Event-based Aspect-Oriented Programming

Rémi Douence — Jacques Noyé

N° 5838

Février 2006

Thème COM

Rapport de recherche

ISRN INRIA/RR--5838--FR+ENG

N 0249-6399

Towards a Concurrent Model of Event-based Aspect-Oriented Programming

Rémi Douence* , Jacques Noyé*

Thème COM — Systèmes communicants
Projet Obasco

Rapport de recherche n° 5838 — Février 2006 — 11 pages

Abstract: The Event-based Aspect-Oriented Programming model (EAOP) makes it possible to define pointcuts in terms of sequences of events emitted by the base program. The current formalization of the model relies on a monolithic entity, the monitor, which observes the execution of the base program and executes the actions associated to the matching pointcut. This model is not intrinsically sequential but its current formalization favors a sequential point of view. In this paper, we present a new formalization of EAOP as finite state processes. This new formalization paves the way to reasoning about aspects in a concurrent setting and to the definition and implementation of concurrent EAOP languages.

Key-words: programming languages, separation of concerns, event-based aspect-oriented programming, concurent programming.

* École des Mines de Nantes

Vers un modèle concurrent de la programmation par aspects événementiels

Résumé : La programmation par aspects événementiels (EAOP) permet la définition de coupes en termes de séquences d'événements émises par le programme de base. La formalisation actuelle du modèle repose sur une entité monolithique, le moniteur, qui observe l'exécution du programme de base et exécute les actions associées aux points de coupes détectés. Ce modèle n'est pas intrinsèquement séquentiel, mais sa formalisation actuelle favorise un point de vue séquentiel. Dans cet article, nous présentons une nouvelle formalisation d'EAOP à l'aide de processus à états finis. Cette nouvelle formalisation ouvre la voie à l'étude des aspects dans un cadre concurrent, ainsi qu'à la définition et la mise en oeuvre de langages EAOP concurrents.

Mots-clés : langages de programmation, séparation des préoccupations, programmation par aspects événementiels, programmation concurrente.

Contents

1 Introduction

The *Event-based Aspect-Oriented Programming* (EAOP) model was initially presented in [7] and further refined in [4, 5]. In this model, independent from any specific programming language, the points of interest of a (base) program execution (or *join points* in "standard" AOP terminology) are abstracted as *events* emitted during program execution. The key feature of the model is that an aspect can associate an action (actually a series of actions) to a *sequence of events*, rather than to an individual event. This makes it possible to build very expressive aspect languages that can talk about the history of the base program execution beyond the ad hoc `cflow` pointcut descriptors of AspectJ.

Although the use of the term "event" may lead to the thought that the model is well-adapted to the concurrent and distributed world, this is not quite the case. Indeed, the matching of event patterns is described in terms of a monolithic monitor, which has a global view of the program execution (in [7], EAOP is referred to as *Monitor-based AOP*). The monitor receives the events emitted by the base program and executes the aspects. This includes matching but also executing the associated actions. Here, concurrency is explicitly restricted: the base program cannot proceed when the monitor is active, although they are cases where this restriction is clearly not necessary (let us consider a simple aspect tracing the events of the base program).

On the one hand, all this makes it difficult to combine EAOP and concurrency, at the model level, and a fortiori at the implementation level. As a result, all the EAOP implementations [8, 6] so far have been sequential. This is unfortunate as many interesting applications of AOP are concurrent. Actually they are even distributed, *e.g.*, distributed components and web services. In this context, it is interesting to be able to consider the base program as a collection of concurrent *components*, where each component is described by its static and dynamic interfaces, i.e., the signatures of its required and provided services, together with a protocol describing, in terms of service requests and replies, the interaction of the component with its environment (see, for instance, [13, 9, 12]).

On the other hand, there are many similarities between the EAOP model and an asynchronous concurrency model. Basically, there are both interested in execution traces. Actually the modeling of the execution of the base program as a sequence of events directly leads to the representation of this program as a transition system. Also, aspect composition and pattern composition are expressed using operators (sequence, choice, parallel composition) which look very much like concurrency operators.

This strongly suggests to consider recasting the EAOP model in terms of an asynchronous concurrency model. This would have a number of benefits:

- Such a model, defined using a well-known formalism, would be easier to grasp and extend, for instance, relaxing some of the synchronization constraints between the base program and the aspects could be considered.
- It would be possible to apply a variety of standard tools and techniques (animation, model-checking...) to EAOP models.
- The model would give guidelines for building concurrent (and distributed) implementations of the model.

This paper describes our first attempt to do so. It considers the basic EAOP model as described in [4] and shows how the base programs, the aspects, and the monitor can be encoded in a simple process calculus. This process calculus is a subset of FSP (Finite State Processes) [10], which means that the associated tool, LTSA (Labeled Transition System Analyzer) can directly be used to animate the models, model-check them, and compute the corresponding labeled transition systems.

The paper is structured as follows. Section 2 describes our starting point, a basic "sequential" EAOP model. Section 3 shows how (base) programs and aspects expressed using this model can be encoded into finite state processes. Section 4 weakens the synchronization imposed by the previous encoding in order to get a fully concurrent version of EAOP. Section 5 discusses related work and section 6 concludes.

2 Sequential EAOP

The original version of EAOP considers sequential programs only. A sequential program can be modeled by its set of traces in the following syntax:

$$\begin{array}{rl} S ::= & \mu s.(e_1 \rightarrow S_1 | \ldots | e_n \rightarrow S_n) \\ | & s \\ | & \mathit{Stop} \end{array}$$

A sequential program is composed of sequences of execution events $e \rightarrow S$, branches $e_1 \rightarrow S_1 | e_2 \rightarrow S_2$, and loops $\mu s.(e \rightarrow S)$[1]. A sequence definition ends by jumping to the beginning of a loop (i.e., a label s of the recursion operator μ) or by halting (i.e., *Stop*).

For instance, let us consider a sequential program that modifies files (as specified by the execution event `write`), either within or outside of work sessions (delimited by `login` and `logout`). Such a program can be modeled as:

$$\begin{array}{l} \mu s_1.\ (\texttt{login} \rightarrow \mu s_2.(\texttt{write} \rightarrow s_2 | \texttt{logout} \rightarrow s_1) \\ \qquad |\texttt{write} \rightarrow s_1) \end{array}$$

Either there is a session that contains file modifications (first line), or there are file modifications outside of a session. These two patterns can be repeated and alternated an arbitrary number of times.

EAOP is a sequential model for aspect-oriented programming based on execution events. The base program execution emits events. Aspects monitor the base program execution and trigger advice execution when a given sequence of events is detected. EAOP provides a specialized language A to define aspects:

$$\begin{array}{rl} A ::= & \mu a.(e_1 \triangleright S_1; A_1 \Box \ldots \Box e_n \triangleright S_n; A_n) \\ | & a \\ | & \mathit{Stop} \end{array}$$

The structure of an aspect in A defines sequences of execution events to be matched, hence closely follows the structure of a sequential program in S, with recursion, sequences (operator ;), choices (operator $\Box$), and halting (operator *Stop*). A new feature is that an advice S_i, which is an arbitrarily complex sequential program, is associated to each event e_i, using the operator $\triangleright$. The intuitive meaning of $e_i \triangleright S_i$ is that if e_i matches, then the advice S_i should be performed.

Advices S_i are arbitrarily complex sequential programs that can occur at every step of the sequences. For instance, let us consider a backup aspect that commit file modifications in a versioning system but only within a session. This aspect can be defined as follows:

$$\begin{array}{l} \mu a_1.\ (\texttt{login} \triangleright \mathit{Stop}; \mu a_2.\ (\texttt{write} \triangleright \texttt{cvs\_ci} \rightarrow \mathit{Stop}; a_2 \\ \qquad\qquad\qquad\qquad\quad \Box \texttt{logout} \triangleright \mathit{Stop}; a_1)) \end{array}$$

First, the aspect waits for the beginning of a session (i.e., an event `login`). When this happens, the corresponding empty advice *Stop* is executed and the aspect now waits for either a file modification (i.e., an event `write`) or the end of the session (i.e., an event `logout`). When a file is modified, the corresponding advice, `cvs_ci` $\rightarrow$ *Stop*, versions the file, then the aspect waits for the next modification or the end of the session. When the session ends, the aspect waits for the next one.

This aspect versions modifications that only occur during a session. For instance, let us consider the following execution trace of a base program:

`login` $\rightarrow$ `write` $\rightarrow$ `write` $\rightarrow$ `logout` $\rightarrow$ `write` $\rightarrow$ *Stop*

[1] In the following, for the sake of readability, we simplify $\mu s.S$ to S when s does not occur free in S.

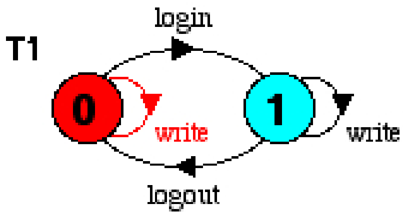

Figure 1: LTS for the Base Program `T1`

The third modification (i.e., `write` event) does not occur between a `login` and a `logout`, so it will not be registered in the versioning system. When this third modification happens, the aspect is waiting for the beginning of a new session (i.e., `login`).

Basically, EAOP is sequential. It provides a coroutining mechanism between the execution of the base program and the execution of the aspects. Its semantics has been defined in [4] based on a small step operational semantics of the base program and a sequent system in order to let the monitor interleave advice execution when a sequence of small steps is matched. This previous work focuses on aspects interaction detection and resolution: when two aspects match the same sequence, two advices must be interleaved with the base program at the very same execution point. In this case, the sequential composition of the advices is non-deterministic, which requires support for explicit composition beyond ordering. We now focus on concurrency and show how sequential EAOP can be encoded in a concurrent world. This paves the way to studying concurrent versions of EAOP by releasing some synchronization.

3 Sequential EAOP in a concurrent framework

We now encode sequential EAOP in a simple process calculus: FSP (Finite State Process) [10]. A key feature of FSP is that FSP specifications generate *finite* Labeled Transition Systems (LTS). First we model the base program. A base program specification in S, as defined in the previous section, can be directly translated into an FSP primitive process, using the sequence `->` and choice `|` operators. Each event corresponds then to an FSP (atomic) action (and to a label of the corresponding LTS).

For instance, for

$$\begin{aligned}&\mu s_1.\ (\texttt{login} \rightarrow \mu s_2.(\texttt{write} \rightarrow s_2 | \texttt{logout} \rightarrow s_1)\\ &\qquad |\texttt{write} \rightarrow s_1)\end{aligned}$$

we get:

```
T1 = (login -> T2 | write -> T1),
T2 = (write -> T2 | logout -> T1).
```

The corresponding LTS, generated using LTSA, is given in Figure 1. It makes clear that a file modification can happen outside of a session (`write` in state 0) or within a session (`write` in state 1). However this simple translation is not enough in order to model sequential EAOP (i.e., coroutining between base and aspects). Indeed, each time an execution event e_i is generated, the base program must suspend its execution until advices have been executed (i.e., until they generate an execution event $rete_i$ for "return from e_i"). This simple instrumentation can be defined as the following program transformation:

$$\begin{aligned}&T_1 :: S \rightarrow S\\ &T_1[\![\mu s.(e_1 \rightarrow S_1 | \dots | e_n \rightarrow S_n)]\!] =\\ &\quad \mu s.(e_1 \rightarrow rete_1 \rightarrow T_1[\![S_1]\!] | \dots | e_n \rightarrow rete_n \rightarrow T_1[\![S_n]\!])\\ &T_1[\![s]\!] = s\\ &T_1[\![Stop]\!] = Stop\end{aligned}$$

For the previous base program example we get:

$$\begin{aligned}&\mu s_1.(\texttt{login} \rightarrow \texttt{retlogin} \rightarrow\\ &\qquad\qquad \mu s_2.(\texttt{write} \rightarrow \texttt{retwrite} \rightarrow s_2\end{aligned}$$

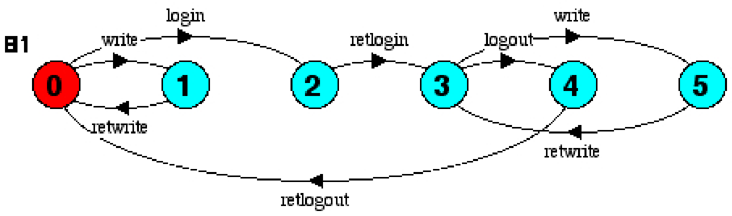

Figure 2: LTS for the Instrumented Base Program `B1`

$|\texttt{logout} \rightarrow \texttt{retlogout} \rightarrow s_1)$
$|\texttt{write} \rightarrow \texttt{retwrite} \rightarrow s_1)$

The corresponding LTS is:

```
B1 = (login -> retlogin -> B2
     | write -> retwrite -> B1),
B2 = (write -> retwrite -> B2
     | logout -> retlogout -> B1).
```

Each transition of `T` has been decomposed into a sequence of two transitions (see also Figure 2). Extra intermediate states are 1, 2, 4 and 5. The two other states 0 and 3 correspond to the states 0 and 1 in the original LTS.

Let us consider a base program S and its aspects $A_1, \ldots, A_n$. The semantics of the complete system is defined as the parallel composition of a process representing the base program and the processes corresponding to aspects:

$$T_1[\![S]\!] || (T_2[\![A_1]\!]\ \alpha(a_1)) || \ldots || (T_2[\![A_n]\!]\ \alpha(a_n))$$

By definition of the parallel composition operator $||$, these processes synchronize together by rendez-vous on shared events. Here, T_2 is a program transformation that generates a process definition in S for each aspect in A and α computes the alphabet (i.e., relevant events) for an aspect as follows:

$$\begin{array}{ll} \alpha :: A \rightarrow 2^{Evt} & \\ \alpha(\mu a.(e_1 \triangleright S_1; A_1 \Box \ldots \Box e_n \triangleright S_n; A_n)) & \\ \quad = \bigcup_{i=1}^{n} \{e_i\} \cup \alpha(A_i) & \\ \alpha(a) & = \emptyset \\ \alpha(\mathit{Stop}) & = \emptyset \end{array}$$

and

$$\begin{array}{l} T_2 \quad :: A \rightarrow 2^{\alpha} \rightarrow S \\ T_2[\![\mu a.(e_1 \triangleright S_1; A_1 \Box \ldots \Box e_n \triangleright S_n; A_n)]\!]\ \alpha_A \\ \qquad = \mu a.\ (e_1 \rightarrow S_1[\mathit{rete}_1 \rightarrow T_2[\![A_1]\!]\ \alpha_A / \mathit{Stop}] \\ \qquad\qquad |\ldots \\ \qquad\qquad |e_n \rightarrow S_n[\mathit{rete}_n \rightarrow T_2[\![A_n]\!]\ \alpha_A / \mathit{Stop}] \\ \qquad\qquad |e \rightarrow \mathit{rete} \rightarrow a) \\ \qquad\qquad \forall e\ \mathit{in}\ \alpha_A - \{e_1, \ldots, e_n\} \\ T_2[\![a]\!]\ \alpha_A = a \\ T_2[\![\mathit{Stop}]\!]\ \alpha_A = \mathit{Stop} \end{array}$$

The role of the transformation T_2 is twofold:

- First, it introduces the synchronization necessary to implement the coroutining between the base program and the aspects. The end *Stop* of each advice (defined in S) is replaced by the event rete_i that resumes the base program execution, followed by the rest of the transformed definition. This is denoted by the substitution $E[e/c]$, which corresponds to the substitution of e for c (here *Stop*) in the expression E.
- Second, each choice point is completed with empty branches for all events that the aspect is not currently expecting. This can be seen as an active waiting loop. It is required so that the aspect translated as a process does not introduce a deadlock. Either the aspect is interested in the current base program event, or it is not and waits for the next event.

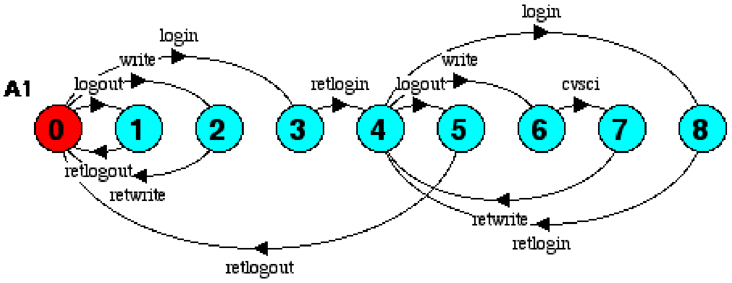

Figure 3: LST for the Instrumented Aspect `A1`

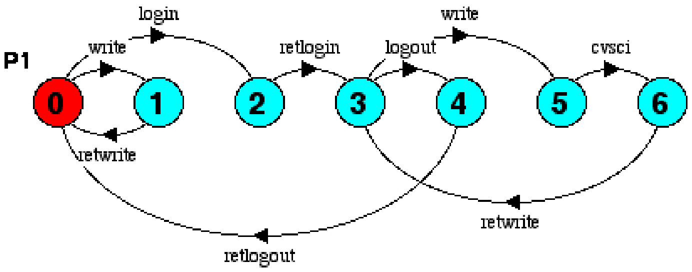

Figure 4: LTS for the Woven Program `P1`

The result of the transformation on our backup aspect is the following:

$$
\begin{aligned}
\mu a_1.\ (&\texttt{login} \rightarrow \texttt{retlogin} \rightarrow \\
&\qquad \mu a_2.\ (\texttt{write} \rightarrow \texttt{cvs\_ci} \rightarrow \texttt{retwrite} \rightarrow a_2 \\
&\qquad\qquad |\texttt{logout} \rightarrow \texttt{retlogout} \rightarrow a_1 \\
&\qquad\qquad |\texttt{login} \rightarrow \texttt{retlogin} \rightarrow a_2) \\
&|\texttt{write} \rightarrow \texttt{retwrite} \rightarrow a_1 \\
&|\texttt{logout} \rightarrow \texttt{retlogout} \rightarrow a_1)
\end{aligned}
$$

That is, in concrete FSP syntax:

```
A1 = (login -> retlogin -> A2
     | write -> retwrite -> A1
     | logout -> retlogout -> A1),
A2 = (write -> cvsci -> retwrite -> A2
     | logout -> retlogout -> A1
     | login -> retlogin -> A2).
```

The corresponding LTS is given in Figure 3. Waiting loops appear naturally as state transitions 0-1-0, 0-2-0, and 4-8-4.

The FSP of the base program `B1` and the FSP of the aspect `A1` share all the actions (e.g. `login`, `write`, ...) but `cvs_ci`. They synchronize by rendez-vous on these shared actions. So the complete woven program is defined as:

```
||P1 = (B1||A1).
```

The corresponding LTS in Figure 4 shows how file versioning `cvs_ci` has been woven inside sessions.

This semantics makes it easy to formally prove properties. For instance, we can specify in FSP the safety property that "during sessions file modifications are versioned" as:

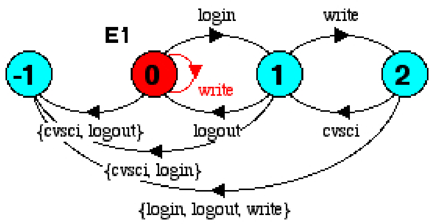

Figure 5: LTS for the Safety Property `E1`

```
property
E1 = (login -> E2 | write -> E1),
E2 = (write -> cvsci -> E2 | logout -> E1).
```

The LST of this property, given in Figure 5, introduces an error state -1 that must never be reached. The property can be checked by composing it with the woven program `||P2 = (P1||E1)`. This composition returns exactly the LTS of the woven program (Figure 4) where the error state does not occur: the property is satisfied by the woven program.

Note that this translation in FSP provides a slightly less sequential semantics than the one introduced in [4]. Indeed, in this previous work, when several advices are to be executed at the same execution point, they are executed one after the other in a non-deterministic order. Our translation ensures that all advices begin with e and finish with *rete* but allows them to be interleaved (or executed in parallel). In the next section we show how to take advantage of the concurrent framework.

4 Towards Concurrent EAOP

Modifying the previous transformation in order to provide a fully concurrent version of EAOP is quite simple. First, we consider concurrent base programs in B:

$$B \quad ::= \quad S_1 || \dots || S_n$$

A base program definition in B is a parallel composition of sequential programs S_i. In fact, B is a superset of S. The transformation T_1 is straightforwardly extended to B as follows:

$$T_1 :: B \rightarrow B$$
$$T_1[\![S_1 || \dots || S_n]\!] = T_1[\![S_1]\!] || \dots || T_1[\![S_n]\!]$$

Second, in T_1 and T_2, the events *rete* ensure that the base program is paused and waits for the end of advice execution before resuming. When these events are suppressed from the translation schemes, the base program and the advices can be executed in parallel.

For instance, let us consider the base program without the synchronization events *rete*:

```
T1 = (login -> T2 | write -> T1),
T2 = (write -> T2 | logout -> T1).
```

and the concurrent aspect definition without the synchronization events, but still with "waiting loops" for `logout` and `write` in state 0 and `login` in state 1:

```
CA1 = (login -> CA2
      | write -> CA1
      | logout -> CA1),
CA2 = (write -> cvsci -> CA2
      | logout -> CA1
      | login -> CA2).
```

The corresponding LTS is given in Figure 6. The woven program is modeled by the parallel composition `||CP1 = (T1 || CA1)`. The resulting LTS in Figure 7 performs `cvs_ci` after each file modification within a session. Note that this simple example does not display much parallelism opportunity. The reason is that

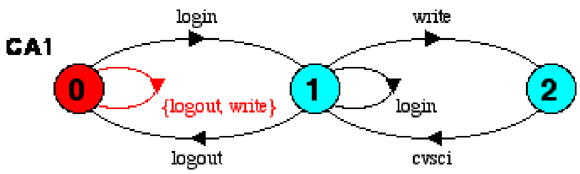

Figure 6: LTS for the Concurrent Aspect `CA1`

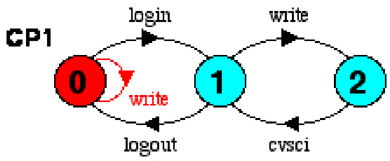

Figure 7: LTS for the Concurrent Woven Program `CP1`

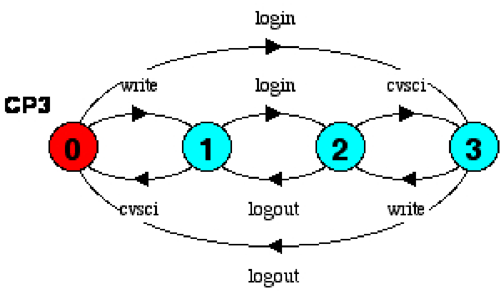

Figure 8: LTS for the Concurrent Woven Program `CP3`

the aspect is interested in all the events produced by the base program and moreover matches very closely its behaviour.

Let us consider a slightly different aspect, which commits each write modification independently from its occurrence within a session:

$$\mu a.(\mathtt{write} \triangleright \mathtt{cvs\_ci}; a)$$

The alphabet of this aspect, $\{\mathtt{write}\}$, is a strict subset of the set of events, $\{\mathtt{login}, \mathtt{write}, \mathtt{logout}\}$, emitted by the base program. The aspect can straightforwardly be translated into the following process:

```
CA3 = (write -> cvsci -> CA3).
```

The LTS resulting of the weaving `||CP3 = (T1 || CA3)` is given in Figure 8. One can see that, here, the last write of a session can be committed either before or after logout, i.e., this last write and logout can take place in parallel. Also, the last write between two sessions can be committed either before or after login.

Such a loose synchronization is not always what is required. For instance, our backup aspect that commits the current version of a file requires that the base program does not modify the file before the current modification has been committed. When the base program performs the sequence $\mathtt{write} \rightarrow \mathtt{compress}$ rather than just `write`, the base function `compress` and the advice `cvs_ci` both modify the file and should not be executed in parallel. In general, when the advice accesses or modifies the base program state, this state should be locked. This synchronization can be directly expressed in the concurrent framework with extra shared events. For instance, we can introduce a `lock` event in the base program $\mathtt{write} \rightarrow \mathtt{lock} \rightarrow \mathtt{compress}$ and in the aspect advice $\mathtt{write} \triangleright \mathtt{cvs\_ci} \rightarrow \mathtt{lock} \rightarrow \mathit{Stop}$ in order to synchronize the two functions. This event is similar to our *rete* events in the previous section.

5 Related work

There are now numerous proposals for AOP, but there is little work devoted to concurrent AOP.

In AspectJ, the base program is paused when an advice is executed. Moreover, AspectJ does not provide explicit support for concurrent programs. So, the advices must explicitly create threads and the programmer must manually deal with synchronization.

The pointcut model of AspectJ can be extended with trace matching in order to define sequences of joinpoints (i.e., execution events) [1]. Joinpoints in a sequence definition can share variables (i.e., object references). This allows matching several sequences at the same time in a sequential Java program. Trace matching also provides support for concurrent base programs. An aspect can match the trace of a single thread (as specified by the `perthread` keyword), or the complete trace (i.e., the interleaved traces of all threads). An advice is executed in the thread corresponding to the last event of a sequence (i.e., the base program is paused). However, trace matching does not provide explicit support for concurrent aspects (advices must create threads explicitly). Advices are also simpler than in our model: there is a single advice per aspect, at the end of the corresponding sequence.

Process algebras have already been used to model AOP [2]. However, this related work does not consider concurrent AOP but shows how to encode sequential AOP in a process calculus. It focuses on correctness of aspect-weaving algorithms and discusses different notions of equivalence. The (non)-termination of the weaving algorithm is discussed.

Concurrency has also been considered in a domain close to AOP: reflection. The authors of [11] criticize the standard approach of *procedural* reflection, whereby the base level is blocked when the metalevel is active and suggest that both levels should communicate via asynchronous events. The paper sketches a framework implementing this idea together with its implementation in Java, using J2EE and JMS. There is no support (language or model) to reason about synchronization and composition issues.

6 Conclusion

Building on previous EAOP work, we have presented a simple model of concurrent aspects built on top of a process calculus. Although the model is an extension of the initial EAOP model, it is a simplification of it. Important features such as visible aspects (a.k.a. aspects of aspects) and aspect composition require further investigations. The issue here is to provide the user with the right level of abstraction, which does not seem to be at the level of the process calculus. For instance, composing our backup aspect with a disk optimization aspect, which zips files once they have been written, is not a simple parallel composition. Indeed, commit should occur before the file is zipped. Such a constraint cannot be expressed in a simple compositional manner (i.e., without rewriting the process versions of the backup and zip aspects) using FSP.

In spite (and because) of its simplicity, we find that this model provides an interesting basis for better understanding the relationship between processes, components, and aspects, for reasoning about concurrent aspects, and for designing and implementing concurrent aspect languages.

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In R. P. Gabriel, editor, *ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*. ACM Press, 2005.

[2] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In Yonezawa and Matsuoka [14], pages 187–209.

[3] J.-P. Briot and J. Malenfant, editors. *Langages et modèles à objets - LMO'03*, Vannes, France, Jan. 2003. Hermès. RSTI série L'objet, 9(1-2).

[4] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002 - Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188, Pittsburgh, PA, USA, Oct. 2002. Springer-Verlag.

[5] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In K. Lieberherr, editor, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 141–150, Lancaster, UK, Mar. 2004. ACM Press.

[6] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with arachne. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[7] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In Yonezawa and Matsuoka [14], pages 170–186.

[8] R. Douence and M. Südholt. Un modèle et un outil pour la programmation par aspects événementiels. In Briot and Malenfant [3], pages 105–118. RSTI série L'objet, 9(1-2).

[9] A. Farías and M. Südholt. On components with explicit protocols satisfying a notion of correctness by construction. In R. Meersman and Z. Tari, editors, *CoopIS/DOA/ODBASE*, volume 2519 of *lncs*, Irvine, CA, USA, Oct. 2002. sv.

[10] J. Magee and J. Kramer. *Concurrency: State Models and Java*. Wiley, 1999.

[11] J. Malenfant and S. Denier. ARM : un modèle réflexif asynchrone pour les objets répartis et réactifs. In Briot and Malenfant [3], pages 91–104. RSTI série L'objet, 9(1-2).

[12] S. Pavel, J. Noyé, P. Poizat, and J.-C. Royer. Java implementation of a component model with explicit symbolic protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[13] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(9), 2002.

[14] A. Yonezawa and S. Matsuoka, editors. *Meta-Level Architectures and Separation of Crosscutting Concerns, Third International Conference, Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, Kyoto, Japan, Sept. 2001. Springer-Verlag.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399