An integrated approach to coordination description in distributed multimedia applications

Pierre Gradit*, Khalil Drira and François Vernadat
LAAS-CNRS, 7 Av. Colonel Roche, 31077 Toulouse Cedex 04, France
Tel.: +33 561 336 334; Fax: +33 561 336 411;
E-mail: {gradit,drira,vernadat}@laas.fr

This paper proposes an original integrated approach allowing coordination rules to be described for resource allocation in Distributed Multimedia Applications. Our approach has two major benefits. On the one hand, it allows the scalability and the dynamicity required for today's distributed systems. On the other hand, rather than bringing a new formalism and rather than making a dialect of an existing formal language we work on how two existing description techniques may be integrated to compositionally describe the whole system. An implementation of our approach is presented.

1. Introduction

The dynamic evolution of topology, the distribution of components and the complexities of interactions are important features of Distributed Multimedia Applications. These features make traditional software architecture models and existing description techniques to be likely inappropriate for the design of these emerging systems [23]. Designers and programmers are now facing problems in using conventional description techniques to represent the many points of view in Distributed Multimedia Applications. In such systems, the behaviour of dynamically created and selected cooperating agents as well as the topology which enables and controls their interactions must be able to evolves to fit the needs of the different agents.

Both recent technical and theoretical studies agree on the importance of separating communication and coordination from the other system functionalities. Hackman's research on coordination in group work identifies the central role of the coordination process [13]. The recent standardisation work for *Open Distributed Processing* standard (ODP) and the very popular middleware standard *Common Object Request Broker Architecture* (CORBA) [10] focus on this separation. One of the most important benefits for this separation is scalability and evolutivity of the built systems. But within the scope of formal methods, this separation is not embedded in a description technique:

- Coordination languages [4] focus on coordination description for concurrent systems. Linda coordination language
  [3] and Gamma calculus [2] constitute the most known models in this area. They allow coordinated agents to interact using a shared tuple space. These works focus on the description of the semantics of interaction but leave the description of the coordination process as an exercise for the modeller.
- Although they have been recognised since a long time as appropriate to formalise behaviour of agent models like Petri Nets [20] and Process Algebra [17] lack expressiveness for topology transformation description.
- Recent research works have demonstrated the appropriateness of graph grammars for modelling the coordination process in concurrent systems [16, 21]. These works while focusing on describing the coordination process lack describing its interaction with the co-ordinated agents.

Nevertheless, the relevance of this separation has already produced several extensions or specialisation of the previous description techniques, handling both agent behaviour and topology evolution description, which can be classified into the three following categories:

*Corresponding author.

Integrated Computer-Aided Engineering 8 (2001) 311–324
ISSN 1069-2509 / $8.00

- *Those combining formalisms and descriptions*: In the first category, we find traditional description techniques that have been developed in the eighties and that was devoted to specify point to point communication protocols [9,22,11,20].
- *Those separating descriptions but combining formalisms*: The second category gathers dialects developed on the basis of the techniques of the first category. Approaches of this category aim to adapt existing techniques to separate (or at least to identify) topology management-related actions in agent behaviour description e.g. the "new" and "delete" operators added to standard LOTOS in [18] to specify agent creation and deletion, respectively. In such approaches, dependency of both aspects (topology and behaviour) prevents reusability of topology management specifications with different agent behaviours. In the same category there are other approaches that describe the semantics of behaviour-dedicated techniques in terms of topology-dedicated techniques, e.g. the semantics of Actors [1] in terms of Graph Grammars proposed in [14]. In both approaches of this category, the use of an inappropriate technique leads to a complex and large specification. Again, we obtain combined descriptions preventing from reusing models.
- *Those separating descriptions and formalisms*: Our work belongs to the third category, which includes emergent research focusing on coordination description where topology evolution management is a critical issue.

In this paper, we aim at bringing a trade-off between the interaction semantics definition and the coordination process description. To this end, we introduce ActYve, an integrated technique for distributed multimedia applications modelling based on Graph Grammars (GG) for coordination description and Labelled Transition Systems (LTS) for individual agent behaviour description. These LTS can be produced by exploration of the reachable state graph of a more compact specification such as Petri Nets or Process Algebras. The proposed technique, ActYve, allows on the one hand generic functions of the coordination process to be described including topology management (e.g. communication channels, hierarchical relationships, temporal dependencies, etc. ...) independently from the individual agents behaviour. And on the other hand it allows both, agents' descriptions and the coordination process description to interact with each other. ActYve description and associated engine are mainly devoted to coordination issues. A coordination policy can be written through an ActYve coordination scheme while different behavioural component corresponding to the different user strategy can be merged in the final specification. It allows to check if the combination of user own strategies and the whole coordination system are compliant. Moreover, the conception modes induced by this paradigm are deeply distributed and can be used to specify more general distributed systems.

The first part of the paper details the ActYve paradigm through a case study: a resource allocation coordination problem in Multimedia group conferencing. This part is divided in two part, the first is devoted to the specification without insertion/deletion while the second considers that case. To this purpose transient behavioural components, which can be inserted and deleted, are introduced.

The second part presents the general principles of an implementation of the ActYve paradigm, realised in MAUDE [5]. Maude is a language that implements Rewriting Logic [15], and can be consider as a formal meta-tool. In that logic, a term is a class of terms of a signature modulo a set of equations. A rewrite rule is given by a labelled pair of terms. A key feature of MAUDE is to be reflective, this allows to manipulate behaviour representations themselves as terms in another specification. This kind of terms can handle both LTS representation or directly compact representation of behaviours such as Petri Nets. The key idea is to use step rewriting both to rewrite the dynamic topology representation and the different behavioural component, using *descent functions* which allows the use of term representation of behaviours. The last part details the different calculus steps needed to produce all the possible co-ordinated transitions from a given state according to the standard gluing condition of graph grammars and the behaviour synchronisation. This is presented here only for simulation purpose but it constitutes also the kernel engine of the computation of the reachable state graph, entry point of verification techniques.

2. Channel allocation in multimedia dynamic conferencing

In order to make easier the understanding of the proposed technique, we will use a multimedia group conferencing example to illustrate coordination of resource allocation with dynamic topology evolution. We will highlight four distinct topologies and define a set of coordination rules and agent behaviour which remain

valid whatever the initial topology is. Dynamicity is introduced in two steps. First, only systems composed of a fixed number of behavioural components, called *static*, are considered. In such systems, topology can evolve anyway, but in a fixed set of component references. Then in a second phase, we introduce systems where the number of behavioural components can evolve, called *dynamic*.

We consider, in this section, the problem of resource allocation in Multimedia group conferencing. A group of n participants shares $m \geqslant 2$ channels. To communicate a participant needs two channels one for the audio and the other for the video transmission. The different rules of the system in natural language are listed below:

Static rules

- **Enter**: A waiting participant ( ) shall get two free channels ( ) to communicate representing an audio channel and a video channel. It becomes a communicating participant ( ) holding two busy communication channels ( ).
- **Leave**: A communicating participant ( ) stops communicating by releasing its two busy channels ( ). The participant becomes idle ( ), while their channels are available for another communication.
- **Ask**: An idle participant ( ) can become asking for a communication ( ).

Dynamic rules

- **Join**: A participant may join the group bringing a free communication channel ( ), he is considered to be asking for communication ( ).
- **Quit**: An idle participant ( ) may quit the group with a free communication channel.

Conventional modelling approaches of resource allocation in concurrent systems consider an unique static topology for this problem, mostly a ring. The nature of Multimedia group conferencing problem may lead to different topologies depending on the location of participants and communication resources. We propose a new coordination description technique that takes into account actions specified by individual agents and relation constraints – e.g. introduced by resources allocation (a participant needs two channels) – whatever the topology is (chain, grid, ring, tree, . . .) as shown in Fig. 1.

Moreover, the graph description we handle represents both the system topology (by the graph structure) and the state of each agent (by the node labelling) representing a participant. Considering again Fig. 1, each topology is represented by a graph where the node labelling provides additional information. These labels describe agents' states. The graphical convention used to represent these states are detailed in the informal presentation of rules. Furthermore, in this graph, edges denote dependencies and agent relationships. For instance, edge ($\rightarrow$) indicates that channel ( ) is *used* by participant ( ). Finally, the direction of the edge allows to distinguish different relationships between the same pair of nodes. This is useful for example for the participant situated at the centre of the grid (Fig. 1(b)) to distinguish channels that are *used* by this participant (west and south channels) from those that are accessible by this participant according to the topology but which are *used* by another participant (north and east channels). General rules may be established according to the fan-out of the graph. For our example, for both tree and grid topologies – where fan-out is greater than 2 – directed graph must be used to avoid ambiguity between the "accessibility" and "ownership" relationships.

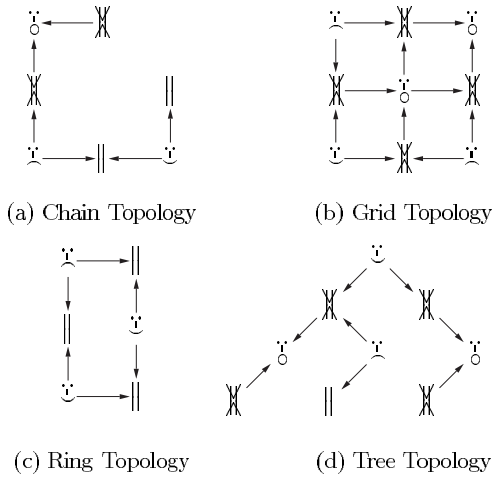

Fig. 1. Different topologies for the cooperating participants problem.

The description approach we present in this paper, uses Graph Grammars [8] to describe topology evolution. In the Graph Grammar production, nodes are labelled by actions performed by agents. To allow Graph Grammars to describe Distributed Multimedia Applications topologies with synchronised actions, we define *ActYve* interface, a specialisation of Goettler's Y-interface [12].

Extending the standard approach which models a complex system as a *static* configuration of *permanent* agents, our approach describes a Distributed Multimedia Applications as a *dynamic* graph, whose nodes represent *dynamically activated and deactivated* cooperative agents and vertices represent relationships be-

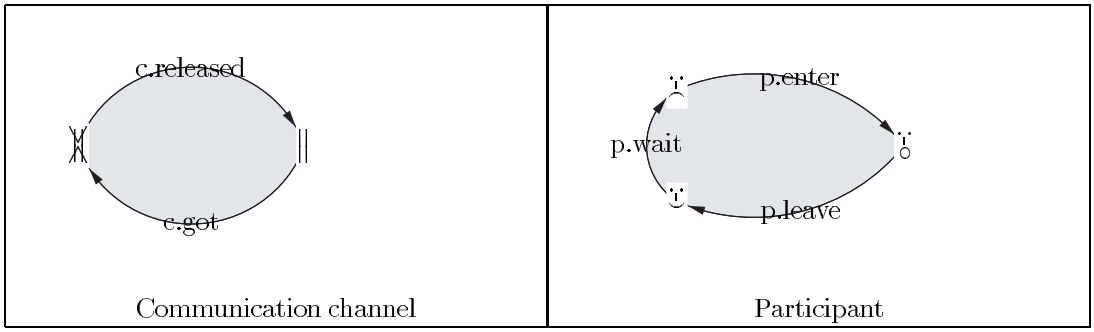

Fig. 2. LTS as behavior representation.

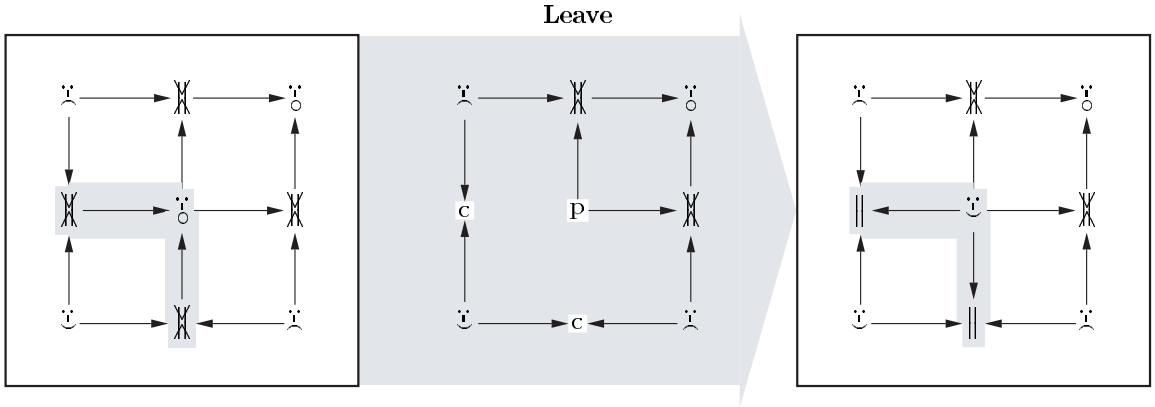

Fig. 3. Instance of rule **Leave** over a grid topology.

tween agents, including: geographical & spatial information (e.g. the "accessibility" of a channel to a participant), architectural information (which computer is connected to which computer through which communication protocol), causal dependency (e.g. two consecutive versions of a document) and application-related relationships (e.g. which channels are used by which participant).

Each reactive node is associated with a behaviour, for the simplicity of the presentation, we use LTS. Without loss of generality, we assume that label sets of the different LTS are pair-wise disjoint, hence each kind of node (channel or participant in our example) may be identified by state or action labels. We assume that LTS have a "silent action" called ϵ which does not change their state.

Coordination rules: As in Graph Grammar productions, coordination rules are described again by a graph and each rule graph describes a coordination rule. A coordination rule is defined as a labelled graph, where labels have particular types: they represent triggering *actions* performed by agent. We define Graph Grammar productions with Goettler's Y-Interfaces [12]. Thus, the rule graph is surrounded by an Y (bold lines), defining which edges are removed (left-sided edges) and which edges are added (right-sided edges), while edges situated in the Y remains unchanged after applying the rule. Figure 4 details the interface with the different "regions" highlighted by the Y.

Coordination rule application: A rule is applicable when the following conditions hold:

– As required for applicability of standard graph grammar productions, the graph rule matches a sub-graph of the system graph through a graph morphism l. This morphism satisfies the *gluing condition* [8]:

  Identification l is injective for the deleted part.
  Binding Neighbouring edges of a deleted node are deleted by the rule application.

– Each automaton identified by graph morphism l must be able to perform the action labelling its associated node in the rule graph. Rules act on agents as follows: every time a rule is applied, involved agents change state as specified by the tran-

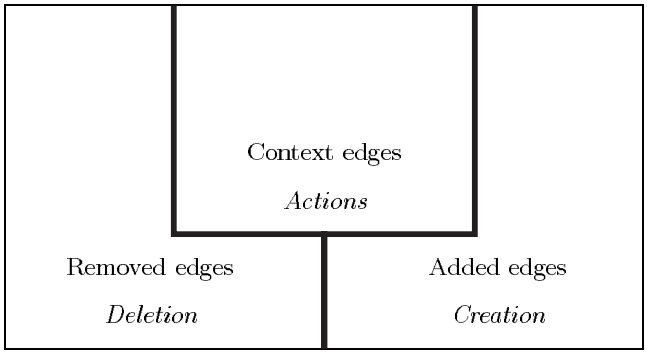

Fig. 4. **ActYve** Interface.

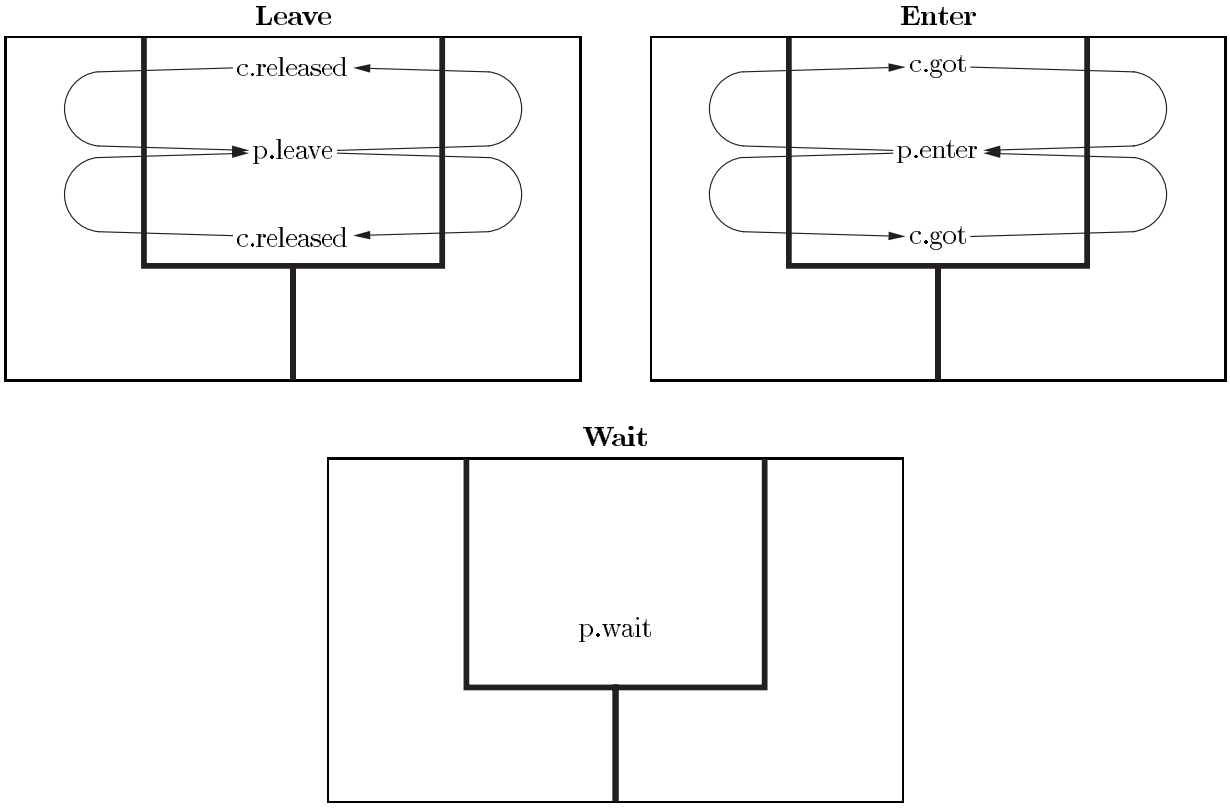

Fig. 5. **ActYve** specification of channel allocation in multimedia group conferencing applications.

sition labelled by the associated action, according to graph morphism l.

To illustrate rules and their application, we will first focus on rules that modify topology without managing creation/deletion of agents, in our example **Enter** and **Leave**. In a second step, we will consider all the rules, and specially those managing insertion/deletion of agents, as **Join** and **Quit**.

2.1. Basic coordination rules

In a first step, we focus on coordination.

Considering again our channel allocation specification, rule **Leave** specifies the possibility for a communicating participant to stop communicating: *p.release* action allows to apply the rule with communicating ( ) participant, while *p.release* actions specifies that used ( ) channels shall surround it. Figure 3 summarises

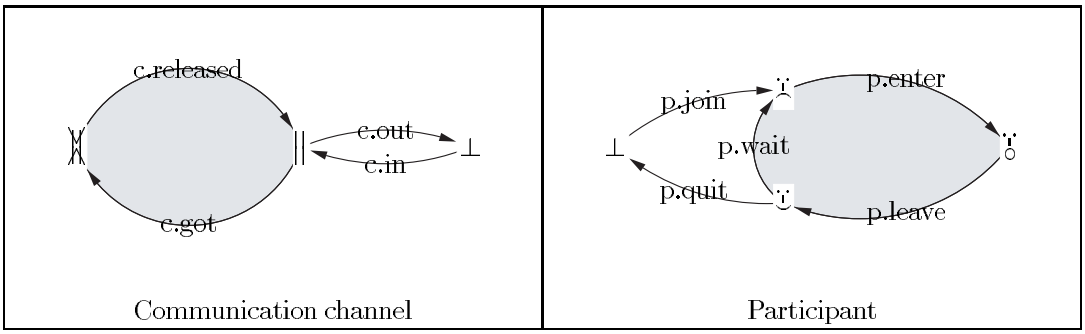

Fig. 6. Extended LTS.

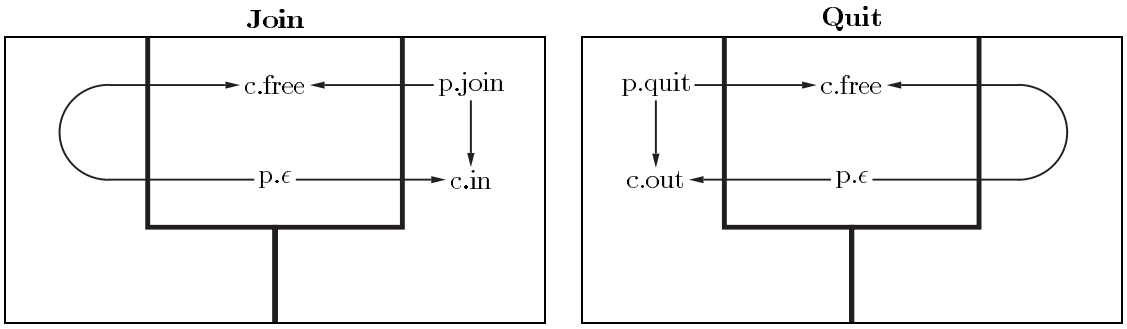

Fig. 7. **ActYve** specification of channel allocation in multimedia dynamic group conferencing applications.

the effect of this coordination: system states before and after synchronisation are represented. Shaded part of picture gives the coordination rule instance and highlights the "region" of the system affected by the transformation. According to their respective behaviour (given in Fig. 2), the two used channels (triggering action *c.released*) become free (||) and the communicating participant (triggering action *c.release*) becomes idle (☺).

As in a graph grammar [8,12,14,16] writing coordination rules does not require a global knowledge of the whole system topology. The previous rules remain valid whatever the system topology is (chain, ring, grid, tree . . .). The comparison with CCS [17] is interesting as on one side it can be considered as prosing a predefined class of pattern of synchronization while our approach allow to define any pattern. On the other side its hierarchical facilities to define these patterns are not yet supported by our paradigm.

Until now, we have seen how the coordination problem can be described as a graph matching between the "rule graph" and the "system graph", we now develop the remaining coordination functions relative to agent creation and deletion management.

2.2. Complete coordination rules

To manage the dynamic case, we first introduce how we manage *transient component* by refinement of the automaton-related conventions. In order to model *creation* and *deletion* of nodes, a particular state (⊥) is added to the set of states, which can be interpreted as "absent". Transition leading to ⊥ makes the agent vanishing (*deletions*), whereas transitions coming from ⊥ makes a *new* agent (*creations*). Figure 6 details the extended behaviour of agents: A channel enters and leaves the system in state free (||), while a participant enters the system in state waiting (☹) and leaves it in state idle (☺).

The second extension concerns the possibility to add nodes in removed and deleted parts of the graph grammar production. Such actions label shall be *deletions* for left sided nodes and *creations* for right sided nodes. The central part represents context part which remains topologically unchanged and associated node labels are *actions*.

Rule **Join** expresses the arrival of two connected objects: a participant (p.join) and a channel (c.in). The coming participant will be connected to a free channel (c.free) already present in the system. Now consider

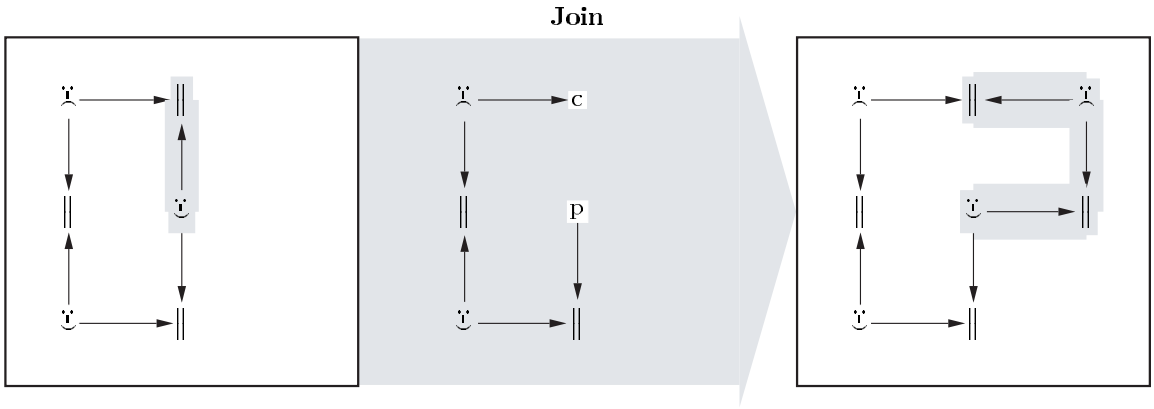

Fig. 8. Addition of a connected pair participant/channel in a ring.

```
mod Participant is
sort State .
ops ¨ı ¨i ¨i : -> State .
rl [wait] : ¨ı => ¨i .
rl [get] : ¨i => ¨i .
rl [release] ¨i => ¨ı .
endm
```

```
mod Channel is
sort State .
ops || ⋈ : -> State .
rl [got] : || => ⋈ .
rl [released] ⋈ => || .
endm
```

Fig. 9. MAUDE module for static behavioral components.

a node which was connected to this channel: the link between this node and the channel is deleted and replaced by a link to the introduced channel. Conversely, rule **Quit** expresses the departure of a connected pair participant/channel.

3. Simulator overview

This section presents the encoding in MAUDE [5] of an ActYve specification and the general features of our dedicated simulator. As presented in the previous section, an ActYve specification is composed of two kinds of specifications: *Behavioural component specifications* and *ActYve coordination scheme*. Then this specification is interpreted through a *step-by-step* simulator. Computation aspects are developed in the last section.

3.1. Behavioural component specifications

Behavioural component specifications are MAUDE modules without any added restriction. For example, Fig. 9 gives the MAUDE specifications of both behavioural components of our case study: participant and channel. These modules are very simple due to the simplicity of the case study.

In order to specify transient components we build a common ancestor to all transient module called TRANSIENT. Following the frame introduced in Section 2.2, this ancestor is very simple, it has a single sort (`All`) and a single operator $\perp$ which is an `All` constant. In order to be transient a module shall include the transient module, rule starting from $\perp$ will be interpreted as apparition while rule going to $\perp$ will be interpreted as vanishing. As both left hand side and right hand side of a rule must be of same sort, possible sort for the current state of the component must be subsorts of `All`. In the frame of our use case, Fig. 10 gives the behaviour of Participants and Channels as transient automata.

3.2. ActYve coordination scheme

ActYve coordination scheme is written in a dedicated syntax. An ActYve coordination scheme is composed of:

– *Sort Declaration*
  `Sort Participant Channel.`

– *Tuple Declaration*

(notice that in graphical form, edges are unlabeled (i.e. `can-access` and `is-used-by` are forgotten) as the edge direction is sufficient to determine the corresponding label.)

```
tuple (_can-access_): Participant
Channel.
tuple (_is-used-by_): Channel
Participant.
```

– *Variable Declaration*:

```
var participant incoming former:
Participant.
var left right: Channel.
```

– *set of Labelled Rules*

(A rule specification can be divided in three parts: 1. the rule label (i.e. **enter**), 2. the topological rewriting and 3. the synchronization of the components involved in the topological rewriting.)

– *Static rules*

(A variable occurs in the left hand side if and only if it occurs in the right hand side. For such variables, a <synchro_> tag is used to describe synchronisation process. Such synchronisation can be gathered by using _|_ operator inside the tag. Synchronisation is optional. Notice that for rule **ask**, the only used variable only occurs in the synchronisation pattern, it highlights the fact that this action is local.)

```
rule [ask]: empty => empty
with <synchro participant[wait]>.
rule [enter]: (participant
can-access left) (participant
can-access right)
=>(left is-used-by participant)
(right is-used-by participant)
with <synchro participant[get] |
left[got] | right[got]>.
rule [leave]: (left is-used-by
participant) (right is-used-by
participant)
=> (participant can-access left)
(participant can-access right)
with <synchro participant
[release] | left[free] |
right[free]>.
```

– *Dynamic rules*

(Some variables occurs in a single side. For each left-only occurring variables (resp. right occurring), a < delete_> tag (resp. < create_>) a must be associated to handle initial and final state.)

```
rule [join]: (participant
can-access left)
=> (participant can-access right)
(incomer can-access left)
(incomer can-access right)
with <create incomer[join] |
right[in]> <synchro left[free]>.
rule [quit]: (participant
can-access right)
(former can-access left)
(former can-access right)
=> (participant can-access left)
with <synchro left[free]>
<delete former[quit] |
right[out]>.
```

This coordination scheme is a textual form for Figs 5 and 7.

3.3. Step-by-step simulation

The standard applications of rewriting techniques are traditionally devoted to compact representation of confluent and terminating computations. Here, we will use the same theoretical framework but for an orthogonal purpose: exploring a reactive system that is not expected to end and for which any intermediate state can be matter of interest, rather than the "last one". So we are more interested in the step-by-step specification unfolding than the output of a specific terminating sequence. In consequence, the default loaded modules used to manage ActYve data allows a step-by-step exploration:

– Show all one-step reachable state of the ActYve system from the current state. It indicates also the rule triggered in the *ActYve coordination scheme* with the substitution used to reach each reachable state.
– User chooses among these states the next current state.

3.4. General set-up

First of all, *ActYve coordination scheme* has to be compiled as MAUDE module, then following modules has to be loaded in the interpreter:

```
fmod TRANSIENT is
sort All .
ops ⊥ :  -> All .
endfm
```

```
mod Participant is including TRANSIENT
sort State .
subsort State < All .
ops ¨ı ¨i ¨ı :  -> State .
rl [wait]      :  ¨ı => ¨i .
rl [get]       :  ¨i => ¨ı .
rl [release]   :  ¨ı => ¨ı .
rl [join]      :  ⊥ => ¨i .
rl [quit]      :  ¨ı => ⊥ .
endm
```

```
mod Channel is including TRANSIENT
sort State .
subsort State < All .
ops ||  ||  :  -> State .
rl [got]       :  ||  => ||  .
rl [released]  :  ||  => ||  .
rl [in]        :  ⊥ => ||  .
rl [out]       :  ||  => ⊥ .
endm
```

Fig. 10. MAUDE modules for transient automata components.

```
mod Participant is including TRANSIENT
sort State .
subsort State < All .
op 0⃗ :  -> State .
op _⊕_ :  State State -> State [assoc comm id:  0⃗] .
ops ¨ı ¨ı :  -> State .
rl [wait]      :  0⃗ => ¨i .
rl [get]       :  ¨i => ¨ı .
rl [release]   :  ¨ı => 0⃗ .
rl [join]      :  ⊥ => ¨i ⊕¨i .
rl [quit]      :  0⃗ => ⊥ .
endm
```

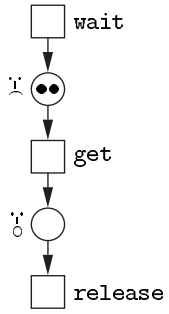

Fig. 11. MAUDE module for transient Place/Transition Petri net participant.

– compiled ActYve specification;
– behavioural modules, one for each sort in the ActYve specification. The name of the model is the name of the sort;
– step-by-step simulator.

At that stage the commands of the simulators allow to: define a state, choose a possible step in the list, and go to previous reached state. Offering the list of possible steps is done after any of these commands.

3.5. Simulation Use case

With the *ActYve coordination scheme* previously defined, and behavioural component given in Fig. 9, consider the following current state depicting a single waiting Participant which can access two Channels. Topology and behaviour descriptions of the state are given in separate squares.

State 0

```
Topology mod= session
state= Participant(1) can-access Channel(1))
(Participant(1) can-access Channel(2))
```

```
Behaviour mod= Participant
          id = 1 state= ¨ı
Behaviour mod= Channel
          id = 1 state= ||
Behaviour mod= Channel
          id = 2 state= ||
```

Then from current state 0, the simulator proposes a list one-step accessible results, [1, 2] given in Fig. 12. Each numbered one-step reachable result is

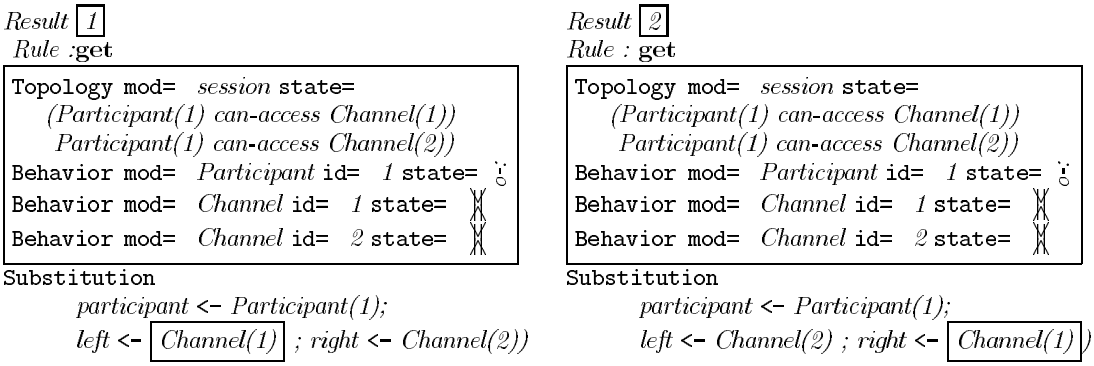

Fig. 12. Enumerated one-step reachable state.

composed of a rule label, a state and the substitution used to reach it. User may choose the next current state by striking its corresponding number: 1 or 2.

Here, by striking `1` or `2`, the user will reach the same state through two different substitutions. Then, the simulator will propose two different transitions leading to the state where an idle participant can access two channels.

From state 0, if the user strikes three times `1`, the current state will be 0 again. Notice that, as striking always `1` can be interpreted as the *default* strategy for rewriting computations, interpreting the specification in a standard rewriting way will not be terminating.

3.6. Different modules leads to different systems

We will consider in that paragraph three configurations differing only by the behavioural components involved in the specification. It follows that all these configurations are built with the *session* ActYve coordination scheme given before.

- *Standard automata configuration* The behaviour components are those given in Fig. 9. Participant module does not include "absent" state. So when performing the first stage of the ActYve step, no Participant can rewrite $\perp$, so no Participant insertion is possible. Conversely, the step 4.1 is never validated as $\perp$ is unreachable. It results from these considerations that reference dynamism is inhibited by the behaviour of the component. The iconic layout of these modules is a consequence of the use of LaTeX to display them, for example ☺ is written in the specification as the `\idle` LaTeX command. It makes the textual form as near as possible of the graphical one (c.f. Fig. 2).
- *Transient automata configuration* Just loading transient modules in place of static ones will give to the whole system a dynamic behaviour.
- *Transient Place/Transition configuration* If the modelled system allows a Participant to have more than a communication at a time, simply replace the behavioural component of Fig. 10 by the transient Place/Transition Petri Net [20] given in Fig. 11. This module corresponds to the net given on the right where the indicated marking corresponds to the marking of the component when it appears. To vanish, a transient Place/Transition participant needs to have an empty net. This remark highlights the key difference between the empty marking ($\vec{0}$) and the "absent" state ($\perp$).

This last example and its use of a Place/Transition Petri Net shows how the use of MAUDE extends the use of LTS as proposed in the presentation section.

4. ActYve step computation in MAUDE

This section presents the actual computations performed by our simulator engine. First, the reflectivity of MAUDE that enables the building of our simulator is presented. Then, the different step rewritings used to gather relevant information throughout the specification are detailed. The last part is devoted to the overall strategy used to perform the computation of the possible ActYve steps.

4.1. MAUDE modules and reflection

In MAUDE system, a module encompasses the definition of the signature, the variable declaration and a set of rewrite rules that can either be silent rule or labelled rule. The set of silent rules are expected to be Church-

Rosser and terminating while there is no expectation on labelled rules as they will be controlled step-by-step.

The effective use of a control is a consequence of the most important feature of MAUDE [5], reflectivity [7]. This notion can be formally depicted as follows: *it exists a finitely represented rewriting theory called U that is universal in the sense of we can represent in U any finitely presented rewrite theory R* (*including U itself*) *as a term* $\overline{\mathcal{R}}$*, any term* t, t' *in R as term* $\overline{t}$*,* $\overline{t'}$ *and any pair* (R, t) *as term* $\langle\overline{\mathcal{R}}; \overline{t}\rangle$*, in such a way that we have the following equivalence [5]*:

$$R \vdash (t \xrightarrow{a} t') \Longleftrightarrow \mathcal{U} \vdash \left(\langle\overline{R}; \overline{t}\rangle \xrightarrow{\overline{a}} \langle\overline{\mathcal{R}}; \overline{t'}\rangle\right) \quad (1)$$

In MAUDE, you can access very efficiently to this reflective feature by the use of *descent functions* [6], accessible through the META-LEVEL module in the standard library of MAUDE. These functions allow to use module representation to rewrite term representation according to Eq. 4.1.

MAUDE provides an another class of functions that allow to automatically produce term meta-representation from a standard input and conversely, to perform this feature it uses module representation as grammar for parsing or pretty-printing meta-terms. These features allow us to present our ActYve engine without going too deeply in the MAUDE encoding of it.

4.2. Step rewritings and information gathering

By combining *descent* functions and *pretty-printing* functions, we consider the `steps` rewriting which for a given module and a state belonging to this module, gives all the one-step rewrite successors according to a rule label.

We introduce in the `step` definition the possibility in MAUDE to constraint the substitution used to perform the step rewriting, and get back the complete substitution for each possible one-step rewrites. Formally, a `step` term is composed of a module, a state, a rule name and a start substitution. Its confluent and terminating rewriting leads to a list of *result pairs* [6] composed of a state and a final substitution.

– Behavior rewriting step
`steps` (*Participant*, $\overset{state}{\ddot{\smile}}$,get,none)
-> [< $\overset{state}{\ddot{\smile}}$; $\overset{subst}{none}$>]

In previous rewriting step, the substitution is empty (i.e. none) because the rewriting only involves constants. In the general case, the possibility to constraint and extract the substitution is a key feature for the linking of the dynamic topology with behavioural components.

The *ActYve coordination scheme* is compiled in MAUDE in order to allow two kinds of rewriting, a *topological rewriting* which gives the one-step accessible topologies together with their substitution, a *synchronization rewriting* which gives for a rule label the different synchronisation needed over component variable. Both rewritings are detailed below for our use case (c.f. 3.5).

– Topological rewriting step
`steps` (session, (Participant(1) can-access Channel(1)) (Participant(1) can-access Channel(2)), enter, none)
-> [<<(Channel(1) is-used-by Participant(1)) (Channel(2) is-used-by Participant(1)); participant <- Participant(1);, left <- Channel(1);, right <- Channel(2), <(Channel(1) is-used-by Participant(1)) (Channel(2) is-used-by Participant(1)); participant <- Participant(1);, left <- Channel(2);, right <- Channel(1)
– Synchronisation extraction step
`steps`(session,synchro,enter,none) ->
[< | (Participant(participant), get) | (Channel(left), got) | (Channel(right), got) ; none>]

4.3. Producing all ActYve steps

From information gathered by the different step rewritings detailed in the previous section, we can produce the set of possible ActYve steps. In the static case, step rewritings occurs before the ActYve computation, but in the dynamic case, the different step rewriting occurs in a more sophisticated algorithm, in order to take into account different constraints raising from reference management issues.

4.3.1. Linking the two levels in the static case

In the static case, an ActYve rewriting step is composed of two steps. First a topological rewriting is performed. Then it performs the synchronisation rewriting to deduce the different rules needed to be trigger over each behavioural component.

Then, according to the synchronisation information and the extracted substitution of the topological rewriting, it performs the `step` command over each involved component.

This intermediate stage of the calculus is represented in Fig. 13 while underlined items in rewriting steps pick

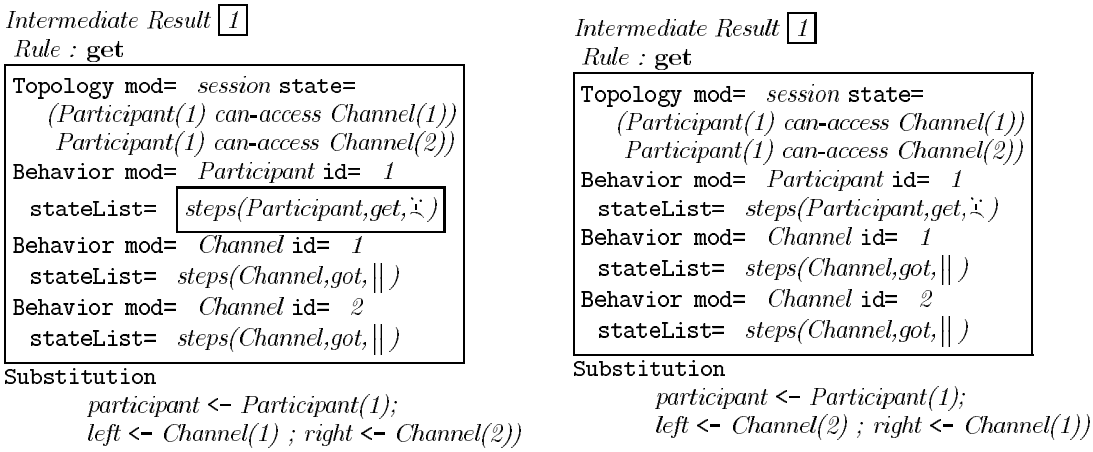

Fig. 13. Enumerated intermediate states including state Lists.

up relevant items for the computation of the *Participant*(*1*).

Finally it builds the list of ActYve states according to the different possibilities of involved components given in Fig. 12.

4.3.2. Linking the two levels in the dynamic case

This stage is more complicated because it needs a reference management. First of all, we say that a state *binds* if any references cited in the topology has a behaviour *and conversely*. As all considered states *bind*, we have to manage a single set of references because topological reference set and behavioural component reference set are identical. In the dynamic frame, an ActYve rewriting is performed in four stages, static case being embedded by a reference allocation stage and a reference deletion stage.

1. *Allocation stage* According to the synchronization rewriting, the engine allocates *new* references for the corresponding variables, and perform the one-step rewriting of $\perp$ for the corresponding modules. The result of the step is twice:
   - A set of *added behavioural components*, which have to be inserted anyway to any solution at the end of the process.
   - A *kernel substitution* associating to each appearing component a reference which is not cited in the current state.
2. *Topological rewriting stage* Rewriting rules with free variable (i.e. cited in the right hand side without being cited in the left hand side) are forbidden in standard rewritings. But, as noticed in the `step` definition, MAUDE engine allows to constraint the substitution used for perform one-step rewrites, this feature can be used to instanciate free variables [6]. Equation 3 summaries informally the use of the step rewriting for the topology rewriting.

$$\texttt{steps}(topology, state, rule, kernel\ subs.) \rightarrow \text{“}\texttt{listof}\text{”}(newstate, completesubs.) \quad (2)$$

3. *Synchronization rewriting stage* This stage is very near from the static case. According to the *complete* substitution (i.e. larger than the *kernel one*), the rewriting rule of all preserved and deleted components are triggered using the *step* command. The list of reachable ActYve state is then expanded according to the state list of each involved component. If a behaviour component has an empty list of possible rewrites, it makes the whole list of reachable ActYve state empty. A particular case has to be discussed, when a node out of the topology rewriting triggers a behaviour rewriting, as for example in rule **ask**. Any component of the corresponding sort, out of the topology substitution, can trigger that rule, and the complete substitution is updated in consequence.
4. *Deletion stage* In order to validate both binding and standard gluing condition for graph rewriting we check at that stage three conditions:
   - The state of any deleted component is the “absent” state ($\perp$).
   - The reference of any deleted component is no more cited in the resulting topology, this condition is also called *binding condition* in the standard terminology [8].

– The substitution is locally injective for any deleted component (i.e. no other variable of the complete substitution maps to the same mapping reference). This condition is called *identification condition* in the standard terminology[8].

Condition 4.2 and 4.3 form the *gluing condition* in the standard terminology [8].

To be used as a tool to verify non-confluent and non-terminating systems, and not only simulate them or use it to drive other software, we need to be able to generate exhaustive exploration of reachable state graph and not only a single path. A difficulty raises from the allocation stage, which can allocate different reference values when several allocations are interleaved. It is known in the Graph Grammar [8] approach that a result of a graph rewrite rule is unique, but only up to isomorphism. A state explorer dedicated to ActYve specification must take into account this unicity. Such state explorer is under development in MAUDE, it relies on the possibility to associate to the set of known states a rewriting module mapping each state to a rule. If a one-step is possible from a new state in that module, we can check the substitution bijectivity to find out if this new state was already reached, up to isomorphism.

5. Conclusion

This paper has proposed, ActYve, an integrated approach to describe distributed multimedia applications as a co-ordinated behaviour of co-operative agents. ActYve uses different formalisms for interaction and coordination description. The implementation in MAUDE of our paradigm takes advantage of this separation. Behavioural components are MAUDE module of any complexity and without any restriction. ActYve coordination scheme are written in a specific syntax, near from the MAUDE one. Such specification has to be translated in a MAUDE module which allows *topological rewriting* and also a *synchronisation rewriting* for synchronisation information retrieval. Notice that *ActYve coordination scheme* can have free variable as long as they are associated to a behaviour component apparition. Even if the simulator attempted to manage the whole in a single language, MAUDE [6], the way topology rewriting is managed makes it closer to Graph Grammar [8] than Rewriting Logic [15] through a rewriting control based on a precise filtering of the substitution involved in the rewriting. Both formalisms, Graph Grammar and Rewriting Logic, have solid formal foundations allowing co-ordinated system correctness to be proved.

With respect to ActYve specification, two extensions can be considered. First, the subtyping feature of MAUDE allows recursive specification: an ActYve specification being itself as a behavioural component of another ActYve specification. This makes the another extension more critical: allowing behavioural component to synchronise over more structured information than label, namely terms. Different strategies can be founded in the literature to perform that kind of exchange: sharing a tuple space [3], defining which component propose information in the term, and which can catch it [22], allow an unification over synchronisation terms [19].

References

[1] G. Agha, *ACTORS: A model of Concurrent computations in Distributed Systems*, The MIT Press, Cambridge, Mass., 1990.

[2] J.P. Banâtre and D. Le Métayer, Introduction to Gamma, in: *Research Directions in High-Level Parallel Programming Languages*, J.P. Banâtre and D. Le Métayer, eds, Springer Lecture Notes in Computer Science 574, June 1991, pp. 197–202.

[3] N. Carriero and D. Gelernter, Generative communication in Linda, *Communications of the ACM* **32**(4) (1989), 444–458.

[4] P. Ciancarini and C. Hankin, eds, *Coordination languages and models: First International Conference COORDINATION '96, Cesena, Italy, April 15–17, 1996: proceedings*, number 1061 in Lecture Notes in Computer Science, Berlin, Germany / Heidelberg, Germany / London, UK / etc., Springer-Verlag, 1996.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J.F. Quesada, The Maude system, in: *Proceedings of the 10th International Conference on Rewriting Techniques and Applications* (*RTA-99*), P. Narendran and M. Rusinowitch, eds, Springer-Verlag LNCS 1631, System Description, Trento, Italy, July 1999, pp. 240–243.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J.F. Quesada, *Maude: Specification and programming in rewriting logic*, Technical report, SRI International, March 1999.

[7] M. Clavel and J. Meseguer, Reflection and strategies in rewriting logic, in: *Proceedings of the First International Workshop on Rewriting Logic and its Applications*(*RWLW96*), Pacific Grove, CA, 3–6 September 1996.

[8] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe, Algebraic approaches to graph transformation I: Basic concepts and double pushout approach, in: *Handbook of Graph Grammars and Computing by Graph transformation*, (Vol. 1) G. Rozenberg, ed., Foundations, World Scientific, 1997.

[9] M. Diaz and J.-P. Ansart et al., *The formal description Technique Estelle*, North-Holland, 1989.

[10] K. Drira, F. Gouezec and M. Diaz, Design and Implementation of coordination protocols for distributed cooperating objects:

a general graph-based technique applied to CORBA, in: *Proc. 3rd IFIP Workshop on Formal Methods for Open Object-based Distributed Systems* (*FMOODS*), P. Ciancarini, A. Fantechi and R. Gorrieri, eds, 1999, pp. 89–104.

[11] J. Ellsberger, D. Hogrefe and A. Sarma, *SDL – formal object-oriented language for communicating systems*, Prentice Hall, 1997.

[12] H. Göttler, Attributed graph grammars for graphics, in: *Graph Grammars and their application to Computer Science*, G. Rozenberg, H. Ehrig and M. Nagl, eds, LNCS 153, 1982, pp. 130–142.

[13] J. Hackman and R. Walton, Leading groups in reorganisations, in: *Designing effective work teams*, P. Goodman, ed., Jossey-Bass New York, 1986.

[14] S. Kaplan, J. Loyall and S. Goering, Specifying concurrent languages and systems with δ-GRAMMARS, in: *Research Directions in Concurrent Object-Oriented Programming*, (Vol. 99), G. Agha, P. Wegner and P. Yonezawa, eds, MIT Press, 1993.

[15] J. Meseguer, Conditional rewriting logic as a unified model of concurency, *Theoretical Computer Science* **96** (1992), 73–155.

[16] D. Le Métayer, Describing software architecture styles using graph grammars, *IEEE Transactions on Software Engineering* **24**(7) (July 1998).

[17] R. Milner, *Communication and Concurrency*, International Series in Computer Science, SU Fisher Research 511/24, Prentice Hall, 1989.

[18] E. Najm, J.-B. Stefani and A. Fevrier, Towards a mobile lotos, in: *Proceedings of the International Workshop on Formal Description Techniques VIII*, G. Bochman, R. Dsouli and O. Rafiq, eds, Chapman & Hall, October 1995.

[19] P. Azéma, F. Vernadat and P. Gradit, A workflow specification environment, in: *Workshop Workflow Management: Net-based Concepts, Models, Techniques, and Tools* (*WFM'98*), June 1998, pp. 5–21.

[20] W. Reisig, *Petri Nets, An introduction*, (Vol. 4), EATCS, Monograph on theoritical Computer Science, Springer Verlag, 1985.

[21] H. Schneider, Graph grammars as a tool to define the behavior of process systems: From petri nets to linda, in: *Fourth International Conference on Graph Grammars*, 1993.

[22] P.H.J. van Eijk, C.A. Vissers and M. Diaz, *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project*, North-Holland, New York, 1989.

[23] P. Wegner, The Paradigm Shift from Algorithms to Interaction, *Communication of the ACM*, May 1997.