ATC : Actors with Temporal Constraints

Boualem Laichi Yamina Sami

Université des Sciences et Technologie Houari Boumedienne
Faculté Génie Électrique
Département d'Informatique
Algeria

blaichi@yahoo.com boumerdes@hotmail.com

Abstract

We present in a first step of this work a new timed actor model : ATC (Actors with Temporal Constraints). We hope to capture most of the temporal constraints arising in real time systems. For this purpose, we have used previous works on timed process algebras. We have chosen to include the two operators of ATP (Algebra of Timed Processes) : the watchdog operator for expressing passive temporal constraints and the urgency operator for modeling the active temporal constraints. The syntax and the rigorous semantic of ATC are given. Moreover, we give in a second step of this work a method to construct the graph of configurations classes. This graph describes the evolution of an ATC program in a sound and complete manner. It will be used for the analysis of ATC programs.

Key words : actor model, temporal constraints, real time systems, syntax, semantic, configurations class.

1. Introduction

In real time systems, results given too late or too early are considered as false and can have catastrophic consequences. This situation has led the standard formalisms of specification and description of systems to endow themselves with mechanisms allowing them to express and to analyze systems whose behavior is conditioned by the progress of time. In this sense, there has been a proliferation of timed models which are in most cases extensions to the time factor of existing formalisms such as process algebras and Petri nets. Models of programming have been also affected by this extension in order to express systems submitted to temporal constraints.

As objects paradigm is wide spreading, many object based models extended with time appeared [6,9,16,18]. However, many of these models integrate in the same object as well the information related to the functional aspects as the temporal ones. This makes the process of modularity and reusability too hard.

In [13,14] and [11] a timed actor model has been suggested whose main objective is to separate the functional aspects from the temporal ones. Then the functional aspects can be expressed by an ordinary actor language and the temporal ones by special entities called : RTSynchronisers. Each RTSynchroniser contains constraints about messages delivery to the actors it supervises. Although RTSynchronisers display many advantages, their power of expression is too limited, since they take into account only the active constraints. The expression of passive constraints like exceptions is not possible in the different versions of RTSynchronisers [13,14,11]. Moreover, RTSynchronisers focus on modeling temporal aspects and no method of validation to check properties on RTSynchronisers programs is given.

We suggest in our work a new temporal extension of actors called ATC (Actors with Temporal Constraints). ATC obeys the principle of separation between the temporal and the functional aspects. Moreover, we try to capture the maximal number of temporal constraints. We are guided for this by previous work on process algebras. Specifically, we adopt the *watchdog* operator of ATP [12,20] to express passive constraints and the urgency operator [12] to express the active ones.

Contrarily to ATP programs where there is a static interconnection between processes and no dynamic creation of them, in ATC programs as in actor programs, dynamic creation of actors, reconfiguration between them and the possibility of change of actor's behavior are allowed. This will make ATC model particularly adapted for the modern real-time systems where the openness and the reconfiguration are needed.

The aim of the first step of this work is not just to suggest a new timed actor model but to define a model that in one hand allows to express modern concurrent systems and in the other hand takes into account most of the temporal constraints arising in real time systems. Note that this latter set of constraints is not rigorously defined in the literature and we should attempt to build it from previous works on the area of real time systems.

In a second step of this work, a method to construct the graph of configurations classes is given. The suggested method is strongly inspired by the work [4] for the construction of occurrence graphs for Interval Timed Coloured Petri Nets (ITCPN) [19]. The most important criteria of this method is the fact that the obtained graph describes the evolution of an ATC program in a sound and complete manner.

2. The Actor Model

Actors were first introduced by Hewitt [5] and redefined by Agha [1]. They constitute a good framework to represent distributed systems. They encapsulate a state and a set of methods that manipulate the state, in a way similar to any object model. However, in opposition to an ordinary object, an actor encapsulates a thread of control.

Each actor operates concurrently with other actors and may send messages to actors of which it knows the addresses. The actors addresses can be communicated, thereby allowing a dynamic reconfiguration of the topology of communication. The communication is asynchronous and point to point. An actor has the possibility to create other actors and to specify their behaviors. Finally, the actor has the possibility to change its behavior. This change has an effect only on the forthcoming messages. In the actor model, an event is defined as the invocation of an actor by a message.

The basic constructs of any actor language are :

- ***send (a,v)*** sends a message that contains *v* to an actor *a*.
- ***newactor()*** creates a new actor and returns its address.
- ***initbeh(a,b)*** initializes newly created actor *a* with behavior b
- ***ready(b)*** shows that an actor has completed the processing of the current message and it is ready to execute another one with the behavior *b*.

3. The ATC Model

In ATC, all the temporal constraints are imposed on events. This means that constraints are imposed on the way in which messages are taken into account. Events are instantaneous. Moreover, as in most of the previous works on the actor model, we assume the processing of a message to be atomic. On the other hand, we obey the so-called *"Time/Action tree"* principle of execution in two steps used in timed process algebras. This stipulates that we alternate between the instantaneous execution of instructions and the progress of time.

3.1 Temporal constraints

We hope to capture the majority of the temporal constraints appearing in real time systems. We have found that real time systems are usually submitted to passive and active constraints. The passive constraint means that an exception can be executed if one or several actions are not executed before a given time. The active one means that several actions must be executed in a given interval otherwise the system reaches a deadlock state and the time stops its progress. So we choose to include these two types of constraints in ATC.

Before describing the way in which ATC handles these two types of constraints, we should recall that messages in an actor model are produced and consumed dynamically. Therefore, we use message patterns similar to those used in [13,14] to identify messages.

Definition 1: *Message Pattern*
A message pattern is a tuple :

((sender,seqNo),receiver(cv))

where *(sender,seqNo)* is a message tag, *receiver* represents the target of the message and *cv* includes the invoked script followed eventually by other parameters.
A message *m* is indicated by a pattern *p* if the following 4 conditions hold :
1. m's sender is the actor specified in the pattern as *sender*;
2. m is the seqNo*th* message sent by the actor *sender*;
3. m's target is the actor specified in the pattern as *receiver*;
4. the script invoked by *m* is that specified in *cv*.
Note that :
a)(sender,seqNo) are related by the system to each message. This has been done in a transparent way for the user.
b)A pattern indicates at most one message.
c)When *sender* or *seqNo* are useless in a given application, they can be replaced by *. Then, we have a reduced pattern in this case and we can represent it only by *Receiver(cv)* instead of *((*,*),Receiver(cv))*.
If a pattern *p* indicates a message *m*, we note this : $m \models p$.

a. Passive constraints

In order to express exceptions in ATC, we use the *watchdog* operator of ATP [12,20]. The ATP algebra is built around this powerful operator.

Definition 2: *Passive Constraint*
A passive constraint is expressed as follows :
P_I *watchdog*$[d_1,d_2]P_J$ where P_I and P_J are sets of patterns.
Intuitively, if no message indicated by one of the patterns contained in P_I is taken into account in the interval $[d_1,d_2]$, then a message indicated by one of the patterns contained in P_J must be taken into account. d_1 and d_2 are real and represent the bounds of the interval with respect to the time of the instantiation of the constraint.

Particular case of the watchdog

Usually, we need to delay the execution of a given action for a given time. For example to say that a given time is necessary for processing an action. To model this, we use the *wait* operator which we found in many timed models. The *wait* operator can be derived from the *watchdog* operator as follows :

Definition 3: *The wait*

$wait(d)P_J \equiv watchdog[0,d]P_J$

b. Active constraints

It is usual to have to impose the execution of an action in a given interval.

Definition 4: active constraint.

An active constraint is expressed as follows:

$P_I \Downarrow^{[d1,d2]}$ where P_I is a set of patterns.

This means that a message indicated by one of the patterns contained in P_I must be taken into account in the interval $[d_1,d_2]$. If the constraint is not satisfied before d_2 units of time, the time will not have the possibility to progress. d_1 and d_2 are real and represent the bounds of the interval with respect to the time of the instantiation of the constraint.

3.2 Syntax of ATC program

An ATC program is defined by three parts :

1. Behaviors definitions,
2. Temporal constraints definitions,
3. Initialization.

- For the first part, i.e. the behaviors definitions, we follow [3]. So we maintain the same syntax for the primitives : ***send***, ***ready***, ***newactor*** and ***initbeh***. Moreover, we introduce two primitives ***actconst*** and ***pasconst*** for instantiating respectively active and passive constraints. These two primitives will be explained at the end of the following part.
- The second part is a set of definitions of constraints : each passive constraint is defined as follows :

$const_1 (actor_1, actor_2, \ldots, actor_n) : P_I$ **watchdog**$[d_1,d_2]\ P_J$

where $const_1$ is the name of the constraint, P_I and P_J are the sets of patterns which indicate messages targeted to the actors having behaviors contained in $\{actor_1, \ldots, actor_n\}$.

each active constraint is defined as follows :

$const_2 (actor_1, actor_2, \ldots, actor_k) : P_I \Downarrow^{[d1,d2]}$

where $const_2$ is the name of the constraint, P_I is the set of patterns which indicate messages targeted to the actors having behaviors contained in $\{actor_1, actor_2, \ldots, actor_k\}$.

Note that when the set of behaviors is pointless in a given constraint, it will be replaced by the variable actor. Then, actor(select) means any actor which may process a message corresponding to the script select.

We introduce two instructions: *pasconst* and *actconst* to instantiate respectively passive and active constraint. These instructions can be used inside the scripts of actors, in the behaviors definitions part i.e. they will be executed during the processing of messages. A constraint instantiation is carried out in this way :

pasconst $const_1(a_1, a_2, \ldots, a_n)$;

actconst $const_2(a_1, a_2, \ldots, a_k)$;

where $a_1, a_2, \ldots, a_n$ (resp $a_1, a_2, \ldots, a_k$) are the addresses of the actors which will receive messages constrained by $const_1$ (resp $const_2$).

- In the third part which is the initialization part, first actors are created and first messages are send to them.

3.3 Examples

In the following section, we will illustrate the syntax of ATC by giving two examples. We begin by the example of *Vending machine* where the processing of exceptions is underlined. In a second example, we give the control system of a *steam boiler,* where we use the urgency operator.

Example 1: Vending machine

A Vending machine that sells drinks is ready to accept a piece of money at any time [20]. An actor having the behavior *Vending-machine*, after receiving a message that invokes the script *money,* instanciates a passive constraint stipulating that the drink must be chosen before *d* units of time otherwise the money is given back to the user and the Vending machine returns to its initial state. In order to do this, the actor sends to itself the message *r_money*. However, the processing of this message is done if any message of choice is carried out before *d* units of time. In the other case, when a choice is done before *d* units of time, the distributor produces the ordered drink : coffee or tea. The preparing of a drink takes some time : *c* units for coffee and *t* units for tea. Note that in this example, *sender* and *seqNo* are useless.

```
actor Vending-machine( ){
    method money( ){
        pasconst constraint1(self);
        send (self,r_money);}
    method coffee( ){
        pasconst constraint2(self);
        send(self,p_coffee); ...//preparing coffee}
    method tea( ){
        pasconst constraint3(self);
        send (self,p_tea); ...//preparing tea      }
    method r_money( ){
   ...//gives back the money and returns to its initial state}
    method p_coffee( ){ ...//provides coffee}
    method p_tea( ){ ...//provides tea}}
constraints{
  constraint1 (actor) :
   (actor(coffee),actor(tea))watchdog[o.d] actor(r_money);
  constraint2 (actor) : wait(c) actor(p_coffee);
  constraint3 (actor) : wait(t) actor(p_tea);}
```

Suppose we need to change the temporal constraints of the Vending machine. We allow the users to cancel their choices before the drink is provided. In order to adapt the system to this new way of operating, we should just change the definition of the constraints *constraint*$_2$ and *constraint*$_3$ as follows :

```
constraint2 (actor) :
      actor(r_money) watchdog[o.c] actor(p_coffee);
constraint3 (actor) :
      actor(r_money)watchdog[o.t]actor(p_tea);
```

Then, the separation between temporal and functional behaviors makes the change of temporal constraints easier. We have done this without modifying the scripts.
Remark : in this example we have omitted the initialization part. We will use it in the following example.

Example 2: Steam boiler
In this example [11], we describe a part of a simple steam boiler control system consisting of a pressure sensor, a controller and a valve actuator. On request, the pressure sensor sends a message containing its pressure value back to its customer (the controller). Based on the pressure value, the controller computes an updated steam-valve position. The controller repeats this procedure periodically (every 20 time units) by sending itself a message. The controller must receive sensor data from the pressure sensor within 10 time units measured from the start of the period, and it must update the steam valve position no later than 5 time units after receiving sensor data. This working is described in the ATC model by the following program :

```
actor Controller (actor sensor,valve){
      method loop( ){
            actconst constraint1(self);
            send(self,loop);
            actconst constraint2(self);
            send(sensor,read(self)); }
      method reading (real pressure){
            newValvePos=computeValvePos(pressure);
            actconst constraint3(self);
            send(valve,move(newValvePos)); } }
actor pressureSensor( ){
            real valeur;
            method read(actor customer){
            send(customer,reading(value)); } }
actor steamValve( ){
      method move(real NewPosition){
            …//move the valve } }
constraints{
      constraint1 (actor) : actor(loop)⇓[20,20];
      constraint2 (actor) : actor(reading)⇓[0,10];
      constraint3 (actor) : actor(move)⇓[0,5]; }
c=newactor(); initbeh(c,pressureSensor); //initialisation part
a=newactor() ;  initbeh(a, steamValve) ;
x=newactor() ;  initbeh(x, Controller(c,a)) ;
send (x, periode) ;
```

4. Semantic of ATC

After giving the syntax of ATC, we will formalize our model in this section and give it a rigorous semantic. This one is an extension with the notion of time of the semantic given in [3].

4.1 The basic semantic of actors

In an actor system, there is a finite set of actors and a finite set of untreated messages. These two sets are sufficient to fully describe the state of a given actor system which we call *configuration.* At a given time, an actor a is in one of the following states :

- ***busy :*** the actor is processing a message. This state is denoted by $[e]_a$, where e is the expression being currently executed.
- ***unknown :*** after its creation and before its initialization by the creator. This state is denoted by $(?x)_a$ where x is the creator actor.
- ***idle :*** ready to process a message with behavior ν. This state is denoted by $(\nu)_a$.

Definition 5: Configuration
An actor system configuration is defined like this:

$$\langle \alpha \mid \mu \rangle^{1} \quad \text{where :}$$

α :is a function which associates to each actor its state;
μ :is a multi-set of unprocessed messages;

The operational semantic of an actor system can be defined by the following transition rules [3]:

<fun:a> :

$$e \xrightarrow{\lambda}_{Dom(\alpha)\cup\{a\}} e` \Rightarrow \langle \alpha, [e]_a \mid \mu \rangle \longrightarrow \langle \alpha, [e`]_a \mid \mu \rangle$$

<newactor:a,a`>:

$$\langle \alpha, [R^2[newactor()]]_a \mid \mu \rangle \longrightarrow \langle \alpha, [R[a`]]_a, (?_a)_{a`} \mid \mu \rangle$$

a` is newly created.

<init:a,a`> :

$$\langle \alpha, [R[initbeh(a`,\nu)]]_a, (?_a)_{a`} \mid \mu \rangle \longrightarrow \langle \alpha, [R[nil]]_a, (\nu)_{a`} \mid \mu \rangle$$

<ready:a> :

$$\langle \alpha, [R[ready(\nu)]]_a \mid \mu \rangle \longrightarrow \langle \alpha, (\nu)_a \mid \mu \rangle$$

<send:a,m> :

$$\langle \alpha, [R[send(\nu_0, \nu_1)]]_a \mid \mu \rangle \longrightarrow \langle \alpha, [R[nil]]_a \mid \mu, m \rangle$$

where $m = <\nu_0 \Leftarrow \nu_1>$

<rcv:a,cv> :

$$\langle \alpha, (v)_a \mid <a \Leftarrow cv>, \mu \rangle \longrightarrow \langle \alpha, [app(v,cv)]_a \mid \mu \rangle$$

The <fun:a> transition defines the change of state of an actor when it executes an internal computation. The internal computation can be expressed by any sequential language. Since the semantic is independent of this language, we use

[1] As in [14,11], we have omitted the external and receptionists actors, since they do not influence on the semantic. The complete semantic of actors has been given in [3].
[2] R is the reduction context [3].

$\xrightarrow{\lambda}$ to express this computation. An actor can create other actors according to the rule <newactor:a,a‘>. It must give to each newly created actor an initial behavior following the rule <init:a,a‘>. The rule <ready:a> shows that an actor has changed its behavior and is now ready to process a new message. The transmission of a message *m* to an actor *a* is expressed by the rule <send:a,m> where *m* is added to μ. The taking into account of a message is expressed by the rule <rcv:a,cv>.

4.2 Semantic of ATC

In this section, we present the operational semantic of ATC, that is strongly influenced by the work done in [14,11].

Definition 6: ATC configuration

$$\langle \alpha \mid \mu \mid \sigma \rangle \quad \text{where :}$$

α : is a function which associates to each actor its state;
μ : is a multi-set of unprocessed messages;
σ : is a set of instantiated constraints;
An instance has always the same nature as its associated constraint, i.e. passive or active. Then, we have two types of instances :

Definition 7: The passive constraint instance
It is a 4-tuple $<P_I,d_1,d_2,P_J>$ where P_I and P_J are the sets of patterns of messages, d_1 and d_2 are non-negative reals with $d_1 \leq d_2$.
This construction means : if no message indicated by one of the patterns of P_I is processed between $t+d_1$ and $t+d_2$, where t is the time of the instantiation of the constraint, then a message indicated by one of the patterns of P_J is allowed to be processed.

Definition 8: The active constraint instance
It is a 4-tuple $<P_I,d_1,d_2,\perp>$ where P_I is a set of patterns of messages, d_1 and d_2 are non negative reals with $d_1 \leq d_2$.
$\perp$ is introduced to show the impossibility of time to progress when no message indicated by one of the patterns of P_I is processed between $t+d_1$ and $t+d_2$, where t is the time when a constraint is instantiated.

All the temporal constraints used in ATC are imposed on the taking into account of messages which are interpreted in the semantic by the transition ***rcv***. Then, the transition rules ***fun, newactor, init, ready*** and ***send*** are not modified when the time is introduced. However, the transition rule ***rcv*** need to be modified. We recall that all the instructions are instantaneous. In order to model the progress of time, we use the transition rule **progress**.
Before giving the transition rules of ***pasconst, actconst, progress*** and the new semantic of ***rcv***, we give the following definitions :

Definition 9: Time progressing
Time progressing induces a change in the set of instances of constraints :

$$\sigma\text{-}e=\{<P_I,d_1-e,d_2-e,P_J> / <P_I,d_1,d_2,P_J>\in\sigma\}\cup$$
$$\{<P_I,d_1-e,d_2-e,\perp> / <P_I,d_1,d_2,\perp>\in\sigma\}$$

This means that when the time progresses, the bounds of the intervals associated to the constraint instances σ decrease.

Definition 10: Functions

- $\text{SatInst}(m)=\{<P_I,d_1,d_2,P_J>/<P_I,d_1,d_2,P_J>\in\sigma, \exists p_i\in P_I\ m \models p_i,\ d_1\leq 0 \text{ and } d_2>0\} \cup \{<P_K,d_1,d_2,\perp> / <P_K,d_1,d_2,\perp>\in\sigma, \exists p_k\in P_K,\ m \models p_k,\ d_1\leq 0 \text{ and } d_2>0\}$

Intuitively, SatInst(m) is a function which returns the set of constraint instances satisfied by the deliverance of the message m.

- $\text{SatMess}(m)=\{m_j/\ m_j\in\mu \text{ and } \exists p_j\in P_J,\ m_j \models p_j \text{ and } \exists p_i\in P_I,\ m \models p_i,\ <P_I,d_1,d_2,P_J>\in\sigma,\ d_1\leq 0 \text{ and } d_2>0\}$

This function returns exception messages related to the message m.

$$\text{Dlv}(m)=\begin{cases} \text{True if } (\exists<P_I,d_1,d_2,P_J>\in\sigma) \text{ such that} \\ \quad (\exists p_i/p_i\in P_I,\ m \models p_i, d_1\leq 0 \text{ and } d_2>0) \text{ or} \\ \quad (\exists p_j/\ p_j\in P_J,\ m \models p_j,\ d_2\leq 0)) \text{ or} \\ \quad (\nexists<P_I,d_1,d_2,P_J>\in\sigma) \text{ such that} \\ \quad\ (\exists p_i/p_i\in P_I, m \models p_i) \text{ or } (\exists p_j/p_j\in P_J, m \models p_j) \\ \text{False otherwise} \end{cases}$$

Dlv(m) is a boolean function that indicates whether a message *m* can be delivered at the current time. If a message is not constrained, it can be delivered immediately. Otherwise, a constrained message must be delivered inside its associated interval (P_J may be $\perp$).
We define in the following, the transition rules of *pasconst, actconst, progress* and *rcv*.
<pasconst: $P_I$watchdog$[d_1,d_2]P_J$>

$$\langle \alpha,[R[\text{pasconst}(\text{constraint}_1, cp)]]_a \mid \mu \mid \sigma\rangle \longrightarrow$$
$$\langle \alpha,[R[\text{nil}]]_a \mid \mu \mid \sigma\cup<P'_I,d'_1,d'_2,P'_J>\rangle$$

where constraint_1 has the form $P_I$watchdog$[d_1,d_2]P_J$ *and* *cp* contains the parameters used to instantiate : P_I watchdog$[d_1,d_2]P_J$. P'_I,d'_1,d'_2,P'_J are respectively the instantiations of P_I,d_1,d_2,P_J using cp.
<actconst: $P_I \Downarrow^{[d1,d2]}$>

$$\langle \alpha,[R[\text{constraint}_2, cp)]]_a \mid \mu \mid \sigma\rangle \longrightarrow$$
$$\langle \alpha,[R[\text{nil}]]_a \mid \mu \mid \sigma\cup<P'_I,d'_1,d'_2,\perp>\rangle$$

where constraint_2 has the form $P_I \Downarrow^{[d1,d2]}$, *cp* contains parameters used to instantiate:
$P_I \Downarrow^{[d1,d2]}$. P'_I,d'_1,d'_2 are respectively the instanciations of P_I,d_1,d_2 using cp.
<rcv:a,cv>

$$\langle \alpha,(v)_a \mid m,\mu \mid \sigma\rangle \longrightarrow$$
$$\langle \alpha,[\text{app}(v,cv)]_a \mid \mu\text{-SatMess}(m) \mid \sigma\text{-SatInst}(m)\rangle$$

if m=<tag:a⇐*cv*> and Dlv(m)=True
<progress: e>

$$\langle \alpha \mid \mu \mid \sigma\rangle \longrightarrow \langle \alpha \mid \mu \mid \sigma\text{-}e\rangle$$

if $d_2-e\geq 0$ for every $<P_I,d_1,d_2,\perp>\in\sigma$.

The two first rules concern the instantiation of passive or active constraints. These instantiations are carried out by giving as parameters the addresses of actors concerned by the constraint and eventually other parameters. The rule <rcv:a,cv> defines the conditions and the consequences of taking into account a message. A message can not be taken into account when the current instant is outside its associated interval. As a consequence of taking into account a message, the constraint instances related to this message must be omitted from σ. Moreover, if a message is under the control of a watchdog, the set of exception messages associated to it must be omitted from μ to prevent their processing. Finally, **progress** formalizes the progress of time. Time must not be allowed to progress if an active constraint is violated.

5. Graph of configurations classes

A model is defined by its power of expression as well as by its power of analysis. In order to increase the analysis power of ATC, we propose a method for the construction of the graph of the configurations classes, which describes the evolution of an ATC program. The proposed method is strongly inspired by the work [4] for the construction of occurrence graphs for Interval Timed Coloured Petri Nets (ITCPN) [19]. In fact, this method has been obtained directly from the translation given by the first author in [8] which consists in giving an algorithm of derivation of an ITCPN from an ATC program. This algorithm is a timed extension to the work done by the second author in [15,17,10] where a method for the translation of the basic actor model [1] to colored Petri nets [7] has been proposed. The correctness proof of the algorithm leads us to built the occurrence graph for each ITCPN obtained from an ATC program and to prove the isomorphism between this latter and the graph of the configurations classes that we propose. We stress the point that the proposed method is sound and complete i.e. that for any evolution of the ATC program, we have a corresponding evolution in the configurations classes graph and conversely. Once the graph of the configurations classes obtained, it can be used for the analysis of ATC programs.

Class of configurations :

We recall that in the ATC model, we pass from a configuration to another by executing an elementary instruction of the model or a set of instructions. A configuration represents a state of an ATC program at a given time. But because of the continuity of time, the set of all possible states is infinite. This is why we group all states (configurations) having similar characteristics (i.e. same states of actors, same set of messages and the same set of instantiated constraints) in one class of configurations.

Each configurations class, denoted by CC_n, is an ordered pair constituted by an ATC configuration and a time interval during which this configuration is possible : $CC_n=\{(\langle\alpha_n \mid \mu_n \mid \sigma_n\rangle,\tau_n)$ / $\tau_{n-1}\leq\tau_n\leq\tau_{n-1}+x_n$ where τ_{n-1} represents the time at which the configuration $\langle\alpha_n \mid \mu_n \mid \sigma_n\rangle$ is obtained and x_n is a period of time during which the actor system may stay in this configuration$\}$.

Graph of configurations classes :

Every node of the graph of the configurations classes represents a class of configurations denoted by CC. An arc in the graph joining two classes of configurations CC_{n-1} and CC_n represents the invocation followed by the processing of a message mr_n, i.e. CC_n is obtained from CC_{n-1} by the processing of a message mr_n in an atomic and instantaneous way. This is because, in our model, the execution of scripts invoked by messages is instantaneous and consequently the progress of time is obtained only in the nodes of the graph.

Before presenting formally the classes of configurations and the way they evolve, we give the following notations :
- mr_i : mr_j means that when the message mr_i is processed, a constraint is instantiated that is imposed on the processing of message mr_j (i.e. mr_i constrain mr_j).
- $bound\_inf_i(n)$ is the lower bound of the constraint interval instantiated during the processing of the ith message. This constraint is imposed on the execution of the nth message.
- $bound\_sup_i(n)$ is the upper bound of the constraint interval instantiated during the processing of the ith message. This constraint is imposed on the execution of the nth message.

Then for every pair of messages mr_i and mr_j we distinguish the following cases :

a) if $\neg(mr_i{:}mr_j)$ then $bound\_inf_i(j)=0$ and $bound\_sup_i(j)=N$. (N denotes a sufficiently large number).

b) if $(mr_i{:}mr_j)$ and d_1 and d_2 are the bounds of the interval of the associated constraint, we have :

- if mr_j is an urgency message or it is under the control of a watchdog operator then : $bound\_inf_i(j)=d_1$ and $bound\_sup_i(j)=d_2$.
- if mr_j is an exception message then : $bound\_inf_i(j)=d_2$ and $bound\_sup_i(j)=d_2$.

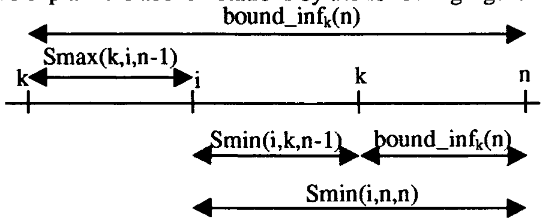
Notations :

Given a sequence of messages mr_1, mr_2, …, mr_n, the processing of each of them represents a step in the computation. Subscripts of messages are relative to their execution order. For a shorthand mr_i is denoted by i. The lower and upper bounds for delays between the processing *of any message mr_i and the mr_n message at the n^{th} step are* obtained recursively as follows :

- $Smin(i,n,n)= \max\{Smin(i,n-1,n-1),$
$\min\{\min_{k:n \wedge 1\leq k<i}\{bound\_inf_k(n)-Smax(k,i,n-1)\},$
$\min_{k:n \wedge i\leq k\leq n-1}\{bound\_inf_k(n)+Smin(i,k,n-1)\}\}\}$

- $\mathrm{Smax}(i,n,n) = \min_{m\in\mu}\{\min\{\mathrm{bound\_sup}_{n-1}(m)+\mathrm{Smax}(i,n-1,n-1), \min\{\min_{k:n \wedge 1\le k<i}\{\mathrm{bound\_sup}_k(m)-\mathrm{Smin}(k,i,n-1)\}, \min_{k:n \wedge i\le k<n-1}\{\mathrm{bound\_sup}_k(m)+\mathrm{Smax}(i,k,n-1)\}\}\}\}$

We explain the above notations by the following figure :



Then, by using Smin(i,n,n) and Smax(i,n,n) we can obtain the lower and upper bounds for delays between the processing of any couple of messages (mr_i,mr_j) at the nth step as follows :

For $i\in[0,j-1]$ and $j\in[1,n-1]$

- $\mathrm{Smin}(i,j,n)=\max(\mathrm{Smin}(i,j,n-1),\mathrm{Smin}(i,n,n)-\mathrm{Smax}(j,n,n))$
- $\mathrm{Smax}(i,j,n)=\min(\mathrm{Smax}(i,j,n-1),\mathrm{Smax}(i,n,n)-\mathrm{Smin}(j,n,n))$

The time of the invocation of a constrained message mr_n contained in a configurations class CC_{n-1}, obtained after the processing of a sequence of messages mr_1, mr_2, ..., mr_{n-1} depends upon the interval of its constraint and upon minimal and maximal delays between processing of previous messages mr_1, mr_2,..., mr_{n-1}. This information is sufficient to compute next classes. Following [4], for defining a configurations class we introduce a function TEC which returns the lower and upper bounds for delays between the processing of any couple of messages (mr_i,mr_j) in an ATC system. By applying the function TEC we can easily deduce the period of time in which the system may stay in a given configuration. Formally :

Definition 11: configurations class

A configurations class CC_{n-1} is a pair

$(\langle\alpha_{n-1} \mid \mu_{n-1} \mid \sigma_{n-1}\rangle, TEC_{n-1})$ where :

- $\langle\alpha_{n-1} \mid \mu_{n-1} \mid \sigma_{n-1}\rangle$ is an ATC configuration.
- TEC_{n-1} is a function $[0,n-1]\times[0,n-1]\rightarrow R$ such that :

  $TEC_{n-1}(i,j)= \mathrm{Smax}(i,j,n-1)$ if $i<j$

  $TEC_{n-1}(i,j)= -\mathrm{Smin}(j,i,n-1)$ if $i>j$

  $TEC_{n-1}(i,j)= 0$ if $i=j$

The processing of a message mr_n from a configurations class CC_{n-1}, gives a class CC_n defined as follows:

Definition 12: firing rules from a configurations class

In a configurations class CC_{n-1}, a message mr_n may be processed iff :

$\exists i$ such that $mr_i{:}mr_n$ then $\mathrm{bound\_inf}_i(n)\le TEC_n(i,n)$ (i.e. $\mathrm{bound\_inf}_i(n)\le \mathrm{Smax}(i,n,n)$)

or $\nexists i$ such that $mr_i{:}mr_n$

The processing of mr_n leads to the class $CC_n=(\langle\alpha_n \mid \mu_n \mid \sigma_n\rangle,TEC_n)$ where :

- $\langle\alpha_n \mid \mu_n \mid \sigma_n\rangle$ is obtained by the ATC rules.
- TEC_n is defined, from TEC_{n-1}, as follows :

$TEC_n(i,n)=\min_{m\in\mu}\{\min\{\min_{j:m}\{TEC_{n-1}(i,j)+\mathrm{bound\_sup}_j(m)\},, TEC_{n-1}(i,n-1)+\mathrm{bound\_sup}_{n-1}(m)\}\}$

$TEC_n(n,i)=\min\{\min_{j:n}\{TEC_{n-1}(j,i)-\mathrm{bound\_inf}_j(n)\}, TEC_{n-1}(n-1,i)\}$

$TEC_n(i,j)=\min\{TEC_{n-1}(i,j), TEC_n(i,n)+TEC_n(n,j)\}$

Example 3:

In this example, we construct the configurations classes graph of the ATC program given in example 1. We assume that the two messages : *money* and *coffee* exist in the initial configurations class and are sent from outside. Moreover, the message coffee must not be executed before money. The message money is not constrained, nevertheless we suppose that it will be executed at the latest N units of time after the beginning of the working of the Vending-machine. The special event 0 represents the origin of the time, i.e. the time of beginning of the working of the Vending-machine. A message will be represented by the target followed by the content of the message.

<u>CC_0 :</u> (initial configurations class)

α = (Vending-machine)$_a$

μ = {a(money), a(coffee)}

σ = { }

TEC_0 = NULL

<u>CC_1 :</u> is obtained from CC_0 by executing the message *money*.

α = (Vending-machine)$_a$

μ = {a(coffee), a(r_money)}

σ = {<(a(coffee), a(tea)),0,d,a(r_money)>}

TEC_1(0,money) = Smax(0,money,1)=N

TEC_1(money,0) = -Smin(0,money,1)=0

<u>CC_2 :</u> is obtained from CC_1 by executing the message *coffee*.

α = (Vending-machine)$_a$

μ = {a(p_coffee)}

σ = {< ,0,c,a(p_coffee)>}

TEC_2(0,coffee) = TEC_1(0,money)+bound_sup$_{money}$(coffee)=N+d

TEC_2(coffee,0)=TEC_1(money,0)-bound_inf$_{money}$(coffee)=0-0=0

(TEC_2(0,coffee) = Smax(0,coffee,2) et TEC_2(coffee,0) = -Smin(0,coffee,2))

TEC_2(0,coffee) and TEC_2(coffee,0) returns the lower and upper bounds for delays between the processing of the message coffee and the beginning of the working of the Vending-machine, i.e. the event 0.

TEC_2(money,coffee)=Smax(money,coffee,2)= TEC_1(money,money)+bound_sup$_{money}$(coffee)=0+d=d

TEC_2(coffee,money)=TEC_2(money,coffee)- bound_inf$_{money}$(coffee)=0

TEC_2(0,money)=min(TEC_1(0,money),TEC_2(0,coffee)+ TEC_2(coffee,money)=min(N,N+d)=N

TEC_2(money,0)=min(TEC_1(money,0),TEC_2(money,coffee) +TEC_2(coffee,money))=min(0,d)=0

Then we have the following graph of configurations classes which corresponds to the Vending machine program. For lack of space, the configurations classes CC_3 and $CC_{2'}$ are not detailed.

$$\begin{tikzcd}
CC_0 \arrow[r, "\text{monney}[0,N]"] & CC_1 \arrow[r, "\text{coffee}[0,d]"] \arrow[d, "\text{r\_monney}[d,d]"'] & CC_2 \arrow[d, "\text{p\_coffee}[c,c]"] \\
 & CC_{2'} & CC_3
\end{tikzcd}$$

6. Conclusion

In this work, a new timed actor model called ATC was given together with a method to describe its evolution in a sound and complete manner. This method constitutes a first step towards the analysis of an ATC program.

The model we suggest obeys the principle of separation between functional aspects and temporal ones. As for the expression power of the functional aspects, we can say that it is the same as in the basic actor model [3]. In fact, an ATC program without temporal constraints is equivalent to an ordinary actor program. The possibility of the actor model to support several models of concurrent computation has been already discussed in [1,2]. We recall that contrarily to most of the models of concurrency, the actor model allows reconfiguration between actors, extension of them and asynchronous communication.

For the expression power of the temporal aspects, we have tried to capture in this model the maximal number of temporal constraints arising in real time systems. After studying in depth several temporal models of concurrency, we have found that real time systems are usually submitted to both kinds of constraints : passive ones, like exceptions, and active ones like urgency. Thus, contrarily to RTsynchronisers which express only active constraints, we have chosen to include in the ATC model both kinds of constraints.

Another contribution of our work is a method of construction of the graph of configurations classes. In particular, we have associated to each class of configurations a function that outputs the minimal and maximal delay between the times invocation of any couple of messages previously processed. The soundness and completeness of the suggested method is an important result since it can be applied to other timed object-based models such as RTSynchronisers.

For a perspective, the graph of configurations classes can be used to check some properties on ATC programs such as reachability, boundedness, liveness, ... Moreover, this graph can be traduced to a timed graph on which a temporal logic can be applied [20].

7. References :

[1] G.Agha. "Actors : a model of concurrent computation in distributed systems". The MIT press, USA, 1986.
[2] G.Agha. "Supporting Multiparadigm Programming on Actor Architecture". PARLE'89, 1989.
[3] G.Agha, I.Mason, S.Smith and C.Talcott. "A foundation for Actor Computation". Journal of Functional Programming, 1996.
[4] G.Berthelot and H.Bouchneb "Occurrence graphs for interval timed coloured nets". LNCS 815, Springer-Verlag, 1994.
[5] C.Hewit. "Viewing control structures as patterns of passing message". An MIT Perspective, Brown & Winston eds, 1977.
[6] Y.Ishikawa, H.Tokuda and C.Mercer "Object-Oriented Real-Time Languages Design: Constructs for Timing Constraints". In Proceeding OOPSLA/ECOOP'90, 1990.
[7] K.Jensen "Coloured Petri Nets" Advances in Petri nets, Part1, LNCS 254, W.Brauer, W.Reisig and G. Rozenberg eds, 1986.
[8] B. Laichi. "ATC : Acteurs avec Contraintes Temporelles et leur Sémantique par les Réseaux de Petri Colorés Temporisés". Thèse de Magister, USTHB, Algeria, 2000.
[9] K.Lin and J.W.Liu "Flex : A language for Real-Time Systems Programming". Technical Report 1634, UIUC, 1990.
[10] S.Miriyala, G.Agha and Y.Sami. "Visualising Actor Programs Using Predicate Transition Nets". Journal of Visual Languages and Computing 3, 195-220, Academic Press, 1992.
[11]B.Nielsen, S.Ren and G.Agha "Specification of Real-Time Interaction Constraints". Proceedings of the First International Symposium on Object-Oriented Real-Time Computing, IEEE Computer Society, 1998.
[12] X.Nicollin. "ATP : une algèbre pour la spécification et l'analyse des systèmes temps réel". Thèse de Doctorat, INP de Grenoble, France, 1992.
[13] S.Ren, G.Agha and M.Saiton "A Modular Approach for Programming Distributed Real-Time Systems". Journal of Parallel and Distributed Computing, 36(1):4-12, 1996.
[14]S.Ren. "An Actor-Based Framework for Real-Time Coordination". PhD thesis, University of Illinois, USA, 1997.
[15]Y.Sami. "Sémantique et Validation des Langages d'Acteurs à l'aide des Réseaux de Petri Colorés". Thèse de Doctorat, Paris XI, France, 1993.
[16]I.Satoh and M.Tokoro "A Formalism for Real-Time Concurrent Object-Oriented Computing". Proceedings of ACM OOPSLA'92, 1992.
[17]Y.Sami and G.Vidal-Naquet "Formalisation of the behaviour of actors by coloured Petri Net and some applications". LNCS 506, PARLE'91, 1991.
[18]K.Takashio and M.Tokoro "DROL :An Object-Oriented Programming Languages for Distributed Real-Time Systems". In Proceedings OOPSLA'92, 1992.
[19]W.M.P van der Aalst "Interval Timed Coloured Petri Nets and their Analysis", 14th International conference on Application and theory of Petri nets, Chicago, LNCS 691, Springer-Verlag, 93.
[20]S.Yovine."Méthodes et Outils pour la vérification Symbolique des Systèmes Temporisés". Thèse de Doctorat, INP de Grenoble, France, 1993.