7e Conférence Internationale de MOdélisation et SIMulation - MOSIM'08 - du 31 mars au 2 avril 2008 – Paris- France
« Modélisation, Optimisation et Simulation des Systèmes : Communication, Coopération et Coordination.».

SPECIFICATION AND PROTOTYPING OF REACTIVE DISTRIBUTED SYSTEMS BASED ON CONCURRENT OBJECT-ORIENTED FORMALISM

F. BOUCHOUL, M. MOSTEFAI

Laboratoire d'Automatique de Sétif (L.A.S)
Université Ferhat Abbas Sétif Algérie
fbouchoul@univ-setif.dz , mostefai@univ-setif.dz

K. BARKAOUI

CEDRIC-CNAM
Paris France
barkaoui@cnam.fr

ABSTRACT : The specification of the dynamic behaviour of reactive distributed systems must exhibit the structures of control and has to imply explicitly the relevant aspects of the distribution, such as the concurrency, the reactivity and the interaction between the entities. The most common reactive distributed systems are industrial ones; distributed networks occur for example in telecommunications, Internet, power and energy, transportation and manufacturing. Distributed computing will play an increasingly critical role in the global industrial-infrastructure. The need for trustworthy systems has received tremendous researchers' attention. The usage of formal tools for simulation and prototyping designed to facilitate the modelling of such systems is of great interest. Improved methods are needed to insure reliability, security and robustness of industrial distributed systems. This paper proposes the fundamentals of a formal approach for the specification of reactive distributed systems based on object-oriented paradigm. Object's behaviour is modelled as REACTNets. The REACTNets enhance the ECATNets that are a kind of high level algebraic Petri nets with explicit distribution and reactivity. Furthermore we associate to the classic ECATNETS MAUDE rules to handle interactions between objects. The two formalisms have a common semantics in term of rewriting logic so that interesting prospects are opened for their integration.

KEYWORDS : *reactive distributed systems, object oriented paradigm, rewriting logic, ECATNets, Maude, rapid prototyping*

1. INTRODUCTION

Industrial systems are inherently and massively distributed. A distributed system can be seen as a number of heterogeneous and autonomous entities which can interact by the means of suitable interfaces. The complexity of these systems increase with the number of entities which compose them. Various works relating to formal modelling of such systems are continuously proposed for the purpose of verification or rapid prototyping; each one with different objectives, concepts, tools and possibilities.

These models must exhibit the structures of control and have to explicitly imply the relevant distribution aspects, such as the concurrency, the reactivity and the interaction between the entities. In particular, the expression of concurrency and reactivity constitutes a crucial aspect during the development of the model. Concurrency can arise between the system entities (inter-entities concurrency) and also inside the same entity (intra-entities concurrency). Reactivity deal with the possibility for the system to react dynamically to its environment.

Thanks to their logical autonomy and to their modularity, objects are naturally predisposed for the role of concurrency units. Not only they make it possible to describe the structural properties of the system but also to handle naturally the distribution (Wegner 1990).

However, the object oriented approach presents an evident weakness to suitably express the dynamic aspects of distributed systems. For this reason, the objects are often enhanced with a formalism for the description of the dynamic aspects of their behaviour. In particular, the approaches associating PNets and objects are more and more gaining the interest of several groups of researchers.

The aim of this work is to propose the fundamentals of a formal approach for the specification of distributed reactive systems with a true concurrency semantics at inter- and intra-entities level. The idea is to integrate the ECATNets (Bettaz et al. 1993) and the theory of concurrent objects proposed by Meseguer in MAUDE (Meseguer 1992),(Clavel et al. 2003). The ECATNets (Extended Concurrent Algebraïc Term Nets) are a kind of high level algebraic Petri nets with a rewriting logic semantics.

First we propose the REACTNets that enhance traditional ECATNets with reactivity. REACTNets should be used to describe individual objects' behaviours so that to provide them with a true intra–object

concurrency; and to express not only the actions which the object carries out but also its interactions with its environment in term of messages emitting/receiving.

2. OBJECT/PNET ASSOCIATION FOR DISTRIBUTED SYSTEMS.

2.1 The object/PNet complementarity

Object/Pnet association is based mainly on the interesting complementarity of the two formalisms for the specification of distributed systems. The PNet deal with the most crucial aspects of concurrency, objects offer the tools necessary to express the various distribution aspects.
Furthermore, distributed systems are often reactive and the behaviour of a reactive system is usually modelled by event-condition-actions rules called commonly production rules or simply ECA rules (Event-Condition-Action); The significance of an ECA rule is that if the event in the environment occurs, and the condition is true, the reactive system performs the action.
The problem is that the token-game semantics of Petri nets does not model behaviour of reactive systems, the non-reactivity of the token-game semantics can be seen immediately from the definition of the firing rule. A transition in a Petri net is enabled once the conditions of firing are true however the environment of the Petri net does not influence in any way its firing. In contrast, in a reactive system a relevant transition needs some additional input event to become enabled. So, the token-game semantics models closed systems, whereas a reactive system is open, otherwise it cannot interact with its environment. Furthermore, in a reactive system an enabled transition must fire immediately. In the token-game semantics, an enabled transition may fire, but does not necessarily have to.
A Reactive PNet can simply be built by changing for internal transitions the rule "the transition may fire" by the rule "the transition must fire"; (Eshuis and Dehnert 2003); while for external transitions expressing the interactions with the environment the traditional rule can be preserved to ensure the network stability. Thus a reactive PNet has two possible states: stable and unstable. The system must continue to fire the internal enabled transitions as a long time as it does not reach a stable state, in other words until no internal transition is enabled; before being able to fire external transitions from the environment. But, the PNet must explicitly comprise sufficient constructions to model the interaction with the environment by external transitions handling the events that influence its internal behavior and expressing the reactivity. For this purpose object paradigm offers to PNets an elegant solution. And we can conclude that the complementarity of the two approaches is twofold, on one hand objects need PNets to express their dynamic behaviour and on the other hand PNets need objects to have modularity and reactivity through object interaction mechanisms.

2.2. Object / PNet Approaches: state of the art.

The object/PNet association is not new, and among the multitude of works integrating objects and PNets, two tendencies are distinguishable, designated successively by "Objects in the Petri nets" and "Petri nets in the objects" (Bastide 1995). The principle of the approach "objects in the Petri nets" is to model a system by a single PNet, whose tokens are objects. This single network can be structured by using a hierarchical decomposition, typically in the form of super-transitions or super-places. The type of tokens are described in an external formalism to the Petri nets, for instance an algebraic notation or a programming language.

The formalism POP/POT (Engelfriet et al. 1990) belongs to this type of approaches. POT (Parallel Object –based Transition) system is an other example: A POT is a simple PNet where objects are tokens with associated structures of memories; The state of the object is explicitly modelled by places. Another example is given by LOOPN (Lakos and ken 1991) which is a language for simulation and specification of distributed systems with timed coloured Petri nets. It includes object properties such as the sub-typing, the inheritance and the polymorphism which allow an adequate modularization of complex specifications.

The approach "Petri nets in objects" consists in using the Petri nets to describe the internal behaviour of the objects. This approach proposes to model the system by several independent PNets (objects) which can interact.
The network marking models the internal state of the object and the transitions model the execution of its methods. The fundamental interest of this type of approach is to allow the use of the concepts resulting from the object paradigm (classification, encapsulation) to describe the structure of the system, instead of using a purely hierarchical structuring.

The COOPN (Competitor Object Oriented Petri Net) (Buchs and Guelfi 1990) and PROTOB (Baldassari et al. 1991) belong to this type of formalisms. In particular, PROTOB is a C.A.S.E (Computer Aided Software Engineering) for the specification, simulation and prototyping of the concurrent systems. A PROTOB Object is defined by its attributes, actions and communication ports. The behavior is described by a PROT which is a high level PNet which integrates PNets and DFDs (DataFlow-Diagramms). In (Wang and Wu 1998) an other similar formalism is presented : the CTOPN (Colored Timed Object- Oriented Petri-Nets) are proposed for the modeling of the automated manufacturing systems. Objective-Linda (Holvoet and Kielmann 1998) is a formalism for the formal specification of active objects' behaviour, using high level Petri nets (HLPN). The EP-Nets (Guan and Lim 2002) associating objects and Petri nets are proposed for the modelling of the interactive multi-media orchestrations.

In (Barezzi and Pezzè 2001) the dynamic model of UML is enhanced by high level timed Petri nets to cover the language gaps. Another example is given by HOONets (Hierarchical Object- Oriented Petri Net). HOONEts deal with several oriented object aspects such as abstraction, encapsulation , modularity , the interaction by messages, the inheritance and polymorphism. (Hong and Bae 2000). However the work closest to the proposed approach is probably the CO-Nets (Aoumeur and Saake 2002); the CO-Nets constitute a multi-paradigm integrating algebraic Petri nets and the object-oriented paradigm, the model is semantically interpreted by a rewriting logic theory largely inspired from that of ECATNets.

3. THE REWRITING LOGIC.

The rewriting logic is nothing but a generalization of equational logic in order to adapt it to changes (Meseguer 1991). The rules are similar to those of equational logic but have a completely different significance. A rule T $\Rightarrow$ T' do not mean any more T equal T' but T becomes T'. The rule is a basic action allowing the transition of the system from one state to another. The rewriting logic describes the changes of the system so that the state is represented by an algebraic term, the transition becomes a rewriting rule and the distributed structure, an algebraic structure modulo a set of axioms E. Syntax in logic of rewriting is given by a signature (Σ,E) where Σ is a set of functions and E a set of axioms. A rewriting theory T=(Σ, E, L, R) in rewriting logic is composed of a signature (Σ,E) and by a set of labeled rules R with labels in L. These rules describe the behaviour of the system and the rewritings are performed on the classes of equivalences of the terms modulo the axioms E. In practice a theory of rewriting T=(Σ, E, L, R) can be used as an executable specification allowing a rapid prototyping of the modelled system and its checking.

One of the most powerful applications of this logic consists of the theory of concurrent objects; it is a theory enabling description of the system as a configuration of objects . Object systems from simplest to most complex can be modelled in this theory. This theory is at the origin of MAUDE language. The latter is a high level specification language for concurrent oriented objects systems where each elementary action is described by a rewriting logic rule. By integrating functional, object and concurrent programming, MAUDE enables specification of object systems in a declarative way with a high degree of abstraction and generality. It adopts OBJ3 (Goguen et al. 1988) as a functional sub-language for the specification of data types. The behaviour of the system is described by a set of rewriting rules. Each rule called event of communication can imply several objects and several messages. The object is the unit of concurrency of the system (granule of concurrency) and evolves according to an interleaving semantics. Another application is the ECATNets

ECATNets (Concurrent Algebraïc Term Nets) are a kind of high level PNets which associate rewriting logic to PNets. ECATNets integrate the NPN (Numerical Petri Nets) (Wilbur Ham 1987), algebraic data types (ADTs) and the rewriting logic. By these three formalisms, ECATNets offer a powerful tool for the specification, prototyping and validation of concurrent systems. NPNs and ADTs define structural and syntactic aspects of ECATNets whereas the rewriting logic defines its semantics. ECATNets were subjects to several applications and extensions (Bettaz and Maouche 1992),(Bettaz and Maouche 1994),(Bounoua and Bettaz 1992), (Bettaz and Maouche 1995), (Djemame et al. 1998) and (Boudiaf et al. 2006), the last work (Hicheur et al. 2006) proposed the RECATNets that enhance the ECATNets with the recursion and possibilities to specify complex workflow patterns.

4. THE REACTNETS

The REACTNets results from the integration of the ECATNets and MAUDE; in addition to the advantages of an object/PNets association as explained above, the two formalisms have the same semantics based on rewriting logic; on one hand this common semantic enable an homogeneous integration and attenuates the difficulties often encountered during the integration of ad hoc formalisms, on the other hand this association makes it possible to specify not determinist distributed systems with a true concurrency semantics at inter-object level (thanks to MAUDE rules) and intra-object level (thanks to ECATNets); finally the object paradigm adds the distribution and reactivity which are missing in traditional PNets to ECATNets.

4.1. The ECAObjects.

The representation of the object is inspired from that proposed in (Jungclaus et al. 1991a),(Jungclaus et al. 1991b) but differs by the consideration of intra–object concurrency. The object that we call ECAObject (Fig. 1) is described by its structural aspects and its behavioural aspects. The structure of an ECAObject consists of its static description in term of its name (unique identifier), its attributes, its communication ports and the events describing its behaviour.

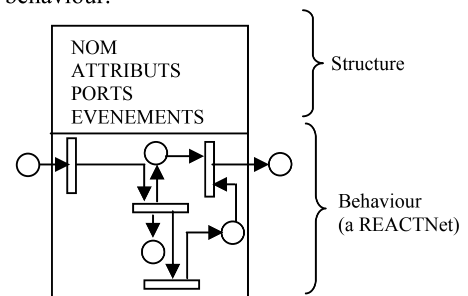

Fig. 1. Abstract Architecture of an ECAObject

The attributes model the ECAObject's static properties such as:

- parameters of ECAObject (name, first name, age…).
- states of ECAObject (busy, idle …).
- references to other ECAObject.

The ports are the ECAObject's access points used for the emission and reception of messages. The current state of an ECAObject is given by the set of its attributes. The event is the elementary activity of the ECAObject dependent on its state and modifying it. It is the granule of its concurrent behaviour. An ECAObject can carry out several events in parallel. The identification of the events depends on the level of abstraction agreed to describe this behaviour. The events can be either internal (local operations in the ECAObject) or visible (emission or interception of messages). The visible events constitute the interface of ECAObject and model the services needed or offered by him. The behaviour of the ECAObject consists of its dynamic evolution and can be described by the set of its acceptable life cycles. A life cycle represents a possible succession of events implying this ECAObject during its evolution and can comprise concurrency, mutual exclusion, and sequencing.

A place may be :

- An attribute of the ECAObject
- A port for an external interaction
- An intermediate place added for the needs of specifications

The behavior of an ECAObject is described by a REACTNet exhibiting not only its internal events but also its external events expressing its interaction with the environment through emission or reception of message in specific ports (Fig. 2). The places P-out (emission) and P-in (reception) in Fig. 2 are communication ports for the ECAObject's visible events. This case of figure could be brought back to a composition by transition (also called by rendez-vous) of the two PNets (Fig. 3). It is a particular case of the composition by a sequential process and it was proven that properties of aliveness and boundness are preserved in the composite network (Souissi and Memmi 1990). In addition we agree that REACTNets are considered with respect to the stability rules of classical reactive PNets theory as presented in (Eshuis and Dehnert 2003).

The communication ports allow to specify the simultaneous emission and interception of several different messages in parallel whereas The input/output places of classical Object/PNets approaches are generally managed in FIFO in accordance with the traditional vision of communication ports of concurrent objects.

The transition T models an internal event which is an action undertaken by the ECAObject. Let us note here that any change which can affect the state of an ECAObject (therefore its attributes) constitutes a stage of one of its possible life cycles and have to be expressed in the REACTNet

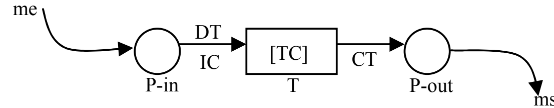

Fig. 2. A generic REACTNet.

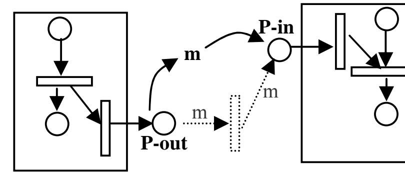

Fig. 3. Emission/reception of messages as a composition by "rendez-vous"

4.2. The REACTNets' semantics.

The state of the system called configuration is specified as a multi-set of ECAObjects and messages, provided with an operator ACI , with the identity element $\varnothing$.

The pair (P, M(P)) defines the current state of the place P. the set of these pairs (place, marking) has a structure of a multi-set with $\otimes$ union on this multi-set and $\varnothing_B$ the identity element. The state of the ECOBJect is the union ($\otimes$) of the states of all its places and is expressed with the term $< O: C/ P_1: m_1 ... P_n: m_n >$
Where,

O : name of the ECAOBject.
C : classe of the ECAOBject P_i: i^{th} place of the associated REACTNets.
m_i : marquing of the ith place of the associated REACTNets.

The tokens are algebraic terms. IC(input condition), DT(Destroyed Tokens), CT(Created Tokens) are multi-sets of terms (tokens), where $\oplus$, $\cap$, $\subset$, Θ stand for respectively union, intersection, inclusion and difference on the multi-sets and $\varnothing_M$ the element identity. [TC] is a Boolean algebraic expression eventually containing variables appearing in IC, DT and CT. to each place P are associated a sort S(P) and a capacity C(P) defined as a multi-set of closed terms (constants). The marking M(P) of a place is defined in respect of its capacity (which can be infinite).

The transition T materializes an internal event and is enabled if the following conditions are true:

- IC(P1,T) is enabled : IC indique the multi-set of tokens that have to be present in P1

- TC(T) is true.
- the addition of CT to the place P2 must not result in exceeding its capacity

When T is fired

- The multi-set $M(P1) \cap DT$) is removed from the input place P1.
- The multi-set CT is added to P2.

4.3. The rewriting theory of the system.

A REACTNets-based specification has a rewriting logic semantics combining the semantics of ECATNets and that of MAUDE and therefore it is a particular case of a conditional rewriting theory. The rewriting system obtained inherits the four groups of ECATNets' rules to which we add two other groups derived from MAUDE, the first one expresses the reactivity by the means of the interaction with environment and the second models the creation/destruction of objects.

4.3.1. equational logic rules.

These rules are derived from the algebraic equations describing the types of tokens (by ADTs). Usually, the ECATNets use OBJ3 [18] as a functional sub-language. The evaluation of the tokens can be done using a concurrent equational rewriting.

4.3.2. transitions rules.

The form of the rules derived from the transitions depends on the form of IC. The form of the rule is derived from the ECATNets rules as well as MAUDE events of communication in the sense that we explicitly express the object nature of the REACTNet. If we suppose that the generic REACTNet presented at the preceding paragraph is associated to an object O of class C which we represent in accordance with MAUDE notation by the expression < O:C >, we will have the following cases:

Case 1 : IC is of the form $[\ m\ ]_{\oplus}$

- IC = DT

We agree to express the rule as follows:

$T: < O:C/P1: IC > \;\Rightarrow < O:C/P2: CT >$

Where expressions P1: IC and P2: CT are in conformity with the ECATNet notation i.e. they respectively express the suppression of IC of P1 and the addition of CT to P2.

- $IC \cap DT = \varnothing_M$

The multi-set IC must be included in M(P) but does not have to be removed after firing, to express it the idea is to transform IC into itself:

$T: < O:C/P1: IC; P1:DT \cap M(P1) >$
$\Rightarrow < O:C/P1:IC; P2:CT >$

- $IC \cap DT \neq \varnothing M$

For this case, it was shown that it is possible to split the transition T in two transitions T1 and T2 of the simple type (two preceding cases) whose simultaneous firing is equivalent to that of T so we derive two rules.

$T1 : < O:C/P1: IC_1 > \;\Rightarrow < O:C/P2: CT_1 >$

$T2 : <O:C/P1, IC2> \otimes <O:C/P, DT2> \;\Rightarrow\; <O:C/P, IC2> \otimes\; <O :C/P2, CT2>$ With :
$IC= IC1 \cup IC2 , DT= DT1\cup DT2$
$IC1 = DT1, IC2 \cap DT2 = \varnothing_M$

Case 2 : IC is of the form $\sim [\ m\ ]_{\oplus}$

The form of the rule is given by:

$T:< O:C/P1: DT\cap M(p) > \;\Rightarrow < O:C/P2: CT >$ if $(IC\backslash (IC\cap M(p) = \varnothing_M) \Rightarrow$ false

Case 3 : IC= $\varnothing_M$

The form of the rule is given by:
$T: <O:C/P1, DT\cap M(p)) \Rightarrow (O:C/P2,CT)$
if $(M(p) =\varnothing_M) \Rightarrow$ true

When the place capacity C(p) is finite, the conditional part of the rewrite rule will include the following component: $(CT_{\neq} \oplus (M(p)_{\neq} \cap C(p)) \Rightarrow CT \oplus M(p_{\neq}$**(Cap**)

In the case where there is a transition condition TC, the conditional part of our rewrite rule must contain the following component: $TC\Rightarrow$ true

4.3.3. identity rules.

$\varnothing_M \oplus X \;\Rightarrow X$
$\varnothing_B \otimes Z \;\Rightarrow Z$

4.3.4. inferences rules.

The two following rules allow by splitting and recombination of the set of tokens, to carry out the rewriting rules with a maximum of concurrency at the level of the ECAOBJect itself, in fact this splitting/recombination of the state of the ECAOBject exhibits explicitly intra-object concurrency which is missing in MAUDE.

- splitting:

$< O:C/ P:X \oplus Y > \;\Rightarrow < O:C/ P:X >\otimes< O:C/ P:Y >$

- Recombination :

$< O:C/ P:X >\otimes< O:C/ P:Y >\Rightarrow< O:C/ P:X \oplus Y >$

4.3.5. Visible events rules.

They are asynchronous events related to the ports of the ECAObject. The explicit separation between the communication interface and the other activities for the same object makes it possible to have an additional level of intra–object concurrency. The communications can be done in a completely independent manner of the internal activities.

- Intercepting a message

This rule can be expressed according to the adopted syntax as follows:

$m < O:C > \Rightarrow < O:C/(P\text{-in}, m) >$

- Emitting a message.

The agreed rule is as follows:

$< O:C/(P\text{-out}, m) > \Rightarrow m < O:C >$

4.3.6. object creation/destruction rules

The object creation/destruction model considered is borrowed from that of MAUDE and inherits in particular, its declarative nature.

- Object creation.

The creation of an object requires a rule which makes it possible to specify explicitly that a message m_C is a creation message, while revealing the object created on the right of the rule in accordance with MAUDE syntax .
example:

$$m_C \Rightarrow < O:C/S >$$

This rule specifies that m_C is a message of creation; the effect is the generation of an object O of class C; S is the initial state of the associated REACTNet , i.e. the pairs set (place: marking) which starts the life cycle of the ECAObject created.
The identity of the ECAObject O and its initial state S can be the message parameters. Creation can be made, as presented in (Meseguer 1992) in two stages, initially the sending of a message to a particular object (Meta–object) associated to the class then the emission by this last of the effective message of creation. The objective is to manage the unicity of the identity and the validity of the creation.

- Object destruction.

The destruction can be specified by the interaction of a destroying message and the object to be destroyed, which will have to disappear from the right of the rule.

Example: $m_D < O:C > \Rightarrow \varnothing$

Just as for creation, the destruction of an object can be processed by a particular object (a priori the same charged by creation) in order to check that the object to be destroyed really exists and to eliminate it in the affirmative from the list of the objects of the current configuration, by transmitting the destructive message.

5. Case study (the router system)

The usage of multiple switches to connect test points or devices to instruments for the purpose of testing, measuring or monitoring industrial systems is very common. The router system seems to be a good example for our approach. This choice is also motivated by the high degree of parallelism implied in such systems.

5.1. Abstract specification.

The system is composed of several senders and several receivers communicating via the router. A sender emits from a queue of packets. Each emitted packet must be acknowledged. The sender does not send a new packet to a given receiver if its predecessor is not acknowledged yet. The receiver receives the packets in a queue. For each received packet, an acknowledgement is sent to the sender. The router has at a given moment a set of packets and acknowledgements to treat. It can intercept many packets and acknowledgements in parallel and rout them in the same time to the receivers.

5.2. The router formal model.

The system is composed of three ECAObjects classes: Sender, Receiver and Router. The messages' exchange between these three ECAObjects can be done according to the protocol presented in Fig. 4 where S, RT and R are respectively, the ECAObjects of the Sender class, the Router class and the Receiver class: The sender S sends a Pck(S, D, R) message to the router RT who transmits it to the concerned receiver R in the form of the routed message (S, D, R). D (Data) is the contents of the message.
After the message reception, the receiver R returns an acknowledgement Ack(R, S) which is routed to S in the form (R,S). The distinction between packets and acknowledgements before and after routing is necessary since each message type is associated to a distinct visible event. Indeed, the Pck(S message, D, R) have to be intercepted by the router RT whereas the message (S, D, R) have to be intercepted by the receiver S.

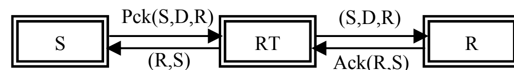

Fig 4. interaction protocol between the ECAObjects

The ECAObject Receiver.

- Attributes:

Recq : queue of received packets.

- Ports :

Ack_out, Pck_in

- internal events:

Treating–Pck : processing of a received packet (queuing in Recq and emission of an acknowledgement)

- visibles events :

Output messages: {Ack(R,S) }
Input messages : { (S,D,R) }

The ECAObject Router.

- Attributes:

Acknowledgement : a set of packages and acknowledgements to be treated at a given moment.

- Ports :

Ack_in, Pck_in , Ack_out, Pck_out

- Internal events:

Routing–Pck : Routing of a packet.

Routing–Ack : Routing of an acknowledgement.

- Visible events:

Output Messages: { (S,D,R), (S, R) }
Input Messages : {Ack(R,S), Pck(S,D,R)}

The ECAObject Sender.

- Attributes:

Sendq : queue of the packets to emit.
Receiver : identifier of the receiver from which an acknowledgement is expected.

- Ports :

Ack_in, Pck_out

- Internal events:

Emitting–Pck : this action consists in emitting a packet when the conditions are true (file not empty and no acknowledgement waited from the receiver)
Treating–Ack : processing of a received acknowledgement.

- Visible events:

Output Messages : { Pck(S,D,R) }.
Input Messages : { (R,S) }.

5.3. The router's REACTNets.

The type of Queue[elt data] is supposed to be predefined with the usual operations Remove, Empty, Add. we consider the functions Send, Rec and Data which give respectively for a packet or an acknowledgement the sender (S), the receiver (R) and the data (D):

5.3.1 The REACTNet "Receiver".

The packets (S, D, R) are received in the input port Pck In (Fig. 5). For each received message, an acknowledgement is emitted via the output port Ack–Out. The data D is added to the file Q in the Recq place. The parameter id used is supposed refering the identity of the object associated to the REACTNet.

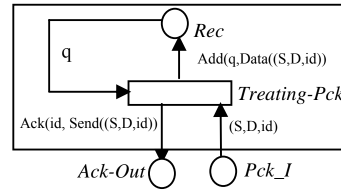

Fig. 5 The REACTNet Receiver

5.3.2. The REACTNet "Sender".

Sendq contains the file Q of the packets (Pck(S, D, R)) to emit. The packets are emitted via the output port Pck–Out (Fig. 6). For any emission a reference of the receiver R whose a acknowledgement is awaited is stored in the Receiver place. A packet Pck(S, D, R) is emitted only if no acknowledgement is awaited from receiver R. The expression ~Rec(Head (Q)) expresses that the identity of the receiver of the packet at the head of file should not be in the Receiver place and $\varnothing_m$ indicates that no token is destroyed. The acknowledgements are received in the input port Ack–In.

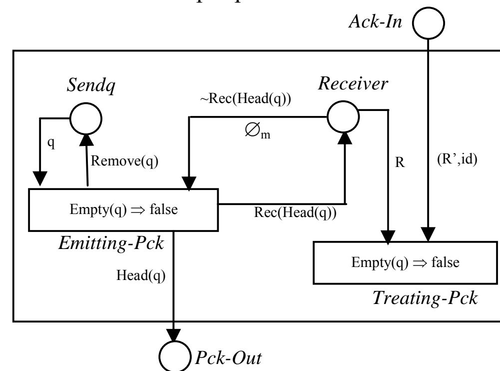

Fig. 6. The REACTNet Sender

5.3.3. The REACTNet "Router".

The Router ECAObject (Fig. 7) communicates with the environment through two input ports Pck–In and Ack–In, respectively for the packets and the acknowledgements and two output ports Pck–Out and Ack–Out. The various messages and received acknowledgements are collected in the place

"Messages". The transitions Receiving–Pck and Receiving–Ack are used to pass the received messages of the input ports to the place "Messages"

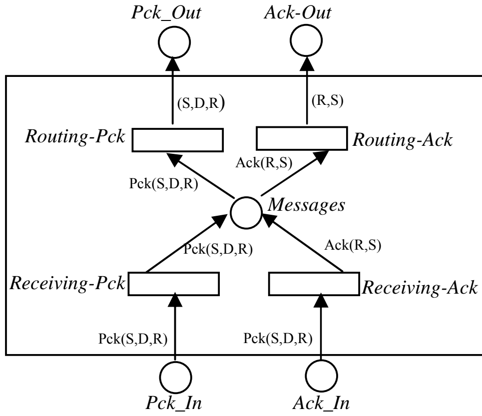

Fig. 7. The REACTNet Router

6. Specification of the system.

6.1 The Object module "ROUTER".

The module object ROUTER (Box given below) is the specification in the adopted notation of the example of the router introduced in precedent paragraphs. The following types are supposed to be predefined: Mset[elt] (multi-set of elements), Queue[elt] (file of elements) and Bits (sequence of bits). Syntax is borrowed of course from MAUDE but with different concurrency semantics, MAUDE has an interleaving semantics whereas our approach has a true concurrency semantics.

OMOD ROUTER
protecting configuration / specification of the configuration withsortes msg, objects and Oid(object identifier) and communication events
protecting Queue[elt
protecting Mset[elt]
protecting Bits
make Msg–queue **is** Queue[msg] **endmk**
make Msg–mset **is** Mset[msg] **endmk**
msg Pck(–,–,–) **:** Oid Bits Oid → **msg**
msg **(–,–,–) :** Oid Bits Oid → **msg**
msg Ack(–,–) **:** Oid Oid → **msg**
msg (–,–) **:** Oid Oid → **msg**
var q**:**msg–Queue
var S,R,R',RT **:** Oid
var D **:** Bits
Class Sender **/ Atts:** Sendq:Msg–queue, Receiver:Oid; **Ports :**Ack–In,Pck–Out:Msg–mset
Emitting–Pck : <S:Sender/(Sendq,q) ⊗ Receiver,$\varnothing_m$)> ⇒ <Sender/(Sendq,Remove(q)) ⊗ (Pck–Out,Head(q)) ⊗ (Receiver, Rec(Head(q))> **if** ((Empty(q)⇒false) and (M(Receiver)Θ(M(Receiver) ∩ Rec(Head(q)))= $\varnothing_m$)⇒false)

Treating–Ack : <S:Sender/(Receiver,R) ⊗ (Ack–In, (R',S))> ⇒ $\varnothing_B$ **if** ((R=Rec ((R',S))) ⇒ true)

R1 : <S:Sender/(Pck–Out,Pck(S,D,R))> ⇒ <S:Sender> Pck(S,D,R) / Pck–Out rule
R2 : (R',S) <S:Sender> ⇒ <S:Sender/(Ack–In, (R',S))> / Ack–In rule
Class Receiver **/Atts:** Recq:Msg–queue**; Ports :**Ack–Out,Pck–In:Msg–mset.

Treating–Pck : <R:Receiver/(Pck–In,(S,D,R)) ⊗ (Recq,q) > ⇒
<R:Receiver/(Recq, Add(q,Data((S,D,R)))) ⊗ (Ack–Out,Ack(R,Send (S,D,R))>
R10 : <R:Receiver,(Ack–Out,Ack(R,S))**>** ⇒ <R:Receiver> Ack(R,S)/ règle associée à la place Ack–Out.
R20 : (S,D,R) <R:Receiver> ⇒ <R:Receiver/(Pck–In,(S,D,R))>/ règle associée à la place Pck–In.
Class Router **/ Atts :**Messages**; Ports:** Pck–Out,Ack–out,Ack–In,Pck–In : Msg–mset
Receiving–Pck : <RT:Router/(Pck–In,Pck(S,D,R)> ⇒ <RT:Router/(Messages,Pck(S,D,R))>
Receiving–Ack : < RT:Router /(Ack–in,Ack(R,S)> ⇒ <RT:Router/(Messages,Ack(R,S))>
Routing–Pck : <RT:Router/(Messages,Pck(S,D,R)> ⇒<RT:Router/(Pck–Out, (S,D,R))>
Routing–Ack : < RT:Router /(Messages,Ack(R,S)> ⇒ <RT:Router/(Ack–Out, (R,S))>
R100 : Pck(S,D,R)<RT:Router> ⇒ <RT:Router/(Pck–In,Pck(S,D,R))> /règle associée à la place Pck–In.
R200 : Ack(R,S)<RT:Router> ⇒ <RT:Router/(Ack–In, Ack(R,S))> / règle associée à la place Ack–In.
R300 : <RT:Router/(Pck–Out, (S,D,R))> ⇒ <RT:Router> (S,D,R) / règle associée à la place Pck–Out.
R400 : <RT:Router/(Ack–Out, (R,S))> ⇒<RT:Router > (R,S) /règle associée à la place Ack–Out.
ENDOMOD

6.2 A prototyping scenario

We show in what follows how the specification above can be used for a rapid prototyping of the system, we start from a given configuration and execute the prototype.

initial Configuration :

<RT:Router/(Pck–Out,$\varnothing_m$)⊗(Ack–Out,$\varnothing_m$) ⊗ (Messages,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$) ⊗(Ack–In,$\varnothing_m$)>
<S1:Sender/(Sendq, Pck(S1,D1,R1)**.**Pck(S1,D2,R2))⊗(Pck–Out,$\varnothing_m$) ⊗ (Ack–In,$\varnothing_m$) ⊗(Receiver,$\varnothing_m$)>
<S2:Sender/(Sendq, Pck(S2,D3,R2)) ⊗(Pck–Out,$\varnothing_m$) ⊗(Ack–In,$\varnothing_m$)⊗(Receiver,$\varnothing_m$)>
<R1:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$)> <R2:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$)>

Step (1)
Firable rules :
object S1 : Emitting–Pck
object S2 : Emitting–Pck

concurrency
inter–object between S1 and S2.

Configuration after rewriting
<RT:Router/(Pck–Out,$\varnothing_m$)⊗(Ack–Out,$\varnothing_m$) ⊗(Messages,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$) ⊗(Ack–In,$\varnothing_m$)>
<S1:Sender/(Sendq,Pck(S1,D2,R2)) ⊗ (Pck–Out, Pck(S1,D1,R1))⊗(Ack–In,$\varnothing_m$)⊗(Receiver,R1)>
<S2:Sender/(Sendq, $\varnothing_m$))⊗(Pck–Out, Pck(S2,D3,R2) ⊗(Ack–In,$\varnothing_m$)⊗(Receiver,R2)>
<R1:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$) ⊗ (Pck–In,$\varnothing_m$)>
<R2:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$)>

Step (2)
Firable rules:
object S1 : R1
Emitting–Pck
object S2 : R1

concurrency
inter–object between S1 and S2
intra-object for S1

Configuration after rewriting
<RT:Router/(Pck–Out,$\varnothing_m$)⊗(Ack–Out,$\varnothing_m$)⊗(Messages,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$) ⊗(Ack–In,$\varnothing_m$)>
<S1:Sender/(Sendq,$\varnothing_m$)⊗(Pck–Out, Pck(S1,D2,R2))⊗(Ack–In,$\varnothing_m$)⊗(Receiver,R1⊕R2)>
<S2:Sender/(Sendq,$\varnothing_m$))⊗(Pck–Out,$\varnothing_m$)⊗(Ack–In,$\varnothing_m$)⊗(Receiver,R2)>
<R1:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$)>
<R2:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$)>
Pck(S1,D1,R1) Pck(S2,D3,R2)

Step (3)
Firable rules:
object S1 : R1
object RT : R100 (twice)

concurrency
inter–object between S1 and RT
intra-object for RT

Configuration after rewriting
<RT:Router/(Pck–Out,$\varnothing_m$)⊗(Ack–Out,$\varnothing_m$)⊗(Messages,$\varnothing_m$)⊗
(Pck–In,Pck(S1,D1,R1)⊕Pck(S2,D3,R2))⊗
(Ack–In,$\varnothing_m$)>
<S1:Sender/(Sendq,$\varnothing_m$)⊗(Pck–Out,$\varnothing_m$)⊗(Ack–In,$\varnothing_m$)⊗(Receiver,R1⊕R2)>
<S2:Sender/(Sendq,$\varnothing_m$))⊗(Pck–Out,$\varnothing_m$)⊗(Ack–In,$\varnothing_m$)⊗(Receiver,R2)>
<R1:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$)>
<R2:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$)>
Pck(S1,D2,R2)

Step (4)
Firable rules:
object RT : R100
Receiving–Pck (twice)

concurrency
intra-object for RT

Configuration after rewriting
<RT:Router/(Pck–Out,$\varnothing_m$)⊗(Ack–Out,$\varnothing_m$)⊗(Messages,Pck(S1,D1,R1)⊕Pck(S2,D3,R2))
⊗(Pck–In,Pck(S1,D2,R2))⊗ (Ack–In,$\varnothing_m$)>
<S1:Sender/(Sendq,$\varnothing_m$)⊗(Pck–Out,$\varnothing_m$)⊗(Ack–In,$\varnothing_m$)⊗(Receiver,R1⊕R2)>
<S2:Sender/(Sendq,$\varnothing_m$))⊗(Pck–Out,$\varnothing_m$)⊗(Ack–In,$\varnothing_m$)⊗(Receiver,R2)>
<R1:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$)>
<R2:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$)>

Step (5) and final
Firable rules:
object RT : Receiving–Pck
Routing–Pck (twice)
concurrency
intra-object for RT

final Configuration
<RT:Router/(Pck–Out, (S1,D1,R1)⊕
(S2,D3,R2))⊗(Ack–Out,$\varnothing_m$)⊗
(Messages, Pck(S1,D2,R2)) ⊗(Pck–In$\varnothing_m$,)⊗ (Ack–In,$\varnothing_m$)>
<S1:Sender/(Sendq,$\varnothing_m$)⊗(Pck–Out,$\varnothing_m$)⊗(Ack–In,$\varnothing_m$)⊗(Receiver,R1⊕R2)>
<S2:Sender/(Sendq,$\varnothing_m$))⊗(Pck–Out,$\varnothing_m$)⊗(Ack–In,$\varnothing_m$)⊗(Receiver,R2)>
<R1:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$)>
<R2:Receiver/ (Recq,$\varnothing_m$)⊗ (Ack–Out,$\varnothing_m$)⊗(Pck–In,$\varnothing_m$)>

CONCLUSION

Industrial systems are inherently distributed. In this paper we have proposed the fundamentals of a new approach for the specification of object oriented distributed systems with true concurrency at both intra and inter-object levels.

We associates two formalisms the ECATNets and MAUDE and thus inherits the advantages of both formalisms: the high degree of concurrency and expressiveness of the ECATNets and the object-orientness of MAUDE. The main strength of our approach is probably its rewriting logic semantics; therfore The obtained prototype can be executed and analyzed under the MAUDE environment.

REFERENCES

Aoumeur N. and G. Saake 2002. A component-based Petri net model for specifying and validating cooperative information systems. *Data & Knowledge Engineering*, Volume 42, Issue 2, P. 143-187

Baldassari M.., G. Bruno, A. Castella 1991. PROTOB: an object oriented CASE tool for modelling and prototyping distributed systems. *Software practice and experience* vol 21

Baresi L. and M. Pezzè 2001. Improving UML with Petri nets. *Electronic Notes in Theoretical Computer Science,* Volume 44, Issue 4, PP 107-119

Bastide R.1995. Approaches in unifying Petri nets and the object-orientation approach. *In Proceedings of the "Application and Theory of Petri Nets 1995*" *workshop on "Object-Oriented Programming and Models of Concurrency"*, Torino, Italy

Bettaz M., M. Maouche, M. Soualmi and M. Boukebeche 1993. Protocol specification using ECATNets. *Networking and distributed computing, Hermes*, Paris, Vol 3-1

Bettaz M., M. Maouche 1992. How to specify non determinism and true concurrency with algebraic term Nets. *LNCS 655 springer Verlag*.

Bettaz M., M. Maouche 1994. On the specification of protocol objects. *10th ADT + Compass general meeting'94*, Genova, Italy

Bettaz M., M..Maouche 1995. Modeling of object based systems with hidden sorted ECATNets. *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, *MASCOTS apos*

Boudiaf N., K. Barkaoui A. Chaoui. Implémentation des règles de réduction Des ECATNets dans MAUDE. *6e Conférence Francophone de MOdélisation et SIMulation -MOSIM'06* - du 3 au 5 avril 2006 – Rabat – Maroc

Bounoua O., M.Bettaz 1992. A graphical editor-simulator for Algebraic term Nets. *Proceeding of the second maghrebian conference on SE and AI*, Tunis

Buchs D., N. Guelfi 1990. *COOPN: a concurent object oriented Petri Nets models*. rappot LRI N°616 1990

Clavel M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott 2003. The MAUDE 2.0 System. *In Proc. Rewriting Techniques and Applications, 2003, Springer-Verlag* LNCS 2706, 76-87.

Djemame K., D.C. Gilles, L.M. Mackenzie and M. Bettaz 1998. Performance comparison of high-level algebraic nets distributed simulation protocols. *Journal of Systems Architecture*, Volume 44, Issues 6-7, March 1998, PP 457-472

Engelfriet J., G. Leth, G. Rozenberg 1990. *Net based description of parallel object based systems or POTs and POP*s. Technical report, Noordwij Kerhool fool workshop.

Eshuis R. and J. Dehnert 2003. Reactive Petri nets for workflow modelling. *In ICATPN*, pp 296-315.

Goguen C., H. Kirshner, J. Meseguer, A Megrelis and T.Winkler 1998. An Introduction To OBJ3. *IN J-P Jouannaud and stepen Kaplan editors, proceeding conference on conditional term rewriting* Orsay France LNCS 308

Guan S-U and S-S Lim2002. Modeling with enhanced prioritized Petri nets: EP-nets. *Computer Communications*, Volume 25, Issue 8, 15, PP 812-824

Hicheur A., K. Barkaoui, and N. Boudiaf 2006. Modeling Workflows with Recursive ECATNets. *synasc, pp. 389-398, Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'06).*

Holvoet T. and T. Kielmann. Behaviour specification of parallel active objects. *Parallel Computing*, Volume 24, Issue 7, July 1998, PP. 1107-1135

Hong J.E. and D.H. Bae 2000. Software modeling and analysis using a hierarchical object-oriented Petri net. *Information Sciences*, Volume 130, Issues 1-4, PP 133-164

Jungclaus R., G. Saake, and T. Hartmann 1991a. langage features, for object oriented conceptual modeling. *In proceeding 10th international conference on the ER Approach, SAN MAKO*

Jungclaus, R. G. Saake, and C. Sernadas 1991b. Introduction to TROLL In *Saake, G. And Sernadas, A., editors information systems, correctness and reusability TU Banshweig*

Lakos C.A. and C.D. Keen 1991. Modelling Layered Protocols in LOOPN, *proceedings of the Fourth International Workshop on Petri Nets and Performance Models, Melbourne*, 1991

Meseguer J. 1991. *Conditional rewriting logic as a unified model of concurrenc*y. technnical report SRI CSL 91.

Meseguer J. 1992. *a logical theory of concurrent objects and its realization in the MAUDE language* . Technical report SRI CSL 92, SRI international

Souissi Y., G. Memmi 1990. composition of nets via a communication medium. *LNCS 483 advances in Petri Net*s

Wang L. C. and S. Y. Wu 1998. Modeling with colored timed object-oriented Petri nets for automated manufacturing systems. *Computers & Industrial Engineering,* Volume 34, Issue 2, April PP. 463-480

Wegner P.1990. concepts and Paradigm of object oriented programming. *OOPSLA 1990.*

Wilbur Ham M.C 1987. Numerical Petri nets, a guide version 2, *Telecom Australia, Research Laboratories*, 1987