

# Analysing Mailboxes of Asynchronous Communicating Components

Jean-Claude Royer and Michael Xu

Equipe LOAC, Ecole des Mines de Nantes, 4, rue Alfred Kastler - BP 20722, F-44307 Nantes Cedex 3  
Jean-Claude.Royer@emn.fr, Kaiye.Xu@eleve.emn.fr

**Abstract.** Asynchronous communications are prominent in distributed and mobile systems. Often concurrent systems consider an abstract point of view with synchronous communications. However it seems more realistic and finer to consider asynchronous communicating systems, since it provides a more primitive communication protocol and maximize the concurrency. Several languages and models have been defined using this communication mode: agent, actor, mobile computation, and so on. Here we reconsider a previous component model with full data types and synchronous communications with an asynchronous flavour. The dynamic behaviour of a component is represented as a structured symbolic transition system with mailboxes. We also present an algorithm devoted to an analysis of the dynamic behaviour of the system. This algorithm decides if the system has bound mailboxes and computes the reachable mailbox contents of the system. The component model and the algorithm are illustrated on a flight system reservation.

**KEYWORDS:** Asynchronous Communication, Component, Architecture, Dynamic Behaviour, Unbound or Bound Mailbox

This is a preliminary version of our DOA'2003 paper, it contains a comprehensive description of the flight reservation example.

## 1 Introduction

Architectures and components [24, 17, 16, 21, 34, 6, 2, 26] are nowadays technologies in software development. They promote software architectures based on communicating software entities. In this domain previous experiences are the actor model [4, 3, 19], concurrent object-oriented programming [10, 9], mobile computation [32] and the multi-agent systems [33]. Actors and agents are rather based on asynchronous message sending while often components and objects promote synchronous communications or Remote Procedure Calls. Nevertheless there are few examples of object-oriented languages with asynchronous call, for instance ProActive [1] and Piccola [2].

In the context of distributed computing, asynchronism of communications should be the default policy, especially in wide area networks. [22] presents a comprehensive discussion about distributed computing and its main characteristics. We may note that some applications like news or mail are naturally asynchronous. Asynchronous communications are simpler and more primitive than synchronous one, even if each one can simulate the other. Asynchronism is the choice done by several theoretical approaches but less often by real platforms, for example client-server has generally synchronous communications. Many infrastructures or component languages have basically synchronous communications: EJB, CORBA, RMI [17], this is also

true with several classic models and languages like ADA, CCS, CSP or LOTOS. Asynchronous communications are less constraining from a concurrency point of view but the emitter does not know if a message will be received by the receiver. Other important facts with asynchronism are the impossibility to differentiate a slow component from a stopped one and the impact of failures. Thus it implies more complex descriptions, we need to cope with more errors and it produces complex dynamic behaviours. On the other hand this communication mode is simpler to implement. To the contrary synchronous communications are more time consuming, more abstract and produce simpler dynamic behaviours.

There are already several language categories which use asynchronous communications: some concurrent object-oriented languages, some component languages, actor languages and multi-agent systems. We think that this use will grow conjointly with the emergence of distributed applications and distributed languages. Thus there is a need to provide analytic tools to study or understand behavioural problems with asynchronous communications. The ability to compute the global dynamic behaviour of such system and to reason about it are important both at the specification level and at the coding/testing level. Here we focus on asynchronous communications with safe communications, without real time aspect and process failure. One classic way to simulate asynchronism in a synchronous context is to use buffers which memorize messages between emitters and receivers. It introduces two kinds of entity: the components and the buffers. Furthermore the resulting dynamic behaviour is complex and it requires a similar analysis technique than for pure asynchronism. So we prefer to use native asynchronism with buffers inside components. It is much more uniform and simpler to implement in a distributed context.

In this paper we introduce a component model with asynchronous communications. It is based on some previous experiences with synchronous communications. This model defines interfaces and protocols for components, it also introduces architecture and communication schemes. We illustrate some graphical notations with a simple example of flight reservation. Our purpose is to study the global behaviour of such a system. We assume that sent messages always arrive to the receiver, in other words we have reliable or safe communications. Message lost are out of the scope of our study, see [29] for a related study. We provide means to have a look at the dynamic behaviour and to help designers to detect problems. We briefly describe a data structure for structured state and transition systems with mailboxes to code the global dynamic behaviour of such systems. There is a need to know the size of the mailboxes and if there are bound or not. Such informations are important for semantics reasons but also for optimisations ones. We propose a general algorithm to decide if the system is bound or not and which builds a complete simulation of the behaviour.

The paper is organised as follows. Section 2 presents the principles of the model, some notations for behavioural aspects and an example. Section 3 shows how to compute the global behaviour of such a system. It also describes our algorithm which calculates a view of the dynamic behaviour taking into account the contents of the mailboxes. Finally, some choices and related works are discussed and a conclusion finishes this presentation.

## **2 The Asynchronous Model**

Our current work is based on some previous experiences about architectures and components. These works are rather at the specification level, since we are firstly interested in designing good architectures. The KORRIGAN model [13, 12] is devoted to the structured formal specification of mixed systems through a model

based on a hierarchy of views. It allows one to specify in a uniform and structured way both data types and behaviours using Symbolic Transitions Systems (or STS) and algebraic specifications. Symbolic Transition Systems are finite state machines with guards and variables in addition to traditional labels. The main interest with these systems is that (i) they avoid state explosion problems, and (ii) they define equivalence classes (one per state) and hence strongly relate the behavioural and the algebraic representations of a data type. STS may be related to Statecharts [18] but they are simpler and stricter on the semantic side. KORRIGAN is relevant to describe reusable components, architectures and communication schemes. The Graphic Abstract data Type model [31] improves the KORRIGAN model on specification method and verification. It proposes a general way to prove properties for the system which is also successful to prove temporal logic properties. The technique [30, 5] uses the synchronous product of STS and first-order logic to write temporal properties.

In this paper we reused some principles coming from our previous work: component, architecture, symbolic state machine and the synchronous product. Note that in our previous work we take into account full description of component and system with abstract data types. But here we are mainly interested in behavioural descriptions and we avoid full data type descriptions. However sometimes we need guards in the state machine descriptions. Some of our current work may be analysed with other concepts like Petri Nets or automata. However to take into account, hierarchy, guard and full data type in a uniform way would be more difficult.

## 2.1 An Example

We model a simple example with four concurrent components, a previous version with synchronous communications is [27]. This is a part of a flight seat reservation system with a component for the seat reservation, one for simulating the bank, one for the flight company and a last one for the client. The client gives its account number when he requests a seat to the counter. The counter asks the company to know if there is a seat. This may fail or succeed, in this last case the seat reservation orders the price to the bank. The order may fail or if it succeeds then the counter prints a ticket and the company books the reservation.

## 2.2 Component Principles

Strictly speaking we model types of component rather than components but this has no consequence on our current discussion. Another common choice is to consider atomic components and to define complex components as an architecture of asynchronously communicating subcomponents.

Asynchronous communication distinguishes message sending and its execution. If  $\text{op}$  is an operation call we note  $\text{>op}$  the message sending and  $\text{op>}$  its execution. The emitter does not memorize messages; this is done by the receiver in a specific buffer. This buffer acts as an asynchronous channel but it is part of the component as for an actor. Sometimes operation may be executed but no message receipt is needed: we call them autonomous operations and simply note them  $\text{op}$ .

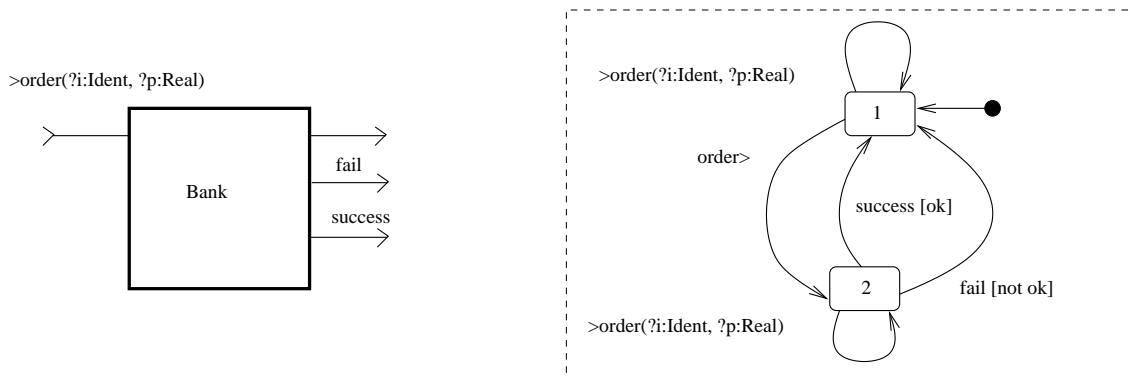
Thus we define the following vocabulary:

- *Autonomous operation*:  $\text{op}$  is an action which does not need a receipt to be executed. It takes into account the fact that a component may know sufficient information to trigger this action.

- *Receipt*:  $\>op$  means the receipt of a message in the mailbox. A specific guard [`not fullMailBox`] is used to ensure that the mailbox is not full.
- *Action*:  $op\>$  denotes an operation which will be triggered when the component receives the corresponding message receipt ( $\>op$ ). The fact that the component received the message is denoted by the guard [ $\&op$ ].

Message sending is done during the execution of an operation. The notation  $op\>$  stands for the usual provided service and  $\>op$  for the required service of many other component languages. Autonomous operations are not strictly fundamental, we may see them as a kind of syntactic sugar. Mailboxes are buffers for messages received by a component.

Graphically a component is a box with *input pins* corresponding to message receipts and *output pins* corresponding to message sending. Input pins (resp. output pins) are put on the left of the component (resp. on the right). An autonomous operation has only a right output pin, other operations have a left input pin (receipt) and a right output pin (sending). For example the bank is represented in the left part of Fig. 1. There is an operation for message receipt  $\>order$  with two arguments: the account number of the client ( $?i:Ident$ ) and the price of the ticket ( $?p:Real$ ). The operations `fail` and `success` are autonomous. The different pins may be (or may not be) connected in a given architecture, it expresses the receipt of messages (on the left) or message sending (on the right).



**Fig. 1.** The Bank Component and Its Dynamic Behaviour

The previous figure describes the interface part of the bank, we also take interest in the dynamic behaviours or protocols. Such a protocol is represented in the right part of Fig. 1. Note that each state has transition loops for message receipts,  $\>order$  in the bank example. A sender does not block except if the buffer is full and nothing else is possible, this is also true for the receiver except if the message buffer is empty and without autonomous operation. Indeed this behaviour may be computed as the free concurrent product of a buffer which receives messages and a business dynamic behaviour for the bank as illustrated in Figure 2.

In the sequel we only present the simplified dynamic behaviour of the atomic components. Hence a component will be described by a box with pins outside and a business dynamic behaviour inside.

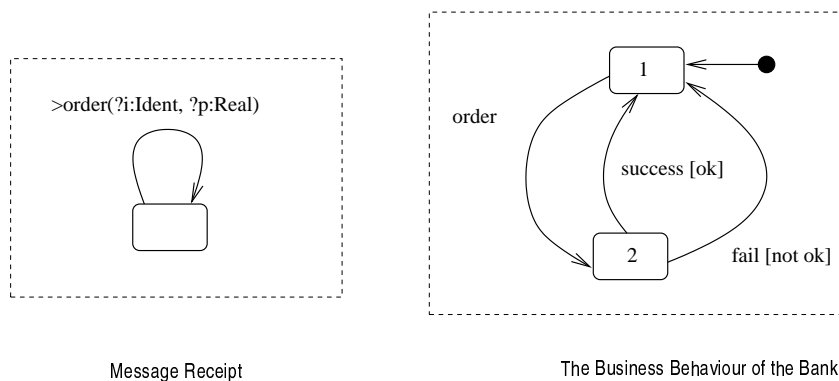


Fig. 2. Decomposition of the Dynamic Behaviour of the Bank

### 2.3 The counter, the client and the company Components

Fig. 3 represents the component for the counter and the meanings of its ports. The two other components: company and client are respectively depicted in Fig. 4 and 5.

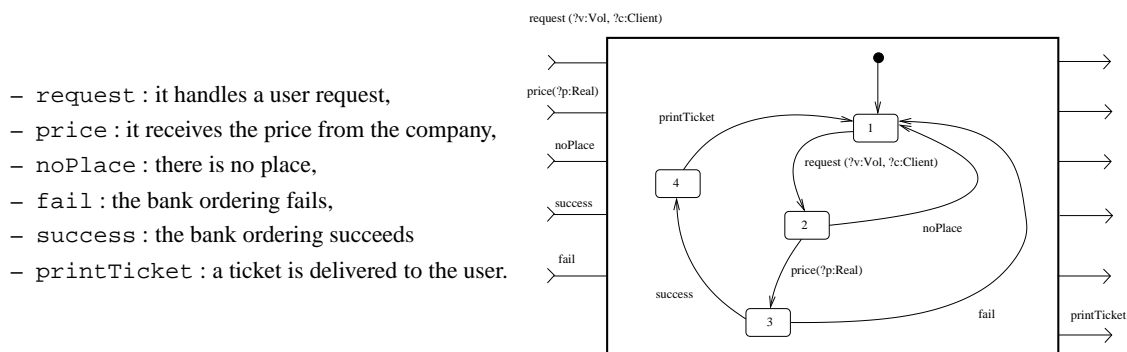


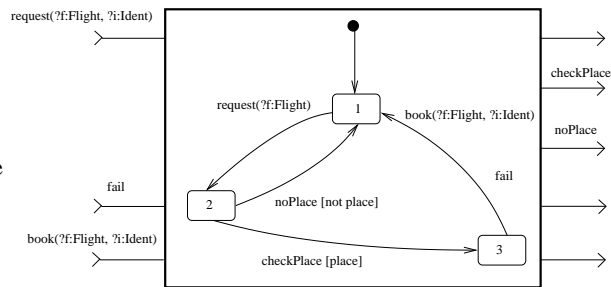
Fig. 3. The Counter Component

### 2.4 The Architecture

One key issue in designing good architectures is to separate the communications from the component to get more reusability. This has several consequences: to have a local naming of operation per component and to define a glue language to denote communications in the architecture. Figure 6 represents the architecture of our flight reservation system. A message transmission is graphically denoted by a thin line from an output pin to an input pin. Some pins like request> and book> for the company, fail for the counter or order> for the bank are not connected since they do not send messages in this configuration.

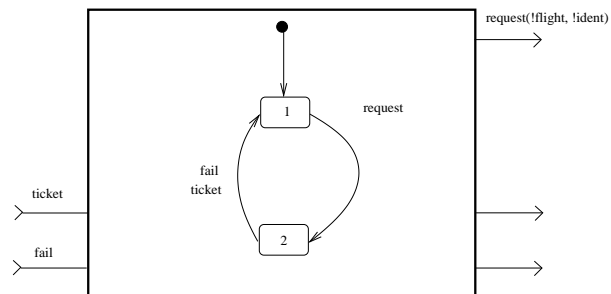
To simplify the figures we avoid the effective arguments and guards of messages and transitions. Such an architecture may be transformed into a component, however nested components are not required in this example.

- request : the request has been received,
- checkPlace : it checks if there is some place,
- noPlace : there is no place,
- fail : the reservation fails (coming from the counter),
- book : the reservation succeeds.



**Fig. 4.** The Company Component

- request : the client asks for a flight reservation,
- ticket : it get its flight ticket,
- fail : the reservation fails.



**Fig. 5.** The Client Component

## 2.5 Some Advanced Communication Schemes

We provide some other communicating schemes which are represented in Fig. 7.

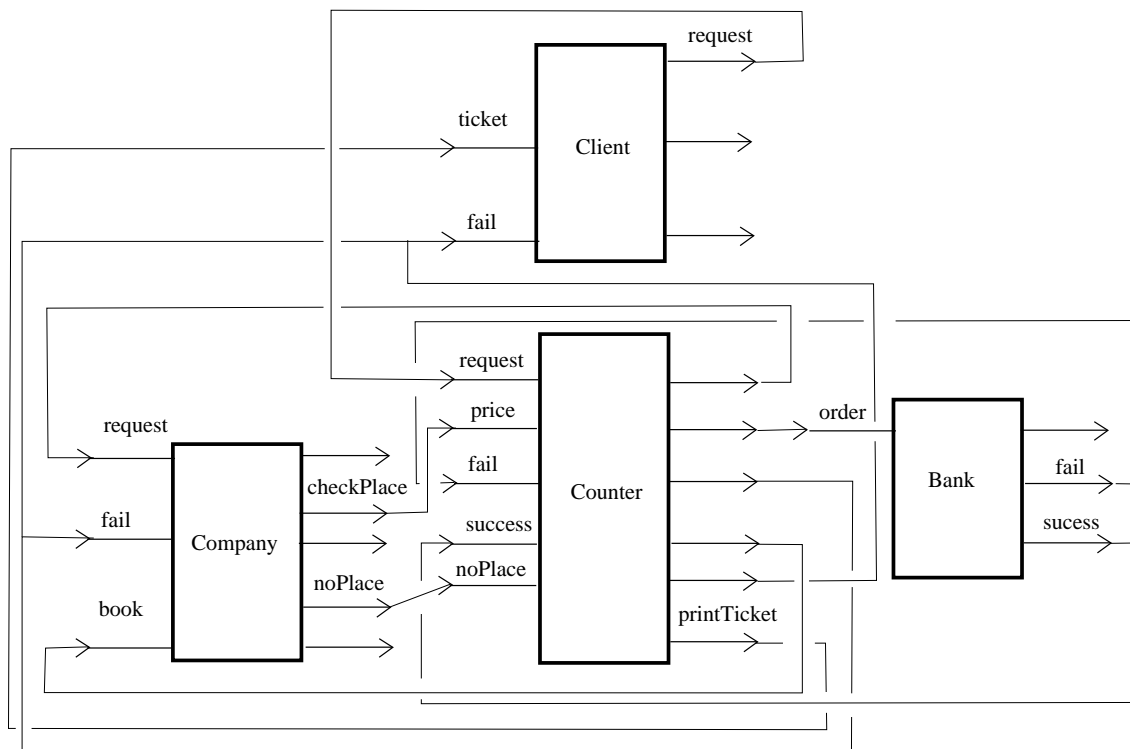
- *Broadcasting* : to broadcast a message a link is drawn from the emitter pin to the destination pins.
- *Multiple inputs* : one input pin is linked with several output pins, it obviously denotes exclusive communications.
- *Conditional Communication*: in some cases we need to send messages, under some conditions, to several pins. For example an advanced design of our architecture may connect the output pin `request` of the company to the input pins `fail` and `success` of the counter component. In this case we add the guards `[noPlace]` and `[place]` to control the exclusivity of the communications.

An architecture may mixed these different schemes, provided that some rules are checked. The use of guards on links provides dynamicity in communications which is an important feature of distributed computing. For instance in our example, guards in communications allows us to get a stable architecture relatively to the number of clients. The current behaviour of the counter serializes the client requests but it is possible to change it and to allow request interleaving.

## 2.6 To Build the Architecture

To elaborate such an architecture we briefly recall the main steps:

1. To decompose the system and to identify atomic components.
2. To specify each atomic components:



**Fig. 6.** The System Architecture

- to define its input and output pins and its name and parameters,
  - and to draw the business dynamic behaviour with a STS.
3. To architecture the system with the subcomponents and their communications.

Of course this is a simple view and there are many difficulties ; a full discussion is out of the scope of this paper, see [28, 11] for more details.

We implemented the flight reservation system in ProActive [1] which is an asynchronous language based on Java. Every atomic component is mapped to a ProActive component composed of an active object and some internal passive objects. An input pin of a component is easily transformed into a public method of the active objects with the correct name, type and parameters. For the output pin of a component we have to implement its action and the message emissions. For the action part, the output pin is mapped to the public method of the corresponding active object. The message sending from an output pin to an input pin is translated into a method call inside the process of the ProActive component associated to the receiver. These experimentations confirm the feasibility and the simplicity of such an implementation and give us a first view of translation rules from our component model to the ProActive language.

### 3 To Analyse the Global Behaviour

As explained in the introduction asynchronism gives more complex descriptions and behaviours. Generally a synchronous model does not work if it is embedded in an asynchronous communication framework, it

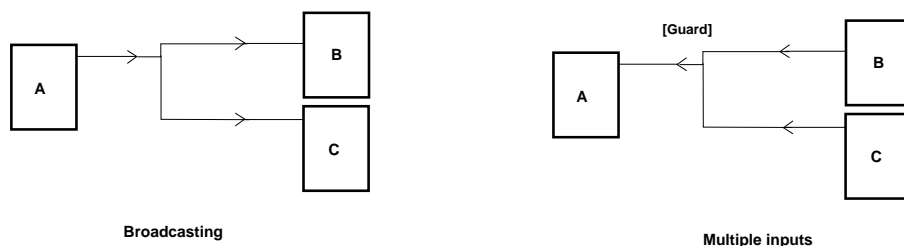


Fig. 7. Other Communicating Schemes

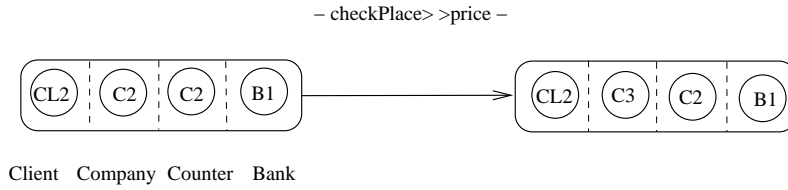
must be redesigned carefully. It is often mandatory to provide a failure associated to each business action, for example in some previous designs we avoid `fail` for the company and for the client. Our analysis shows some wrong results (deadlocks, unbound buffers, ...) but it was definitively not prominent in the architectural design. Another example of problem was the occurrence of a deadlock in a bus controller due to a wrong choice in the buffer strategy. Thus once the components and the architecture have been designed, the problem is to get some confidence in the global behaviour. We first compute the global protocol from the component protocols and the architecture.

### 3.1 The Concurrent and Communicating Product

There are several ways to express concurrency, synchronisation and communication. Mainly there are: process algebra expression, temporal logic formula or state machine. Our semantics of concurrency is based on the synchronous product of STS associated to the components. The synchronous product originating from [8] has been adapted to our STS [31]. Each state of the product of STS is a compound state which has two or more inner states corresponding to the component states. The transitions of the product are also compound in the way depicted in Fig. 8. To take into account the fact that a component may act asynchronously, we use a special nil transition noted `-`. A message sending, from the `foo` output pin to the `bar` input pin, is represented as a synchronisation between the emitter when it triggers `foo>` and the (buffer) receiver when it triggers `>bar`. The message is received in the mailbox of the receiver and asynchronously executed by the receiver. Thus the message sendings of the architecture are translated into synchronisations and expressed by the so-called *synchronous vector*.

One example of synchronisation concerns the emission of the `>price` message to the `counter` by the `checkPlace>` operation of the `company`. During this communication the first and the fourth component (the client and the bank) do nothing. This is a structured transition (`- checkPlace> >price -`) which starts from a structured state `CL2, C2, C2 B1`, which means `client` in state 1, `company` in state 2, `counter` in state 2 and `bank` in state 1. In this example (`- checkPlace> >price -`) is an element of the synchronous vector. The drawing of the global behaviour, in a real case study, becomes too complex, but it can be automatically computed from the component protocols and the architecture.

The synchronous vector collects the tuples of operation which are allowed to synchronize. The general conformance rule is one output pin and one or more input pins, and if an input pin receives more than one message a guard must guaranty the exclusivity of the messages. There are several possible synchronisation



**Fig. 8.** Communication Representation

rules to handle operations not in the synchronous vector (as in LOTOS or CCS for example). Here we choose the same as in LOTOS [20] since we do not have other synchronisation or communication.

From the architecture we built the synchronous vector in the following way:

- Basically a communication link from the output pin  $f_{oo}>$  to input pin  $>bar$  leads to a tuple with  $f_{oo}>$  and  $>bar$  at the right place and – elsewhere.
- Broadcasting simply extends this principle to more than one input pin.
- Multiple inputs express a quantification over output pins, it is translated into several broadcasting communications, one for each output pin. It must result in a legal synchronous vector, we assume that guards on multiple inputs are exclusive.
- A conditional communication is considered as multiple communication links with guards.

### 3.2 Some Remarks

The global product is a symbolic machine however it may be complex. This is partly due to the complex nature of asynchronous model. We obtain a system with 48 states and 296 transitions from our architecture which has a rather small size. The similar example with synchronous communication mode, see [27], has nearly 10 states and 15 transitions. However we must precisely compare it with a synchronous system simulating asynchronous communication. A quick analysis shows that the number of states and transitions would be nearly the same. This increases the need for analysis tools, preferably automatic tools.

Of course some analysis may be conducted using Petri Net tools, model-checkers, or other automata related tools. Generally our systems are not adequate for this and a preliminary translation is needed. However the main reason to try another way is that we have structured systems with data types and guards. Thus we need a powerful approach taking into account the full description of the system. From this powerful approach it seems relevant to propose more specific but automatic tools.

Once we get the global behaviour it is a structured STS and it is not really adequate for tools like classic model-checkers. However several transformations may be done to get a labelled transition system, see [30] for example. The general idea is to simulate the STS choosing some limits for data types. Here we need a similar transformation, we only consider the  $[\&op]$  and  $[not\ fullMailBox]$  guards and the contents of mailboxes. The  $[\&op]$  means that there is at least one  $>op$  message in the mailbox and  $[not\ fullMailBox]$  checks if the mailbox is full or not. In our example it is obvious that some actions cannot be triggered since the corresponding message has not been received. Another interesting fact is about the size of the mailboxes, it must be bound or not.

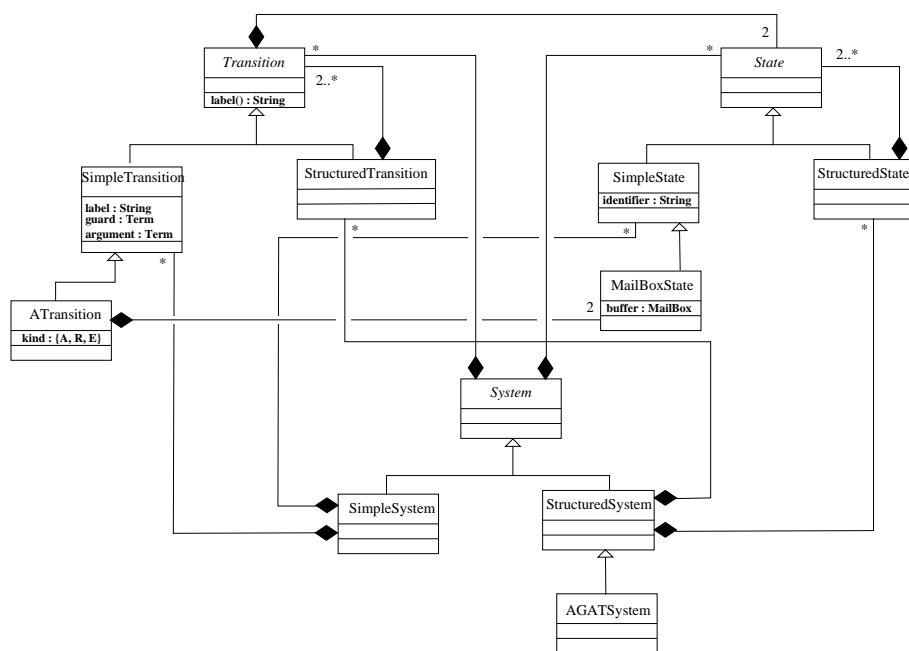
We think that it is better to define several dedicated algorithms to increase the reusability by allowing the composition of analyses. A first algorithm, called the `width` algorithm, produces a simulation of the

system taking into account fixed capacity mailboxes. To handle unbound mailboxes is more subtle, this is the goal of the bound algorithm. To have both, fixed capacity mailboxes and unbound mailboxes, may be done in two ways. The first is to process the `split` algorithm after a bound analysis. The `split` algorithm duplicates the states with fixed capacity mailboxes and the edges starting from these states. The second way is to define a specific algorithm mixing the `bound` and the `split` algorithms. Since the most original algorithm is the `bound` one we avoid a comprehensive discussion about the `width` and the `split` algorithms.

### 3.3 The Bound Analysis

We have designed a bound algorithm which is able to simulate the behaviour coping with unbound mailboxes. The algorithm searches in the dynamic system and computes the states with their mailbox contents. When a mailbox has a possible infinite contents a star is put to avoid the construction of an infinite set of states. We have experimented several examples and the different experimentations are based on dictionaries which memorizes the number of received messages.

**The Data Structures.** We briefly give a look at the data structures used to represent the dynamic behaviour of our systems.



**Fig. 9.** The UML Class Diagram for Structured STS

These data structures are described with a UML class diagram in Fig. 9, but we avoid some specialisations and constraints to do not overload too much the diagram. Simple states are simply identifiers, we have

also structured states, both are organised along a composite pattern. A simple transition is a source state, a target state, a label plus various parameters, we also structured them along a composite pattern manner. The `ATransition` class describes three kinds of transition labels: autonomous (A), receipt (R) and action/emission (E). These transitions have mailboxes in their source and target states. A labelled transition system is a set of simple states, and a list of simple transitions. A structured system is compound from structured states and structured transitions. The `AGATSystem` class is used to represent the dynamic behaviour of our architecture and it is the input and the output data structure of the bound algorithm. The `AGATSystem` class defines structured labelled systems where transitions are compound from `ATransition` instances. It seems possible to have a general `Mailbox` class which may be specialised to code any specific buffer strategy. However our actual algorithm only deals with dictionaries of the number of received messages.

**The Algorithm.** The goal of the bound algorithm is to compute the complete simulation of a structured system and to put `*` when the mailbox capacity becomes unbound. In the algorithm page 12: `[ , ]` are lists, `( , )` tuples, and `<-` assignments. The algorithm takes as input a `self:AGATSystem` and produces another `AGATSystem` instance with, as labels, the labels of the input system and the contents of the mailboxes. Mailboxes are dictionaries counting the number of received messages, it may contain star which represents an infinite number of messages. A mailbox overlaps another one if the former has the same values except some stars. The algorithm ensures the following invariant: for all label of `self`, and for all reachable mailbox contents, it exists a single output state with the same state label and a mailbox which is the same or an overlapping one. In other words it computes a finite accessibility graph associated to the input system. Basically this algorithm visits the state, the `listState` contains the new states to visit. The `history` and `listHistory` are variables used to detect cycles in the input system. The `history` variable denotes the list of visited states from the initial one to `currentState`. `newState` denotes a reachable label with a new mailbox contents. There are four exclusive cases:

1. `newState` already exists in the current result or it exists an overlapping state (`overlap` function); in this case we simply add a new transition.
2. `newState` exists in `history` but with a lesser mailbox contents (`superior` function), in this case we have a cycle and we must propagate stars in the graph. `propagateStar` propagates stars and also rebuilds the graph since to add stars may collapse several existing states.
3. If there is no cycle there may be existing states in the current result which are overlapped by `newState`. In this case we consider one of them and we propagate stars and collapse states as in the previous case.
4. The last case is the simplest since it adds the new state and a new transition in the resulting graph.

Careful attention must be paid in already existing states and loops which complicates a little the management of histories. This algorithm needs to define several operations to compare mailboxes, mainly the `overlap` and the `superior` functions.

- `overlap(d1, d2): d1 != d2` and for all `msg, d1[msg] = * or d1[msg] = d2[msg]`
- `superior(d1, d2): d1 != d2` and for all `msg, d1[msg] => d2[msg]`

where `d[msg]` is the number of received messages of name `msg` in the `d` dictionary.

The algorithm looks like this:

```

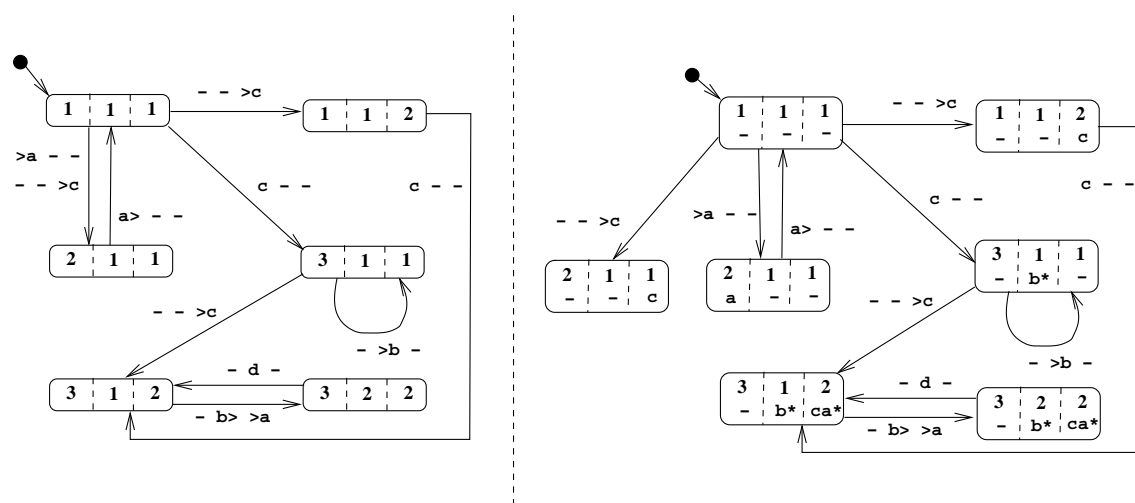
bound : self:AGATSystem ----> AGATSystem

BEGIN
listState <- [initial of self]           # list of states to visit
historyList <- [[]]
history <- []
newAGAT <- init the resulting Atag
WHILE listState != empty DO             # main loop on states to visit
  currentState <- first listState       # current state of newAGAT
  stateLabel <- identifier(currentState)
  mailbox <- buffer(currentState)
  history <- add currentState in historyList.pop()
  listNeighbours <- neighbours of stateLabel in self
  WHILE listNeighbours != empty DO     # visit the neighbours of the currentState
    transition <- listNeighbours.pop()
    IF transition is possible from mailbox THEN
      newMailBox <- apply transition to mailbox
      target <- target(transition)
      label <- label(transition)
      newState <- createState(target, newMailBox)
      IF newState already exists in newAGAT # the state already exists
        THEN
          newAGAT.addTransition(label, currentState, newState)
        ELSE
          config <- find the mailbox <= newMailBox in history
          IF config exists
            THEN
              indices <- newMailBox.dicoIndices(config)
              newAGAT.addTransition(label, currentState, config)
              removeStates <- newAGAT.propagateStar(newState, indices)
              removeStates from history, listHistory, listState
            ELSE
              config <- newAGAT.findInf(newState) # finds an overlapped state
              IF config exists
                THEN
                  newAGAT.addTransition(label, currentState, config)
                  config.copyNotOmega(newTable)
                  removeStates <- newAGAT.propagateStar(newState, newTable.findOmega())
                  removeStates from history, listHistory, listState
                  IF not config in listState
                    THEN
                      add newState in listState
                      add newState in historyList
                    ENDIF
                ELSE
                  newAGAT.addState(newState) # simple addition
                  newAGAT.addTransition(label, currentState, newState)
                  add newState in listState
                  add newState in historyList
                ENDIF
              # overlapped config exists
            ENDIF # config exists in history
          # newState exist
        ENDIF # transition
      remove currentState from listState
    ENDWHILE # listNeighbours
  ENDWHILE
END

```

**Application Examples.** We have processed several examples of protocols, communication patterns, and some simple applications. The bound algorithm gives some relevant informations about the dynamic behaviour and helps to improve the architecture design.

A fictitious example is described in Fig. 10 and the result shows the mailbox contents which are dictionaries of the number of received messages. It demonstrates a deadlock in state  $(2, 1, 1)$  after the receipt of a  $c$  message on the third component. States  $(3, 1, 1)$ ,  $(3, 1, 2)$  and  $(3, 2, 2)$  have stars in their mailboxes since they are parts of cycles in the graph (a loop and a 2-states cycle) which accumulate receipts.



**Fig. 10.** A Fictitious AGATSystem Example and its Resulting Analysis

Figure 11 shows the simulation of the flight reservation system with dictionaries of messages. Now the system is reduced to 22 states and 31 transitions and represents the global behaviour of the system in a more concise form. We have a bound system and we may use model-checking to analyse its properties. It also shows that we can optimise the data structure for buffers since we have at most two messages in each buffer. In this example the result is simpler in terms of state and transition, that is not always the case. However the bound algorithm simplifies the global behaviour in the sense that it removes some impossible transitions and it provides a compact description for infinite mailboxes. Note that in this simulation we have only one client, then it is important to extend it to any number of clients. Of course it is possible to compute this simulation when the maximum number of client is known. For example with 3 clients we get a global system with 192 states 2048 transitions, the bound analysis gives a result with 1174 states and 3809 transitions. But if the maximum number of clients is not known our current approach is not able to compute the global behaviour of the system. In this case one solution is to extend our symbolic machines to take into account  $n$ -ary state and transition. Another way would be to use an algebraic or a temporal logic description of the dynamic behaviour. This is a point which needs future researches.

The datas and the algorithm have been implemented in Python [23] and also in Java. Two different versions of the algorithm (a breadth-first and a depth-first traversal) have been checked. We are currently proving the algorithm and analysing its complexity. At first glance the complexity seems lesser than  $O(n \times$

$m^2 \times max^{\#msg}$ ) where  $n$  and  $m$  are the number of edges and the number of states of the input system,  $max$  is the maximum capacity of the bound mailboxes and  $\#msg$  the number of operation receipts. We expect to improve efficiency, one related and feasible optimisation is to compute directly this analysis from the architecture without calculating explicitly the global product.

## 4 Related Work

We first briefly comment some other possible choices about our model. One first choice was to explicit receipt and action/emission. A finer approach would be to distinguish receipt, action and emission. In fact we think that emission or receipt need to be associated with action else we cannot get a right model where events result from some activities related to the components. Another point is about the association of receipts and actions in the dynamic behaviour. We have at least three main possibilities. The first is to sequentialize the message receipt  $\text{>op}$  and its execution  $\text{op>}$ , but this blocks the sender as with synchronous communication. A less blocking policy is to have a receipt loop on each source state for  $\text{op>}$ . A more liberal than the latter one, which is used here, is to receive a message in each state. Notes that these three choices change neither our overall model nor our algorithms, it only impacts the dynamic behaviour of the atomic components.

We assume here safe communication but unsafe ones may be simply simulated by emission transition without synchronisation and reception. We have also an approach which provides synchronous communications without difficulty in a uniform context. Lastly, it is possible to take into account non-determinism in communication by the use of guards.

Our component and architecture description is related to architectural description languages (ADL), see [24] for a good survey. We have atomic and complex components with interfaces and dynamic behaviours. Our approach gives a way to specify mixed systems *i.e.* with both full data types and dynamic behaviours. Here we only present a graphical representation of the architectural description language, this is not generally sufficient for automatic processing and full code generation. A main difference is the use of asynchronous communications while most of the time ADLs promotes synchronous communications.

At this stage it is interesting to compare our approach with WRIGHT [7, 6]. WRIGHT is a formal architectural description language with first class components and connectors. It may be seen as relook of CSP, since the notations and the semantics are inspired by this process algebra. A component has a set of ports and a behaviour part. A connector defines a set of roles and a glue specification. Roles are expected to describe the local behaviour of the interacting parts. The glue describes how the local activities of the different parts are coordinated. The semantics of these constructions is defined by a translation into CSP. This has the advantages to get effective model checking for CSP and related work about behavioural refinement. However, most of these verifications are limited by the state explosion problem and consider simple data types. WRIGHT proposes a deep analysis about automatic checking for architectural languages. It allows connector consistency, configuror consistency, and attachment consistency using mainly techniques to prove deadlock freedom and behavioural refinement. We improve readability by graphic notations, this is important for large scale applications. WRIGHT is not really adequate for this due to several reasons: no graphic presentations and no-ary composition. In our approach we consider both dynamic and functional properties not only dynamic properties with restricted data types. This is a first important difference but others are the use of symbolic transitions systems and asynchronous communications.

One related work to our communicating component is [15]. This paper presents a model for concurrent distributed object with asynchronous communications. The semantics of the model is based on labelled event structures and a new communication relation that expresses asynchronous object interaction.

Except quoted work about Petri Nets, we have not yet found a related mailboxes analysis. The use of Petri Net and the reachability/coverability algorithms [25] may solve our mailbox contents analysis, but it needs, at least, a translation into the Petri Net world. However we have a structured systems with data and guards and we want to define not only this bound analysis but other in the future. Then we think more appropriate to define a specific approach. Our algorithm is also different on several point: we have abstract buffers (not only marks), we have a complete simulation (all the states of the result are reachable), and we do not compute a tree but directly the reachability graph. We have an algorithm, limited to our special case, which represents the reachability graph, even if the system is not bound, using stars as in the coverability algorithm.

Model checking is a technique to verify automatically temporal properties of finite state systems. Tool examples are CADP, MEC, VIS, see [8, 14] for more details. Model checking is useful to quickly prove deadlock or other related properties. The state explosion problem of model checking can be limited by using BDD coding. This technique is called *symbolic model checking*, although it does not address the worst-case complexity of the problem. In practice it is useful and allows the verification of very big systems with more than one million of states. It is possible to compute the set of the reachable states which verify some properties like: one  $\text{>op}$  must always occur before an  $\text{op}>$ . But it is more difficult to cope with guards like `[not fullMailBox]`. However model-checking may be helpful to quickly prove some properties of the finite system forgetting guards and variables. For example, to know if the finite state machine has a deadlock or if there are transitions starting from a given state, and so on. There are two main points which are generally not covered by classical model checking and temporal logic. The first is the use of guards, variables and full data types. The second is that we do not deal, generally, with a bound labelled transition system. There are several different approaches which introduce variables and guards in model checking. Amongst them the  $\mu\text{CRL}$  language is relevant to our problem. The temporal logic proposed seems complex to read and to understand. We have not yet investigated the comparison in depth, but clearly our approach is more suited to engineering practice (first-order logic framework). We also have more readable state-transitions systems and a more powerful framework.

In [29] the authors propose a notion of stuckness, that is to eliminate programs waiting to receive or trying to send messages in vain. That is a more formal but complementary work in the context of unsafe communications.

## 5 Conclusion

We provide an approach to design component and architecture with asynchronous communications and dynamic behaviours. To handle asynchronous communications we distinguish emission and receipt operations rather than the use of specific buffers to memorize messages. We show how to compute the global behaviour of an architecture and to represent it without lost of information. We propose an algorithm which simulates such a dynamic system and computes the reachable configurations of the mailboxes. The algorithm is able to decide if a mailbox has a bound size or not. The result may be used to verify the dynamic behaviour

but also to optimise the architecture deployment. This also may be used as a general framework for other class of systems like actors, multi-agent, synchronous communication, or channel based communications. We have done sensible choices which are not too constraining and our current model is able to cope easily with extensions to unsafe and non-determinism of communications. This approach may be generalised on the mailbox policy.

One trend of future researches is to extend our approach to cope with other analysis, for example with guards in communications. Another one is to consider a variable number of components. These are important features to fit with more realistic systems.

## References

1. ProActive. <http://www-sop.inria.fr/oasis/ProActive/>.
2. Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.
3. Gul Agha. Concurrent Object-Oriented Programming. *Communication of the ACM*, 33(9):125–141, September 1990.
4. Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 19–40. Springer-Verlag, Berlin-Heidelberg-New York, 1985.
5. Michel Allemand and Jean-Claude Royer. Mixed Formal Specification with PVS. In *Proceedings of the 15th IPDPS 2002 Symposium, FMPPTA*. IEEE Computer Society, 2002.
6. Robert Allen, Remi Douence, and David Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer-Verlag, 1998.
7. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
8. André Arnold. *Finite Transition Systems*. International Series in Computer Science. Prentice-Hall, 1994. ISBN 0-13-092990-5.
9. Denis Caromel. Object Based Concurrency: Ten Language Features to Achieve Reuse. In *Workshop on Object Based Concurrency and Reuse*, Utrecht, the Netherlands, June 1992. ECOOP'92.
10. Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.
11. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. Wing and J. Woodcock and J. Davies, editor, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 939–962. Springer-Verlag, 1999.
12. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Formal Specification of Mixed Components with Korrikan. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference, APSEC'2001*, pages 169–176. IEEE, 2001.
13. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Specification of Mixed Systems in KORRIGAN with the Support of a UML-Inspired Graphical Notation. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001*, volume 2029 of *LNCS*, pages 124–139. Springer, 2001.

14. Edmund M. Clarke, Orna Grumberg, and David E. Long. Verification tools for finite-state concurrent systems. In *A Decade of concurrency – Reflections and Perspectives*, volume 603 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
15. G. Denker and J. Küster-Filipe. Towards a Model for Asynchronously Communicating Objects. In *Proc. 2nd Int. Baltic Workshop on Databases and Information Systems*, pages 182–193, 1996.
16. Wolfgang Emmerich. Distributed component technologies and their software engineering implications. In *Proceedings of the 24th Conference on Software Engineering (ICSE 02)*, pages 537–546. ACM Press, 2002.
17. Wolfgang Emmerich and Nima Kaveh. F2: Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 311–312. ACM Press, 2001.
18. David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
19. Kohei Honda and Mario Tokoro. An object calculus for synchronous communication. In Pierre America, editor, *European Conference on Object Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, Geneva, Switzerland, 1991. Springer-Verlag.
20. ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.
21. Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.
22. Leslie Lamport and Nancy Lynch. *Distributed Computing: Models and Methods*, pages 1156–1199. Elsevier Science Publishers, 1990.
23. Mark Lutz. *Programming Python*. O'Reilly & Associates, 1996.
24. Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
25. T. Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
26. OMG. CORBA Component Model Specification, v3.0. Technical report, Object Management Group, 2002. [www.omg.org/technology/documents/](http://www.omg.org/technology/documents/).
27. Liang Peng, Anya Romanczuk, and Jean-Claude Royer. A Translation of UML Components into Formal Specifications. In Theo D'Hondt, editor, *TOOLS East Europe 2002*, pages 60–75. Kluwer Academic Publishers, 2003. ISBN: 1-4020-7428-X.
28. Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Concurrency and Data Types: a Specification Method. An Example with LOTOS. In J. Fiadero, editor, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 13th Workshop on Algebraic Development Techniques, WADT'98*, volume 1589 of *Lecture Notes in Computer Science*, pages 276–291. Springer-Verlag, 1999.
29. Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. *Lecture Notes in Computer Science*, 2404:166–179, 2002.
30. Jean-Claude Royer. Formal Specification and Temporal Proof Techniques for Mixed Systems. In *Proceedings of the 15th IPDPS 2001 Symposium, FMPPTA*, San Francisco, USA, 2001. IEEE Computer Society.
31. Jean-Claude Royer. The GAT Approach to Specify Mixed Systems. *Informatica*, 27(1):89–103, 2003.
32. T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.
33. Mike Wooldridge and P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In P. Ciancarini and M. Wooldridge, editors, *First Int. Workshop on Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in Computer Science*, pages 1–28. Springer-Verlag, Berlin, 2000.
34. Pamela Zave and Michael Jackson. A component-based approach to telecommunication software. *IEEE Software*, 15(5):70–78, 1998.

