DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs

Matthieu Leclercq, Vivien Quéma and Jean-Bernard Stefani
Sardes Project - LSR/IMAG - INRIA
655, avenue de l’Europe, F-38330 Montbonnot, France
first.last@inrialpes.fr

Abstract

In this paper, we present DREAM, a component-based software framework for the construction of resource-aware message-oriented middleware that can be assembled statically or dynamically. DREAM is based on Fractal, a recent component model featuring hierarchical composition, component sharing and component binding. DREAM allows building different forms of message-oriented middleware, from distributed message queues to content-based publish/subscribe systems. This paper describes the main elements of the DREAM framework, discusses its implementation, and evaluates its effectiveness. We show that message-oriented middleware built using DREAM compare very favorably with non-configurable, functionally equivalent middleware, while allowing finer-grain control over resource consumptions and providing run-time configurability.

1 Introduction

The use of asynchronous communications (MOM for Message-Oriented Middleware) is recognized as a mean of achieving scalability in applications made of loosely coupled autonomous components that communicate on large-scale networks [4]. Several MOMs have been developed in the past ten years [1, 23]. The research work has primarily focused on the support of various non functional properties like message ordering, reliability, security, scalability, etc. Less emphasis has been placed on MOM configurability. From the functional point of view, existing MOMs implement a fixed programming interface (API) that provides a fixed subset of asynchronous communication models (publish/subscribe, event/reaction, message queues, etc.).

From the non-functional point of view, existing MOMs often provide the same non-functional properties for all message exchanges. This reduces their performance and makes them difficult or impossible to use with devices having limited computational resources. As these non-functional properties have
not been developed as independent (removable) modules, removing them often requires the code to be totally re-engineered.

To overcome these limitations, it is necessary to build modular and composable architectures. Work on configurable systems has lead, in particular, to the development of component-based and reflective middleware [14]. The idea is to build a middleware as an assembly of interacting components, which can be statically or dynamically configured to meet different design requirements or environment constraints. While in principle applicable to different forms of middleware, existing component-based middleware have mostly been used to construct classical middleware with synchronous interactions, and with a few exceptions [8, 17], have not dealt systematically with resource configurability (i.e. the ability to control the use of resources within the middleware). Modular architectures have also been proposed to build routers [13] or communication sub-systems [7]. The main limitation of these systems lies in their restricted component model, which can mainly be used for static configuration, does not support hierarchical composition, and which does not provide any control capability, thus making it hard to administer and configure systems during execution.

This paper presents DREAM (Dynamic REflective Asynchronous Middleware), a software framework dedicated to the construction of asynchronous middleware. DREAM provides a component library and a set of tools to build, configure and deploy middleware implementing various asynchronous communication paradigms: message passing, event-reaction, publish-subscribe, etc. It builds upon the Fractal generic component framework [10], which provides support for hierarchical and dynamic composition. Hierarchical composition supports the construction of systems from the assembly of structured (hierarchical) sets of components. Dynamic composition provides the basis for dynamic re-configuration, a useful feature for long-running systems. Dynamic composition is achieved by means of reflection mechanisms which allow the execution of a system to be monitored and controlled at the level of individual components.

The contributions of this paper are as follows:

- We present a software framework that enables message-oriented middleware to be constructed and configured from a library of components, which can be assembled statically or dynamically (at deployment-time or at runtime).

- We describe a MOM personality that has been built using DREAM and we show how our framework can be used to dynamically control resource consumption and concurrency in this personality.

- We show that the performance of dynamically configurable message-oriented middleware built with the DREAM framework compares very favorably to a monolithic, functionally equivalent middleware.

The article is structured as follows. The DREAM component architecture is presented in Section 2. Section 3 describes the DREAM component library
and associated tools. Section 5 presents an evaluation of our work. Section 6
discusses related work and section 7 concludes this paper.

2 Dream component architecture

The component model used in Dream is an extension of Fractal [10], a com-
ponent model for Java. We distinguish two kinds of components: primitive
components and composite components, which provide a means to deal with a
group of components as a whole. An original feature of the model is that a given
component can be included in several other components. Such a component is
said to be shared between these components. Shared components are useful to
model access to low-level system resources.

Dream components communicate through interfaces. Interfaces can be of
two kinds: server interfaces, which correspond to access points accepting in-
coming method calls, and client interfaces, which correspond to access points
supporting outgoing method calls. Particular interfaces, called input and output,
allow Dream components to exchange messages. Messages are always sent
from outputs to inputs. We distinguish the push connection mode — where
message exchanges are initiated by the output — and the pull connection mode
— where message exchanges are initiated by the input.

A component is made of two parts: a controller part, which comprises in-
terceptors and controllers, and a content part, which can be either a standard
Java class in the case of a primitive component, or other components (called
subcomponents), in the case of a composite component. The controller part can
provide different levels of structural and behavioral reflection. The following are
examples of controllers.

**Binding controller:** A component can provide the Binding Controller in-
terface to allow binding and unbinding its client interfaces to server interfaces.

**Content controller:** A component can provide the Content Controller
interface to list, add and remove subcomponents in its contents.

**Life-cycle controller:** A component can provide the Life-cycle Controller
interface to allow control over its main behavioral phases, in support for dy-
namic reconfiguration. Basic methods supported by a Life-cycle Controller in-
terface include methods to start and stop the execution of the component.

Figure 1 illustrates the different constructs in a typical Dream component.
Thick grey boxes denote the controller part of a component, while the interior of
these boxes correspond to the content part of a component. Arrows correspond
to bindings, and tau-like structures protruding from grey boxes are interfaces.
Input and output interfaces are represented by blue triangles. Interfaces appear-
ing on the top of a component represent controller interfaces such as Binding
Controller or Content Controller interfaces. The two shaded boxes represent a
shared component.
3 The Dream library

This section presents the Dream library. We first describe abstractions and components dedicated to resource management. Then we present the functional components, i.e. the components that implement functions and behaviors commonly found in an asynchronous middleware.

3.1 Abstractions and components for resource management

The Dream library defines abstractions and provides components for managing resources found in asynchronous middleware, i.e. messages and activities. These components allow fine-grained control over consumed resources, which is a required feature to build scalable MOMs.

3.1.1 Message management

**Dream messages**  Dream messages are Java objects that encapsulate named chunks. Each chunk implements an interface that defines its type. As an example, messages that need to be causally ordered have a chunk that implements the Causal interface. This interface defines methods to set and get a matrix clock. A message may also encapsulate other messages and is identified by an interface called Message. This interface allows accessing/adding/removing chunks and encapsulated messages.

**Message managers**  Messages are managed by shared components, called message managers, that allow Dream components to create, duplicate or delete messages. The goal of message managers is to manage memory resources in a MOM personality. By implementing pools of messages, they avoid unnecessary object creations, and allow resource management policies to be implemented. For that purpose, all the methods defined in the MessageManager interface have a parameter (called consumer) that specifies the component calling the method. Message managers can use this parameter, for instance, to control the amount of messages created for the different components.
### 3.1.2 Activity management

A DREAM component can either be passive or active. An active component defines tasks\(^1\) to be executed; a passive component doesn’t, i.e. calls to other component interfaces can only be made in the tasks of a calling component. For a task to be executed, it must be registered to one of the dedicated shared components, called *activity managers*, that encapsulate tasks and schedulers.

- **Schedulers** are components with a Schedule server interface, and an Execute client interface. The role of a scheduler is to map higher-level tasks (to which its Execute client interface is bound) onto lower-level tasks (that are bound to its Schedule server interface). The number of scheduler levels is not limited. The DREAM framework currently provides various schedulers: round-robin, FIFO, periodic, and with priority.

- **Tasks** are components with an Execute server interface, and a Schedule client interface. The former interface defines a method execute. We distinguish (i) *highest-level tasks* that are registered by the MOM’s components and that contain functional code, (ii) *lowest-level tasks* that wrap Java threads, and (iii) *inter-scheduling tasks* (IS tasks) that are created by schedulers to be scheduled by lower-level schedulers.

![Activity management diagram](image)

**Figure 2: Activity management**

Figure 2 depicts an example of activity management. Components A and B have registered three tasks that are scheduled by two hierarchically organized\(^2\)

---

\(^1\)Active components must implement the TaskController interface, which allows a third party to access the component’s tasks.
schedulers. This produces the following result: the periodic scheduler periodically executes the IS and B tasks. The IS task execution consists in triggering the sequential execution of tasks A1 and A2 (using the FIFO scheduler). Note that the periodic scheduler is executed by two lower-level tasks that wrap threads.

3.2 Functional components

This section describes the core components of the DREAM library, i.e. the components encapsulating functions and behaviors commonly found in an asynchronous middleware. Note that the library also contains specific components developed for particular middleware: for instance, components implementing event-reaction processing. For space limitations, we do not present these components.

Message queues are used to store messages. Queues differ by the way messages are sorted (FIFO, LIFO, causal order, etc.), and the behavior of the queue when the capacity is exceeded (blocks vs. removes messages), when the queue is empty, etc.

Transformers have one input and one output. Every message received on the input is transformed, and delivered on the output interface. The message transformation is only governed by one rule: the message identity must be preserved, i.e. the Message interface uniquely identifying the message must not be changed. As a consequence, a transformer cannot deliver a newly created message. On the other hand, the content of the message can be changed: it may encapsulate new chunks and new sub-messages.

Pump have one pull input and one push output. They have an activity whose role is to pull messages on the input and then push them on the output. Note that this activity is stateless. As a consequence, it can be executed by several threads simultaneously.

Routers have one input and several outputs (also called “routes”), and route messages received on their input to one or several routes. The routing process may involve message transformations (e.g. to remove or manipulate routing information).

Duplicators have one input and several outputs. They duplicate messages received on their input to all their outputs.

Aggregators have one or several inputs to receive the messages to be aggregated, and one output to deliver the aggregated message.

De-aggregators implement aggregators’ reverse behavior, i.e. they take an aggregated message and generate appropriate individual messages from it.

Channels allow message exchanges between different address spaces. Channels are distributed composite components that encapsulate, at least, two components: a ChannelOut — which aims at sending messages to another address space —, and a ChannelIn — which can receive messages sent by the ChannelOut.
4 The Dream tools

The DREAM framework currently provides two tools:

- a deployment tool that uses a description of the component configuration to be deployed into an Architecture Description Language (ADL) in order to proceed to its distributed deployment and configuration. This tool is very similar to other ADL-based deployment tools [18].

- a type checking tool that defines a type system for DREAM messages allowing checking that a component configuration is correct, i.e. that each component of the configuration will receive messages containing chunks with appropriate types.

The rest of this section is dedicated to the description of the type checking tool. We first motivate the use of such a tool. Then, we describe the type system and show how it can be used to check DREAM configurations.

4.1 Motivations

A system built out of DREAM components comprises several components which exchange messages, which may modify them (e.g. adding/removing a chunk), and which may behave differently according to their contents (e.g. routing a message). In the current implementation of the DREAM framework, every message has for type the Message Java interface, independently of its contents. As a consequence, certain assemblages of DREAM components type-check and compile correctly but lead to one of the three following run-time failures:

- a chunk is absent when it should be present (e.g. for a read, remove, or update).
- a chunk is present when it should be absent (e.g. for an add).
- a chunk does not have the expected type (e.g. for a read).

Figure 3 gives the example of an incorrect configuration: component readTS expects messages with a TS chunk, whereas component addTS expects messages without TS chunk. Since both components receive exactly the same messages (duplicated by the duplicator component), one of them will fail. The typing annotations are clearly insufficient to allow the previous analysis.

4.2 A type system for Dream components

This section describes a polymorphic type system that allows the specification of the more common behaviors of DREAM components. It provides the guarantees that, if components conform individually to their type, the composed system will not fail with any of the runtime errors identified in the previous section.
The type system is an adaptation of existing work on type systems for extensible records. A record is a finite set of associations between names and values, also called fields. In [21, 22], Rémy describes an extension of ML where all common operations on records are supported. In particular, addition or removal of fields and concatenation of records. He then defines a static type system that guarantees that the resulting programs will not produce runtime errors, such as the access to a missing field.

4.2.1 Message types

We type messages as extensible records. Informally, the type of a message consists of a list of pairwise distinct names together with the type of the corresponding chunk, or a special tag if the message does not contain a given name. Moreover, a final information specifies the content of the (infinitely many) remaining names.

$$\mu_1 = \{a : \text{pre}(\text{CausalChunk}); b : \text{pre}(\text{IPChunk}); \text{abs}\}$$
$$\mu_2 = \{a : \text{pre}(X); \text{abs}\}$$
$$\mu_3 = \{a : Y; \text{abs}\}$$
$$\mu_4 = \{a : \text{pre}(\text{CausalChunk}); Z\}$$

Figure 4 defines several examples of message types. A message $m$ of type $\mu_1$ contains exactly two chunks named $a$ and $b$, associated to chunks of type CausalChunk and IPChunk, respectively (the importance of the pre constructor will be made clear later). It does not contain any other name, as specified by the abs tag. Richer types can be constructed using type variables. In type $\mu_2$, $X$ represents an arbitrary type. Informally, a message of type $\mu_2$ must contain a name $a$, but the type of the associated chunk is not specified. Similarly, in $\mu_3$, $Y$ is a field variable. It can be either abs, or pre$(X)$ where $X$ is a type variable. Hence, the pre constructor allow us to impose the presence of a given
field, even if its type is unspecified. Finally, in $\mu_4$, $Z$ is a row variable that represent either abs or any list of associations.

4.2.2 Component types

Besides traditional interfaces, DREAM components have inputs and outputs interfaces that allow them to exchange messages. Each input/output is characterized by its name, and the type of the messages it can carry. The type of a component is a polymorphic function type relating outputs to inputs. Figure 5 gives examples of components and component types. The duplicator component has a polymorphic type. Its input and its outputs can be used with any type $X$. It duplicates the messages it receives on its input to all its outputs. The adda component adds a new IPChunk with name $a$ to the messages it receives on its input $i$. Note that these messages do not contain a chunk with name $a$. The removea component removes the chunk named $a$, that may or may not be present.

\[
duplicator: \forall X.\{i : \{X\}\} \rightarrow \{o_1 : \{X\}; o_2 : \{X\}\}
\]
\[
adda: \forall X.\{i : \{a : \text{abs}; X\}\} \rightarrow \{o : \{a : \text{pre}(\text{IPChunk}); X\}\}
\]
\[
removea: \forall X,Y.\{i : \{a : Y; X\}\} \rightarrow \{o : \{a : \text{abs}; X\}\}
\]

Figure 5: Examples of component types

4.3 Example

Figure 6 depicts the same configuration as in figure 3, using the type system defined in the previous section. The configuration will be well-typed if and only if we can solve the equations:

\[
\{X\} = \{ts : \text{pre}(A); Y\}
\]
\[
\{X\} = \{ts : \text{abs}; Z\}
\]
The equations do not have any solution, and thus the system is not well-typed.

5 Evaluation

In this section, we describe an experiment we have done to evaluate the DREAM framework: an implementation of the Joram MOM [3]. We show that DREAM, while preserving Joram’s performance, solves part of its limitations. In particular, it allows a finer-grain control over resource consumptions, provides Joram with dynamic reconfiguration capabilities, and offers the ability to easily derive a variant of Joram targeting embedded computing devices.

5.1 A brief introduction to Joram

Joram is an open-source JMS-compliant middleware (Java Messaging Service). It comprises two parts: the ScalAgent MOM [5], and a software layer on top of it to support the JMS API. The Scalagent MOM provides a distributed programming model based on autonomous software entities called agents that behave according to an “event → reaction” model. The ScalAgent MOM comprises a set of agent servers (Figure 7). Each agent server is made up of three entities: the Engine is responsible for the creation and execution of agents; it ensures their persistency and atomic reaction. The Conduit routes messages from the engine to the networks. The Networks ensure reliable messages delivery and causal order between servers.

![Figure 7: Two interconnected agent servers](image)

5.2 Implementing JORAM using DREAM

We have implemented the ScalAgent MOM using DREAM (see Figure 8). Its main structures (networks, engine and conduit) have been preserved to facilitate the functional comparison between the ScalAgent MOM and its DREAM re-implementation.

The engine comprises two main components: the AtomicityProtocol composite that ensures the atomic execution of agents; the Repository composite, which is in charge of creating and executing agents.
Two typical networks are depicted. Both are composite components encapsulating a ChannelIn, a ChannelOut and a DestinationResolver component. The latter is a transformer that adds the information required by the TCPChannelOut component (i.e. IP address, and port number). The Network 2 composite contains two more components: the CausalSorter causally orders messages; the message queue decouples the workflows of the engine and the network. The conduit is implemented by a router.

5.3 Configurability assessment

A first benefit of the DREAM implementation comes from the ability to easily change provided non-functional properties. For instance, it is straightforward to remove causal ordering, or to remove the atomic protocol ensuring transactional execution of agents. Both modifications only require a modification of the ADL description, or can be programmatically done at runtime. On the other hand, removing these properties from the ScalAgent MOM requires modifying and recompiling its source code.

Another benefit brought by implementing the MOM with DREAM is that it is easy to change the number of active components encapsulated within the agent server. The architecture we have presented in Figure 8 involves three active components for an agent server with one network. A mono-threaded
architecture can be obtained by removing the message queues encapsulated within the engine and the network. The only remaining active component is the ChannelIn component that listens on a socket.

Finally, DREAM allows building MOMs for embedded devices. We developed an agent server for mobile equipment with limited resources. This kind of equipment presents two characteristics: it may be temporarily disconnected from the network and it has a limited storage capacity. Figure 9 shows how an agent server can be re-designed to take into account these characteristics.

![Figure 9: An agent server for mobile devices](image)

The agent server (on the left part of the picture) is made of one composite. This composite is an engine whose message queue has been replaced by a ChannelIn component, and which encapsulates a ChannelOut component to send messages. Messages intended to the mobile device are stored on another device (on the right part of the picture). This device hosts a component that acts as a proxy for the mobile device’s engine. It is plugged to a conduit router (like a traditional engine). It has two functions: (1) it receives messages intended to the mobile device and stores them into the queue; these messages can then be pulled by the mobile device. (2) it forwards messages sent by the mobile devices.

This architecture preserves the MOM functionality, while saving memory (the mobile device part is mono-threaded; messages are pulled instead of pushed; it does not have the CausalSorter and DestinationResolver components). Moreover it allows the mobile device to be disconnected, since messages are remotely stored.

### 5.4 Performance comparisons

Measurements have been performed to compare the efficiency of the same application running on the ScalAgent MOM and on its DREAM implementation. The application involves four agent servers; each one hosts one agent. Agents in the application are organized in a virtual ring. One agent is an initiator of
rounds. Each round consists in forwarding the message originated by the initiator around the ring. We did two series of tests: messages without payload and messages embedding a 1kB payload. Experiments have been done on four PC Bi-Xeon 1.8 GHz with 1Go, connected by a Gigabit Ethernet adapter, running Linux kernel 2.4.20.

Table 1: Performance of the DREAM implementation versus ScalAgent implementation

<table>
<thead>
<tr>
<th>MOM</th>
<th>Number of rounds</th>
<th>Memory footprint (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 kB</td>
<td>1 kB</td>
</tr>
<tr>
<td>ScalAgent</td>
<td>325</td>
<td>255</td>
</tr>
<tr>
<td>DREAM (non-reconf.)</td>
<td>329</td>
<td>260</td>
</tr>
<tr>
<td>DREAM (reconf.)</td>
<td>318</td>
<td>250</td>
</tr>
</tbody>
</table>

Table 2: Impact of the concurrency level

<table>
<thead>
<tr>
<th>MOM</th>
<th>Number of rounds</th>
<th>Memory footprint (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 kB</td>
<td>1 kB</td>
</tr>
<tr>
<td>DREAM (3 threads)</td>
<td>329</td>
<td>260</td>
</tr>
<tr>
<td>DREAM (2 threads)</td>
<td>346</td>
<td>268</td>
</tr>
<tr>
<td>DREAM (1 thread)</td>
<td>370</td>
<td>279</td>
</tr>
</tbody>
</table>

Table 2 reports on experiments we have done to assess the impact of the concurrency level on the performances of the ScalAgent MOM. We compare three
architectures built using DREAM that differ by the number of active components they involve. In the 2-thread architecture the message queue encapsulated in the network has been removed. In the mono-threaded architecture, both active message queues have been removed (Engine and Network). We see that, in this particular case, reducing the number of active components improves the number of rounds (+3 to 5% for the 2-thread architecture, and +7 to 12% for the mono-threaded architecture). This can be explained by the fact that agents are organized in a virtual ring, thus each agent server only processes one message at a time. As a consequence, only one thread is necessary.

6 Related work

There are several types of work related to DREAM. We review them briefly below.

Reflective adaptable middleware
The past ten years have seen a lot of activity related to the construction of reflective middleware, as exemplified by the OpenORB [9], DynamicTAO [15], QuO [17], or Hadas [6] projects. DREAM differs by three main characteristics.

- DREAM is using and extending the Fractal component model, an original component model for Java, which unlike component models used in other reflective middleware does not impose predefined reflective capabilities. On the contrary, it allows the middleware developer to design arbitrary meta-object protocols.

- DREAM and its component library targets the construction of asynchronous middleware services. To our knowledge, existing reflective middleware have focused on synchronous interactions.

- DREAM provides a set of tools allowing configuring and deploying asynchronous middleware built using the component library. In particular, DREAM provides a type checking tool that defines a rich type system allowing detecting incorrect architectures.

- DREAM provides resource management functions, in particular for activities. The only middleware we know of that has integrated such functionalities is openORB. The proposed activity model shares similarity, but is less flexible than the one we propose in this paper. QuO provides interesting means to enforce QoS in an ORB. The main contribution of QuO is the definition of languages that allow QoS requirements to be expressed. QoS enforcement is mainly obtained by the way of interceptors, which could be implemented by Fractal interceptors.
Communication subsystems

Research work has also been conducted on component-based communication subsystems. Several frameworks have been designed: Click [13], Coyote/Cactus [7], OSKit/Knit [12, 20], and APPIA [19]. Proposed component models are limited. They do not allow the dynamic manipulation of composite components available in Fractal. They provide mainly for static configuration and not fully dynamic reconfiguration as in Fractal. In contrast to DREAM, changing the concurrency structure of a Click or Coyote protocol, for instance, can only be done at runtime if explicitly programmed in the protocol implementation.

Asynchronous middleware

Several MOMs have been developed in the past ten years: Astrolabe [24], Gryphon [23], MSMQ [1], Siena [11], SonicMQ [2]. The research work has primarily focused on the support of various non functional properties, as exemplified by [16]. Less emphasis has been placed on MOM configurability. The only work relating to the construction of configurable message-oriented middleware we now of is the work discussing reflective features of Gryphon [23] (a content-based asynchronous middleware), which, to our knowledge, have not been implemented and come short of providing the level of reconfigurability provided in DREAM.

7 Conclusion

We have presented the DREAM framework for the construction of configurable asynchronous middleware. DREAM is using and extending the Fractal framework, which provides for hierarchical composition of components, component sharing and fully dynamic reconfiguration of software architectures. Moreover, DREAM defines a component library that, besides functional components, encapsulate components for resource management. Finally, DREAM provides tools for the configuration and deployment of middleware built using the component library. In particular, DREAM provides a type checker that allows detecting incorrect component configurations.

We have shown in the paper that it was possible to implement with DREAM the Joram middleware. We have shown that the gains in flexibility and configurability come without significant loss of performance for applications. Actually, we have shown in our evaluation that the high configurability of DREAM can yield significant performance improvements by adapting a middleware architecture to its environment and application load (e.g., as was done in our evaluation, by changing the concurrency structure of a middleware).

Availability

DREAM is freely available under an LGPL license at the following URL: http://dream.objectweb.org
References


About the authors

Matthieu Leclercq holds an MSc in computer science from ENSIMAG - Grenoble Institute of Technology (INPG). He is currently working at INRIA. His research interests include component-based programming, architectures and designs of message-oriented middleware, and resource management in J2EE clusters. Contact him at INRIA, 655 avenue de l’Europe, F-38334 Saint-Ismier Cedex, France; matthieu.leclercq@inrialpes.fr.

Vivien Quéma is a PhD candidate in computer science at Grenoble Institute of Technology (INPG). He is currently working at INRIA, where he is doing research on component-based middleware, architecture description languages, and Grid resource monitoring. He holds a MSc in computer science from Grenoble Institute of Technology. Contact him at INRIA, 655 avenue de l’Europe, F-38334 Saint-Ismier Cedex, France; vivien.quema@inrialpes.fr.

Jean-Bernard Stefani is research director at INRIA. He is graduated from Ecole Polytechnique. His research interests are process calculi, component-based systems design, and distributed algorithms. Contact him at INRIA, 655 avenue de l’Europe, F-38334 Saint-Ismier Cedex, France; jean-bernard.stefani@inrialpes.fr.