Concurrent Aspects

Rémi Douence, Didier Le Botlan, Jacques Noyé, Mario Südholt
OBASCO project
École des Mines de Nantes/INRIA, LINA
Département Informatik, École des Mines de Nantes, 44307 Nantes cedex 3, France
www.emn.fr/{douence,lebotlan,noye,sudholt}

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. Finally, we sketch a Java prototype implementation for concurrent aspects, which generates coordination specific code from the FSP model defining the concurrent AO application.

Categories and Subject Descriptors D.1.3. Software [*Programming Techniques*]: Concurrent Programming

General Terms Languages, Verification

Keywords Aspect-oriented programming, concurrency, formal verification, implementation, Java

1. Introduction

Aspect-Oriented Programming (AOP) [1, 14] 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.

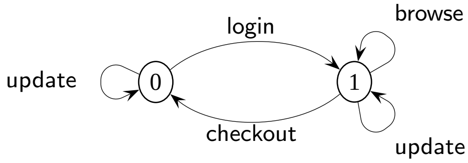
Up to now a large number of approaches for AOP of sequential programs have been proposed, most notably AspectJ [4]. In these systems, aspects, which allow modularization of crosscutting concerns, are woven into a base application resulting in an executable sequential application. Concurrency can be added 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. Such an approach would permit, for instance, to synchronize more easily advice of different aspects applied at the same join point, or conversely, to allow these advice to be executed in parallel.

As a running example, we consider a simplified e-commerce application. Its concurrent activities include actions of remote users as well as (potentially extensive) changes to the underlying databases of that system (which, in turn, entail modifications to the current view of the user of that database). We wish to deal with these crosscutting functionalities concisely, by expressing actions in terms of advice executed at appropriate times and by controlling their concurrent execution directly with high-level coordination operators, that is, without the need of a different mechanism (aspect-oriented or otherwise) whose only purpose is to introduce synchronization-related code. As a result, the programmer can concisely express synchronization constraints between advice without having to consider low-level synchronization issues.

In this paper we address three major issues concerning the coordination of concurrent aspects: (i) aspects should be defined in terms of sequences of execution events triggering actions which are to be coordinated, (ii) coordination not only of complete actions ("advice") but also parts thereof should be supported in order to allow flexible coordination policies, and (iii) different coordination strategies should be supported in case that multiple advice apply at one execution point, for instance, in order to compose independent advice in a concurrent fashion while enabling prioritization through synchronization if necessary. 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, in particular supporting generation of implementations from the models used to define concurrent aspects.

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) [10, 11]. This model provides an intuitive and simple model for sequential AOP, whose notion of aspects,

GPCE '06 Oct. 22–26, Oregon.



$$
\begin{array}{llcl}
(0) & \mathsf{Server} & = & \mathsf{login} \rightarrow \mathsf{InSession} \\
 & & | & \mathsf{update} \rightarrow \mathsf{Server}, \\
(1) & \mathsf{InSession} & = & \mathsf{checkout} \rightarrow \mathsf{Server} \\
 & & | & \mathsf{update} \rightarrow \mathsf{InSession}, \\
 & & | & \mathsf{browse} \rightarrow \mathsf{InSession}.
\end{array}
$$

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

defined in terms of regular sequences of execution events, is readily amenable for extension to concurrent executions. We present the model of Concurrent EAOP (CEAOP), which explicitly addresses the AO-specific coordination issues and requirements mentioned above. First, CEAOP aspects, more specifically advice and part of advice, are concurrent entities that can be woven with a concurrent base program. Aspects and aspect weaving is formally defined by an automatic transformation into the calculus of Finite Sequential Processes (FSP) [15]. Second, coordination is supported by a set of general composition operators. Third, aspects and AO programs can be manipulated, simulated and can be automatically model checked using the tool Labeled Transition Systems Analyzer (LTSA) [15]. Finally, we present an implementation of CEAOP in Java which is realized by the generation of coordination-specific code from the FSP model defining the concurrent AO application.

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

In this section we briefly review how regular aspects are defined in sequential EAOP (which also forms the sequential subset of CEAOP) and introduce the weaving strategy of CEAOP for this special case. 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 at the top of Figure 1 or its equivalent textual definition (bottom of Figure 1).

Let us now consider the problem of cancelling updates during sessions to the client-specific view of the e-commerce shop, *e.g.*, to ensure consistent pricing to the client. Using EAOP [10, 11] we can define a suitable aspect, called Consistency as follows:

$$\mu a.\ \big(\mathsf{login};\mu a'.\ \big((\mathsf{update} \triangleright \mathsf{skip\ \ log}; a') \ \square\ (\mathsf{checkout}; a)\big)\big)$$

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.

Our approach essentially provides mechanisms to synchronize an advice with the base program execution as well as with other advice that applies at the same execution point. In the example, an advice that generates new indexes when an update takes place would have to be synchronized with the consistency aspect above.

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 first for the sequential case for its simpler presentation. FSP is one of the formalisms used by LTSA (Labelled Transition System Analyser) [15]. 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 Processes (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 →, 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 therefore 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.

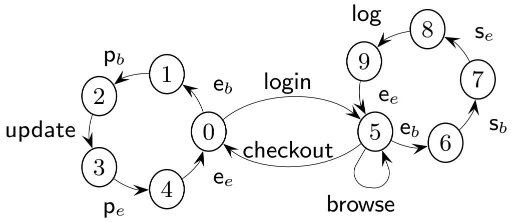
The consistency aspect defined above is translated as shown in Figure 2. In this and the following figure, non-highlighted code stems verbatim from the aspect definition, while highlighted code corresponds to the instrumentation, namely waiting loops and synchronization events. As an example of synchronization 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. Note that the partitioning between skippable and non-skippable events has to be determined by the programmer.

Similarly, the base program abstraction of Figure 1 bottom 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.

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. To this end we can check trace-based quivalence of FSP expressions using the model checking techniques introduced by Magee and Kramer (see, *e.g.,* [12]). 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.

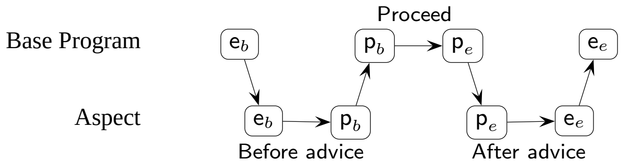
We can picture the control flow between the base program and the aspect as shown below for the case of a proceeding advice. Only the four synchronization events are shown, which are denoted e_b, e_e, p_b and p_e, as in Figure 4. The arrows represent the control flow.



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



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).

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.

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).

$$
\begin{array}{rcl}
\mathsf{Server} & = & \mathsf{login} \rightarrow \mathsf{InSession}, \\
\mathsf{InSession} & = & \mathsf{checkout} \rightarrow \mathsf{Server} \\
 & | & \mathsf{browse} \rightarrow \mathsf{InSession}. \\
\mathsf{Update} & = & \mathsf{update} \rightarrow \mathsf{Update}. \\
\\
\| \ \mathsf{Base} & = & (\mathsf{Server} \parallel \mathsf{Update}).
\end{array}
$$

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

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.

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 $\backslash\{\mathsf{eventE\_update}\}$ before composing them.

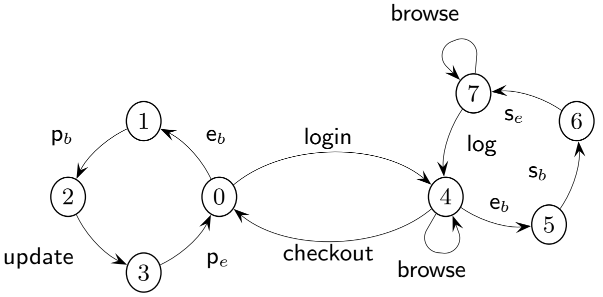
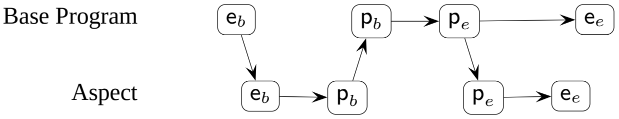


Figure 9. A concurrent program woven with an aspect

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:

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

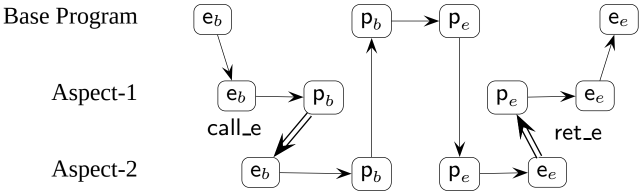
Each time the website is updated (*i.e.,* the administrator publishes an internal working version), this safety aspect rehashes a database of links before the publication, and backups the database afterward. Note that rehashing and backups are rather time-consuming operations whose concurrent execution, if possible, with other e-commerce functionalities is desirable. We translate this aspect into FSP using the technique described previously and obtain the following result:

$$
\begin{array}{l}
\mathsf{a''} \ = \ (\ \mathsf{eventB\_update} \rightarrow \mathsf{rehash} \rightarrow \mathsf{eventB\_proceed} \\
\qquad \rightarrow \mathsf{eventE\_proceed} \rightarrow \mathsf{backup} \rightarrow \mathsf{eventE\_update} \rightarrow \mathsf{a''})
\end{array}
$$

Sequential functional composition. The first operator, Fun, we consider corresponds to a fully sequentialized functional composition of two aspects. When two advice 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 $advice_1[advice_2/\mathtt{proceed}]$. Furthermore, the (optional) action sequences before and after the proceed are correspondingly nested.

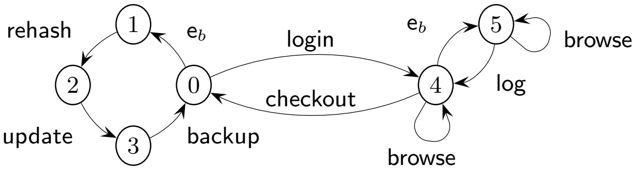
Reconsidering our example aspects, the advice of the consistency aspect of Section 2 applies to update events only during sessions, where it skips updates and adds a log. In contrast, the advice of the safety aspect applies both during and out of a session. So, as the composition Fun(Consistency,Safety) 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 advice 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:



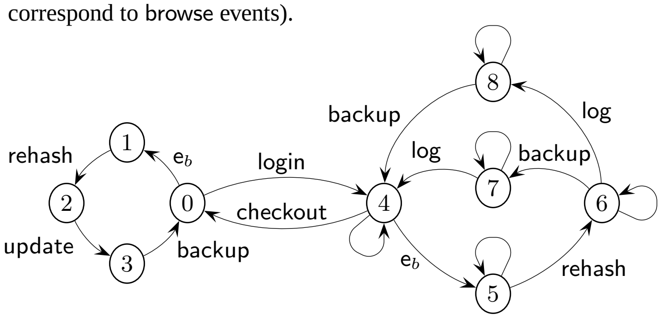
For the sake of clarity, we have hidden most synchronization events in the woven program, keeping only essential ones. Outside of session, only the safety aspect is woven (see the left hand side cycle). During sessions, as shown by the right hand side cycle, the consistency aspect is woven. It skips 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 advice 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 advice 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 advice are synchronized. Before (and after) skip or proceed, advice of the aspects are executed in parallel. The woven program is represented by the following automaton, where unlabeled loops correspond to browse events).



It makes clear that the advice 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 advice composed with ParOr proceed when at least one of them proceeds.

5. Implementation in Java

We have implemented a prototype of CEAOP for Java as a model-driven aspect-weaver. This weaver takes a base program written in Java, complemented with a description of the events of interest, and a composition of aspects and produces a concurrent program, which is correct with respect to the model by construction.

Each element of the the whole composition (base program, aspects, operators), implemented as an active object, can be seen as an LTS that progresses concurrently with the other LTSs. This progress is controlled by a monitor (a monitor in the general acceptation of the term, actually implemented as a passive Java object using the monitor associated to the object). The role of this monitor is to guarantee that the concurrent shared actions performed by the individual LTSs are performed in an atomic way, *i.e.*, that they strictly follow the semantics of the LTS synchronized product.

The weaver is responsible for instrumenting the base program so that it properly interacts with the monitor as well as for fully generating the active objects for the aspects and the operators. It does so by reusing Java building blocks provided as an `LTS` interface and a corresponding prototypical implementation as an `LTSImpl` class, which are used to interact with a generic monitor, implemented as an instance of the `Monitor` class.

In the following, we discuss the principles of the execution essential to get a behavior consistent with the model, we describe the above-mentioned building blocks and show how they are used by the aspect weaver to generate the application.

5.1 Principles of LTS execution

In this section we discuss the crucial point in executing LTSs: the sharing of actions between different processes and how to implement choices between shared actions.

In order to execute shared actions atomically, two barrier synchronizations are required. A first barrier stops the LTSs ready to perform a shared action until all the LTSs sharing the action are ready to proceed with the action. When all the LTSs have joined, a second *exit* barrier is used in order to make sure that all the LTSs have indeed performed the action before proceeding. Without this second barrier, some LTSs could proceed too fast and therefore perform new actions that would be wrongly interleaved with the execution of the current shared action.

Of course, all the LTSs composing the application are competing for performing the action their are interested in. Following a standard implementation of synchronization barriers, the monitor manages counters, one counter per shared action, recording which LTSs are ready to proceed with which action. As soon as a counter reaches its bound, the associated LTSs can proceed with the shared action.

In order to keep the counters coherent, new attempts to modify the counters are then blocked (we will say that the monitor is *busy*) until all the LTSs have exited the exit barrier, at which stage the related counter value is again zero. Useless notification and awakening of blocked LTSs are avoided by using the wait sets of the LTSs as well as the wait set of the monitor. After an LTS has incremented a counter of the monitor but is not ready yet to proceed, it waits for a notification on its own wait set. The last LTS reaching the barrier can then, via the counter, notify all the other LTSs that they can proceed. When the monitor is busy, the new LTSs ready to execute a shared action wait on the monitor wait set. They are notified that they can proceed by the last LTS leaving the exit barrier.

Finally, a last point to take care of is the case when an LTS has a choice of shared actions, which then depends on the progress of other LTSs. This happens, for instance, when the base program may proceed or skip, depending on the execution of an advice. In this case, the LTS increments several counters: the counters related to its choice of actions. If one of the counters reaches its bound, this means that the corresponding action is selected. The LTS decrements the counters that it may have just been incremented for other possible actions and notifies the other LTSs after having set their `selectedAction` instance variable (which plays the role of a condition variable). Before proceeding with the action, these other LTSs decrement the counters associated to the rejected actions of the choice.

5.2 Java building blocks

The Java implementation relies on three building blocks: classes representing the monitor supervising the execution, individual LTS processes, and actions. We now discuss these in turn.

5.2.1 The monitor

The monitor is implemented as a class `Monitor`. It includes an instance variable `selectedAction`, which is non-null when the monitor is busy (its value is the selected action), and a collection of counters. It provides the following methods (which are `public` and `synchronized`):

- `void register(LTS lts, List<Action> actions)` is used to initialize the system. Before starting its own thread, each LTS, seen as an object implementing the interface `LTS` (see below), has to register to the monitor with this method. The parameter `actions` corresponds to the alphabet of the LTS. It is used by the monitor to set up the counters.
- `void synchronizeOnEntry(List<Action> choice)` is called by an LTS ready to perform a choice. It starts with a guard requiring the monitor not be busy. It increments the appropriate counters. If one of the counter bounds is reached, the monitor is busy. The appropriate counters are decremented and the LTSs on the wait set of the selected counter are notified (their `selectedAction` instance variable is set to the selected action). A specialized version of this method with a single action as a parameter is used to deal with the cases when there is no choice (no counter decrementing is required).
- `void synchronizeOnExit()` is called when an LTS has performed its part of the shared action. The counter corresponding to the selected action is decremented. When it reaches zero, the monitor is no longer busy. The LTSs on its wait set are notified.

5.2.2 The `LTS` interface and the `LTSImpl` class

Each LTS, on top of being `Runnable` as an active object, implements the `LTS` interface, which includes the single method required by the monitor in order to interact with the LTSs: `void setAction(Action action)`. This method must be `synchronized`. It is used by the monitor to inform the LTS that it can proceed with the selected action `action`. It simply sets the `selectedAction` instance of the LTS and calls `notifyAll` to wake up the waiting LTS.

Its corresponding prototypical implementation, the `LTSImpl` class, is a basic implementation of an LTS based on a definition of the LTS by its alphabet (instance variable `alphabet`) and its transfer function, implemented as a combination of a hash map associating an action to an index (`actionMap`) and an array associating a (source) state and an index to a (target) state (`target`). `LTSImpl` can be subclassed to define alternative implementations, for instance to implement aspects and operators (see below). Some of its methods are used by the aspect weaver to instrument the base program:

- `void synchronizeOnEntry(List<Action> choice)` initializes `selectedAction` to `null`, tells the monitor about the choice by calling the monitor version of `synchronizeOnEntry`, waits for an action to be selected and finally decrements the appropriate counters. A specialized version of this method deals with single actions.
- `void synchronizeOnExit()` simply calls the monitor version of `synchronizeOnExit`.

Once registered to the monitor, an instance of `LTSImpl` repeatedly evaluates the action choice associated to the current state, calls its method `synchronizeOnEntry`, calls its method `void eval(int actionIndex)`, which, by default, change states depending on the selected action, and calls its method `synchronizeOnEntry`.

5.2.3 The `Action` class

An `Action` instance carries two pieces of information: a name and a value. This value is used to transmit the parameters of the `send` occurrence of an action to its `receive` occurrences. For instance, when an action of the base program (typically a method call) includes parameters, these parameters are made part of the corresponding `eventB` action so that they are available to the interested aspects. On the receiver's side, the action value is initially `null`, but when the action has been selected, the sender's action is passed back as the selected action to the receivers.

5.3 The aspect weaver

Finally, weaving requires the base program to be instrumented and the suitable code for the aspects and composition operators to be generated.

5.3.1 Instrumentation of the base program

We assume the availability of the source code of the base program, implemented as an active object, and of its actions of interest, separated into skippable and non-skippable events, described as AspectJ named pointcuts.

AspectJ is then used to turn the base program into an LTS as follows. First, inter-type declarations are used to add the LTS interface to the base class and complement this class with the necessary fields and methods taken from LTSImpl: monitor, selectedAction, alphabet, actionMap, setAction(), getActionIndex() (this returns the index of the selected action), register(), synchronizeOnEntry(), synchronizeOnExit(), and synchronize(), a combination of the two previous methods. Inter-type declarations are also used to add methods required for the instrumentation of the skippable events. For instance, here is the method used to instrument the update event in our example (we have used strings rather than proper actions in order to simplify the code), the pattern is always the same:

```
void instrumentedUpdate() {
  synchronize("eventB_update");
  synchronizeOnEntry(buildChoice("proceedB_update",
                                 "skipB_update"));
  if (getActionIndex() == 6) { // proceed
    synchronizeOnExit();
    synchronizeOnEntry("update");
    proceed();
    synchronizeOnExit();
    synchronize("proceedEnd");
    synchronize("eventEnd");
  } else {                     // skip
    synchronizeOnExit();
    synchronize("skipEnd");
    synchronize("eventEnd");
  }
}
```

Second, AspectJ advice is used to:

- Include a call to register before starting the thread associated to the active object (assuming that nothing else than the method start is used).
- Include a call to synchronizeOnEntry and synchronizeOnExit, respectively before and after each non-skippable event.
- Replace each skippable event with its instrumentation.

Alternatively, the same kind of transformation can be performed on bytecode using a tool able to conveniently perform both structural and behavioral transformations such as Reflex [19, 18].

5.4 Generation of the aspects and operators

The aspects are essentially described as FSPs but with a very simple Java syntax, which makes it possible to add parameters to actions for parameter passing purposes. Here is how the consistency aspect looks like (blocks are used to denote choices):

```
Aspect Consistency {
  void consistency() {
    login(); consistency1();
  }
  void consistency1() {
    {update(); skip(); log();}
    {checkout(); consistency()}
  }
}
```

The basic idea consists of subclassing LTSImpl while providing the proper alphabet and transfer function computed from the aspect FSP. The method eval() makes it possible to perform the advice. Here is how the consistency aspect looks like (-2 is used to complete the target array for the pairs (source state, action) which do not appear in the automaton):

```
public class Consistency extends LTS {
  public Consistency() {
    super("Consistency");
    String [] actions =
      {"login", "checkout", "browse", "update",
       "eB", "eE", "pB", "pE", "sB", "sE", "log"};
    setActions(actions);
    int [][] target =
    //   l  c  b  u eB eE pB pE sB sE log
      {{ 4, 0, 0,-2, 1,-2,-2,-2,-2,-2,-2}, // state 0
       {-2,-2,-2,-2,-2,-2, 2,-2, 2,-2,-2}, // state 1
       {-2,-2,-2,-2,-2,-2,-2, 3,-2,-2,-2}, // state 2
       {-2,-2,-2,-2,-2, 0,-2,-2,-2,-2,-2}, // state 3
       { 4, 0, 4,-2, 5,-2,-2,-2,-2,-2,-2}, // state 4
       {-2,-2,-2,-2,-2,-2,-2,-2, 6,-2,-2}, // state 5
       {-2,-2,-2,-2,-2,-2,-2,-2,-2, 7,-2}, // state 6
       {-2,-2,-2,-2,-2,-2,-2,-2,-2,-2, 8}, // state 7
       {-2,-2,-2,-2,-2, 4,-2,-2,-2,-2,-2}, // state 8
      };
    setTarget(target);
    }
    protected void eval(int actionIndex) {
if (actionIndex == 10)
  log();
        super.eval(actionIndex);
    }
}
```

The principle is the same for the operators.

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: advice 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 (advice must create threads explicitly). Advice 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 advice 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 [16], *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 [9], 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 [17] 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.

There have been several approaches using regular expressions for the specification of concurrent systems, *e.g.,* path expressions [6] nad concurrent regular expressions [13]. However, these approaches have not considered problems specific to synchronization in an AO setting. For instance, contrary to that work our composition operators provide means for the synchronization of modifications, *i.e.,* synchronization of an advice, in terms of the structure of that advice. Furthermore, an AO setting is different in that advice may introduce new events which are themselves relevant for synchronization.

Finally, aspects have been considered as a way to implement coordination [7, 8]. 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 advice 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, means for the synchronization of complex systems have to be investigated. Moreover, 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. A first target are optimisations of our Java implementation by partially evaluating the monitor interactions with respect to aspect definitions in order generate more efficient code.

Acknowledgements. The work on the implementation has benefited of fruitful discussions with Sebastian Pavel, Jean-Claude Royer and Angel Nuñez.

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] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In E. Gelenbe and C. Kaiser, editors, *Int. Symp. on Operating Systems*, volume 16 of *LNCS*, pages 89–102. Springer-Verlag, 1974.

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

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

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

[10] 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.

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

[12] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proc. of the 18th IEEE Int. Conf. on Automated Software Engineering (ASE'03)*, pages 152–163. IEEE Computer Society, 2003.

[13] V. K. Garg. Modeling of distributed systems by concurrent regular expressions. In *2nd Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols*, Vancouver, Canada, Dec. 1989.

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

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

[16] 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).

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

[18] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In R. Glück and M. Lowry, editors, *Proceedings of GPCE'05*, LNCS 3676, pages 173–188, Tallinn, Estonia, Sept./Oct. 2005.

[19] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proc. of OOPSLA 2003*, pages 27–46. ACM Press, Oct. 2003.

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

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

Figure 2. The consistency aspect in FSP

```
Server = login → InSession
    | eventB_update →
        ( proceedB_update → update → proceedE_update → eventE_update → Server
        | skipB_update → skipE_update → eventE_update → Server) ,

InSession = checkout → Server
    | eventB_update →
        ( proceedB_update → update → proceedE_update → eventE_update → InSession
        | skipB_update → skipE_update → eventE_update → InSession) ,
    | browse → InSession.
```

Figure 3. Instrumented base program in FSP

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

$$\begin{array}{ll}
\mathcal{T}_B(\mu b.(\Box_i\, e_i; B_i)) & \triangleq b = \Big|_i (\mathcal{T}'_B(e_i; B_i)) \text{ , } \mathcal{T}_B(B_1) \text{ , .. , } \mathcal{T}_B(B_n)\text{ .} \\
\mathcal{T}_B(b) & \triangleq \epsilon
\end{array}$$

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

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

Figure 5. Abstract base program syntax and instrumentation

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

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

Figure 6. Aspect syntax and their transformation into FSP

$$
\begin{array}{lcl}
\mathcal{T}_A(\mu a.(\Box_i\ e_i S_i; A_i)) & \triangleq & a\text{=} \quad \big|_i (\mathcal{T}'_A(e_i S_i; A_i) \ \mid \ \big|_{e \in \mathcal{E} \cup \mathcal{S} \setminus (\bigcup_i e_i)} loop(a,e), \\
 & & \qquad \mathcal{T}_A(A_1),\ ..\ ,\ \mathcal{T}_A(A_n). \\
\mathcal{T}_A(a) & \triangleq & \epsilon
\end{array}
$$

$$
\begin{array}{lcll}
loop(a,e) & \triangleq & e \rightarrow a & \text{if } e \in \mathcal{E} \\
loop(a,e) & \triangleq & \mathsf{eventB\_}e \rightarrow \mathsf{proceedB\_}e \rightarrow \mathsf{proceedE\_}e \rightarrow \mathsf{eventE\_}e \rightarrow a & \text{if } e \in \mathcal{S}
\end{array}
$$

Figure 7. Translation with waiting loops

```
||FunArg1 = a/{call_e/proceedB_e, ret_e/proceedE_e, skipB_e1/skipB_e, skipE_e1/skipE_e}.

||FunArg2 = a"/{call_e/eventB_e, ret_e/eventE_e, skipB_e2/skipB_e, skipE_e2/skipE_e}.

 Fun      = ( skipB_e1 →  skipB_e →  skipE_e →  skipE_e1 →  Fun
            | skipB_e2 →  skipB_e →  skipE_e →  skipE_e2 →  Fun).
```

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

```
||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