TEMPORAL ANALYSIS OF COMPLEX TIME-DEPENDENT SYSTEMS: AN APPROACH BASED ON TIME PETRI NETS, ACTORDEVS AND HLA

Franco Cicirelli[(a)], Angelo Furfaro[(b)], Libero Nigro[(c)], Francesco Pupo[(d)]

[(a) (b) (c) (d)] Laboratorio di Ingegneria del Software
(www.lis.deis.unical.it)
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria
87036 Rende (CS) – Italy

[(a)]f.cicirelli@deis.unical.it, [(b)]a.furfaro@deis.unical.it, [(c)] l.nigro@unical.it, [(d)]f.pupo@unical.it

ABSTRACT
The design of time-dependent systems is challenging because it must fulfil both functional and temporal requirements. A properly abstracted model of one such a system, with temporal aspects only, is often derived and analyzed in order to evaluate the temporal behaviour of the system. Temporal analysis can be based on simulation or (hopefully) on exhaustive state space exploration. The latter techniques, though, are difficult to practice for large system models. In the work described in this paper, Time Petri Nets (TPNs) are preferred to formalize a time-dependent system because they facilitate the expression of concurrency, distribution, synchronization, mutual exclusion etc. concerns. An approach is proposed where a TPN model is mapped on ActorDEVS, a minimal and efficient Java framework supporting parallel or interleaved DEVS model execution. Complex TPN models can be analyzed using distributed simulation of ActorDEVS over HLA. The approach is demonstrated by means of a real-time realistic example.

Keywords: DEVS modelling and simulation, time Petri nets, actors, HLA

1. INTRODUCTION

The design of systems with timing constraints (e.g. embedded real-time systems, communication protocols, flexible manufacturing systems etc.) is difficult because it must fulfil both functional and (most importantly) temporal requirements. A properly abstracted model of one such a system, where only temporal aspects are explicitly modelled, is often derived and analyzed in order to evaluate the temporal behaviour of the system.

Temporal analysis can be based on simulation or (hopefully) on exhaustive state space exploration using e.g. model checking techniques (Cicirelli et al., 2007c). The latter techniques, though, are difficult to practice in the case of large system models which can have a large or even unbounded state graph.

This paper proposes an approach to modelling and temporal analysis of embedded control systems (Furfaro and Nigro, 2007) which is based on Time Petri Nets (TPNs) (Merlin and Farber, 1976; Cicirelli *et al.*, 2007c) and simulation. The approach is novel and maps preliminarily a TPN model on to ActorDEVS (Cicirelli *et al.*, 2008), a lean and efficient agent-based framework in Java supporting Parallel DEVS (Zeigler *et al.*, 2000). A distinguishing feature of ActorDEVS with respect to standard DEVS tools like DEVSJAVA (Zeigler and Sarjoughian, 2003) concerns the possibility of customizing the simulation engine in order to cope with different execution semantics. As a significant example, an interleaved parallel simulation engine was developed which is able to manage at runtime conflicts existing among TPN transitions (Cicirelli *et al.*, 2007b). The realization is beyond the scope of standard DEVS because the built-in semantics of maximal parallelism assumed by conventional simulation infrastructure (Zeigler and Sarjoughian, 2003), implies that all components which can undergo a simultaneous state transition at a given time, must do so and then cannot take care of conflicting situations.

Application of the proposed approach proceeds as follows. First a TPN model is visually designed and modularized in the context of the TPN Designer toolbox (Carullo *et al.*, 2003; Cicirelli *et al.*, 2007c). Then the model is translated into PNML (Billington *et al.*, 2003). The PNML version, finally, is partitioned, deployed and executed over a certain number of computing nodes, using the runtime support for TPNs achieved with ActorDEVS and the services of Theatre/HLA (Cicirelli *et al.*, 2008) for distributed simulation.

The paper demonstrates the use of the approach by modelling and analysis of a realistic real-time system related to a traffic light controller which is capable of responding to the exceptional situation corresponding to the arrival of an ambulance which must be handled within required safety and timing constraints.

2. BASIC CONCEPTS OF TIME PETRI NETS

A TPN is assumed to be a tuple $TPN = (P, T, A, I_{nh}, W, M_0, I^S)$ where

- P and T are non empty and disjoint sets respectively of places and transitions of the underlying Petri net (Murata, 1989)

- A is a set of arcs: $A \subseteq P \times T \cup T \times P$
- I_{nh} is a set of inhibitor arcs: $I_{nh} \subseteq P \times T$
- W associates weights to arcs: $W : A \cup I_{nh} \to \mathbf{N}$, with $\mathbf{N}$ the set of natural numbers. Weights are assumed strictly positive for arcs in A, 0 for inhibitor arcs
- M_0 is the initial marking: $M_0 : P \to \mathbf{N}$ in the usual sense of Petri nets
- I^S is the static firing interval function: $I^S : T \to \mathbf{R} \times (\mathbf{R} \cup \{\infty\})$.

Place $p \in P$ is an *input place* for transition t if there is an arc (p,t) in A. Place p is an *inhibitor place* for t if $(p,t) \in I_{nh}$, i.e. there exists an *inhibitor arc* connecting p to t. An inhibitor arc is graphically represented by a dot terminated line. A place p is an *output place* for t if there exists an arc (t,p) in A. The set of input and inhibitor places of t is said its *preset* and denoted by ${}^\bullet t$. The set of output places constitutes the transition *postset* which is denoted by $t^\bullet$.

I^S associates with each transition t a *dense* firing interval whose bounds are assumed to be specified by non negative reals: $I^S(t) = [a,b]$ with $0 \le a \le b$, b can be ∞. Bound a is said the (static) *earliest firing time* (EFT^s) of t, b the (static) *latest firing time* of t (LFT^s).

Let M be a marking. Transition t is said enabled in M, denoted by $M[t>$, iff

$$M[t> \Leftrightarrow \forall p \in {}^\bullet t \begin{cases} M(p) \ge W(p,t) & if\ (p,t) \notin I_{nh} \\ M(p) = 0 & if\ (p,t) \in I_{nh} \end{cases}$$

As soon as a transition t is enabled, it *starts* firing (server semantics). The firing end event is constrained to occur in the time interval associated with the transition. Let τ be an instant in time when transition t is enabled. Provided t is continuously enabled, *t* cannot fire before $\tau + a$ but *must* fire before or at $\tau + b$, unless it is disabled by the firing of another transition. At the time transition firing ends, tokens are removed from the input places and new tokens are generated in to output places as in classic Petri nets. Let M_{before} be the net marking just before t completes its firing. Firing end of t transforms M_{before} in M_{after}, denoted by $M_{before}[t > M_{after}$, by an instantaneous and atomic process in two phases:

(phase 1-token withdrawal)

$\forall p \in P$ ***if*** $p \in {}^\bullet t$ ***then*** $M'(p) = M_{before}(p) - W(p,t)$

else $M'(p) = M_{before}(p)$ ***endif***

(phase 2-token deposit)

$\forall p \in P$ ***if*** $p \in t^\bullet$ ***then*** $M_{after}(p) = M'(p) + W(t,p)$

else $M_{after}(p) = M'(p)$ ***endif***

where M' represents the intermediate marking generated after token withdrawal. It is worth noting that an enabled transition t', *i.e.* $M_{before}[t \wedge M_{before}[t'$, can be disabled (its firing stopped) by the firing of t, either in marking M' (because of a conflict due to the sharing of some input places with t, i.e. ${}^\bullet t \cap {}^\bullet t' \neq \varnothing$, or in the reached marking M_{after} (because of the existence of some inhibitor arc: $\exists p \in t^\bullet : (p,t') \in I_{nh}$). Similarly, a disabled transition t'', i.e. $M_{before}[t \wedge \neg M_{before}[t''$, due to the firing of t can become enabled in M' or in M_{after}. Single server firing semantics is assumed. After its own firing, would t be still enabled it is considered as any new enabled transition.

3. A TRAFFIC LIGHT CONTROLLER

In order to illustrate the approach, modelling and simulation of a Traffic Light Control system (TLC) (Raju and Shaw, 1994) are described in the following. In the proposed scenario, the traffic flow at an intersection between an avenue and a street is regulated by two traffic lights. The lights are operated by a control device (controller) that, in normal conditions, alternates in a periodic way the traffic flow in the two directions. In addition, the controller is able to detect the arrival of an ambulance and to handle this exceptional situation by allowing the ambulance crossing as soon as possible and in a safe way. For the sake of simplicity, it is assumed that at most one ambulance can be in the closeness of the intersection at a given time.

During normal operation conditions, the sequence green-yellow-red is alternated on the two directions with the light held green for 45 seconds, yellow for 5 seconds and red on both directions for 1 second. The intersection is equipped with sensors able to detect the presence of an ambulance at three different positions during its crossing. As soon as the ambulance arrival is detected, a signal named “approaching” is sent to the controller. When the ambulance reaches the nearness of the intersection the signal “before” is issued. After the ambulance completes the crossing the signal “after” is generated. The controller reacts to the “approaching” event by leading the intersection to a safe state, i.e. bringing both lights on red.

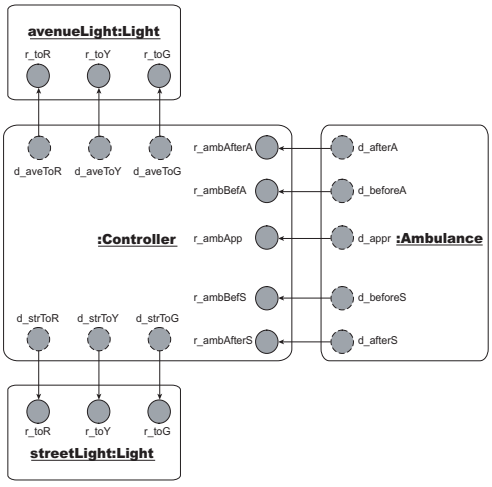

Figure 1: TLC system model

When the signal "before" is received, the controller switches to green the light on the ambulance's arrival direction. After the ambulance leaves the intersection ("after" event) the controller turns the green light to red and resumes its normal sequence.

Fig. 1 illustrates a model of the TLC system which is made of four connected components: there are two instances of the Light component, which respectively correspond to the light on the avenue and that on the street, one Ambulance component, which models the behaviour of the sensing equipments of the intersection and one Controller component, which implements the above described control logic.

Each component is specified by exploiting the *module* construct available in PNML (Billington *et al*., 2003). Modules support the creation of several independent instances of a given sub-net, usable in different contexts. For example, in Fig. 1, aveLight and streetLight are two instances of the module Light. Module interfaces are defined by means of *import* and *export* places. An export place (represented by gray-shaded disc with continuous border) is a place that is made visible outside the module; an import place (represented by gray-shaded disc with dashed border) is a reference to a place owned by another module instance. Connections among module instances are achieved, as in Fig. 1, by linking each import place to an export place.

The behaviour of a module is modelled by a TPN sub net. Fig. 2 details the TPN model of the module Light. The places red, yellow and green model the status of the traffic light. Places r_toR, r_toY, and r_ToG are the export places of the component interface.

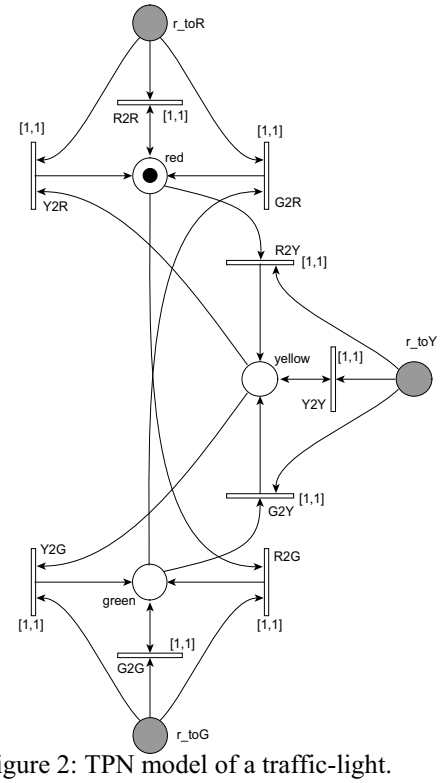

Figure 2: TPN model of a traffic-light.

An external component may ask to switch the light on green, yellow or red by respectively putting a token in the place r_toG, r_toY or r_toR. The initial marking of this sub net, where only place red contains a single token, models the fact that at start-up the red light is on and the others are off.

As can be seen from the time-windows of the transitions if Fig. 2, the handling of each request to change the status of a light, requires 1 time unit to be served.

Fig. 3 depicts the TPN model of the Ambulance module, which is used to simulate the sporadic arrival of ambulances needing to cross the intersection. This module has an interface made of five import places, which are used to notify the ambulance movements. It can be easily noticed that this net is symmetrical, with the upper part which models the crossing along the avenue and the lower part that along the street. The initial marking corresponds to a situation where the ambulance chooses its next crossing direction in a non deterministic way. This is modelled by the conflict existing between the transitions NextS and NextA.

The ambulance approaching on the avenue (street) is signalled by the firing of transition ApprA (ApprS) that puts a token into the import place d_appr. Timing specifications of these two transitions corresponds to the minimal and maximal interval between two successive ambulance arrival events. Transition BefA (BefS) puts a token into the import place d_beforeA signalling that the ambulance finds itself just before the intersection and that it is coming from the avenue (street). Timing constraints of this transition correspond to the time interval that may elapse between an approaching and a before event. Firing of transition AfterA (AfterS) corresponds to the completion of the ambulance crossing along the avenue (street) and results in a token put in the import place d_afterA (d_afterS).

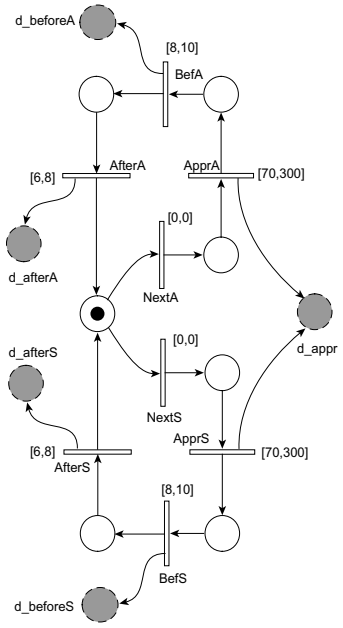

Figure 3: TPN model of the ambulance

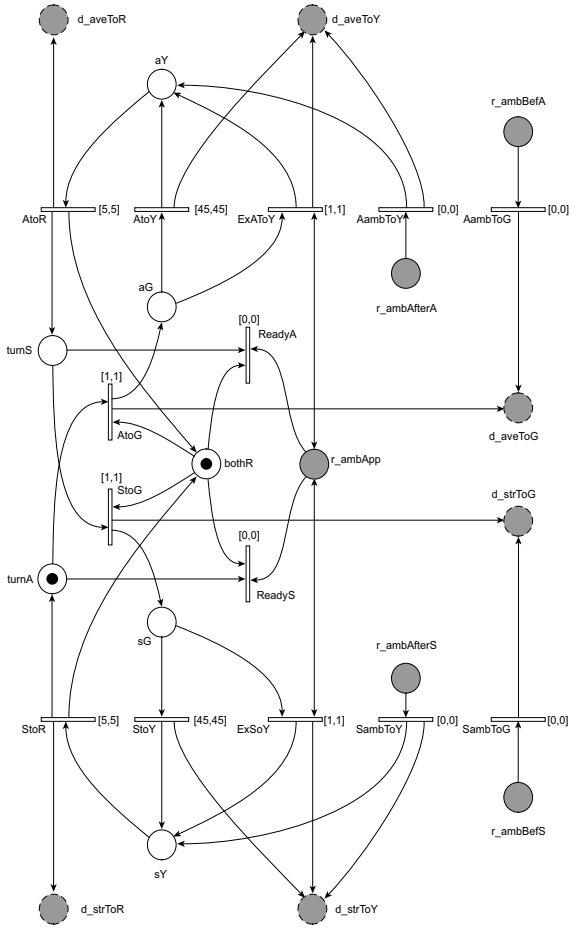

Figure 4: TPN model of the controller

The time interval associated to AfterA (AfterS) models the time needed to the ambulance to complete its crossing.

Fig. 4 is the TPN model for the controller which is the most complex component of the system. Also this net is symmetrical with the upper part interacting with the avenue light and the lower part with the street light. At start-up, the controller assumes that both lights are red and this is reflected by the presence of a token in the place bothR. The marking of places turnA and turnS determines whether the sequence green-yellow-red must starts respectively on the avenue or on the street. Under a normal operation mode, if there is a token into turnA, transition AtoG fires after 1 sec, puts one token into aG and another one into the export place d_aveG asking the light in the avenue to switch on green. After 45 sec, transition AtoY fires and puts one token into aY and another one into d_aveY, then after 5 sec transition AtoR fires and generates one token into places bothR, turnS, and d_aveR. Place turnS is now marked and, after 1 sec, the sequence starts again but on the street. This cycle continues until the approaching of an ambulance is notified by a token arrival into the export place r_ambAppr. The controller reaction to this event depends on the current status of the lights. It must bring both lights to red in a safe way in order to be ready to handle the ambulance crossing. The best case occurs when both light are already red and then one of the two transitions readyS or readyT can fire and disable the beginning of another sequence. In the case the light on the avenue (street) is green, the presence of one token into r_ambAppr and of one another into aG (sG) enables the immediate transition ExAtoY (ExStoY) whose firing has the same effect as the firing of aY (sY), i.e. asking the avenue (street) light to switch on yellow. After that, the sequence continues as in the normal case until both lights become red and the start of the next sequence is avoided as before. No special provision has to be taken when one of the two light is yellow. After handling the ambulance approaching, the controller maintains both lights red until the event of a token arrival into place r_ambBefA (r_ambBefS) notifies that the ambulance is just before crossing the intersection along the avenue (street). The controller reacts switching to green the light on the avenue (street) by firing transition AambToG (SambToG). The light is maintained green until the ambulance completes the crossing, event that is notified by a token into the import place r_ambAfterA (r_ambAfterB). This token enables the immediate transition AambToY (SambToY) whose firing has the same effect as that of AtoY (StoY), i.e. asking the light to switch on yellow. After 5 sec transition AtoR (StoR) fires and regenerates one token into place turnS (turnA). At this point, the controller restarts its normal operation mode.

4. TLC PROPERTY ANALYSIS

The behaviour of the TLC system can be validated by simulating its TPN model. System validation rests on checking that a set of assertions about its logical and temporal behaviour are satisfied at certain points during simulation, i.e. when events of interest occurs.

An example of safety property, i.e. one that must always be satisfied during system evolution, regards the consistent status of the traffic lights.

In order to avoid accidents among vehicles crossing the intersection, when on a direction the light is green or yellow, thus allowing the traffic on this direction, the light on the opposite direction must be red. This property can be checked by inspecting the marking of the two instances of the Light component each time one of their transitions fires. When such an event occurs, if one between avenueLight.green and avenueLigth.yellow is marked then streetLight.red must be marked and if one between streetLight.green and streenLigth.yellow is marked then avenueLight must be marked.

Another safety property concerns the status of the intersection at the time a “before” event is received. When such an event occurs no vehicle should be allowed to cross the intersection, i.e. the lights should be red on both directions. This property can be checked by inspecting the marking of places avenueLight.red and streetLight.red when one between transitions BefA and BefS fires in the Ambulance component.

Assuming that it takes at least 4 sec for the ambulance to reach the intersection from the time instant of the before signal, it follows that there exists a

deadline of 3 sec for turning green the light on the arriving direction. This accounts for the fact that a light takes 1 sec for changing its status. This property can be checked by recording the occurrence time of last firing of transition BefA (BefS) and measuring the elapsed time when the corresponding light is turned green, i.e. when one of the transitions having avenueLight.green (streetlight.green) in its postset fires.

The correct sequencing of the lights on each direction can also be easily checked by listening transition firing of Light components. A correct behaviour requires that only transitions RtoG, GtoY, and YtoR may fire: the firing of a transition out of this set denotes a wrong sequence.

5. CONCEPTS OF ACTORDEVS OVER THEATRE/HLA

A modular TPN model can be partitioned and deployed for execution over an instance of the Theatre architecture (Cicirelli *et al*., 2007a). ActorDEVS (Cicirelli *et al*., 2007b-2008) supports Parallel DEVS component development "in-the-small". Theatre furnishes the mechanisms required for distributed simulation, e.g. built on top of HLA middleware (DMSO, on-line; Kuhl *et al*., 2000) which provides time management and communication services to the theatres (federates) which compose the whole system (federation). For each ordered pair of communicating theatres a corresponding *interaction class* (Cicirelli *et al*., 2007a) is introduced and used as a communication channel.

ActorDEVS is agent-based. Actors with asynchronous message-passing are the basic building blocks, supporting DEVS atomic/coupled models. Message processing is atomic and cannot be pre-empted. In the mapping of TPN onto ActorDEVS, transitions correspond to atomic components. Places are topological entities which, as soon as they change their internal markings, alert dependent transitions to check their enabling status. The sub net assigned to a given theatre is a flattened coupled component. Flattening allows the use of one simulator per theatre, thus the simulation infrastructure of standard DEVS which associates a simulator to each atomic or coupled model (a multi-threaded organization) is avoided. ActorDEVS design minimizes the number of exchanged messages during simulation and then favours high-performance execution.

Fig. 5 is a snapshot of a typical Theatre system over HLA. A fundamental component in a theatre is the ControlMachine which is responsible of local message scheduling and dispatching. Local actors are held within the Local Actor Table (LAT). Since ActorDEVS actors can migrate between theatres, a Network Class Loader (NCL) is in charge to retrieving "on-the-fly" the class of a foreign received object from a network repository and loading it in the local JVM.

Specific control machines for ActorDEVS, working with HLA under conservative distributed simulation, were developed. They include Parallel-SimulationEngine and InterleavedSimulationEngine.

ParallelSimulationEngine, detailed in (Cicirelli *et al*., 2008), follows standard Parallel DEVS execution semantics. InterleavedSimulationEngine, on the other hand, fires one component (e.g. one transition of a TPN sub model) at a time. This is a key for proper management of conflicts (Cicirelli *et al*., 2007b), when the firing of a transition can disable transitions which share some input places. Other possibilities for disabling are related to the use of inhibitor arcs (see also next section).

ParallelSimulationEngine uses a combination of virtual (simulation) and logical times in order to ensure causality relationships among simultaneous events, when concurrent and interacting components are allocated to different physical nodes, are ultimately fulfilled.

6. TPN MODEL PARTITIONING

The enabling of a transition depends on the marking of the places of its preset. In this work, network message exchanges are purposely avoided during the enabling process. For this reason a model partitioning is assumed where a transition and the places of its preset are deployed on the same node (theatre). This implies that two transitions whose presets share at least one place (structural conflict) must also find themselves on the same node. This fact restricts the number of ways a model can be partitioned. By tracking this type of dependency among the transitions of a TPN model it is possible to determine the maximal number of partitions that can be obtained. These basic partitions are defined as equivalence classes induced by an equivalence relation.

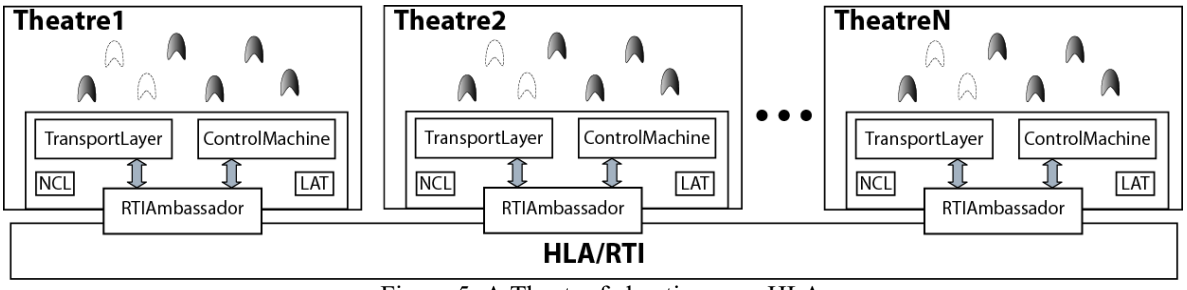

Figure 5: A Theatre federation over HLA

Let R be a binary relation among the transitions of a TPN defined as $R = \{(t_1, t_2) \in T \times T : {}^\bullet t_1 \cap {}^\bullet t_2 \neq \varnothing\}$, where $\varnothing$ denotes the empty set. R is reflexive and symmetrical. Let R^* be the transitive closure of R. R^* is an equivalence relation and as such it naturally induces a partitioning of T given by the quotient set T/R. The equivalence classes induced by R^* constitute the basic partitions of the model. Transitions belonging to the same equivalence class must be allocated on the same node. A model where R^* induces a unique equivalence class cannot obviously be partitioned.

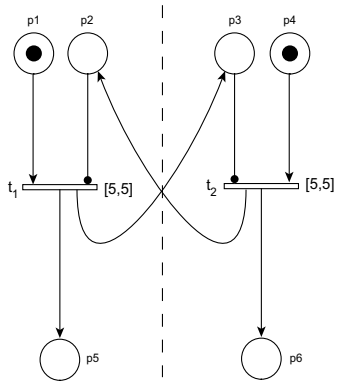

Figure 6: A conflict situation due to inhibitor arcs

The preceding concepts are sufficient to handle TPN models without inhibitor arcs. However, when inhibitor arcs are used, another type of dependency among transitions must be taken into account (see also the example in Fig. 6).

Transitions t_1 and t_2 of Fig. 6, would have been deployed on two different nodes because they are part of two distinct equivalence classes of R^*. However, between these transitions there exists a conflict situation due to the presence of inhibitor arcs. The firing of t_1 disables t_2 by putting a token into place p_3. For an analogous reason the firing of t_2 disables t_1.

To account for these type of conflicts, a coarser equivalence relation has to be used. Let RH be a binary relation defined as:

$$RH = \{(t_1, t_2) \in T \times T : ((t_1, t_2) \in R \vee \exists p \in t_1^\bullet : (p, t_2) \in I_{nh})\}$$

RH takes care of dependencies induced by inhibitor arcs and the equivalence classes induced by RH^*, its reflexive-symmetrical-transitive closure, can be used to establish basic partitions.

7. DEPLOYMENT AND SIMULATION

The TLC TPN model was partitioned according to the techniques described in the previous section and actually deployed as a federation of two theatres over HLA. One theatre hosts the traffic lights and the ambulance components, the other hosts the controller. Listing 1 shows a fragment of PNML describing the module for street lights.

```
<pnml>
 <module name="Light">
  <interface>
   <exportPlace ref="toR" id="r_toR"/>
   <exportPlace ref="toY" id="r_toY"/>
   <exportPlace ref="toG" id="r_toG"/>
  </interface>
  <!-- places -->
  <place id="toR"/>
  <place id="red">
   <initialMarking><text>1</text></initialMarking>
  </place>
  <place id="toY"/>
  <place id="yellow"/>
  …
  <!-- transitions -->
  <transition id="R2R">
   <firetime>
    <lBound>1</lBound>
    <uBound>1</uBound>
   </firetime>
  </transition>
  <transition id="R2G">
      <firetime>
       <lBound>1</lBound>
       <uBound>1</uBound>
      </firetime>
  </transition>
  ...
  <!-- arcs -->
  <arc source="toR" target="R2R">
   <weight><text>1</text></weight>
   <type direction="PT"/>
  </arc>
  <arc source="toR" target="R2G">
    <weight><text>1</text></weight>
    <type direction="PT"/>
  </arc>
  <arc source="toR" target="R2Y">
    <weight><text>1</text></weight>
    <type direction="PT"/>
  </arc>
  <arc source="red" target="R2R">
    <weight><text>1</text></weight>
    <type direction="PT"/>
  </arc>
  <arc source="R2R" target="red">
    <weight><text>1</text></weight>
    <type direction="TP"/>
  </arc>
  ...
 </module>
</pnml>
```
Listing 1: A portion of PNML module Light

Listing 2 illustrates the content of a PNML file which describes how the TLC model is built by instantiating and connecting the modules of the various components and how it should be partitioned and deployed in order to be simulated. Model partitioning is achieved by exploiting the PNML page construct. Each page contains a part of the model that constitutes a unit of deployment. One or more pages can be deployed for execution on a given physical node. In this example there are two pages respectively named left and right.
The first page contains two instances of the module Light and one instance of the module Ambulance. The second page contains only one single instance of the Controller. Each page also describes how the various instances are interconnected, i.e. the bindings among reference places and actual places.

```
<?xml version="1.0" encoding="utf-8"?>
<pnml>
 <net id="TLC_Prova" type="TPNNet">
  <toolspecific>
   <lp ip="HYDRA" id="0" port="8000"/>
   <lp ip="PERSEUS" id="1" port="8001"/>
   <singlemap lp="0" pageid="left"/>
   <singlemap lp="1" pageid="right"/>
  </toolspecific>

  <page id="left">
   <instance id="aveLight"
        ref="../../pnmls_nets/TLC/Light.xml#Light"/>
   <instance id="strLight"
        ref="../../pnmls_nets/TLC/Light.xml#Light"/>
   <instance id="ambulance"
        ref="../../pnmls_nets/TLC/Ambulance.xml#Ambulance">
    <importPlace parameter="d_appr" instance="controller"
        ref="r_appr"/>
    <importPlace parameter="d_beforeA" instance="controller"
        ref="r_beforeA"/>
    <importPlace parameter="d_beforeS" instance="controller"
        ref="r_beforeS"/>
    <importPlace parameter="d_afterA" instance="controller"
        ref="r_afterA"/>
    <importPlace parameter="d_afterS" instance="controller"
        ref="r_afterS"/>
   </instance>
  </page>

   <page id="right">
    <instance id="controller"
         ref="../../pnmls_nets/TLC/Controller.xml#Controller">
     <importPlace parameter="d_ALG" instance="aveLight"
         ref="r_toG"/>
     <importPlace parameter="d_ALY" instance="aveLight"
         ref="r_toY"/>
     <importPlace parameter="d_ALR" instance="aveLight"
         ref="r_toR"/>
     <importPlace parameter="d_SLG" instance="strLight"
         ref="r_toG"/>
     <importPlace parameter="d_SLY" instance="strLight"
         ref="r_toY"/>
     <importPlace parameter="d_SLR" instance="strLight"
         ref="r_toR"/>
    </instance>
   </page>
 </net>
</pnml>
```

Listing 2: PNML file for model deployment and partitioning

The mapping between pages and physical computing nodes is defined in the first part of Listing 2 delimited by the tag <toolspecific>. Here, a list of <lp> tags associates each node identifier with a pair (Internet address, port) of the relevant physical node and then a list of <singlemap> tags establishes the mapping between pages and nodes.

At simulation start-up a DEVS component, named TPNDEVSDeployer, is created on a computing node and it is feed with Listing 2 file and with files defining the single modules. TPNDEVSDeployer is in charge of parsing these files and of creating an in-memory representation of each page. After the parsing phase is completed, TPNDEVSDeployer creates as many instances of SubnetManager component, as there are computing nodes. Each SubnetManager receives the representation of the pages it has to handle and thereafter it is migrated on the relevant node. Finally, when a SubnetManager reaches its destination, it creates the DEVS components corresponding to the local transitions and instantiates the data structures corresponding to places. In the case the postset of a transition resides on a different node, for each place of this postset a corresponding ReferencePlace object is created. ReferencePlace objects are responsible of transparently notifying the effect of a transition firing to the SubnetManager where actual places of the postset reside.

A transducer (statistical) object is used in the first theatre to follow the firing of transitions of lights and ambulance, as well as to check marking of relevant places. The transducer has a fire() method that gets called on each transition firing, receiving the transition id and the current time. Simulation experiments were carried out using a simulation time limit of 10^6 for each run. Listing 3 shows a portion of the method fire() checking TLC properties. For brevity, only the most important properties are shown. The places object is an hash map for retrieving a place on the local theatre from its name.

```
public void fire(String id, long now) {
      Place aveR=places.get("aveLight.toR");
      Place aveY=places.get("aveLight.toY");
      Place aveG=places.get("aveLight.toG");
      Place strR=places.get("strLight.toR");
      Place strY=places.get("strLight.toY");
      Place strG=places.get("strLight.toG");
      if( (strY.getMarking()>0 || strG.getMarking()>0) &&
          aveR.getMarking()==0 )
              log.put("Traffic allowed on both directions!! @"+now);
      if( (aveY.getMarking() > 0 || aveG.getMarking() > 0) &&
          strR.getMarking() == 0 )
              log.put("Traffic allowed on both directions!! @"+now);
      if(id.equals("ambulance.BefA") || id.equals("ambulance.BefS"))
       {
              if(strR.getMarking() == 0 || aveR.getMarking() == 0)
                    log.put("Intersection not safe at before @"+now);
              beforeT=now;
              if( id.equals("ambulance.BefA") ) beforeA = true;
              else beforeS = true;
      }
      if( (id.equals("aveLight.R2G") ) && beforeA ) {
              beforeA = false;
              long v=now-beforeT;
              if( v>max ) max=v;
              if( max>3 ) log.put("Avenue light should be green!!
       Deadline missed @"+now);
      }
      if( (id.equals("strLight.R2G") ) && beforeS ) {
              beforeS=false;
              long v=now-beforeT;
              if(v>max) max=v;
              if( max>3 ) log.put("Street light should be green!!
       Deadline missed @"+now);
      }
      ...
}//fire
```

Listing 3: A fragment of method fire() for property checking

All the previously stated properties of the TLC model were found satisfied. Therefore, from the viewpoint of simulation, the system was found to be temporally correct.

8. CONCLUSIONS

This paper extends previous work carried out by the authors and concerning an achievement with ActorDEVS of the runtime support of Time Petri Nets (Cicirelli *et al*., 2007b), i.e. an application where it is required to dynamically handle conflicting components. The proposed approach aims at providing distributed simulation for property analysis of large TPN models of time-dependent systems.

The approach proceeds according to the following steps:

- first a TPN model is graphically designed in the context of the TPN Designer toolbox (Carullo *et al*., 2003; Cicirelli *et al*., 2007c)
- then a PNML version of the model is generated where distinct sub nets are associated with distinct PNML modules. A module interface publishes a set of import/export reference places whose connection is responsibility of the configuration process. Module boundaries can be conveniently exploited for partitioning so as to fulfil local semantics requirements of transitions
- a partitioned PNML model is then parsed, instantiated and deployed on a certain number of computing nodes of the Theatre/HLA architecture, in the presence of a conservative simulation conflict-aware control engine.

On-going work is geared at:

- improving the PNML generation process from TPN Designer. At the present time the generation occurs in two steps. First TPN Designer generates an XML version of the model (externalization). Then a stylesheet is used for converting the achieved XML into PNML. The goal is to extend TPN Designer in order to produce directly the PNML
- tuning the distributed simulation infrastructure based on HLA to Pitch pRTI 1516 (Pitch) product.

REFERENCES

Billington, J., Christensen, S., van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C. and Weber, M., 2003. The petri net markup language: concepts, technology, and tools. *Proceedings of the 24th Int. Conf. on Application and Theory of Petri Nets*, LNCS 2679, pp. 483–505. Springer.

Carullo, L., Furfaro, A., Nigro, L. and Pupo, F., 2003. Modelling and simulation of complex systems using TPN DESIGNER. *Simulation Modelling Practice and Theory*, 11 (7-8), 503-532.

Cicirelli, F., Furfaro, A., Giordano, A. and Nigro, L., 2007a. An Agent Infrastructure for distributed simulation over HLA and a case study using unmanned aerial vehicles. *Proceedings of 40th Annual Simulation Symposium (ANSS'07)*, pp. 231-238. March 26 – 28, Norfolk (VA, USA).

Cicirelli, F., Furfaro, A. and Nigro, L., 2007b. Conflict management in PDEVS: An experience in modelling and simulation of time Petri nets. *Proceedings of Summer Computer Simulation Conference (SCSC'07)*, pp. 349-356. July 15-18, S. Diego (CA, USA).

Cicirelli, F., Furfaro, A. and Nigro, L., 2007c. Using TPN/Designer and Uppaal for modular modelling and analysis of time-critical systems. *Int. J. of Simulation Systems, Science & Technology*, 8 (4), Special Issue on: Frameworks and Applications in Science and Engineering, 8-20.

Cicirelli, F., Furfaro, A. and Nigro, L., 2008. Actor-based Simulation of PDEVS Systems over HLA. *Proceedings of 41st Annual Simulation Symposium (ANSS'08)*, pp. 229-236. April 14–16, Ottawa (Canada).

DMSO, 2008. HLA-RTI, *Defense Modeling and Simulation Office*, http://www.dmso.mil/public/transition/hla. [accessed on April 2008]

Furfaro, A. and Nigro, L., 2007. Modelling and schedulability analysis of real-time sequence patterns using Time Petri Nets and Uppaal. *Proceedings of Int. Workshop on Real Time Software (RTS'07),* pp. 821-835. October 16, Wisla (Poland).

Kuhl, F., Dahmann, J. and Weatherly, R., 2000. *Creating computer simulation systems: An introduction to the High Level Architecture*. Prentice Hall.

Merlin, P. and Farber, D., 1976. Recoverability of communication protocols – implications of a theoretical study. *IEEE Transactions on Communications*, 24 (9), 1036–1043.

Murata, T., 1989. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77 (4), 541-580.

Pitch pRTI 1516, Pitch Kunskapsutveckling AB, http://www.pitch.se/prti1516/default.asp.

Raju, S.C.V., and Shaw, A.C., 1994. A prototyping environment for specifying and checking Communicating Real-time State Machines. *Software–Practice and Experience*, 24 (2),175–195.

Zeigler, B.P., Praehofer, H. and Kim, T.G., 2000. *Theory of modeling and simulation*. 2nd edition, New York: Academic Press.

Zeigler, B.P. and Sarjoughian, H.S., 2003. *Introduction to DEVS modelling and simulation with Java: developing component-based simulation models*. *http://www.acims.arizona.edu*. [accessed on April 2008]