Concurrent aspects

Rémi Douence, Didier Le Botlan, Jacques Noyé, Mario Südholt

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Concurrent aspects

Rémi Douence, Didier Le Botlan, Jacques Noyé and Mario Südholt

N° 5873

Mars 2006

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5873--FR+ENG

Concurrent aspects

Rémi Douence*, Didier Le Botlan†, Jacques Noyé* and Mario Südholt

Thème COM — Systèmes communicants
Projet Obasco

Rapport de recherche n° 5873 — Mars 2006 — 18 pages

Abstract: Aspect-Oriented Programming (AOP) promises the modularization of so-called crosscutting functionalities 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. In particular, there is no formally-defined concurrent approach to AOP, with the result that coordination issues between aspects and base programs as well as between aspects cannot precisely be investigated.

This paper presents Concurrent Event-based AOP (CEAOP), which addresses this issue. Our contribution can be detailed as follows. First, we formally define a model for concurrent aspects which extends the sequential Event-based AOP approach. The definition is given as a translation into concurrent specifications using Finite Sequential Processes (FSP), thus enabling use of the Labelled Transition System Analyzer (LTSA) for formal property verification. Further, we show how to compose concurrent aspects using a set of general composition operators and sketch a Java prototype implementation for concurrent aspects we have realized.

Key-words: Aspect-Oriented Programming, concurrent programming, CEAOP

This work has been supported by AOSD-Europe, the European Network of Excellence in AOSD (`www.aosd-europe.net`).

* École des Mines de Nantes
† CNRS/École des Mines de Nantes

Des aspects concurrents

Résumé : La programmation par aspects, *A*spect-Oriented Programming (AOP) en anglais, est une technique prometteuse qui améliore la modularisation des applications de grande taille en évitant l'entrelacement de certaines fonctionnalités de ces applications au sein de leur code de base. Actuellement, presque toutes les approches pour la programmation par aspect fournissent des moyens pour la description d'aspects séquentiels qui sont appliqués à des programmes de base également séquentiels. En particulier, il n'y a pas encore de modèle formel d'aspects concurrents, ce qui entrave l'étude des problèmes de coordination entre aspects et programme de base ainsi qu'entre aspects. Dans cette article, nous introduisant un modèle d' *a*spects événementiels concurrents qui permet une telle étude. Nos contributions peuvent être détaillées comme suit. D'abord, nous définissons formellement un modèle d'aspects concurrents qui étend l'approche des aspects événementiels séquentiels. Cette définition est donnée sous forme d'une traduction dans des spécifications d'activités concurrentes exprimées à l'aide de *F*inite Sequential Processes, ce qui permet l'utilisation de l'outil *L*abelled Transition System Analyzer (LTSA) pour la vérification formelle de propriétés. En outre, nous montrons comment des aspects concurrents peuvent être composés à l'aide d'un ensemble d'opérateurs généraux de composition et esquissons l'implémentation d'un prototype en Java de notre modèle.

Mots-clés : Programmation par aspects, programmation concurrente, CEAOP

Contents

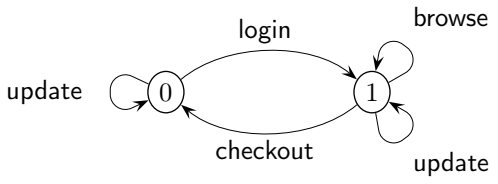
1 Introduction

Aspect-Oriented Programming (AOP) [1, 11] promises means for the modularization of so-called crosscutting functionalities, which cannot be reasonably modularized using traditional programming means, such as objects and components. The proper modularization of such concerns constitutes a major problem for the development of large-scale applications. Crosscutting concerns occur, in particular, in many concurrent applications, at various levels. Let us consider, for instance, request handling in web servers, event handling in graphical user interfaces, monitoring and debugging, and coordination.

Up to now a large number of approaches for AOP of sequential programs have been proposed, most notably AspectJ [4]. In these systems, aspects, which allow modularization of crosscutting concerns, are woven into a base application resulting in an executable sequential application. Concurrency can be added in these systems only by exploiting existing libraries for concurrent programming. By contrast, there are no AOP languages with facilities for the definition of concurrently executing aspects as well as for the coordination of aspects and concurrent base programs directly in terms of AOP-specific concepts. This state-of-affairs is all the more problematic as AOP features basic program structures and corresponding execution patterns that do not admit simple reuse of traditional coordination and concurrency control mechanisms. In this paper we address 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 segments 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 usefully applied in the concurrent setting (in contrast to the standard "advice chaining" strategy used in sequential AOP).

In this paper we investigate means to address such AO-specific coordination issues. We base our investigation on the model of Event-based AOP (EAOP) [9, 10]. This model provides an intuitive and simple model for sequential AOP, whose notion of aspects, defined in terms of regular sequences of execution events, is readily amenable for extension to concurrent executions. Furthermore, we strive for a compositional model of concurrent AOP, which supports coordination through suitable aspect composition operators applied to arbitrary aspects In addition, because of the inherent difficulty of developing correct concurrent programs, to which aspects may even contribute (through their scattered effects on base executions), a model for concurrent aspects should support the use of automatic verification techniques, such as model checking techniques. Finally, the model should be intuitive and enable practical implementations.

Concretely, we present the model Concurrent EAOP (CEAOP), which explicitly addresses the three AO-specific coordination issues and meets the requirements mentioned above. Aspects in CEAOP are concurrent entities that can be woven with a concurrent base program; coordination is supported by a set of general composition operators; aspects and AO programs can be manipulated and model checked using the tool Labeled Transition Systems Analyzer (LTSA) [12] but can also be readily implemented in Java.



a) Automata representation

(0) Server = login → InSession
| update → Server,

(1) InSession = checkout → Server
| update → InSession,
| browse → InSession.

b) FSP representation

Figure 1: A model of a simple e-commerce base program

The remainder of the paper is structured as follows. In Section 2, we introduce and formally define the basic instrumentation technique underlying the coordination of concurrent aspects and base programs in the context of the special case of sequential AO programs. Section 3 generalizes the model to concurrent aspects and base programs, while Section 4 presents concurrent composition operators. An implementation in Java is sketched in Section 5. Section 6 details related work and Section 7 gives a conclusion and presents future work.

2 Sequential EAOP

An aspect is a modular unit whose purpose is to modify the execution of a program, called the base program, by inserting behavior and possibly skipping some of its steps. The piece of code describing the modification is called an advice. Pointcut expressions match sets of execution point (joinpoints) and thus define execution points where control has to be transferred in order to execute the advice. In most AOP approaches, an aspect is a collection of pairs (pointcut, advice) and pointcut matching as well as advice execution depends on the local state of the base program at the joinpoint matched by the pointcut. The EAOP model is richer. Instead of denoting a set of individual joinpoints, a pointcut denotes a set of joinpoint sequences. Along a given sequence, different parts of the advice are executed, depending on the history of the execution, *i.e.,* EAOP directly supports stateful aspects.

To illustrate the concepts introduced in this paper, we use a running example inspired by typical e-commerce applications. Let us consider the following e-commerce base program. Clients connect to a website and must log in to identify themselves, then they may browse an online 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 a simple control flow automaton as shown in Figure 1a or its equivalent textual definition in FSP as shown in 1b.

Let us now consider the problem of cancelling updates during sessions, *e.g.,* to ensure consistent pricing to the client. Using EAOP [9, 10] we can define a suitable aspect as follows:

$$\mathsf{Consistency} \triangleq \mu a.\,\big(\mathsf{login}; \mu a'.\,\big((\mathsf{update} \rhd \mathsf{skip}\ \ \mathsf{log}; a') \,\square\, (\mathsf{checkout}; a)\big)\big)$$

This aspect initially starts in state a 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 a' in which it waits for either an update event or a checkout (other events being ignored). If update occurs first, the associated advice skip log causes the base program to skip the update command (skip is a keyword) and the aspect performs the log command. Then the base program resumes and the aspect returns to state a'. If checkout occurs first, the aspect returns to state a and the base program execution resumes. Since updates are ignored in state a, updates occurring out of a session are performed, while those occurring within sessions (state a') are skipped.

Modeling of aspects. This behavior can be formally defined by translating the aspect definition into FSP. Note that this translation remains entirely valid for the concurrent case. Yet, we chose to present and define it at first for the sequential case for its simpler presentation. FSP is one of the formalisms used by LTSA (Labelled Transition System Analyser) [12]. This tool combines two formalisms, FSP and LTS, to model concurrent behavior using finite state machines that can be either described textually as Finite State Proceses (FSP) and graphically as Labelled Transition Systems (LTS). Labelled transition systems can then be used to animate or verify the model using standard model checking techniques.

A sequential process is modeled as a sequence of atomic actions using recursion, the sequence operator $\rightarrow$, the choice $|$ operator, and guarded actions (that we will not use in the following). Sequential processes can be combined into concurrent processes using the parallel operator $\|$. Interactions are modeled by shared actions. When an action is shared among several processes, the shared actions must be executed at the same time by these processes. Two operators on actions, a renaming operator and a hiding operator, make it possible to define generic processes that can be "connected" in various ways to other processes. Any sequential process can be straightforwardly represented as a labelled transition system (subprocesses of the process correspond then to states in the automaton), and parallel composition is a form of synchronized product. In the following, in order to simplify the presentation, we will sometimes ignore some specific syntactic details of FSP (*e.g.,* we will not use capital letters for process names).

The interest of FSP/LTS is that it provides a fairly simple model of concurrency that is well documented (with a good understading of how to implement models in Java) and supported by a valuable tool, LTSA (all the examples in this paper have been tested using this tool).

Let us come back to the translation of our example. The translation performs two operations. First, it introduces synchronization events that will be used to coordinate the aspect and the base program. In fact, we consider advice to consist of three parts $b\ ps\ a$ where ps is one of the keywords proceed or skip, specifying respectively whether the base action at the matched joinpoint is executed or not, and b, a denote sequences of actions that are executed respectively before and after ps. Synchronization events are used to be able to synchronize these three sequences of actions with the base program. Synchronization events will there-

/sf

```
1  a = ( login → a'
2      | eventB_update → proceedB_update → proceedE_update → eventE_update → a
3      | checkout → a | browse → a ),

4  a' =( eventB_update → skipB_update → skipE_update → log → eventE_update → a'
5      | checkout → a
6      | browse → a | login → a' ).
```

Figure 2: The consistency aspect in FSP

fore be introduced in the base program as well. Second, it deals with events ignored by the aspect by introducing loops (henceforth called waiting loops) that automatically resume the base program.

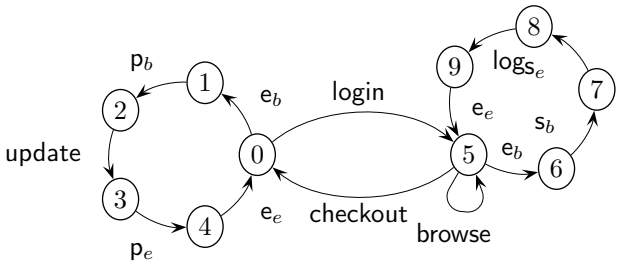
The consistency aspect defined above is translated as shown in Figure 2. In this and the following figure, code set black on white stems verbatim from the aspect definition, while highlighted code corresponds to the instrumentation, namely waiting loops and synchronization events. As an example of synchronization events, line 4 introduces two pairs of events: the pair eventB_update and eventE_update mark respectively the beginning and end of the advice attached to the update event. The second pair skipB_update and skipE_update is used to control the base program by sending a (begin and end) skip message. We show below how the base program is instrumented accordingly to deal with such control messages.

As an example of waiting loops, in state a, loops have been added for the events update on line 2, and for checkout and browse on line 3. The nature of update is different from the two others. Indeed, update is a so-called *skippable* event, that is, it corresponds to an operation of the base program that may be skipped. As a consequence, it needs synchronization events that control the base program. On the contrary, checkout and browse are simple non-skippable events.

Similarly, the base program abstraction of Figure 1b must be transformed as shown in Figure 3. Transitions on the non-skippable events login, checkout, and browse are preserved without changes (lines 1, 5, and 9, respectively). Transitions for the skippable event update is translated to eventB_update followed by a choice (lines 2 and 6). Intuitively, the base program emits this event to the aspect. Then the base program waits for a control event from the aspect which is either proceedB_update (lines 3 and 7) or skipB_update (lines 4 and 8). In the first case, the original event update is emitted (*i.e.,* the base program performs the update operation, which may take some time), then it emits proceedE_update to yield control to the aspect. Finally, it waits for the end of the advice eventE_update. In the second case, the original base program resumes the advice by emitting skipE_update and waits for the end of the advice in order to resume its own execution.

```
1 Server = login → InSession
2     | eventB_update →
3         ( proceedB_update →  update → proceedE_update → eventE_update → Server
4         | skipB_update → skipE_update → eventE_update → Server) ,
5 InSession = checkout → Server
6     | eventB_update →
7         ( proceedB_update →  update → proceedE_update → eventE_update → InSession
8         | skipB_update → skipE_update → eventE_update → InSession) ,
9     | browse → InSession.
```

Figure 3: Instrumented base program in FSP



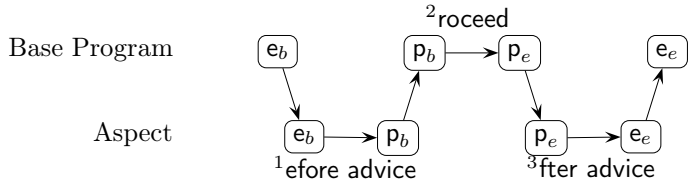
where the following abbreviations are used:

e_b	=	eventB_update
e_e	=	eventE_update
p_b	=	proceedB_update
p_e	=	proceedE_update
s_b	=	skipB_update
s_e	=	skipE_update

Figure 4: Woven example

The semantics of the woven program is modeled by the parallel composition of the base program and the aspect in FSP. Automata representations of such compositions can be generated using the LTSA tool, thus enabling property checking based on model checking techniques (in order to verify, *e.g.,* that no updates but only logs occur during sessions). We show the output of our example composition in Figure 4. The left-hand side cycle performs updates outside of sessions. The right-hand side cycle skips update commands during sessions and does some logging. The middle cycle starts and ends sessions.

We can picture the control flow between the base program and the aspect as shown below for the case of a proceeding advice. Only the four synchronization events are shown, which are denoted e_b, e_e, p_b and p_e, as in Figure 4. The arrows represent the control flow.



Formal definition of instrumentation. Now that we have shown the idea of the encoding on our example, we formally define the general transformation (which is also valid for the concurrent case). The control flow of the base program is abstracted into a finite state automaton described by the grammar B at the top of Figure 5. Each variable b represents a state and the e_i represent transitions labels. The recursion operator μ makes it possible to define cycles. In the following, we assume that all variables b are different (this can be ensured easily by α-renaming). The transformation $\mathcal{T}_B$ translates such a base program into FSP. The first rule generates a list of equations for b and its successors. The second rule stops the equation generation. The third rule translates a sequence $(e; B)$ starting with a non-skippable event, identified by $e \in \mathcal{E}$, into a FSP sequence $e \rightarrow name(B)$, where $name(B)$ is the process name associated to B. Finally, the fourth equation introduces the synchronization events and translates a transition with a skippable event ($e \in \mathcal{S}$) into two branches (either the base program proceeds, executing the base instruction, or it skips the instruction).

$$B \quad ::= \quad \mu b.(\square_{i=1\ldots n}\ e_i; B_i) \quad \Big| \quad b$$

$$\mathcal{T}_B(\mu b.(\square_i\ e_i; B_i)) \triangleq b = \Big|_i (\mathcal{T}'_B(e_i; B_i))\ ,\ \mathcal{T}_B(B_1)\ ,\ ..\ ,\ \mathcal{T}_B(B_n).$$
$$\mathcal{T}_B(b) \triangleq \epsilon$$

$$\mathcal{T}'_B(e; B) \triangleq e \rightarrow name(B) \qquad \text{if } e \in \mathcal{E}$$
$$\mathcal{T}'_B(e; B) \triangleq \mathsf{eventB\_}e \rightarrow (\ \mathsf{skipB\_}e \rightarrow \mathsf{skipE\_}e \rightarrow \mathsf{eventE\_}e \rightarrow name(B)$$
$$\mid\ \mathsf{proceedB\_}e \rightarrow e \rightarrow \mathsf{proceedE\_}e \rightarrow$$
$$\mathsf{eventE\_}e \rightarrow name(B)\ ) \qquad \text{if } e \in \mathcal{S}$$

$$name(\mu b.(..)) \triangleq b \qquad name(b) \triangleq b$$

Figure 5: Abstract base program syntax and instrumentation

Similarly, Figure 6 defines the syntax of aspects and their translation $\mathcal{T}_A$ into FSP. The grammar A of aspects is similar to the grammar B, but each event is followed by an advice S (for the sake of simplicity, either a sequence of events introduced by $\triangleright$ and containing proceed or skip, or an empty advice ϵ). Moreover, we assume that an advice is empty if

and only if its associated event cannot be skipped. The first and the second rule of the transformation $\mathcal{T}_A$ are similar to $\mathcal{T}_B$ in Figure 5. When the event e is non-skippable, the transition is translated directly. When the event e is skippable, the advice is translated by inserting synchronization events in its definition.

$$
\begin{array}{lll}
A & ::= & \mu a.(\Box_{i=1\ldots n}\ e_i S_i; A_i) \ \mid\ a \\
S & ::= & \epsilon \ \mid\ \rhd e_{1b} \ldots e_{nb}\ ps\ e_{1a} \ldots e_{ma} \quad \text{where } ps \in \{\mathsf{proceed}, \mathsf{skip}\}
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{T}_A\,(\mu a.(\Box_i\ e_i S_i; A_i)) & \triangleq & a = \Big|_i \mathcal{T}'_A\,(e_i S_i; A_i)\,,\ \mathcal{T}_A(A_1)\,,\ ..\ ,\ \mathcal{T}_A(A_n). & \\
\mathcal{T}_A\,(a) & \triangleq & \epsilon & \\
 & & & \\
\mathcal{T}'_A\,(e; A) & \triangleq & e \rightarrow name(A) & \text{if } e \in \mathcal{E} \\
\mathcal{T}'_A\,(eS; A) & \triangleq & e \rightarrow \mathcal{T}''_A(S, e) \rightarrow name(A) & \text{if } e \in \mathcal{S} \\
 & & & \\
\mathcal{T}''_A(\rhd e_{1b} \ldots e_{nb}\ ps\ e_{1a} \ldots e_{ma}, e) & \triangleq & \mathsf{eventB\_}e \rightarrow e_{1b} \rightarrow \ldots \rightarrow e_{nb} \rightarrow ps\mathsf{B\_}e & \\
 & & \quad \rightarrow ps\mathsf{E\_}e \rightarrow e_{1a} \rightarrow \ldots \rightarrow e_{ma} \rightarrow \mathsf{eventE\_}e & \\
 & & \text{where } ps \in \{\mathsf{proceed}, \mathsf{skip}\} &
\end{array}
$$

Figure 6: Aspect syntax and their transformation into FSP

The previous transformation translates an aspect into FSP but does not account for waiting loops. In order to take ignored events into account and to avoid deadlock, we modify and extend the previous transformation by introducing waiting loops as shown in Figure 7. The core of the transformation remains the same, but the transformation now completes the transitions of every state with waiting loops that ignore the other events (and also allow the base program to proceed in case of a skippable event).

$$
\begin{array}{lcll}
\mathcal{T}_A(\mu a.(\Box_i\ e_i S_i; A_i)) & \triangleq & a = & \Big|_i (\mathcal{T}'_A\,(e_i S_i; A_i)\ \mid\ \Big|_{e \in \mathcal{E} \cup \mathcal{S} \setminus (\bigcup_i e_i)} loop(a, e)\,, \\
 & & & \mathcal{T}_A(A_1)\,,\ ..\ ,\ \mathcal{T}_A(A_n). \\
\mathcal{T}_A(a) & \triangleq & \epsilon & \\
 & & & \\
loop(a, e) & \triangleq & e \rightarrow a & \text{if } e \in \mathcal{E} \\
loop(a, e) & \triangleq & \mathsf{eventB\_}e \rightarrow \mathsf{proceedB\_}e \rightarrow \mathsf{proceedE\_}e \rightarrow \mathsf{eventE\_}e \rightarrow a & \text{if } e \in \mathcal{S}
\end{array}
$$

Figure 7: Translation with waiting loops

This transformation concludes our formal semantics of sequential EAOP. In the next section, we show how some synchronization events can be ignored in order to introduce concurrency.

```
Server    =  login → InSession,          Update  =  update → Update.
InSession =  checkout → Server
          |  browse → InSession.         || Base  =  (Server || Update).
```

Figure 8: A model of a simple e-commerce base concurrent program

3 Concurrent EAOP

Our model in Section 2 is purely sequential: the base program consists of a unique thread and weaving of aspects is modeled with coroutining, that is, synchronization events ensure that only the aspect or the base program runs at a given time. We now adapt our model to the concurrent world by modifying two parameters.

First, the base program is no longer a single process but a combination of several processes. Each thread is modeled by an FSP automaton and the base program is defined as their parallel composition as exemplified in Figure 8.

Second, aspects are viewed as independent processes that run in parallel and synchronize with the base program. Possible synchronization points between the aspect and the base program are pointcuts (*e.g.,* eventB_update), end of advice (*e.g.,* eventE_update), and control events (proceed or skip). Yet, there is some variability in the amount of synchronization that may be introduced. One extremum enforces strong synchronization by weaving sequential aspects in a concurrent program. This corresponds to the encoding done in Section 2. Another option is not to synchronize on some synchronization events, so as to allow more concurrency between the program and the aspect. In particular, aspect and base program may not synchronize on the event eventE_update at the end of the advice. This is expressed in FSP simply by removing this event from the base program and the aspect definitions using the hiding operator \{eventE-update} before composing them.

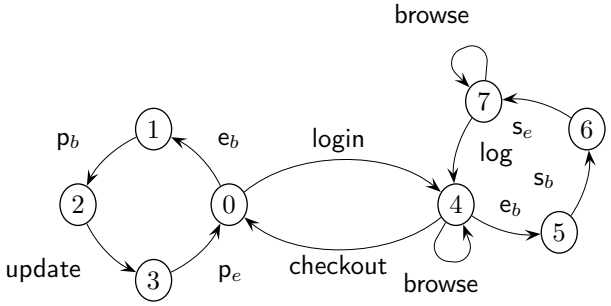
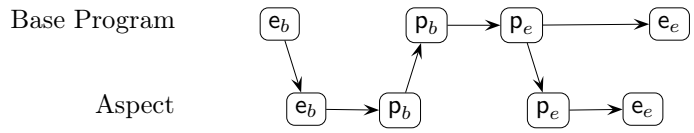


Figure 9: A concurrent program woven with an aspect

In our example, once the update is skipped, the base program resumes while the advice concurrently creates a log. Figure 9 shows the automaton of the woven program computed by LTSA: the user can resume its browsing before a log is created (note the browse transition in

state 7). The picture below illustrates the control flow in this case between the base program and the concurrent aspect, using the abbreviations defined in the previous section.



Another option can be considered. When the events updateE-proceed and updateE-skip are hidden, the rest of the advice is executed in parallel with the update event (which may be executed or not).

4 Concurrent Aspect Composition

The previous sections have shown how to coordinate concurrent execution of a single aspect applied to a base program. In this section, we consider two aspects and show how composition operators can be designed to compose them in different ways. We illustrate our approach by detailing two composition operators and by discussing a few more. The generalization to more than two aspects is possible either by iterating binary composition, or by extending the operators so that they accept more than two arguments.

First, let us augment the e-commerce example introduced in Section 2 by a second aspect:

$$\mathsf{Safety} \triangleq \mu a''.(\mathsf{update} \triangleright \mathsf{rehash\ proceed\ backup}; a'')$$

Each time the website is updated (*i.e.,* the administrator publishes an internal working version), this safety aspect rehashes a database of links before the publication, and backups the database afterward. We translate this aspect into FSP using the technique described previously and obtain the following result:

```
a'' = ( eventB_update → rehash → eventB_proceed
      → eventE_proceed → backup → eventE_update → a'')
```

Sequential functional composition. The first operator, Fun, we consider corresponds to a fully sequentialized functional composition of two aspects. When two advices must be executed at the same joinpoint, the composition Fun($aspect_1$,$aspect_2$) executes the advice of its first argument. If this advice proceeds, it executes the advice of the second argument. If this second advice proceeds, it executes the corresponding action in the base program. Using an informal notion of substitution, Fun($aspect_1$,$aspect_2$) is a composite advice that intuitively behaves like $advice_1[advice_2/\texttt{proceed}]$. Furthermore, the (optional) action sequences before and after the proceed are correspondingly nested.

Reconsidering our example aspects, the advice of the consistency aspect of Section 2 applies to update events only during sessions, where it skips updates and adds a log. In

$\|\mathsf{FunArg}_1 = \mathsf{a}/\{\mathsf{call\_e/proceedB\_e}, \mathsf{ret\_e/proceedE\_e}, \mathsf{skipB\_e1/skipB\_e}, \mathsf{skipE\_e1/skipE\_e}\}.$

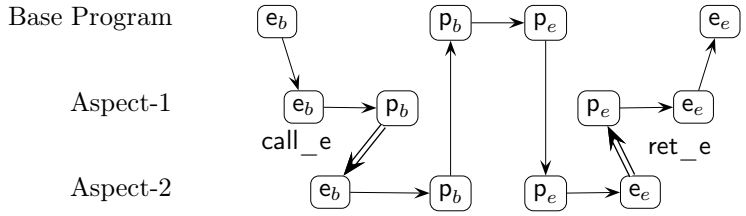
$\|\mathsf{FunArg}_2 = \mathsf{a''}/\{\mathsf{call\_e/eventB\_e}, \mathsf{ret\_e/eventE\_e}, \mathsf{skipB\_e2/skipB\_e}, \mathsf{skipE\_e2/skipE\_e}\}.$

$$\begin{aligned}\mathsf{Fun} \quad = \; & (\ \mathsf{skipB\_e1} \rightarrow\ \mathsf{skipB\_e} \rightarrow\ \mathsf{skipE\_e} \rightarrow\ \mathsf{skipE\_e1} \rightarrow\ \mathsf{Fun} \\ & |\ \mathsf{skipB\_e2} \rightarrow\ \mathsf{skipB\_e} \rightarrow\ \mathsf{skipE\_e} \rightarrow\ \mathsf{skipE\_e2} \rightarrow\ \mathsf{Fun}).\end{aligned}$$

Figure 10: The Fun composition operator in FSP for the event e

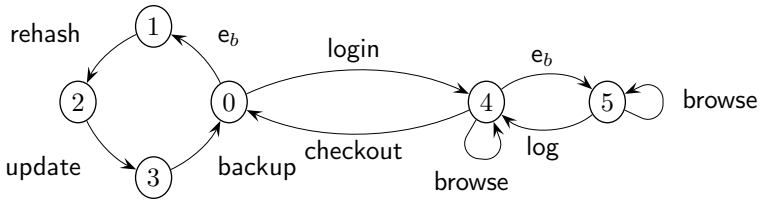
contrast, the advice of the safety aspect applies both during and out of a session. So, as the composition Fun(Consistency,Safety) executes the consistency advice first, it skips updates during sessions and adds a log, but does not run the safety advice. On the contrary, out of a session, the safety advice is applied to update events and the update is performed.

This composition is modeled in FSP by renaming some synchronization events in the aspect definitions and by defining a process Fun that dynamically renames skip messages. Its definition is shown in Figure 10. To ease the understanding of this composition, we represent its control flow in the case where both advices proceed. Here, double arrows correspond to renamings.



When the base program emits a eventB_update event, here denoted by e_b, the advice of the first aspect is executed until it proceeds (event p_b above). In order to link the beginning of the second aspect to the proceed command of the first aspect, we rename both p_b in the first aspect and e_b in the second aspect to the same label call_e. This renaming is depicted by a double arrow. The events p_b and p_e of the second aspect are not renamed, so that they synchronize with the base program. Both the end of the second advice e_e and p_e of the first aspect are renamed to the same label ret_e, so that when the second advice ends, it resumes the execution of the first aspect. Finally, the end of the first aspect emits e_e, which resumes the base program. These renamings appear in Figure 10, using FSP syntax. The processes FunArg_1 and FunArg_2 correspond to the renamings of the first and second aspect, respectively. As for skip commands, they cannot be handled by renamings only. Indeed, both the first aspect and the second aspect may emit a skip command to the base program. In contrast, only the second aspect may emit a proceed command to the base program. As a result, we must rename skip commands differently in each aspect (by appending an identifier 1 or 2) and introduce an extra process Fun that performs dynamic renaming.

The semantics of the woven program is the parallel composition of the three FSP processes in Figure 10 with the base program. The resulting automaton is:



For the sake of clarity, we have hidden most synchronization events in the woven program, keeping only essential ones. Outside of session, only the safety aspect is woven (see the left hand side cycle). During sessions, as shown by the right hand side cycle, the consistency aspect is woven. It skips the safety aspect and only creates a log. In the meantime the user can still browse in parallel with the advice as modeled by the transition browse in each state.

As in Section 3, concurrency can be introduced by hiding synchronization events before composing in parallel the FSP definitions. For instance, when eventE_e is hidden, the post-proceed part of the first advice is executed in parallel with the base program.

Parallel conjunctive composition. Concurrency can also be introduced by considering composition operators that impose less synchronization. For instance, let us consider the ParAnd operator. 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.

The ParAnd operator is defined in FSP as shown in Figure 11. First, the skip and proceed events of aspects are renamed so that they do not synchronize anymore together or with the base program. Second, the process ParAnd implements a rendez-vous between these events of the two aspects by distinguishing four cases. In the first three cases, there is at least one event skip so the base program must also skip. If both aspects proceed, the base program also proceeds.

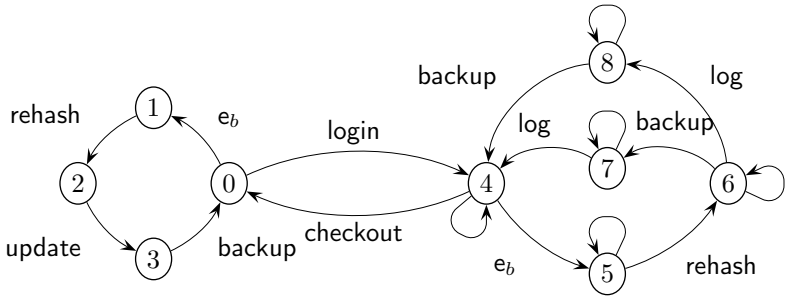
The semantics of the woven program is the parallel composition of the processes of Figure 11 with the base program. Both aspects share the events eventB_e and eventE_e so the beginning and the end of advices are synchronized. Before (and after) skip or proceed, advices of the aspects are executed in parallel. The woven program is represented by the following automaton, where unlabeled loops correspond to browse events).

```
||ParAndArg1 = a/{proceedB_e1/proceedB_e, proceedE_e1/proceedE_e,
                  skipB_e1/skipB_e, skipE_e1/skipE_e}.
||ParAndArg2 = a'' /{proceedB_e2/proceedB_e, proceedE_e2/proceedE_e,
                   skipB_e2/skipB_e, skipE_e2/skipE_e}.

 ParAnd      =
 ( skipB_e1 → ( skipB_e2 → skipB_e → skipE_e → skipE_e1 → skipE_e2 → ParAnd
              | proceedB_e2 → skipB_e → skipE_e → skipE_e1 → proceedE_e2 → ParAnd)

 | proceedB_e1 → ( skipB_e2 → skipB_e → skipE_e → skipE_e2 → proceedE_e1 → ParAnd
                 | proceedB_e2 → proceedB_e → proceedE_e →
                         proceedE_e1 → proceedE_e2 → ParAnd)).
```

Figure 11: The ParAnd composition operator in FSP for the event e



It makes clear that the advices are executed in parallel: both sequences log backup and backup log are valid. Furthermore, the user can still browse in parallel with the advice. As previously discussed, concurrency can be introduced by hiding the event eventE_e before the parallel composition.

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

5 Implementation in Java

We have implemented a prototype of CEAOP for Java. This implementation is realized in form of a framework, which provides classes, *e.g.,* `Aspect` to be subclassed in order to define an aspect. Each aspect has its own thread and provides a blocking method `nextEvent` to specify which events should resume the aspect execution. The consistency aspect, *e.g.,* can be implemented as follows:

```
class Consistency extends Aspect {
  public void run() {
```

```
while (true) {
  nextEvent(Monitorable.Event.login);
  while (nextEvents(Event.update, Event.checkout) == Event.update) {
    skip(); log();
  } } } }
```

Here, the first event of interest is `login`. Then, the aspect waits for either `update` and skips it but creates a log, or `checkout` and waits for the next session.

In our prototype, a composition of aspects is represented by a binary tree whose leaves are aspects and nodes are composition operators such as `Fun` and `ParAnd`. These classes propagate the current execution event of the base program to the aspects and synchronize advices as required. The base program is currently manually instrumented (by explicitly calling a method `emitEvent()`) but this task can be easily automated. Our previous prototype for sequential EAOP performs such an automatic transformation of the base program with the tool Recoder from University of Karlsruhe. Note also that our prototype is not a direct translation of our FSP model for efficiency concerns. Moreover, the power of expression of full Java enables us to experiment with advanced features such as dynamic instantiation of aspects, several composition trees of aspects, and decentralized monitors.

6 Related Work

There are many proposals for AOP, but little work devoted to concurrent AOP. In AspectJ, the base program is paused when an advice is executed. AspectJ also does not provide explicit support for concurrent programs: 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) [2]. 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. Benavides *et al.* introduce AWED [5], an aspect language for distributed programming, which includes regular sequence aspects. Concurrent execution is supported on the language level (i) by pointcuts referring to threads similar to tracematches but also (ii) by remote advice which can be executed asynchronously or synchronously w.r.t. the executions of the (distributed) base program and other aspects. However, this approach, as the others, does not include explicit means for the synchronization of multiple advices applying at an execution point.

Process algebras have already been used to model AOP [3]. However, this 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.

Concurrency has also been considered in a domain close to AOP: reflection. The authors of [13], *e.g.,* 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. Yet, there is no support (language or model) to reason about synchronization and composition issues.

In the area of distributed algorithms, starting with the work of Dijkstra on termination detection [8], there is a long tradition of *superimposing* specific algorithms to base applications with a motivation similar to the aspect approach. Dealing with distributed applications, base applications are naturally modeled as interacting processes. However, the general focus is geared more towards specification and verification than towards providing proper language support for building distributed applications, whereas we are interested in bridging this gap. The work of Sihman and Katz [14] is especially close to ours in that it explores composition issues and suggests that there are two ways of composing superimpositions: sequential composition, similar in spirit to the composition obtained with our `Fun` operator, and merging. But the introduction of a specific aspect construct such as `proceed` changes the overall picture and leads to a richer set of composition operators as demonstrated in our work by the ParAnd operator.

Finally, aspects have been considered as a way to implement coordination [6, 7]. We take here a different point of view. The aspects are basic reusable components whose coordination is specified by the aspect language itself, including the composition operators, and its underlying semantics.

7 Conclusion

In this article, we have presented general requirements for models of concurrent aspects and a concrete formally-defined model, CEAOP, meeting these requirements. In particular, our model supports concurrency in base programs, concurrent execution of aspects and advices with base programs, and composition operators for the coordination of concurrent aspects and base programs. Thanks to our FSP-based semantics, woven programs may be model-checked with LTSA, *e.g.,* verifying absence of deadlocks, progress, and trace properties. We have presented a set of composition operators of concurrent aspects and base programs, as well as evidence that this set can easily be extended. Finally, we have sketched a lightweight prototype implementation in Java.

Our proposal paves the way towards a complete study of concurrent aspect languages and systems. In particular, we consider future work on the inclusion of a notion of aspects of aspects, on property preservation of composition operators, and on efficiently implementing concurrent aspects in a distributed setting.

References

[1] M. Akşit, S. Clarke, T. Elrad, and R. E. Filman, editors. *Aspect-Oriented Software Development*. Addison-Wesley Professional, Sept. 2004.

[2] 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 *Proceedings of OOPSLA'05*. ACM Press, 2005.

[3] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of Reflection 2001*, LNCS 2192, 2001.

[4] AspectJ home page. `http://www.eclipse.org/aspectj/`.

[5] L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly distributed AOP using AWED. In *Proceedings of AOSD'06*. ACM Press, 2006. To appear.

[6] S. Capizzi, R. Solmi, and G. Zavattaro. From endogenous to exogenous coordination using aspect-oriented programming. In *Proceedings of COORDINATION'04*, LNCS 2949, 2004.

[7] A. Colman and J. Han. Coordination systems in role-based adaptive software. In *Proceedings of COORDINATION'05*, LNCS 3454, 2005.

[8] E. W. Dijkstra and C. S. Sholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, Aug. 1980.

[9] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of GPCE'02*, LNCS 2487, 2002.

[10] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of AOSD'04*. ACM Press, 2004.

[11] G. Kiczales, J. Lamping, A. Mendhekar, et al. Aspect-oriented programming. In *Proceedings of ECOOP'97*, LNCS 1241, 1997.

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

[13] J. Malenfant and S. Denier. ARM : un modèle réflexif asynchrone pour les objets répartis et réactifs. In *Proceedings of LMO'03*. Hermès, 2003. RSTI série L'objet, 9(1-2).

[14] M. Sihman and S. Katz. A calculus of superimpositions for distributed systems. In *Proceedings of AOSD'02*. ACM Press, 2002.

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