Brenda: Towards a Composition Framework for Non-orthogonal Non-functional Properties

Mikaël Beauvois

France Télécom R&D
Distributed Systems Architecture Department
28 Chemin du Vieux Chêne, BP 98 38243 Meylan, France
mikael.beauvois@rd.francetelecom.com

Abstract. Software infrastructures have to manage distributed and heterogenous entities making them more and more complex. To abstract complexity, the concept of technical services has been introduced, also called non functional properties (functional properties are related to business level) to implement functions related to security, mobility, transactional behaviour, etc. The main challenge is the composition of non functional properties to build these infrastructures.
Our goal is to identify the right abstractions which enable the composition of non functional properties when they are non orthogonal.
In this paper, we analyse how this issue is tackled in AOP (Aspect Oriented Programming) where composition of non functional properties is a composition of aspects. Then, we study a new approach where compositions of non functional properties are compositions of components and automata and show how we have implemented these concepts in a composition framework.

1 Introduction

Composition of non orthogonal non functional properties appears when two or more computations are activated at the same moment and some execution orders between these computations have to be respected.

In this paper, we try to show that the composition of non orthogonal non functional properties must have some formal basis to be resolved [9][4]. Our contribution is an identification of some composition concepts and their implementation in a Java composition framework.

This issue can be found both in industry like execution supports used in application servers (EJB containers [15]), and academic research like Separation of Concerns techniques (e.g. AOP (Aspect Oriented Programming) [10], Composition Filters [5], etc) where non functional properties are defined as *concerns* (*aspects* in AOP, etc). The building process of execution supports used in application servers (such as EJB servers) is a manual composition of technical services, where all these services are statically mixed. The AOP paradigm captures non functional properties as concerns at the design and weaves them in business applications.

J.-B. Stefani, I. Demeure, and D. Hagimont (Eds.): DAIS 2003, LNCS 2893, pp. 29–40, 2003.
© IFIP International Federation for Information Processing 2003

We first describe how this issue is tackled in the AOP research domain (section 3). We then present a new approach to resolve the composition of non orthogonal non functional properties (section 4). This approach is based on components architecture enhanced with behavioural composition. It introduces a behaviour model and a component model, both detailed in sections 5 and 6. Finally, we describe the framework, called Brenda, that implements these models. Our approach is illustrated and motivated throughout the paper by an example defined in the next section.

2 Example

In our example, we consider the following non functional properties, that have to be composed:

1. *persistence* is implemented by a *storage manager* which loads compressed data of components and decompresses them before assigning values to business component,
2. *security* is implemented by a *security manager* which checks if authenticated users can execute a business method and logs all operations, and an *encryption manager* which encrypts business component data before sending them over the network.

The above managers are separate components that can be designed and run independently. For example, (1) the storage manager can load data without checking the user-permissions; it doesn't encrypt loaded data; (2) the security manager doesn't load or encrypt any data; (3) the encryption manager encrypts all data of business components without knowing which data are stored. The above managers are structurally independent. This example will be used in what follows.

3 Non-orthogonal Aspects in AOP Paradigm

The non orthogonal composition of non functional properties is addressed in the AOP (Aspect Oriented Programming) paradigm as a composition of non orthogonal aspects when several aspects are defined and activated on the same joinpoint. An execution order between those aspects has to be defined. A joinpoint is the realization of the crosscut on an application target.

The AOP paradigm focused on how to define aspects to apply them in applications, and how to compose them especially when they are not orthogonal. AOP implementations such as AspectJ [2], AspectIX [1], Jac [12], etc are more focused on how to define aspects to apply them in applications than investigating how aspects interact with each other during composition. Some derived concepts like events in EAOP [8] are only introduced to express more sophisticated crosscuts.

3.1 Example in AOP

In the AOP approach, the focus is on the activation of concerns implemented by storage, security and encryption managers: *when* (business method call, field

access, etc) and *what* (available manager functions) we have to use. In the example, these concerns are used on all public business methods calls (more precisely before method invocation) of class BusinessObject: they are non orthogonal due to activation on the same joinpoints when they are weaved together. Using the AspectJ tool [2], implementations are:

```
aspect StorageAspect {
    pointcut businessmethodcalls(BusinessObject bo) :
        target(bo) && call(public * * (..));
    before(BusinessObject bo) : businessmethodcalls(bo) {
        if (!bo.isLoaded) load(bo);
    }
...}

aspect SecurityAspect {
    before(BusinessObject bo) : businessmethodcalls(bo) {
        checkPermissions(bo);}
    private void checkPermissions (BusinessObject bo)
        throws SecurityException {
        ...
        log.notify(''Access to object owned by ''
            +bo.getOwner());
    }
...}

aspect EncryptionAspect {
    before(BusinessObject bo) : businessmethodcalls(bo) {
        cryptAndSend(bo);}
...}
```

The weaving process composes these three aspects (StorageAspect, SecurityAspect, EncryptionAspect): a conflict is raised because these three aspects are activated on the same joinpoint. Their executions have to be ordered.

3.2 Drawbacks

The following drawbacks focus on the AOP approach and not on specific implementation details (dynamic addition of aspects, event based AOP, etc).

In the AOP approach, the ordering of aspects on a joinpoint introduced in [2][12] is an interesting idea when very little well-known well-defined aspects are considered, but cannot be generalized. Our work is focused on aspect granularity which is coarse in the AOP approach. In the example, when the security aspect checks execution permissions, it logs the operation made by the business component with some business component data. Thus the data have to be loaded and decompressed before logging and after checking permissions. It appears these aspects cannot be composed together using AOP even if an execution order is defined. This example shows aspects composition is related to the *interweaving of execution sequences*. Defining aspects as crosscut definitions sets limits to the composition capabilities.

Implementing concerns in aspects is not well defined and raises several questions: should a concern be captured by in one or more aspects? Should aspects be implemented first and joinpoints defined afterwards or the converse?, etc. It seems too difficult to implement a complex concern in a single aspect: the aspect implementation is related to aspect crosscut definition.

Aspects are seen at the same level: if a concern is addressed by several aspects, we think all aspects of a same concern have the same importance in the weaving. This leads to a composition like an ordered list of aspects which is a particular case of composition.

In the rest of the paper, we propose solutions to resolve these drawbacks. This work has two main objectives: the first is to give a formal description of the issue of composing non orthogonal aspects in the AOP paradigm, and implement this in a component framework approach. The second is to orientate the AOP research domain towards the composition of aspects, and, more generally, of technical services.

4 Description of Proposal

The goal is to resolve the composition of non orthogonal non functional properties. This issue appears in several domains from techniques of "Separation of Concerns" (AOP, composition filters, etc) to building execution support for application servers such as EJB servers. The solution has to be independent of the many concepts introduced in these domains, but should be generic enough to be integrated in any domain.

Composing concerns is related to interweaving execution sequences. Our solution gives a support to define composition constraints applied to the interweaving of execution sequences when composing non orthogonal concerns. Some computations of concerns are conflictual: in the example, the log operation is conflictual with the decompress operation, because log has to be executed after decompress. An operation of a concern is an elementary specification of a computation (a method call like log or decompress). Our approach focuses on interactions between concerns when they are non orthogonal. These interactions are introduced according to the temporal order relations between conflictual operations.

The result of the composition of concerns is a set of interweavings of the execution sequences of the concerns. Concerns are non orthogonal if there exists one or more execution sequences where the execution order of their conflictual operations is not respected. Composition of non orthogonal concerns consists in defining a set of their execution sequences where the temporal order relations between conflictual operations are respected. The definition of this set cannot be statically processed because of a combinatorial explosion of solutions. Thus, we choose a dynamical processing, i.e., the generation of a scheduler from a scheduling policy (the composition of temporal order relations between conflictual operations).

We deal with execution sequences, so in order to model the behaviour of the concerns, a behavioural description model based on automata is introduced. A component-based approach was chosen because of its aim to define and apply semantics to compose together entities called components. These semantics can be expressed in our approach by scheduling constraints that the conflictual operations have to follow. Concerns, defined as technical services in industrial infrastructures or as aspects in the AOP approach, are defined as *reactive*

components in our approach. Thus a concern is a set of reactive components. The temporal order relations are expressed by scheduling constraints that we call composition constraints. They have a particular behavioural semantics expressed using an automaton-based model. Scheduling constraint automata can only react to events coming from changes of the component automata that are impacted by the scheduling constraint. In this model, scheduling constraints are also reactive components, so to apply them to a composition of reactive components represented by a composite, they are added into composite components like any another reactive component.

Non orthogonal concerns are structure independent, but behaviour dependent: they interact according signals initiated by scheduling constraints that define temporal order relations. To sum up, weaving non orthogonal aspects in the AOP approach is, in our approach, a composition process of behaviours of reactive components, including technical components and scheduling constraints. The composition leads to an interweaving of the execution sequences of concerns that respect temporal order relations between conflictual operations.

Our approach resolves AOP drawbacks:

- Granularity: we focus on fine-grained operations and not on coarse-grained concerns;
- Concern implementation is directed by a component approach that comprises abstractions such as composites and explicit bindings that helps designing concerns;
- Concern level: scheduling constraints can be applied to any component: they can be defined either on the root level, or in a composite component.

5 Models

To implement the approach described above, several models must be defined. One must be able to express the behaviour of component, i.e., their possible execution sequences. A model is therefore defined to describe the behaviour of components. The proposed model is an extension of hierarchical FSM (finite state machines).

5.1 Reactive Component Model

At the behavioural level, components are seen as reactive components. These components are reactive because when they receive signals on their interfaces, their automata react to these signals. It means that method calls on a server interface are transformed into two signals: the first is received by the automata describing the component behaviour (method call); the second is emitted by one automaton (method call return).

In the example, the StorageAspect concern which manages compressed data is implemented in our model as two reactive components called StorageComponent which loads data on persistent support and CompressComponent which

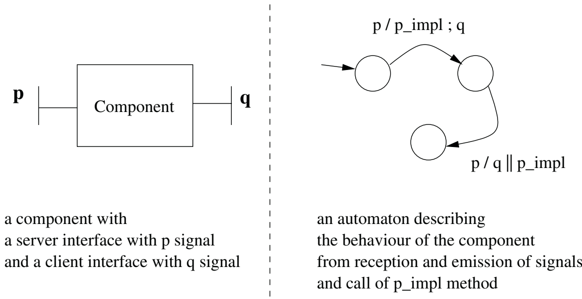

Fig. 1. A component and its automaton.

compresses or decompresses data. The load signal is defined on the server reactive interface of StorageComponent. The SecurityAspect concern which checks execution permissions is implemented as two reactive components called SecurityComponent, and LogComponent which logs permissions checks.

A scheduling constraint is manipulated like a reactive component. It receives signals that indicate activities on the other automata. It can be added in a composite component (i.e. it is activated), removed (i.e. it is desactivated), composed with another constraint (conjunction, disjunction, negation). In conclusion, in our approach, technical components and scheduling constraints are reactive components.

In the example, two scheduling constraints are introduced: (1) between decompress and log operations defined respectively on automata of StorageComponent and SecurityComponent: log cannot be executed until decompress is done, (2) between checkPermissions and load operations: load cannot be executed until checkPermissions is done.

5.2 Behaviour Description Model

Component. The component behaviour can be described by one or several automata (figure 1). Components communicate with each other using their interfaces. The behaviour of a component is described by the signals that constitute component reactive interfaces and by the component implementation. The elements of the automaton model are states and transitions as illustrated in the following figure; a transition is defined by a signal like p or q (provided by interfaces), a guard (defined from boolean methods of component implementation), and an action (a method call on the component implementation like p_impl, a signal emitted on a client interface like q, a sequence action or a parallel action). A parallel action means there is no execution order between two actions.The automaton model is hierarchical: automata can be defined inside an automaton state.

The more the automaton is detailed, the more the composition can be refined. Composition power is related to the automaton description level. Any reactive

component can be described with at least a minimal automaton: with one signal on the single reactive server interface, the component automaton is constitued of one state, one transition with as event the signal defined on its reactive interface and no action.

In the example, the automaton of the StorageComponent component reacts to the load signal and executes in sequence the load operation defined on the implementation, and emits the decompress signal towards the DecompressComponent component through its reactive client interface. The automaton of the SecurityComponent component reacts to the checkPermissions signal and executes in sequence the checkPermissions operation defined on the implementation, and emits the log signal to LogComponent component through its reactive client interface.

Scheduling Constraint. Experiments in defining scheduling constraints in the execution model show us that the constraints can be represented like any another component behaviour. One can thus define them in the same execution language that the one for the component behaviours. The scheduling constraint behaviour model derives from the component behaviour model. In our case, the semantics is based on that of the automaton model.

To schedule the execution of conflictual operations, we are interested in the execution of the actions defined in automaton transitions. The elements are the execution begin and end of the action. The beginning and the end of state are also elements used to express the scheduling of conflictual operations. Those elements are called *automaton element events*. In the automaton, a difference is made between an occurence of an action and a set of actions having the same type.

In the example, the first scheduling constraint (figure 2) between the decompress and log operations can be represented by an automaton with three states and two transitions. The initial state means the decompress operation can be executed. Scheduling constraint automaton moves from initial state to second state when the decompress operation is done (it reacts to the end decompress operation event defined on the transition between initial and second state). The second state means the log operation can be executed. The scheduling constraint automaton moves from second to third state when the log operation is done (it reacts to the end log operation event defined on the transition between second state and third state). The second scheduling constraint follows the same pattern.

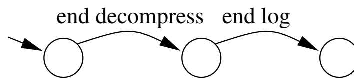

Fig. 2. First scheduling constraint.

Some conditions can be added into a scheduling constraint. A scheduling constraint can be applied only in certain cases (operations are conflictual in

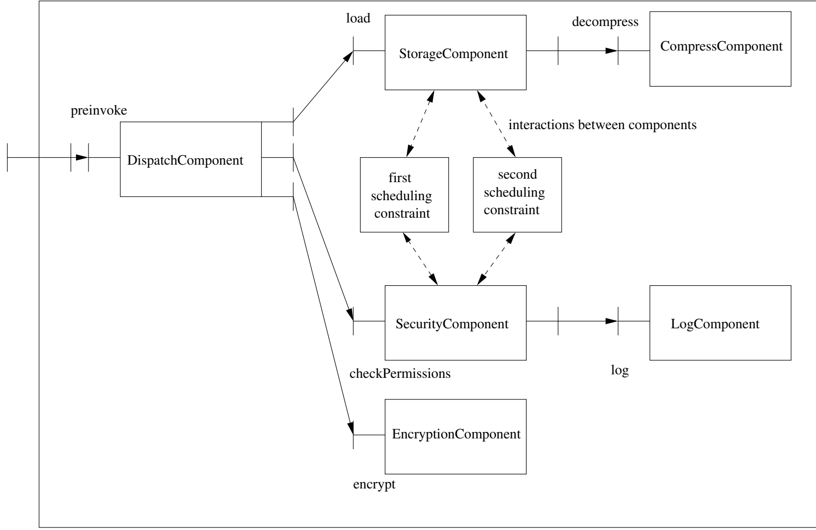

Fig. 3. This figure is the assembly of reactive components introduced in our example. It shows reactive components composition is based on structural composition and results in parallel composition of automata.

some cases only), expressed by a condition. These conditions are expressed either from automaton elements impacted by the scheduling constraint or from boolean methods implemented in scheduling constraint implementation if they cannot be expressed from automaton elements.

6 Composition

This section details the composition process. In our approach, concerns are implemented as reactive components. Thus the concerns composition process is a composition of reactive components. Reactive components behaviours are described by automata. Thus the reactive components composition process is also a composition of automata. We call the composition process the behavioural composition of components. It is based on structural composition of components, i.e., all concepts introduced in structural composition of components (bindings, etc) are present in behavioural composition (figure 3). The automata composition is the parallel composition of the automata describing the behaviours of the components representing the concerns and the scheduling constraints. The interactions between automata are emission and reception of signals defined on the reactive interfaces of components, and events used for synchronization.

At the behaviour level, we believe that adaptability is reached by the capability to modify the behavioural composition process. Composition of concerns is an example of behavioural composition. Adaptability is needed to apply scheduling constraints to the components behaviour.

7 Implementation

This section details the implementation of our proposal: a framework, called Brenda, that implements the concepts introduced in behavioural composition. This framework provides elements to describe components behaviours, to define reactive components, to compose and execute them. The framework design is the best way to introduce and to implement models of our approach. The Brenda framework is an extension of the Fractal framework. The Fractal framework provides a minimal core for components composition. It provides extension mechanisms for any composition type (structural, behavioural, functional) [13].

The behaviour description of a reactive component is given when designing the template of this component. Behavioural composition is based on structural composition as defined in Fractal, so the Brenda framework applies concepts introduced in the Fractal framework (binding, template, composite component, hierarchical model, etc) on behaviour concepts to reach behavioural composition of reactive components.

The Brenda framework architecture is decomposed into two layers: the behaviour layer and an extension of the Fractal specification and of the Julia implementation [7] [3].

Behaviour layer. This layer is dedicated to component behaviour. There is the implementation of automata-based model. Several processes are identified:

1. description: to build automata with automaton elements (state, transitions, actions, etc),
2. generation: to load the behaviour description of a reactive component,
3. composition: to add subcomponents into composite components,
4. binding: to bind signals defined on the interfaces,
5. synchronization: to apply synchronization points on automata impacted by scheduling constraints
6. instantiation: to generate the result of the composition of reactive components.

Each process constructs entities that are elements of behavioural composition: the description process constructs representations of automata, the generation process loads automata, the binding process constructs bindings (signal interactions) between automata, the synchronization process creates and inserts synchronization points which will be used to apply temporal order relations between conflictual operations, the instantiation process constructs the scheduler that is the result of the composition of components and scheduling constraints. The *factory* design pattern is used in each process.

Flexibility points. The framework design provides several *flexibility points*: (1) Factories implementations of the behaviour layer can be changed. For example, new optimized algorithms can be implemented for instantiation process. (2) The behaviour description model can be changed. (3) The execution model can be changed. It has to respect the automata semantics.

8 Instantiation Process and Execution Model

The selected execution model is a reactive synchronous model [14]. The reactive execution model is used for automata reactions when they receive signals. The synchronous execution model allows us to apply synchronization points directly on automata.

Several languages (Esterel [6], Lustre [11], etc) are available, but our instantiation process uses the Esterel language because we focus on runtime performance. Compilation techniques of Esterel language generate a reactive machine that reacts to signals emitted by the environment. The Esterel language provides the parallel operator (parallel composition of automata and parallel action in an automaton transition). The automata composition applied in this model is a synchronous composition of the Esterel automata that result from the conversion of automata of reactive components. This synchronous composition is an automata product. Using the Esterel language restricts framework concepts: no dynamical reconfiguration is available, so no unbinding, no scheduling constraint removal, no reactive component removal (structural reconfiguration implies behavioural reconfiguration), no behavioural reconfiguration can be performed after instantiation. Those restrictions are compromises between runtime performances and dynamical reconfiguration. Scheduling (determining a valid interweaving of execution sequences of concerns) is dynamically processed but the composition is performed statically (i.e., no structural reconfiguration, no behavioural reconfiguration).

The execution model is introduced in the instantiation process. From the behaviour description of components of the composite, this process generates the result of the automata composition which is a global automaton. In a reactive approach, this automaton is implemented inside a reactive machine which reacts to signals defined on reactive interfaces. The instantiation process generates dynamically the code of the reactive component that delimits the synchronous bubble (it emits a signal towards the reactive machine when a method is called, it calls methods defined in the reactive components implementations, etc).

The framework design provides extension mechanisms to apply an instantiation process for any selected execution model.

9 Results

This section presents how composite component composition of concerns can be plugged in an application, and shows execution with interactions between reactive components implementing concerns and scheduling constraints. The framework focuses on component composition. The integration process aims at composing the result of the composition of non orthogonal concerns with the application. The integration process is independent from the composition process. Several ways can be considered for integration process:

- AOP approach : there is a single aspect that is the result of the composition. For example, an aspect can be defined for concerns composition:

```
aspect CompositionAspect {
    private CompositeItf getConcernsCompositionComposite() {...}
    pointcut businessmethodcalls(BusinessObject bo) : target(bo) && call(public * * (..));
    before(BusinessObject bo) : businessmethodcalls(bo) { composite.preInvoke(bo); }
...}
```

The getConcernsCompositionComposite method builds the result of composition of concerns with scheduling constraints and returns the composite component.

– Interceptor component: the application is a components configuration. An interceptor is a component that reifies the method calls on a business component which it decorates. The interfaces of the interceptor are bound to the interfaces of the business component and the interfaces of technical component that is the result of the composition of non orthogonal components.

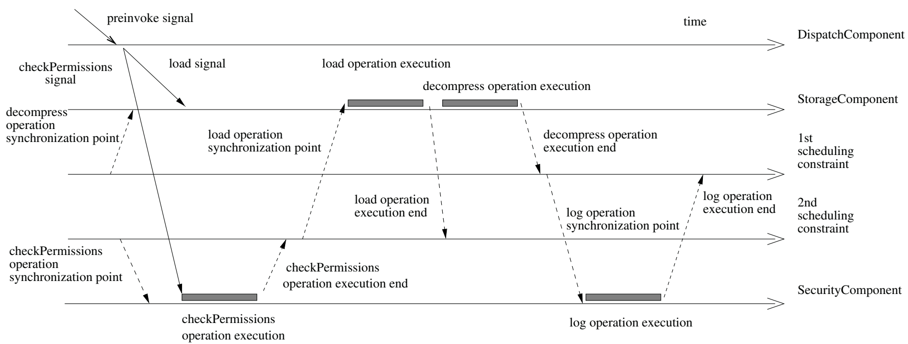

Fig. 4. Interactions between reactive components.

Execution and interactions between reactive components. The figure 4 shows interactions between reactive components implementing concerns and scheduling constraints. DispatchComponent component emits checkPermissions and load signals in parallel. The second scheduling constraint automaton emits a synchronization point signal to the automaton of SecurityComponent component to be able to execute checkPermissions operation. StorageComponent component automaton is waiting for a synchronization point signal from the second scheduling constraint before executing the load operation. At the end of the execution of checkPermissions operation, the automaton of SecurityComponent component emits an event to the second scheduling constraint automaton that moves in a state where the automaton of StorageComponent component can execute load operation (so it emits a synchronization point signal). The same pattern is applied with the first scheduling constraint.

10 Conclusion

This paper presents an approach for composing non orthogonal non functional properties and its Java implementation in the Brenda framework. After detailing the issue of composing non orthogonal concerns, it defines models (behaviour

description and reactive component) and implements them in a framework seen as an increment of Fractal framework. The framework has been implemented and is functional. Our approach provides a way to resolve the composition of non orthogonal concerns by expressing scheduling constraints on component behaviours. The framework generates automatically the results of compositions of non orthogonal concerns, while the AOP approach requires to implement specific weavers for each scheduling constraint.

This work is a combination of several research domains (AOP paradigm, reactive synchronous programming, components architecture and behaviour description modelling). It tries to tackle the well-known but still largely unexplored composition of non orthogonal properties. However, we think progress on those research domains can be directly used to improve our proposal.

We describe an architectural way to express the design of compositions of non orthogonal concerns. We plan to apply these concepts to execution supports in application servers.

Acknowledgements

We'd like to thank Marc Lacoste, Romain Lenglet and Jacques Pulou.

References

1. The AspectIX Project. See `http://www.aspectix.org`.
2. The AspectJ Project. See `http://www.aspectj.org`.
3. The Fractal Project. See `http://fractal.objectweb.org`.
4. J. Andrews. Process-Algebraic Foundations of Aspect-Oriented Programming. In *Proc. REFLECTION'01, LNCS 2192.*
5. L. Bergmans and M. Aksit. Composing Crosscutting Concerns Using Composition Filters. *CACM 44(10)*, 2001.
6. G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
7. E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and Dynamic Software Composition Sharing. In *Proc. WCOP'02.*
8. R. Douence, P. Fradet, and M. Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In *Proc.GPCE'02.*
9. R. Douence, P. Fradet, and M. Südholt. Detection and Resolution of Aspect Interactions. Technical report, INRIA, 2002.
10. G. Kiczales et al. Aspect-Oriented Programming. In *Proc. ECOOP'97, LNCS 1241.*
11. P. Caspi et al. LUSTRE: A Declarative Language for Programming Synchronous Systems. In *Proc. POPL'87.*
12. R. Pawlak et al. JAC: A Flexible and Efficient Framework for AOP in Java. In *Proc. REFLECTION'01.*
13. T. Coupaye et al. Composants et composition dans l'architecture des systèmes répartis. *Journées Composants 2001 Besançon*, 2001.
14. N. Halbwachs. Synchronous Programming of Reactive Systems. In *Computer Aided Verification*, pages 1–16, 1998.
15. Sun Microsystems. *Enterprise JavaBeans Specification version 2.1.* 2003.