Towards a model of concurrent AOP[†]

Rémi Douence, Didier Le Botlan, Jacques Noyé, Mario Südholt
OBASCO project
École des Mines de Nantes - INRIA, LINA
4 rue Alfred Kastler
44307 Nantes cedex 3, France
{douence,lebotlan,noye,sudholt}@emn.fr

ABSTRACT

Aspect-Oriented Programming (AOP) [1, 15] is concerned with the modularization of crosscutting functionalities. Such functionalities are problematic, in particular, for the development of concurrent applications, such as graphical user interfaces or multithreaded server applications. However, few approaches address this problem: there is, in particular, no general model for concurrent AOP that enables coordination among concurrently-executing aspects as well as coordination of concurrent aspects and base applications.

In this paper, we discuss general requirements for such models, briefly present a specific instance meeting these requirements, and propose a set of general composition operators for the construction of concurrent applications using concurrently executing advice.

1. INTRODUCTION

Aspect-Oriented Programming (AOP) [1, 15] is concerned with the modularization of so-called crosscutting functionalities, which cannot be reasonably modularized using traditional programming means, such as objects and components. Crosscutting concerns constitute, in particular, an important problem for many concurrent applications. Request handling in web servers, event handling in graphical user interfaces, monitoring, debugging, and coordination of concurrent activities, for example, have been identified as being crosscutting in many concurrent applications.

Up to now a large number of approaches for AOP of sequential programs have been proposed, most notably AspectJ [4]. Using these systems, concurrency can be introduced and controlled by exploiting existing libraries for concurrent programming, *e.g.,* Java's thread library. These approaches can therefore be characterized as providing a model of aspects for concurrent OO applications. By contrast, there are currently no AOP models 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, *i.e.,* there are no models for *concurrent AOP*.

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. This concerns, in particular, coordination of aspects and the base program in a concurrent setting, coordination of the different parts of advice, and coordination of several advices that apply at the same execution point.

In the following we discuss requirements for a general model of concurrent AOP in Sect. 2). In Sect. 3 we briefly present Concurrent EAOP (CEAOP) [11], a first instance of such a model we are currently developing, based on an initial insight discussed in [12]. Sect. 4 presents a family of general composition operators for concurrent AOP that we have formally examined within the CEAOP model. In Sect. 5 we discuss related work, followed by a conclusion and discussion of future work.

2. REQUIREMENTS FOR A MODEL OF CONCURRENT AOP

The main assumption underlying our quest of a model for concurrent AOP is that AOP uses program structures and execution patterns specific enough to require specially-tailored support for introducing and controlling the execution of concurrent AO applications. As two examples for such specifics, consider that (i) long-running calculations that may be introduced using advice: advice should therefore be built from concurrent entities rather than be treated as one sequential entity which is injected in a concurrent application and (ii) the generally difficult problem of ensuring or even verifying the correctness of concurrent AO applications is made worse by aspects due to their crosscutting nature and oblivious manner of application (the base program is not aware of aspects being applied at some later stage).

To make this discussion (as well as later ones) more concrete let us consider a small example from the e-commerce domain. The base application simply lets users log (*i.e.,* login) into a web shop, perform business transactions (which may consist of, *e.g.,* browsing the catalogue, selecting products to be bought, canceling or confirming purchases), and leave the shop after having checked out. When an administrator concurrently modifies the database, a safety aspect could be used to support fault tolerance. Fig. 1 represents a suitable aspect definition (The use of boxed states and dotted transitions in such definitions is intended to visually indicate that the automaton defines an aspect and not a base program, in particular, that transitions may be annotated with

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

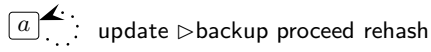

Figure 1: Safety Aspect

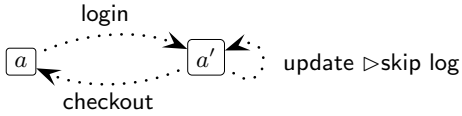

Figure 2: Consistency Aspect

rules of the form event ▷action.) The safety aspect matches occurrences of database update events and applies around advice which first backups the database, performs the update (proceed) and finally rehashes some indexes to speed up database accesses. Note that we pictured the automaton-like representation using dotted transitions: these indicate that the aspect waits for update events and discards other events during this process.

As a second example scenario, consider that database updates should be canceled within sessions of certain users in order to ensure consistent presentation of pricing information. Concretely, consistency could be ensured by delaying modifications involving products in the database affecting the current user's transaction. A suitable aspect is shown in Figure 2. The use of the event skip, which occurs in the advice during sessions (between login and checkout), causes the suppression of the matched base action[1]; additionally the advice records this action by a log event. Within a session, the consistency aspect does not define advice for updates, which are therefore performed. Note that the consistency aspect is stateful [9, 10], because the advice triggered by update depends on the state of the aspect.

These examples scenarios can be used to illustrate four basic requirements for models for concurrent AOP we have investigated:

1. *Aspects should be concurrent entities* (or at least be built from) which can be coordinated with other aspects and the concurrent base program.

   For instance, the different actions of the advice used in the safety aspect above, backup, update (executed through proceed and rehash) can all be very costly: execution speed could be improved by rehashing the database concurrently to the base program execution once the database has been updated. Furthermore, stateful aspects, such as the consistency aspect above, enable concise aspect definitions but should be flexibly scheduled in a concurrent setting, *e.g.,* in particular, in order to allow control over different aspects applying at the same time.

2. Concurrent aspects should be built and applied to base programs using *composition operators* (*e.g.,* to facilitate understanding and reasoning over concurrent AO programs).

   For instance, if the safety and consistency aspects above are both to be applied, the former should probably have priority: in the example, this would ensure that backups are still performed even if the update itself is delayed. This should be expressed using some suitable prioritizing operator instead of hard wiring them according to specific concurrent contexts.

3. Concurrent AO programs should be amenable to automatic or semi-automatic verification techniques.

4. The model should be readily implementable in mainstream programming languages.

In the following, we discuss these requirements in some detail.

2.1 Concurrent entities

Aspects should provide the following support for concurrent executions.

First, aspects are most frequently defined through matching of "pointcuts" on base program executions and executing "advice(s)" when a match occurs, thus modifying the base program execution. Stateful aspects, such as the consistency aspect of Fig. 2, which allow the explicit representation of relations between execution events, are very useful for coordinating aspects and base programs, and provide information crucial for the verification of AO programs. (Note that stateful aspects are only supported in AspectJ through cflow pointcuts.) A model for concurrent aspects should therefore support stateful pointcuts and aspects.

Second, an advice (in this paper we only consider "around" advices) can typically be structured into three parts $b\ c\ a$, where b and a denote an optional sequence of before-actions and after-actions, respectively, and $c \in \{\mathsf{proceed}, \mathsf{skip}\}$ is a mandatory control action. In case of a single applicable advice, proceed triggers execution of the $a$dvised action, *i.e.,* the base program action at the point when the pointcut matched, whereas skip ignores the advised action, which will not be executed. A model for concurrent aspects should make it possible to synchronize or not between the base-program execution and these three different parts.

Third, in case of multiple concurrently-executing aspects triggering execution of multiple advices at one point during base program execution, proceed may trigger the execution of another advice instead of the base action. A model for concurrent aspects should allow defining partial orderings on the advices in such cases and thus define which can execute concurrently and which parts should be synchronized.

2.2 Composition operators

AOP approaches typically provide a translation mechanism that "weaves" aspects into base programs. In a concurrent setting this would correspond to a monolithic weaving strategy yielding a concurrently executing program. In order to

[1]The reader may be surprised that the absence of a call of base functionality is marked explicitly by skip instead of being simply represented, *e.g.,* as in AspectJ, by the absence of proceed in advice: explicit skips are needed, however, to correctly synchronize parts of advice in a concurrent setting.

ease construction of concurrent AO programs and especially support construction of correct programs (cf. the following requirement), a model for concurrent aspects should support a compositional approach featuring generic operators that can be used to weave prefabricated aspects in different ways.

2.3 Correctness

Building correct concurrent programs is generally a very hard task. Unfortunately, aspects introduce an additional (potential) correctness problem because the effect of the crosscutting concern, defined through an aspect, or even a composition of aspects, separately from the base program, may be difficult to understand. Tool support, if possible automatic, is therefore even more important in the case of concurrent aspects.

2.4 Implementation

A model of concurrent aspects should support reasonably efficient implementations and be sufficiently simple in its application. While this is relatively simple to achieve based on explicit support for process calculi, efficient and simple-to-use implementation in mainstream languages, such as Java, should also be supported.

3. CONCURRENT EVENT-BASED AOP

Our endeavor to formally define concurrent aspects resulted in a formal model based on finite-state automata [11] using the FSP language [16]. In this model, called CEAOP, aspects are stateful. CEAOP provides a simple aspect language that allows aspects to be defined concisely but also admit an equivalent graphical representation, which has been used in Section 2.

Aspects are encoded into FSP automata by introducing synchronization events that allow the coordination of aspects and the base program. More precisely, four events are inserted within each advice for a base event (say update), which triggers the advice: update_begin (begin of the advice associated to update), proceed_begin (begin of the `proceed`), proceed_end (end of the `proceed`) and update_end (end of the advice). When the advice skips the execution of the base-program operation, proceed_begin and proceed_end are replaced by skip_begin and skip_end, respectively. Both events are mandatory to implement a two steps rendez-vous when we consider more than one aspect. When there is a single aspect, these two events could be merged. The aspects introduced in the previous section (Figures 1 and 2) are encoded into the automata using this instrumentation as shown in Figures 3 and 4. These figures thus show automata representing regular FSP automata (as displayed, *e.g.,* by the LTSA tool [16]).

The weaving of an aspect into a base program is modeled as the parallel composition of the instrumented aspect with the base program. In order to weave more than one aspect with a concurrent base application, CEAOP defines several composition operators that combine two aspects and a base program, see the following section, where these composition operators are generalized.

The CEAOP model meets the four basic requirements of a

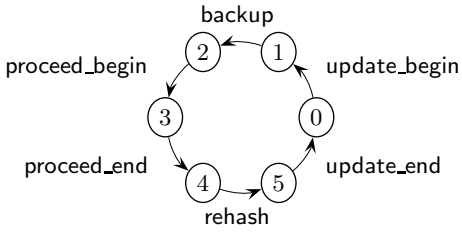

Figure 3: Safety Aspect

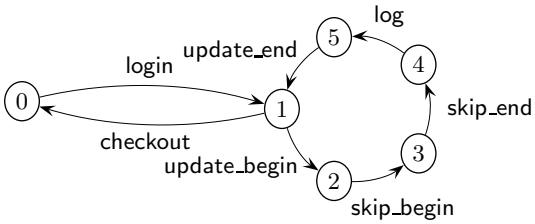

Figure 4: Consistency Aspect

model for concurrent AOP introduced in the previous section:

1. CEAOP supports stateful aspects and concurrent coordination of parts of aspects with other aspects and concurrent base programs.
2. It supports a first set of concurrent composition operators (which can be generalized as shown in the next section).
3. The model is supported by existing tools (LTSA [16]) that allow woven programs to be model-checked.
4. We have implemented a light-weight Java library realizing the CEAOP model [11].

4. COMPOSITION OF CONCURRENT ASPECTS

Interactions between aspects are a usual issue when applying several aspects to the same base program. Most existing work rely on an ordering of aspects to decide which should execute first. In this section, we consider more expressive composition schemes for concurrent aspects following our previous work on operator-based interaction resolution in the sequential case [9, 10].

4.1 A notation for advice composition

In order to more easily describe the combination of several advices, we slightly extend the previously introduced notation. The piece of advice executed before the proceed or skip statement is denoted b_i, where i is the identifier of the advice. The command proceed or skip, *i.e.,* the *control* statement of the advice, is written c_i. The piece of advice executed after the control statement is denoted a_i. Note that, in each execution trace, each advice application must introduce only one control statement. This does not

necessarily mean that an advice should contain only one skip or proceed. For example, assuming an if statement in the advice language, the following advice would be correct: ▷ if (*cond*) then proceed else skip.

Sequences of actions are written with a semi-colon: $a_1; a_2$, indicating that a_1 executes first, then a_2. Actions that are executed in parallel are written with a parallel operator: $a_1 \parallel a_2$, indicating that a_1 and a_2 run concurrently. As an example, $(a_1 \parallel a_2); a_3$ is an advice that first runs both a_1 and a_2 in parallel, until both have finished, and that runs a_3 next.

Despite the informal flavour of this presentation, the resulting expressions can be given formal semantics within the CEAOP model. In that sense, this notation is only a convenient way to describe synchronization of processes, which is otherwise formalized using CEAOP's finite-state automata.

By convention, we identify in boolean contexts proceed with true and skip with false, so that we may apply boolean operators to them. In particular, ¬proceed is skip. If e is an expression, such as $(a_1; a_2)$, and c a control statement, we define $c \times e$ as e if c is proceed and as an empty sequence if c is skip. As a generalization, we define $c\ ?\ e_1 : e_2$ as e_1 if c is proceed and as e_2 if c is skip.

4.2 Examples of composition operators

We illustrate how to use the above notations by defining the ParAnd ("Parallel And") operator, one of the operators originally defined using CEAOP in terms of renamings and FSP automata [11]. Consider two advices A_1 and A_2, both of the form $b_i; c_i; a_i$. The ParAnd operator applied to these advices produces a composite advice of the form

$$\mathsf{ParAnd}(A_1, A_2) = (b_1 \parallel b_2); c_1 \wedge c_2; (a_1 \parallel a_2)$$

This means that both before-advices b_1 and b_2 run in parallel until they reach their control statement (c_1 and c_2, respectively). Then, the base program proceeds if and only if both control statements are proceed. Otherwise, the base program skips. Once the control statement is finished (which may take time in case of proceed), the after-advices are executed in parallel. When both have finished, the base program is resumed. Similarly, the ParOr operator, another CEAOP operator, is defined as follows:

$$\mathsf{ParOr}(A_1, A_2) = (b_1 \parallel b_2); c_1 \vee c_2; (a_1 \parallel a_2)$$

The Fun operator, the third CEAOP operator, for sequentialized scheduling can be defined as follows:

$$\mathsf{Fun}(A_1, A_2) = b_1; c_1 \times (b_2; c_2; a_2); a_1$$

Unlike ParAnd and ParOr, Fun is not commutative: the first advice decides (through c_1 which is either proceed or skip) whether the second advice should execute. Intuitively, $\mathsf{Fun}(A_1, A_2)$ is also equivalent to $A_1\{A_2/\mathsf{proceed}\}$, that is, the advice A_1 in which proceed is replaced by A_2.

Note that these composition operators are defined over advices, not aspects. In order to define composition of two aspects, it suffices to apply the composition operator to their matching advices, pairwise. Actually, our model allows for a finer-grained composition by specifying, for instance, that advices occurring within a session should be combined with ParAnd, while those occurring outside of a session should be combined with Fun.

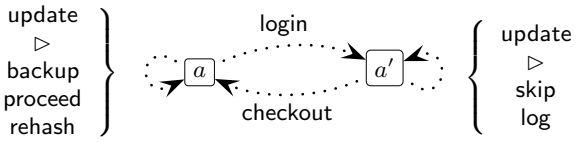

Figure 5: Consistency aspect and safety aspect composed with Fun

As an example, Figure 5 shows the aspect obtained by composing the consistency aspect and the safety aspect of Section 2 with Fun. More precisely, we represent an aspect equivalent to the composition $\mathsf{Fun}(consistency, safety)$. Since the consistency aspect is the first argument, its advice is always applied first. In state a, it proceeds, therefore the advice of the second aspect is inserted (▷ backup proceed rehash). In state a', the advice is ▷ skip log. Since it skips, the second aspect is not applied.

Note that all these examples suspend the base program to run the advices. We propose below a variant of ParAnd that resumes the base program as soon as the before advices are finished. The after advices are run in parallel with the base program.

$$\mathsf{ParAnd'}(A_1, A_2) = (b_1 \parallel b_2); (c_1 \wedge c_2) \parallel a_1 \parallel a_2$$

4.3 Families of general operators

The examples ParAnd and ParOr immediately suggest a family of operators of the form Par_f for any binary boolean operator f. This family can be characterized as follows: let A_1 and A_2 be two advices $b_1; c_1; a_1$ and $b_2; c_2; a_2$, respectively, and f be a binary boolean operator, then we define $\|_f$ as

$$\|_f\ (A_1, A_2) \quad = \quad (b_1 \parallel b_2); f(c_1, c_2); (a_1 \parallel a_2)$$

As expected, ParAnd equals $\|_\wedge$ and ParOr equals $\|_\vee$.

We may then wonder if other families of interesting composition operators can be found. To begin with, we identify a few constraints that are likely to be necessary for composition operators to make sense:

- *before* pieces of advice should occur before the control statement.
- *after* pieces of advice should occur after the control statement.
- If the *before* advice is executed, its associated *after* advice should be executed too.
- If the *before* advice is not executed, its associated *after* advice should not be executed.

Another family that satisfies these requirements is inspired by Fun. It is parameterized by three boolean functions f, g, and h (making 4^3 potential operators).

$$\mathsf{Fun}_{f,g,h}(A_1, A_2) \ = \ b_1; \big(f(c_1)\ ?\ (b_2; g(c_2); a_2) : h(c_1)\big); a_1$$

The advice $\mathsf{Fun}_{f,g,h}(A_1, A_2)$ executes the before part of A_1, then either it goes on with the second advice if $f(c_1)$ is proceed, or it skips the second advice and performs $h(c_1)$ if $f(c_1)$ is skip. In both cases, the after advice a_1 is executed. When the second advice is triggered, it is executed normally, except for its control statement, that is transformed by g.

Writing id for the identity, Fun is defined as $\mathsf{Fun}_{id,id,id}$. Writing $\neg$ for the boolean negation operator and 0 for the constant function equal to skip, we may define $\mathsf{Fun}_{\neg,0,id}$, which executes the second aspect only if the first one skips. If the first aspect proceeds, the base program immediately proceeds without invoking the second aspect. Such a composition is useful to define an aspect that is guaranteed to perform logging every time a given operation is skipped.

4.4 Properties of composition operators

By studying fundamental properties of composition operators, we pave the way towards a more expressive composition language (and model) and towards automated analyses of aspects.

As an example, using LTSA, it is possible to check formally that a given operator is associative. This test is useful since any associative binary operator can be easily extended to n arguments (using the obvious definition $\mathsf{op}(a,b,c) = \mathsf{op}(a,\mathsf{op}(b,c)) = \mathsf{op}(\mathsf{op}(a,b),c)$), thus making it easier to compose many aspects. We have used this technique to check that, as expected, ParAnd is associative, whereas ParNand (where control statements are combined with a Nand operator) is not. The same approach can be used to check algebraic properties of more involved operators, thus providing formal foundations for these operators and facilitating intuitive reasoning.

Analyses can also be performed on the model of the woven program. For instance, one may automatically check that no call to update ever occurs within a session.

5. RELATED WORK

Aspects have been considered as a way to implement coordination [13, 6, 7]. Note that we take here a different point of view: the aspects we consider are basic reusable components whose coordination is specified by the aspect language itself, including the composition operators, and its underlying semantics.

There are many proposals for AOP, but little work devoted to concurrent AOP. So, concurrency is mostly manually programmed in the base language. In particular, 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]. 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). Benavides *et al.* introduce AWED [5], an aspect language for distributed programming, which includes regular sequence aspects. Concurrent execution is supported at the language level (i) by pointcuts referring to threads similar to tracematches but also (ii) by remote advices 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 [17], *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. The work of Katz [14] proposes a control structure to interleave (*i.e.,* synchronously weave) extra operations with those of the processes of distributed applications. The work of Sihman and Katz [18] explores composition issues. But they do not consider specific aspect constructs such as `proceed` that changes the overall picture and can lead to a rich set of composition operators.

6. CONCLUSION

In this paper, we have motivated that a general model for concurrent AOP should satisfy new requirements not considered by the current body of work on AO concurrent programming. We have motivated four basic such requirements — namely aspects as concurrent entities, composition operators for concurrent aspects, tool support for automatic verification, and implementation in mainstream languages — and briefly presented the CEAOP model, which meets these four requirements. Finally, we have presented a new, more intuitive, definition of CEAOP's composition operators and shown how these give rise to families of composition operators.

As future work, we plan to investigate how concurrent aspects can be more flexibly coordinated than currently supported by CEAOP's instrumentation technique. Furthermore, we intend to study the preservation of properties (such as liveness or safety properties) through composition opera-

tors. Such a result is essential since it could guarantee that a given property enforced by a given aspect is guaranteed to be preserved through adequate composition with other aspects.

7. REFERENCES

[1] Mehmet Akşit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors. *Aspect-Oriented Software Development*. Addison-Wesley Professional, September 2004.

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of OOPSLA'05*. ACM Press, 2005.

[3] James 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] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *Proceedings of AOSD'06*. ACM Press, 2006. To appear.

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

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

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

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

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

[11] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. Technical report, INRIA, March 2006.

[12] Rémi Douence and Jacques Noyé. Towards a concurrent model of event-based aspect-oriented programming. In *European Interactive Workshop on Aspects in Software (EIWAS 2005)*, Brussels, Belgium, September 2005.

[13] David Holmes, James Noble, and John Potter. Aspects of synchronization. In *Proc. of the AOP WS at ECOOP'97*.

[14] Shmuel Katz. A superimposition control construct for distributed systems. *TOPLAS*, 15(2):337–356, 1993.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, et al. Aspect-oriented programming. In *Proceedings of ECOOP'97*, LNCS 1241, 1997.

[16] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java*. Wiley, 1999.

[17] Jacques Malenfant and Simon 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).

[18] Marcelo Sihman and Shmuel Katz. A calculus of superimpositions for distributed systems. In *Proceedings of AOSD'02*. ACM Press, 2002.