Proceedings of the 5th WSEAS Int. Conf. on SIMULATION, MODELING AND OPTIMIZATION, Corfu, Greece, August 17-19, 2005 (pp535-540)

Some Equivalences on ATC : Actors with Temporal Constraints

BOUALEM LAICHI
Department of Computer Science
USTHB
Algiers, Algeria

YAMINA SAMI
Department of Computer Science and Engineering
Universite du Quebec en Outaouais
Gatineau, Canada

Abstract : In this paper, we undertake to enhance a general framework for the modeling and validation of real-time concurrent systems which is known as ATC (Actors with Temporal Constraints). To prove that a given system is a valid implementation of its specification, one is often led to decide whether two systems behave in the same manner with respect to a given equivalence relation. In that connection, we propose equivalence relations between systems expressed within the ATC model. As a timed extension of the Actor model, ATC inherits all the functional capabilities of actors and further allows for the expression of most of the temporal constraints pertaining to real-time systems: exceptions, delays and emergencies.

Keywords : Validation, Real time concurrent systems, Actors , Temporal constraints, Temporal equivalences.

1 Introduction

There is a growing tendency for modern systems to be increasingly more open, modular, reconfigurable and concurrent. The Actor Model defined by Hewitt and refined by Agha encompasses all these features in a natural way. With a more and more emphasis being placed on the analysis of real time systems, the standard version of the Actor model soon became outdated, to some extent, which called for a timed extension of the system.

For this purpose, we have proposed in [7] and [8] a timed extension of the Actor Model, denoted by ATC (Actors with Temporal Constraints). As an extension of the Actor Model, ATC inherits all the functional power and the convenience of use provided by actors: dynamic creation of entities, change of behavior at run time, asynchronous communicatio.

As to temporal constraints, as they relate to process algebras and timed Petri nets, they fall into two broad categories : passive temporal constraints apt to trigger exceptions if one or more actions are not executed within the specified deadlines, and active temporal constraints that force the system to execute some actions within specified deadlines.

Accordingly, the basic actor model has been enhanced with temporal constructs to allow it to express these two types of constraints. Its syntax has been defined and its operation formalized by means of an operational semantic. Furthermore, a complete and consistent method has been devised to construct the graph of configuration classes which traces the evolution of an ATC program.

From our perspective, which is MILNER′s for that matter, the semantic of a concurrent system is determined by the combination of its operational semantic that models the program behavior and an equivalence relation between the terms of the model whose definition is a function of the criteria of abstraction that we wish to take into consideration.

In this paper, we undertake to complete the semantic given for ATC in [8] by defining the equivalence relations between the timed expressions and configurations of actors. More specifically, we combine the operational equivalence of PLOTKIN and the test equivalence of NICOLA and HENNESSY [11] to obtain a new timed equivalence, whereby two program expressions are considered to be equivalent if they behave in the same way, regardless of the observing context. Such equivalence will serve, among other purposes, to decide whether two actor systems or subsystems are equivalent. It can be used to formally establish that a given system is a safe or live implementation of an abstract specification.

This paper is organized as follows. In the next section, we review the main ideas of the actor model, the basic concepts of the ATC model and the syntax of a program written in this model. Section 3 is dedicated to describing the rigorous semantic of the ATC model. In section 4 we begin by giving the definition of the graph of configuration classes and some other necessary definitions, then we complete the semantic given in [8] for the ATC model by defining two types of equivalence relations involving actor expressions and actor configurations, respectively. We conclude in section 5.

2 The ATC Model

2.1 The Actor Model

Actors were introduced by Hewitt [4] and redefined by Agha [1]. They provide a good framework for the representation of distributed systems. They encapsulate both a state and a set of methods that manipulate the state, in a way which is similar to that of any object model. Unlike ordinary objects however, an actor encapsulates a thread of control as well. In the actor model, the communication is asynchronous and point-to-point.

The basic constructs of actor languages 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 is ready to execute another one with the behavior *b.*

2.2 The ATC Model

In ATC [8], all the temporal constraints are imposed on the invocation of actors by messages. This means that constraints are imposed on the way in which messages are taken into account. Moreover, as in most of the previous works on the actor model, we assume the processing of a message to be atomic. As well, we take for granted the so-called *"Time/Action tree"* principle stating that the progression of time alternates with the instantaneous execution of instructions.

2.2.1 Temporal constraints

We hope to capture most of the temporal constraints appearing in real time systems. Real time systems are usually subject to passive and active constraints. A passive constraint specifies that an exception can be handled if one or several actions are not executed before a given time. An active constraint states that several actions must be executed within a given time interval, otherwise the system reaches a deadlock state and the time stops its progression. 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 remind the reader that messages in an actor model are produced and consumed dynamically. Therefore, we use message patterns similar to those used in [14,15] 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 four conditions are hold :

1. The sender of m is the actor specified in the pattern as sender;
2. m is the message seqNoth sent by the actor sender;
3. The target of m is the actor specified in the pattern as receiver;
4. the script invoked by m is the one specified in cv.

If a pattern p indicates a message m, we denote this by $m \models p$.

a. Passive constraints

To express exceptions in ATC, we use the *watchdog* operator of ATP [13]. The ATP algebra is built on 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 some message indicated by one of the patterns contained in P_J must be taken into account. d_1 and d_2 are real numbers and stand for the bounds of the interval within which the constraint is to be instantiated.

Particular case of the watchdog

Oftentimes, one might want to delay the execution of a given action for some time to specify, for example, that a certain length of time is necessary for performing an action. To model this, we use the *wait* operator, as found in many timed models. The *wait* operator is derived from the *watchdog* operator as follows :

Definition 3: *The delay*

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

b. Active constraints

It is usual to be willing 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, then time will stand still with no possible progression. d_1 and d_2 are real numbers and represent the bounds of the interval with respect to the constraint instantiation time.

2.2.2 Syntax of ATC program

An ATC program is defined by three components, which are :

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

- For the first component, i.e. the behaviors definitions, we follow [2]. So, we keep the same syntax for the primitives : ***send***, ***ready***, ***newactor*** and ***initbeh***. Moreover, we introduce the two primitives : ***actconst*** and ***pasconst*** to instantiate active and passive constraints respectively.
- The second component consists of a set of constraints definition : 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 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 $\{actor_1, actor_2, \ldots, actor_k\}$.

We introduce two instructions: *pasconst* and *actconst* to instantiate passive and active constraint respectively. These instructions can be used inside the scripts of actors in the behaviors definitions component, 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 that will receive messages constrained by $const_1$ (resp $const_2$).

- In the third component, which is the initialization one, the first actors to be created are created and the first messages are sent to them.

2.2.3 Examples

In the following section, we illustrate the syntax of ATC with an example where the handling of exceptions is highligted.

Example : The Vending Machine

A coin-operated vending machine, which dispenses beverages, is ready for use at any time. Upon the receipt of a message invoking the script *money*, an actor with the *Vending-machine* behavior instanciates a passive constraint, which specifies that a beverage must be chosen within *d* units of time, otherwise, the money will be returned to the user, and the *Vending machine* to its initial state. In order to do this, the actor sends message *r_money* to itself. This message will not be processed unless some choice-specifying message has been entered within *d* units of time. In the other case, when a choice is made before *d* units of time have elapsed, the *Vending Machine* will produce the ordered beverage : coffee or tea. Preparing a beverage takes some time : *c* units for coffee and *t* units for tea. Note that in this example, we omitted the initialization component. *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);}
```

3 Semantic of ATC

After giving the syntax of ATC, we now proceed with defining its rigorous semantic. The latter is a timed extension to the one given in [2].

3.1 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 refer to as a *configuration*. At any point in time, an actor *a* may be in one of three 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 *v*. This state is denoted by $(v)_a$.

Definition 5: Configuration
An actor system configuration is defined as:

$$<\alpha|\mu>^P_X \qquad \text{where :}$$

α : is a function which associates to each actor its state;
μ : is a multi-set of unprocessed messages;
P : is a set of receptionists actors ;
X : is a set of external actors .
The operational semantic of an actor system can be defined by the following transition rules [2]:
<**newactor**:a,a`>:

$$\langle\alpha,[R^1[\text{newactor}()]]_a \,|\, \mu\rangle^P_X \longrightarrow \langle\alpha,[R[a`]]_a,(?_a)_{a`} \,|\, \mu\rangle^P_X$$

a` is newly created.
<**init**:a,a`> :

$$\langle\alpha,[R[\text{initbeh}(a`,v)]]_a,(?_a)_{a`} \,|\, \mu\rangle^P_X \longrightarrow\langle\alpha,[R[\text{nil}]]_a,(v)_{a`} \,|\, \mu\rangle^P_X$$

<**ready**:a> :

$$\langle\alpha,[R[\text{ready}(v)]]_a \,|\, \mu\rangle^P_X \longrightarrow \langle\alpha,(v)_a \,|\, \mu\rangle^P_X$$

<**send**:a,m> :

$$\langle\alpha,[R[\text{send}(v_0,v_1)]]_a \,|\, \mu\rangle^P_X \longrightarrow \langle\alpha,[R[\text{nil}]]_a \,|\, \mu,m\rangle^P_X$$

where $m=<v_0 \Leftarrow v_1>$
<**rcv**:a,cv> :

$$\langle\alpha,(v)_a \,|\, <a\Leftarrow cv>,\mu\rangle^P_X \longrightarrow\langle\alpha,[\text{app}(v,cv)]_a \,|\, \mu\rangle^P_X$$

An actor can create other actors according to the rule <newactor:a,a`>. It must assign 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>. The complete semantic of actors is given in [2].

3.2 Semantic of ATC

In this section, we present the operational semantic of ATC, which is strongly influenced by the work done in [12,14].
Definition 6: ATC configuration

$$<\alpha|\mu|\sigma>^P_X \qquad \text{where :}$$

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;
P : is a set of receptionists actors ;
X : is a set of external actors.
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 sets of message patterns, d_1 and d_2 are non-negative real with $d_1 \le d_2$. This construction is intended to mean : 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, 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 message patterns, d_1 and d_2 are non negative real numbers with $d_1 \le d_2$. $\perp$ 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 the 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 ***newactor, init, ready*** and ***send*** are not modified when the time is introduced. However, the transition rule ***rcv*** needs to be modified. Recall that all the instructions are instantaneous. In order to model the progression of time, we use the transition rule **progress**.
Before proceeding with the transition rules of ***pasconst, actconst, progress*** and the new semantic of ***rcv***, we present the following definitions :

Definition 9: Progression of time
The progression of time induces a change in the set of instances of constraints :

1 R is the reduction context [2].

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

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

Definition 10: Functions

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

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

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

This function returns exception messages that are related to the message m.

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

Dlv(m) is a boolean function which 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 within its associated interval (P_J may be $\perp$).

Now, we define the transition rules for *pasconst*, *actconst*, *progress* and *rcv*.

<**pasconst**: $P_I\mathrm{watchdog}[d_1,d_2]P_J$>

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

where $\mathrm{constraint}_1$ has the form $P_I\mathrm{watchdog}[d_1,d_2]P_J$ and *cp* contains the parameters used to instantiate : $P_I\mathrm{watchdog}[d_1,d_2]P_J$. $P`_I$, $d`_1$, $d`_2$, $P`_J$ are the instantiations of P_I, d_1, d_2, P_J using cp.

<**actconst**: $P_I \Downarrow^{[d1,d2]}$>

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

where $\mathrm{constraint}_2$ 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>

$$\langle\alpha,(v)_a \mid m,\mu \mid \sigma\rangle^{P}_{X} \longrightarrow \langle\alpha,[\mathrm{app}(v,cv)]_a \mid \mu\text{-SatMess}(m) \mid \sigma\text{-SatInst}(m)\rangle^{P}_{X} \quad \text{if } m=<tag:a\Leftarrow cv> \text{ and } \mathrm{Dlv}(m)=\text{True}$$

<**progress**: e>

$$\langle\alpha \mid \mu \mid \sigma\rangle^{P}_{X} \longrightarrow \langle\alpha \mid \mu \mid \sigma\text{-}e\rangle^{P}_{X} \quad \text{if} \quad d_2-e\geq 0 \quad \text{for every} <P_I,d_1,d_2,\perp>\in\sigma.$$

4 Temporal Equivalences for ATC

A model is defined both by its expression power and by its analysis capabilities. In order to increase the analysis capabilities of ATC, we have suggested in [8] a method for the construction of the graph of the configuration classes, which describes the evolution of an ATC program. The proposed method is strongly inspired by work [3] for the construction of occurrence graphs for Interval Timed Coloured Petri Nets (ITCPN). In fact, this method has been obtained directly from the translation given by the first author in [9,7] which consists in giving an algorithm for deriving of an ITCPN from an ATC program. This algorithm is a timed extension to the work done by the second author in [16,17,10] where a method for the translation of the basic actor model [1] to coloured Petri nets [6] has been proposed.

4.1 Class of configurations

Recall that in the ATC model, we move from one configuration to the next 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. Because of the continuity of time, however, the set of all possible states is infinite. This is why we group together all states (configurations) with similar characteristics (i.e. same states of actors, same set of messages and the same set of instantiated constraints) into one class of configurations.

Each configuration class, denoted by CC_n, is an ordered pair constituted by an ATC configuration and a time interval during which this configuration is possible :

$CC_n=\{(\langle\alpha_n \mid \mu_n \mid \sigma_n\rangle^{P}_{X},\tau_n)\ /\ \tau_{n-1}\leq\tau_n\leq\tau_{n-1}+x_n$ where τ_{n-1} represents the time at which the configuration $\langle\alpha_n \mid \mu_n \mid \sigma_n\rangle^{P}_{X}$ is obtained and x_n is a period of time during which the actor system may stay in this configuration$\}$.

4.2 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 linking 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 therefore, the progression of time is obtained only in the nodes of the graph. 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 time between the processing of the previous messages mr_1, mr_2, …, mr_{n-1}. This information is sufficient to compute next classes. Following [3], for defining a configurations class we introduce a function TEC which returns the lower and upper bounds of times 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

$$(\langle\alpha_{n-1} \mid \mu_{n-1} \mid \sigma_{n-1}\rangle^{P}_{X},TEC_{n-1}) \text{ where :}$$

- $\langle\alpha_{n-1} \mid \mu_{n-1} \mid \sigma_{n-1}\rangle^{P}_{X}$ is an ATC configuration.
- TEC_{n-1} is a function

$$[mr_0\ldots mr_{n-1}]\times[mr_0\ldots mr_{n-1}]\rightarrow R \text{ such that :}$$
$$TEC_{n-1}(mr_i,mr_j) = T\max(mr_i,mr_j) \quad \text{if } i<j$$
$$TEC_{n-1}(mr_i,mr_j) = T\min(mr_i,mr_j) \quad \text{if } i>j$$
$$TEC_{n-1}(mr_i,mr_j) = 0 \quad \text{if } i=j$$

Where $T\max(mr_i,mr_j)$ (respectively $T\min(mr_i,mr_j)$) is the maximum (respectively minimum) length of time that elapses between the processing of message mr_i and that of mr_j. mr_0 represents the starting time of the system. Thus, we can calculate the interval of completion for any message relative to the starting time along with the interval of completion for any instruction executed as a result of processing the message. A detailed study of the graph of configurations classes is presented in [8].

Example 2:
The initial configuration class, CC_0, of the ATC program given in example 1, is defined by :
α = (Vending-machine)$_a$, μ = {a(money), a(coffee)}, σ = {}
and the the class, CC_1, obtained from CC_0 by executing the message *money,* is defined by :
α = (Vending-machine)$_a$ μ = {a(coffee), a(r_money)}
σ = {<(a(coffee), a(tea)),0,d,a(r_money)>}

4.3 Timed Equivalences for ATC

In this section we complete the semantic given in [8] for our ATC model by defining two types of equivalence relations involving actor expressions and actor configurations, respectively. Our equivalence relations are timed extensions to those defined in [2] which are mainly based on the test equivalence defined in [11] and adopted later in TPL algebra. Intuitively, two program expressions are considered to be equivalent if they behave in the same way, when placed in any observing context. To define our relations, we make use of the same techniques for constructing the graph of configuration classes as described in the foregoing to determine the completion interval of the actions. The extension of ordinary actor equivalences to temporal ones could not be achieved otherwise, that is, without resorting to methods for constructing the graph of configuration classes and to the computation of the pertaining intervals in particular.

Before presenting the timed equivalence relations, we first introduce some definitions for the sake of clarity. These are straighforward time adaptations of the definitions given in [2].

4.4 Timed Computation Sequences and Paths

Definition 12: timed computation tree
If k is a timed configuration, we define the timed computation tree for k, T(k), to be the set of all finite sequences of timed transitions of the form:
$[k_i \xrightarrow{l_i[t_1,t_2]} k_{i+1} / i<n]$ for some $n \in$ N with k=k_0. l_i denotes the label of a transition defining the model semantic. Such sequences are called computation sequences.
The preceding construction means that configuration k_{i+1} can be reached from k_i by executing l_i between times t_1 and t_2, where t_1 and t_2 are output by the function TEC associated with the configuration classes.

Definition 13: timed computation path
A timed computation path is a maximal linear set of computation sequences in the timed computation tree T(k).
We use T^∞(k) to denote the set of all paths from k.
Drawing upon a test equivalence defined for CCS algebras in [11] and used later in [2], we incorporate a special primitive observation : **event** into the ATC model and check whether it is executed or not in a specified computation path. Because of the non-deterministic feature of the model, three different observations are possible instead of the two in the deterministic case :
1: **event** occurs in all possible executions
2: **event** occurs in some executions and not in others
3: **event** never occurs.
event will be treated in the same way as the all other primitives of the model are treated. Therefore, we may associate a timed completion interval with it.

Definition 14: event
We get the extended ATC model by introducing the new primitive event. The semantic rule associated with this primitive is:

<e :a> $<\alpha,[R[event()]]_a|\mu|\sigma>^P_X \to <\alpha,[R[nil]]_a|\mu|\sigma>^P_X$

Notice now that the observation of completion of *event* in a particular path is considered as success *s* if the event occurs in the path , otherwise, it is a failure *f.*

Definition 15: observation
Let k be a configuration of the extended model and let $\pi=[k_i \xrightarrow{l_i[t_1,t_2]} k_{i+1} / i<n]$ be a path, i.e. $\pi \in T^\infty$(k).We define :

$$obs(\pi) = \begin{cases} s\ [t_1 , t_2] \text{ if } (\exists\ i<n, a\ )\ \ (l_i = \text{<e:a>}\ [t_1 , t_2]) \\ f \text{ otherwise} \end{cases}$$

The criterion for observing the paths splits the set of paths from a given configuration k into success paths and failure paths, i.e:
$T(k) = T_S(k) \cup T_F(k)$ where :
$T_S(k) = \{\pi_i / \pi_i \in T^\infty(k) \text{ and } obs(\pi_i) = s[t_1,t_2]\}$
$T_F(k) = \{\pi_j / \pi_j \in T^\infty(k) \text{ and } obs(\pi_j) = f\ \}$

$$Obs(k) = \begin{cases} (S,I) \text{ if } T_F(k)=\emptyset \text{ and } I=\{[t_1,t_2] / \pi \in T_S(k) \wedge obs(\pi)= s[t_1,t_2]\} \\ (SF,I) \text{ if } T_F(k)\neq\emptyset , T_S(k)\neq\emptyset \text{ and} I=\{[t_1,t_2]/ \pi \in T_S(k) \wedge obs(\pi) = s[t_1,t_2]\} \\ (F,I) \text{ if } T_S(k)=\emptyset \text{ and } I=\emptyset \end{cases}$$

Remark : For simplicity's sake, we suppose that **event** is completed once in a path π at the most. Assumptions to the contrary raise no problem, except that instead of having a time interval associated with S in the observation, we shall need a set of intervals to accommodate for the different executions of **event** in the path.

4.5 Timed equivalences of actor expressions

Two actor expressions are said to be observationally equivalent if they give rise to the same observations. As with the classic case [2], an SF observation may be identified with an S observation, i.e. it may be considered as good as an observation S which defines a new equivalence. Likewise, SF may be identified with an F observation, thus giving rise to another equivalence.

Intuitively an observation context is a timed configuration in which an expression e_i can be evaluated.

Definition 16: $\cong_{1,2,3}$
$e_0 \cong_1 e_1$ iff $Obs(O[e_0]) = Obs(O[e_1])$ for every observing context *O*.
$e_0 \cong_2 e_1$ iff $Obs(O[e_0])=(S,I) \Leftrightarrow Obs(O[e_1])=(S,I)$ for every observing context *O.*
$e_0 \cong_3 e_1$ iff $Obs(O[e_0])=(F,I) \Leftrightarrow Obs(O[e_1])=(F,I)$ for every observing context *O.*

5.6 Equivalences between timed configurations of actors

Now, we extend the notion of equivalence to timed cofigurations. Before this, we give the following definitions :

Definition 17: Composable

Two timed configurations $k_i = <\alpha_i | \mu_i | \sigma_i >_{Xi}^{Pi}$ for i<2, are composable if : $Dom(\alpha_0) \cap Dom(\alpha_1) = \varnothing$, $X_0 \cap Dom(\alpha_1) \subseteq P_1$ and $X_1 \cap Dom(\alpha_0) \subseteq P_0$.

Definition 18: Composition of configurations

The composition $k_0 || k_1$ of two composable configurations $k_i = <\alpha_i | \mu_i | \sigma_i >_{X_i}^{P_i}$ for i<2, is defined by:

$$k_0 \,||\, k_1 = <\alpha_0 \cup \alpha_1 | \mu_0 \cup \mu_1 | \sigma_0 \cup \sigma_1 >_{(X_0 \cup X_1) - (P_0 \cup P_1)}^{P_0 \cup P_1}$$

Now, we define the observing configuration of a given configuartion.

Definition 19: timed observing configuration

For a given ATC configuration $k = <\alpha | \mu | \sigma >_X^P$, the observing configuration is a configuration of the extended ATC model of the form $k' = <\alpha' | \mu' | \sigma' >_P^X$ such that k' is composable with k in the sense of definition 17.

Since only the observation of event is of interest to us and not so much the interactions of the configuration with its environment, we introduce a function *Hide* which hides the receiver actors for a configuration.

Definition 20: Hide(k)

$$Hide(<\alpha | \mu | \sigma >_X^P) = <\alpha | \mu | \sigma >_X^{\varnothing}$$

Finally, a timed observational equivalence between two configurations is defined by :

Definition 21: $k_0 \cong k_1$

Let $k_0 = <\alpha_0 | \mu_0 | \sigma_0 >_X^P$ and $k_1 = <\alpha_1 | \mu_1 | \sigma_1 >_X^P$ be two configurations.

$k_0 \cong k_1$ iff *Obs(Hide(k₀||k))=Obs(Hide(k₁||k))* for all observing configurations *k* of k_j, j<2.

Notice that *Hide(k||k′)* is a closed configuration for an observing configuration *k′* of *k*.

Expression equivalence, as defined previously, carries over to a configuration equivalence as follows : If we substitute an expression with an equivalent one in a given configuration, we get a new configuration observationally equivalent to the original configuration.

Theoreme1 :

If $e_0 \cong e_1$ then

$$k_0 = <\alpha, [C[e_0]]_a | \mu | \sigma >_X^P \cong <\alpha, [C[e_1]]_a | \mu | \sigma >_X^P = k_1$$

5 Conclusion

This paper deals with the modeling and validation of real-time concurrent systems. The ATC model has been of particular interest to us because it features such desirable properties as openness, and reconfigurability. In this paper, we have proposed enhancements to the ATC model in the form of equivalence relations involving actors and configurations, with a view to improve the analysis capabilities of ATC. This method originated in [11] and was later used in the context of actors in [2, 18].

We are now working on designing methods and algorithms that prove the equivalence of actor expressions as defined in this paper. In the near future, we contemplate broadening the scope of our model by considering new timed equivalence relations bearing on the interactions between configurations. We work also on defining a method of reduction of the graph of configurations classes that preserves CTL properties as in [5]

References :

[1] G.Agha. "Actors : a Model of Concurrent Computation in Distributed Systems". The MIT press, USA, 1986.

[2] G.Agha, I.Mason, S.Smith and C.Talcott. "A foundation for Actor Computation".Journal of Functional Programming 1996.

[3]G.Berthelot and H.Bouchneb. "Occurrence graphs for Interval Timed Coloured Nets". LNCS 815,Springer-Verlag,1994

[4]C.Hewit. "Viewing control structures as patterns of passing message". An MIT Perspective, Brown & Winston eds, 1977.

[5] Hadjidj and Boucheneb. "Much compact Time Petri Net state class spaces useful to restore CTL* properties". IEEE ACSD, 2005.

[6] K.Jensen. "Coloured Petri Nets". Advances in Petri nets, Part1, LNCS 254, W.Brauer, W.Reisig and G. Rozenberg eds, 1986.

[7] B.Laichi. "ATC : Acteurs avec Contraintes Temporelles et leur Semantique par les Reseaux de Petri Colores Temporises". These de Magister, USTHB, Algeria, 2000.

[8] B.Laichi and Y.Sami. "ATC : Actors with Temporal Constraints". Fourth International Symposium on Object-Oriented Real-Time Computing, IEEE Computer Society, Magdeburg, 2001.

[9] B.Laichi and Y. Sami. "Formalisation du modele ATC par les reseaux de Petri colores temporises". Fifth International Symposium on Programming and Systems, Algeria, 2001.

[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] R. de Nicola and Henessy, M.C.B. "Testing equivalences for processes". Theoretical Computer Science, 34, 1984.

[12] 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

[13] X.Nicollin. "ATP : une algebre pour la specification et l'analyse des systemes temps reels". These de Doctorat, INP de Grenoble, France, 1992.

[14] S.Ren. "An Actor-Based Framework for Real-Time Coordination". PhD thesis, University of Illinois, USA, 1997.

[15] 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.

[16] Y.Sami. "Semantique et Validation des Langages d'Acteurs a l'aide des Reseaux de Petri Colores". These de Doctorat, Paris XI, France, 1993.

[17] Y.Sami and G.Vidal-Naquet "Formalisation of the behaviours of actors by coloured Petri Net and some applications". LNCS 506, PARLE'91, 1991.

[18] P. Thati "A theory of testing for asynchronous concurrent systems". PhD Thesis, University of Illinois, USA 2003.