Using Design/CPN for the Schedulability Analysis of Actor Systems with Timing Constraints

Libero Nigro and Francesco Pupo

Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria, I-87036 Rende (CS) - Italy
Voice: +39-984-494748 Fax: +39-984-494713 Email: {l.nigro, f.pupo}@unical.it

Abstract. This work is concerned with the use of Design/CPN for the modelling and analysis of actor-based distributed real-time systems. Coloured Petri Nets are used to obtain a formal and operational model of a specified system, as part of an iterative development process. Functional and timing properties of an achieved CPN model can be validated by means of simulation and occurrence graphs. The resultant approach facilitates transformations from analysis down through to the design and implementation in object-oriented languages like C++ and Java.

Keywords: Actors, Modularity, Real Time, Design/CPN, Temporal Analysis, Occurrence Graphs.

1 Introduction

Many distributed software systems can be represented, abstracting away from time aspects, as a set of asynchronous, autonomous components which interact one to another in order to co-ordinate local activity. The Actor model (Agha, 1986) is a well established computational framework suitable for building open and re-configurable general distributed applications.

In real-time systems time management is fundamental: application correctness depends not only on the functional results produced by computations, but also on the time at which such results are generated. Timing correctness is related to guaranteeing that a given logical behaviour is provided at the right time, not before nor after a due time.

In the last years, Agha et al. (Ren and Agha, 1995) (Ren et al., 1996) (Saito and Agha, 1995) have proposed extensions to the actor model in order for it to be applicable to real-time systems. The extensions rely on capturing message *interaction patterns* among actors through a *RTsynchronizer* construct (Ren and Agha, 1995). RTsynchronizers are declarative in character. They refer to groups of actors and specify timing constraints on the execution of relevant messages. Their concrete application depends on the possibility of ensuring a global time notion in a distributed system. Moreover a selected scheduling structure is required in order to guarantee that the timing constraints of messages are ultimately met. A major benefit of the use of RTsynchronizers is modularity. Actors are firstly defined according to functional issues only. Then timing aspects are separately specified through RTsynchronizers which affect scheduling.

The possibility of modelling actor systems through high level Petri nets has been previously investigated. In (Sami and Vidal-Naquet, 1991) an equivalence between actors and the formalism of CPN (Jensen, 1992) is provided with the goal of formalising actor semantics. The work of Agha et al. described in (Agha et al., 1992) is concerned with the use of Predicate Transition nets (Genrich, 1987) in order to support the visualisation of actor programs. Predicate Transition nets (PrT-nets) and CPN are formally equivalent and can be considered as two slightly different dialects of the same language. But none of these approaches has analysis aims. Moreover, only functional aspects are covered.

In (Nigro and Pupo, 1996) the temporal analysis of object-based real-time systems through TER nets (Ghezzi et al., 1991) was explored. However, the lack of mechanisms for modularity and compositionality, which are essential in the modelling of complex systems, was among the motivations leading to the choice of CPN (Jensen, 1994) (Jensen, 1997) in the work described in this paper. CPN temporal extension allows to exploit powerful analysis tools and techniques in the modelling and verification of timing aspects of actor systems.

This paper first describes a variant of the Actor model (Kirk et al., 1997) (Nigro and Pupo, 1997) where reflective actors are used to capture RTsynchronizers, then an embed of the chosen actor model into Design/CPN is proposed as a part of an iterative and incremental system development life (Verber et al., 1997). The proposed actor model centres on an integrated approach where actors, message passing, timing constraints and scheduling are all components of a time-predictable framework. In order to achieve the visualisation of both functional and timing issues of a modelled system, the approach depends upon the facilities and tools provided by Design/CPN, i.e., the high-level character of CPNs, the hierarchical and modular net constructions, the primitive time dimension and the verification of timing properties through the generation of occurrence graphs. As a benefit of the methodology, the validation process is accorded to the final development phases, in the sense that the same concepts and mechanisms move unchanged from analysis down to the design and implementation into an object oriented language like C++ or Java.

2 An actor-based framework for time-dependent systems

A modified version of the Actor model (Agha, 1986) was designed (Kirk et al., 1997) (Nigro and Pupo, 1997) to support the construction of real-time applications. It centres on actors directly modelled as finite state machines. As in the Actor model three basic operations are available:

- *new*, for the creation of a new actor as an instance of a class which directly or indirectly derives from a basic Actor class. The data component of an actor includes a set of *acquaintances*, i.e., the known actors to which messages can be sent
- *send*, for transmitting an *asynchronous message* to a destination actor. The message can carry data values. The sender continues immediately after the send operation
- *become*, for changing the current state of the actor. All of this modifies the way the actor will respond to expected messages. The processing of an unexpected message can be postponed by storing it into states or data.

Differences from the Actor model were introduced for ensuring real-time behaviour. Each object no longer has an internal thread. Rather, concurrency is provided by a *control machine* (see later in this paper) which transparently buffers all the exchanged messages among the actors residing on a same processor, and delivers them according to a control strategy. In other terms, concurrency relies on a light-weight mechanism: *message processing interleaving*, which costs a method invocation in an object-oriented language. Only one message processing can be in progress within an actor at any instant in time. Message execution can't be suspended nor interrupted. All of this contributes to a deterministic computation of message response times.

Instead of associating a single mail queue to every actor, a few message queues (e.g., one) are handled by the control machine. A customisable scheduler hosting timing constraints avoids control machine dependencies from the policies and mechanisms of an operating system.

2.1 Basic Components

At the system level an application consists of a collection of subsystems/processors, linked one to another by a (possibly) deterministic interconnection network (e.g., CAN bus (Kirk, 1995)). A subsystem hosts a group of actors which are orchestrated by a control machine. Basic components are the control machine and the application actors. The control machine is articulated into the following sub-components (see also Fig. 1):

- a *local clock*, which contains a "real" time reference clock for all the actors in the subsystem
- a *plan*, which arranges message invocations along a timeline
- a *scheduler*, which filters message transmissions, applies to them a set of timing control clauses and schedules them on the plan. The scheduler is actually split into two actors: the *input filter* (iFilter) and the *output filter* (oFilter). iFilter is responsible of the scheduling actions of *just sent messages*; oFilter can be specialised to verifying timing violations at the dispatch time of a message
- a *controller,* which provides the basic control engine which repeatedly selects (*selector* block) from the plan the next message to be dispatched, and delivers it (*dispatcher* block) to its receiver actor.

A control machine is naturally event-driven. Asynchronous messages are received either from the external environment or from within a subsystem, and are processed by the relevant actors. A message processing can

generate further messages and so on. From this point of view, the message plan of the control machine could reduce to a FIFO queue.

The time-driven paradigm (Nigro and Tisato, 1996), on the other hand, adds the facility of scheduling messages to occur at certain due times. A possible organisation of the message plan consists of a set of partial ordered *time windows* of message invocations. A *message invocation* includes a message *msg*, its destination actor and two pieces of timing information: t_{min} and t_{max}. It is intended that *msg* cannot be dispatched before t_{min} and should be delivered before t_{max} to avoid a timing violation. The selector component can choose the next dispatch message on the basis of an *Earliest Deadline First* (EDF) strategy.

Figure 1: Architecture of an actor-based RT subsystem.

The control machines of a whole system can interact one to another in order to fulfil system-level requirements. To this purpose a suitable *interaction pattern* (protocol) (Agha, 1996) can be introduced which is responsible of inter-subsystem synchronisation (e.g., real-time barrier synchronisation (Saito and Agha, 1995)). An actor system can require both local and global co-ordination and control. A selected subsystem can host as a *master* control machine, where the operator, through a friendly and asynchronous graphical user interface, can issue configuration or monitoring commands.

2.2 Timing Constraints

A timing constraint normally specifies a time interval between the occurrences of a group of causally connected messages. A violation of a timing constraint can require a recovery action to be carried out. The following operations are provided for the expression of timing constraints and scheduler programming:

- msg.*cause()*, which returns the identity of a message msg' whose processing caused the generation of the message msg
- msg.*iTime()*, which returns the time at which the message msg was dispatched
- msg.*deadline()*, which returns the upper limit of msg's time window
- *schedule*(msg, t_{min}, t_{max}), which schedules on the control machine plan the message msg with the associated time window [t_{min}..t_{max}]
- *now()*, which returns the current value of local clock.

It is worthy of note that for the *iTime* function to be applicable to an incoming network message, a notion of a global time has to be provided (Ren et al., 1996).

A filter is fed of the timing parameters of a subsystem. A timing constraint is identified by a given *pattern*, i.e., a boolean expression involving message parameters and local data of the filter. If the pattern is satisfied the corresponding timing constraint is applied, possibly executing a scheduling action.

The following are some common examples of timing constraints. For simplicity, the are expressed in Java syntax. A framework is assumed where the external environment is sensed by periodic terminator actors and an environment condition starts a chain of messages ("thread of control") whose processing can be required to terminate within a deadline. A terminator synchronously operates a corresponding device (e.g., a sensor).

Periodic message
The time clause is associated with an actor (e.g., a terminator) which has a repetitive message (e.g., ReadCO). After being processed, the message is sent again by the actor to itself. Message periodicity is ensured externally to the actor in the iFilter:

```
if( m instanceof ReadCO && m.cause() == m ) schedule( m, m.cause().iTime()+P, m.cause().iTime()+P );
```

where *P* is the message period, which is data local to the iFilter.

Deadline on a message
A message can be required to be handled within a given deadline *D* measured since the invocation time of its cause:

```
if( m instanceof DeadlineMessage ) schedule( m, now(), m.cause().iTime()+D );
```

Urgent time clause
A message can be scheduled to occur immediately, e.g., for safety conditions:

```
schedule( m, now(), now() );
```

Default time clause
A default time clause can schedule a message to occur according to the deadline of its cause:

```
schedule( m, now(), m.cause().deadline() );
```

A more weak time clause requiring dispatching a message "as soon as possible" can be achieved by:

```
schedule( m, now(), infinite );
```

3 An example

The following considers a simplified model of a software controlled crane (Bergmans and Aksit, 1996) that can lift and carry containers from arriving trucks to a buffer area, from where they are taken for further handling. To carry the containers, the crane uses a magnetic latching mechanism. Actors can naturally model the crane control system (Crane), the buffer area (BoundedBuffer) and the operator (Console). Terminator actors are introduced for controlling the engine, the magnet, and the periodic item detector, left and right position sensor physical devices. Actor Crane understands the following messages:

- *On*, *Off* (turn on and off the magnet)
- *Loaded*, *UnLoaded* (raised by the *ItemDetector*)
- *Forward*, *Backward* (start a forward/backward crane movement)
- *RightStop*, *LeftStop* (raised respectively by *RightSensor* and *LeftSensor*)
- *Emergency* (stops crane and turns off the magnet in emergency situations).

In addition, Crane exports (synchronous) accessor state inquiries: *isOn*, *isOff*, *Moving*, *Stationary*, *isLoaded*, *isUnLoaded*, which return information about crane current state. The Engine and the Magnet actor classes have

messages for turning their state on and off. The Engine has also a message to setup the direction movement (forward/backward). The ItemDetector senses a container into the latching mechanism and sends a Loaded or Unloaded message to crane. Position sensors capture crane limit positions where the engine should be turned off. They transmit a RightStop or LeftStop message respectively.

Starting from a loaded container, first a Loaded message is received by crane. Loaded causes a Forward message to be sent by crane to itself. Processing Forward in turn generates messages for setting up the engine rotation movement and turning it on. Then the crane enters a MOVING state where it waits for a RightStop message. After that, the engine is turned off and a Put message is sent to the buffer area. After receiving a reply from put, the container is released by turning off the magnet and a Backward message is sent to the crane itself for the backward movement.

The crane system can also be controlled by the operator at the console under the restriction that On or Off requests are rejected while crane is moving. However, an Emergency message has to be processed within an assigned deadline *d*. Fig. 2 shows an iFilter which schedules the crane system. It is initialised with the relevant timing attributes: period of sensors and emergency deadline.

```
public class iFilter extends Actor{
   long p /*sensors period*/, d /*emergency deadline*/;
   iFilter( long p, long d ){ this.p=p; this.d=d; }
protected void handler( Message m ){
   if( m instanceof ReadItemDetector || m instanceof ReadLeftSensor || m instanceof ReadRightSensor ) {
      //periodic message
      if( m.cause() == m ) //schedule m according to period p
         ControlMachine.schedule( m, m.cause().iTime()+p,m.cause().iTime()+p );
      else //initialisation
         ControlMachine.schedule( m, now(), now() );
   }
   else if( m instanceof Emergency )
      ControlMachine.schedule( m, now(), now()+d );
   else if(    m.sender instanceof Operator || m.cause() instanceof ReadItemDetector ||
               m.cause() instanceof ReadLeftSensor || m.cause() instanceof ReadRightSensor )  //weak time constraint
      ControlMachine.schedule( m, now(), ControlMachine.infinite )
   else //default time constraint
      ControlMachine.schedule( m, now(), m.cause().deadline() )
}//handler
}//iFilter
```

Figure 2: An iFilter for scheduling the crane system.

It should be noted that self-driving periodic actors send to themselves the first periodic message during initialisation (i.e., within the constructor).
Exchanged messages during normal, automated behaviour, are handled by a default weak timing constraint with t_{max} being infinite. However, an Emergency thread of messages is managed according to its deadline.

4 Modelling Actor Components by CP-nets

Basic system components (actors, control machine, scheduler, ...) can be 1-to-1 mapped on to CPN-subnets (pages). Mapping rules are described by modelling the crane system example presented in section 3. Figure 3 represents the hierarchy page of a CPN model, which has a node for each page in the model. An arc between two nodes indicates that the source node contains a transition (*substitution transition*) whose details are contained in the destination node. The page of the destination node is called a *subpage*. Each node is inscribed by the name and number of the corresponding page, while each arc is inscribed with the name of the corresponding substitution transition.

The hierarchy page shows the net structure of the model. At an highest level it is composed of three parts:

- the System Description part, which consists of the page *Crane_System*. It provides the most abstract view of the system

- the Actors part, which consists of eleven pages (*Actors, Crane, BoundedBuffer, Console, SensorSet, LeftSensor, RightSensor, ItemDetector, Actuators, Magnet, Engine*)
- the Control Machine part, which consists of five pages (*ControlMachine, iFilter, Selector, oFilter, Dispatcher*).

Crane_System is the topmost page in the hierarchy and it is shown in Fig. 4. It contains *Actors* and *ControlMachine* transitions communicating through the interface places CMPIn and CMPOut.

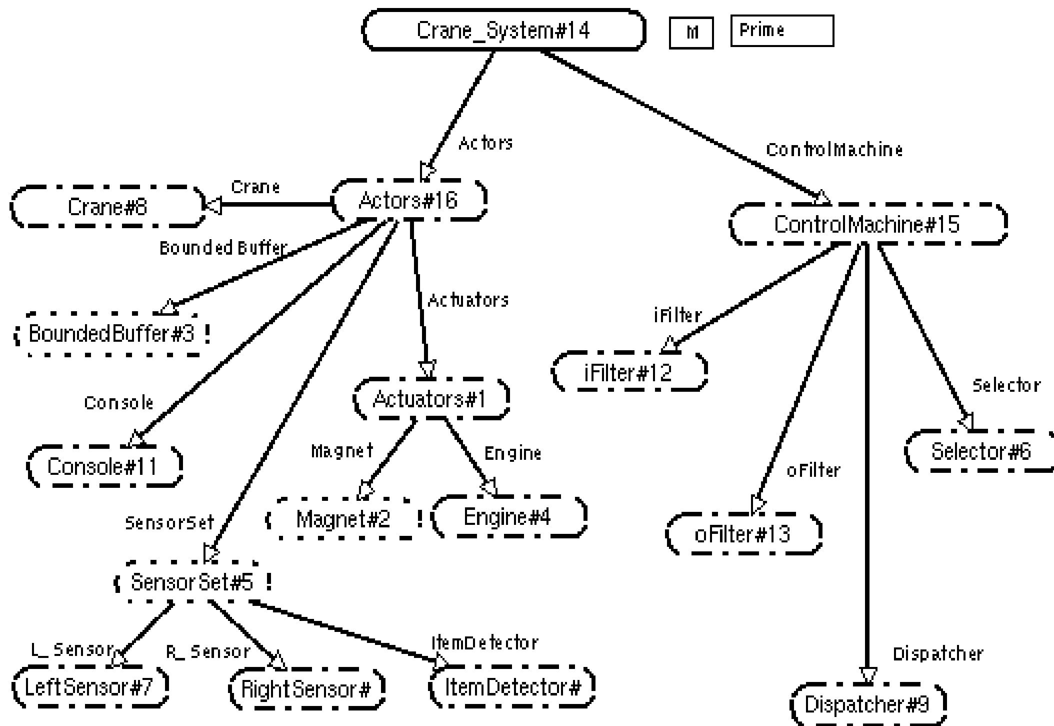

Figure 3: Hierarchy page of the crane system.

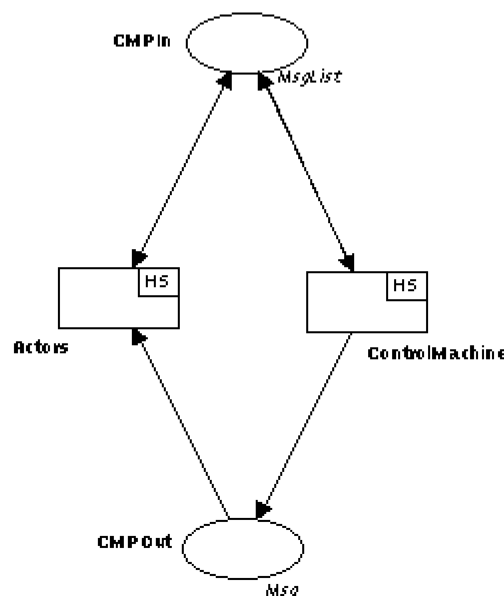

Figure 4: The Crane_System page.

Actors and ControlMachine in Fig. 4 are substitution transitions and from the hierarchy page it can be deduced that they correspond respectively to *Actors*#16 and *ControlMachine*#15 pages. The page Actors#16 is shown in Fig. 5 and contains some substitution transitions which represent the actor group in the system. Some of these

are directly substituted by subpages modelling corresponding actors (e.g., the *BoundedBuffer* page portrayed in Fig. 6), others abstract a group of logically related actors (e.g. SensorSet and Actuators) that are refined at a further sub-level.

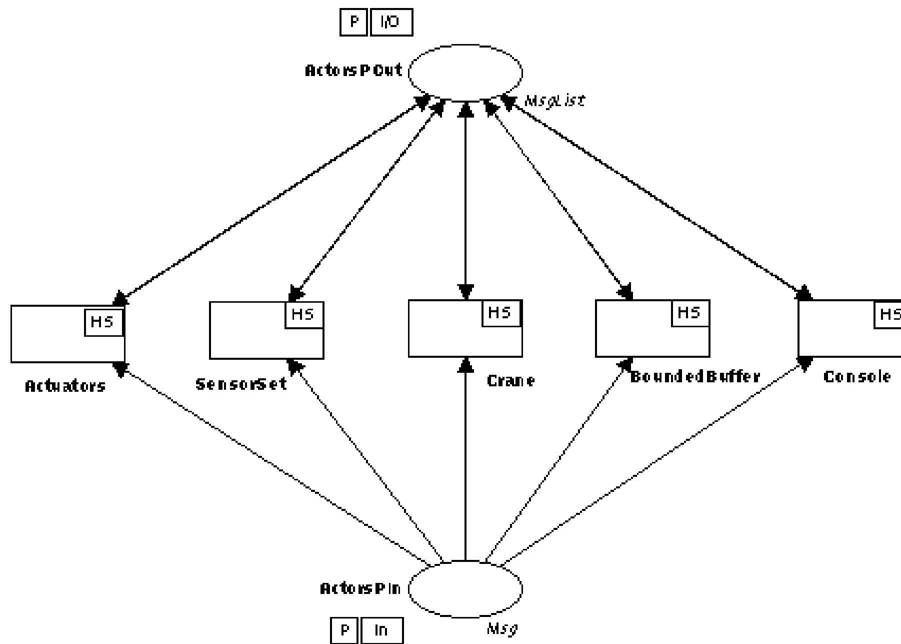

Figure 5: The Actors page.

In order to favour the analysis task of a system, actors can be modelled by a single transition (*action transition*) annotated by an estimation of its worst-case-execution-time. The actor behaviour is then captured by ML arc expressions and some local state places (see Fig. 6). This way two advantages can be achieved. The first one is related to a reduction of the net complexity and therefore to a minimising of the state explosion problem of the occurrence graph method. The second one is concerned with a reduction of the "distortion risks" during the translation of the verified model to the target application. This is obtained by expressing, owing to the high level character of the ML language, actor state transitions and actions in a form very close to the final implementation language.

The BoundedBuffer actor can receive a *Get* or *Put* message and can find itself into one of three states: *Empty*, *Partial* and *Full*. An incoming unexpected message, e.g., a *Put* in the *Full* status, is deferred by storing it in a local queue (token in the DEQ place). Deferred messages are re-scheduled to be received again as soon as the actor changes its current state.

The Control Machine component is modelled by the ControlMachine#15 page shown in Fig. 7. It contains four substitution transitions (*iFilter*, *Selector*, *oFilter* and *Dispatcher*) corresponding to the basic sub-components of the control machine, and five places, two of which (CMPIn and CMPOut) represent the interface with the system actors. CMPIn and CMPOut are port places corresponding to the socket places CMPIn and CMPOut of Crane_System page. These sockets are also assigned respectively to the ports ActorsPOut and ActorsPIn of the page Actors#16. The places Plan, CMP1 and CMP2 allow the control machine components to communicate.

The iFilter component is modelled by the page iFilter#12 depicted in Fig. 8. It contains a transition (iFilter) and four places. At each occurrence of the iFilter transition, the ifilt(...) function is invoked which takes the plan message list from the iFP2 place and the just sent message list from the iFP1 place, applies the time clauses of Fig. 2 and generates a new plan message list. Place iFP3 holds a copy of the last dispatched message which is used as a repository for inquiring about the causal message and its attributes of a message under scheduling.

The CMMonitor place is a member of a Fusion Global set and contains a token of the CMMon colour set which ensures that scheduling, selecting and dispatching activities in the control machine are strictly sequenced.

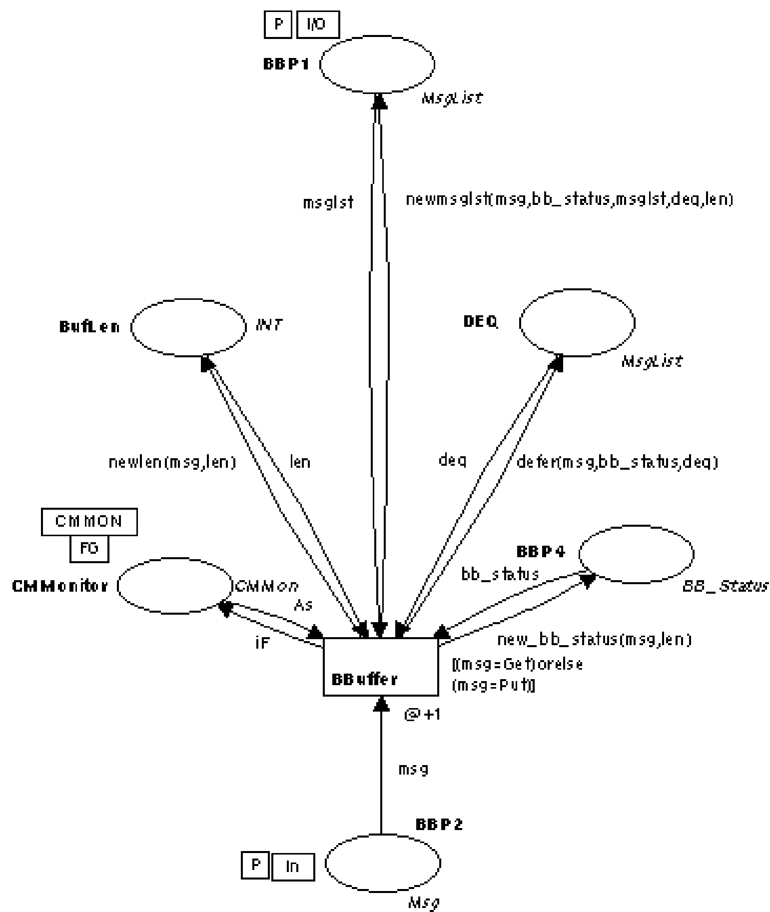

Figure 6: The Bounded Buffer page.

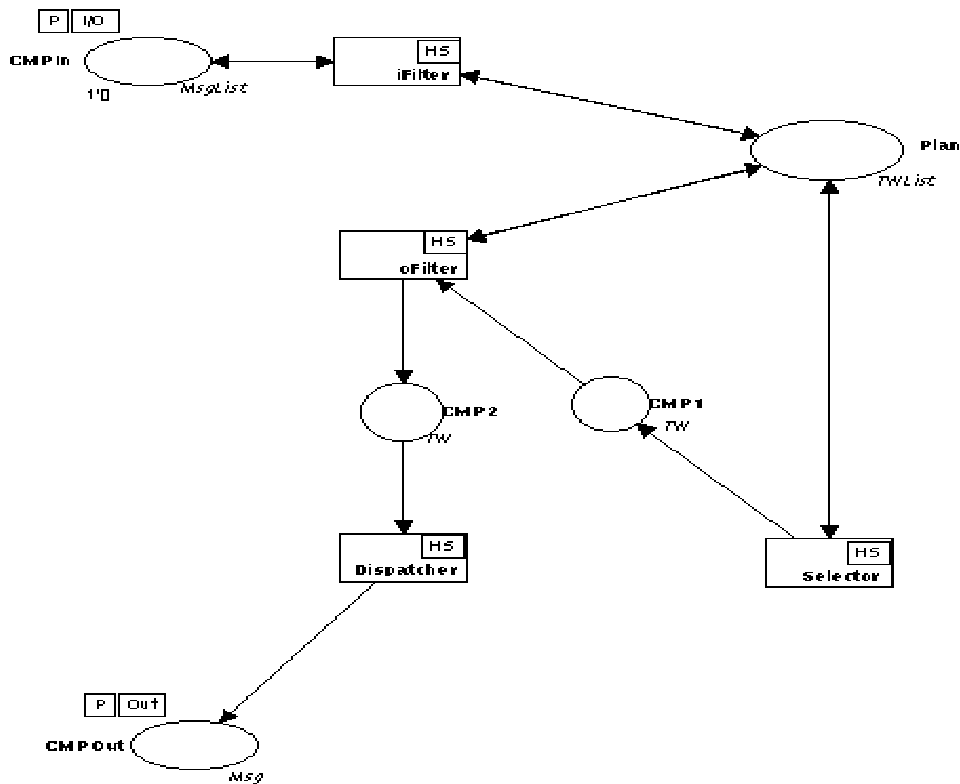

Figure 7: The Control Machine page.

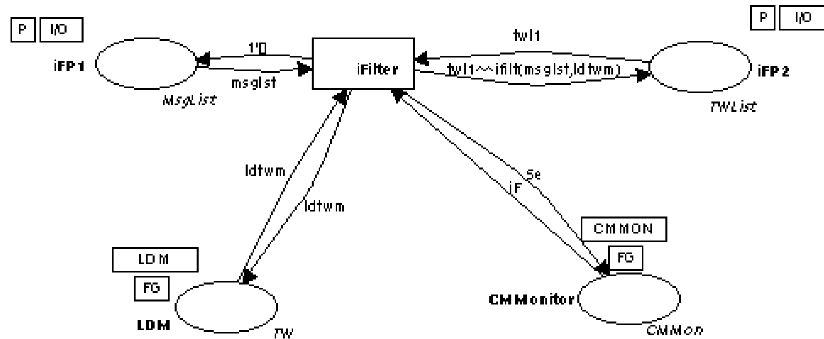

Figure 8: The iFilter page.

The iFP2 place corresponds to the socket place Plan in the page ControlMachine#15. This socket node is assigned also to the Plan port node of the page *Selector#6* which is represented in Fig. 9. This page models the selection of the next message to be dispatched from the Plan. In this case an EDF strategy is used. First the Plan

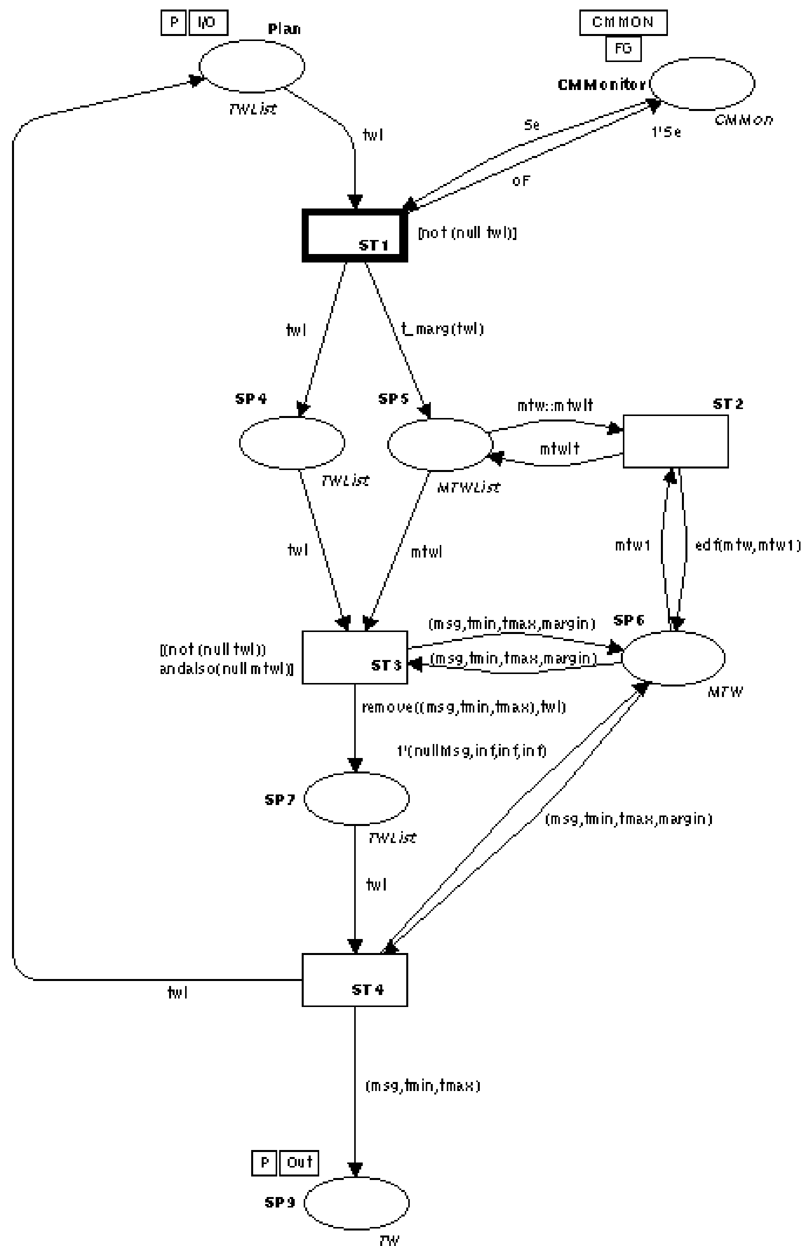

Figure 9: The Selector page.

message list is duplicated into the SP4 and SP5 places. The t_marg() function appends to each message item its *temporal margin*, i.e., the difference between the *tmax* value of the message time window and the current time. Transition ST2 edf() function is responsible of determining in the SP6 place the message with the minimum temporal margin which is removed (ST3 transition) from the message list copy in the place SP4. Finally, the firing of the ST4 transition updates the Plan message list and copies the selected message into the SP9 place for the subsequent dispatch phase.

It should be noted that for the purposes of the Crane example, the oFilter page doesn't apply any function to the message to be dispatched. Therefore, this page is omitted.

The Dispatcher page depicted in Fig. 10 is in charge of delivering the selected message to its relevant actor. The time inscription of the Dispatch transition:

@+if (tmin>time()) then (tmin-time()) else 0

allows to timestamp the message token in the DP2 place with the amount of time possibly required to advance the system time to the message *tmin*.

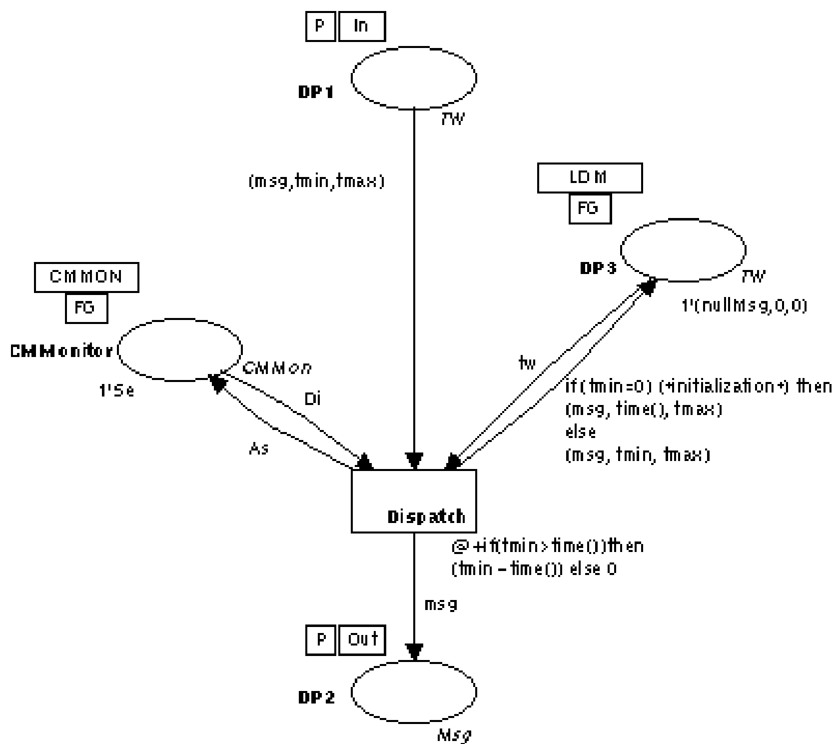

Figure 11: The Dispatcher page.

The destination actor receives the dispatched message, processes it by firing its action transition and generates a list of new messages. All these messages get timestamped with the occurrence time of the action transition augumented by the action duration. After that the control machine resumes its activity into the iFilter component thus starting a new cycle of the control loop.

5 Analysis of an Actor-based CPN Model

The analysis process of an actor-based CPN model aims at verifying and possibly correcting the specification of timing constraints and action worst-case-execution-times in order to ensure *schedulability* (e.g., Tsai et al., 1995). In particular, the analysis has to assert that every thread is completed within its deadline.

Both informal and formal analysis methods can be applied. The informal analysis can be conducted by simulating the model (*specifications testing* (Ghezzi et al., 1991)), i.e., by providing an initial marking and

then by tracing one or more possible resulting behaviours. By observing these behaviours the analyst can realise whether or not the specified system meets functional or timing requirements. Temporal behaviour can be verified by analysing the multiple concurrent threads especially under peak-loads.

Formal analysis consists in defining general properties of the net model which reflect special types of desirable (or undesirable) behaviours of the specified system, and then using the specification to formally prove (or disprove) such properties. For this purpose the *occurrence graph* (OG) method (Jensen, 1994) and the Occurrence Graph Tool (Occ Tool) (Christensen et al., 1996) can be used.

An OG is a directed graph which has a node for each reachable marking and an arc for each occurring binding element. An arc links the node of the marking in which the associated binding element occurs to the node of the marking resulting from the occurrence. Such a graph may become so large, even for relatively small nets, that it cannot possibly be generated even with the most powerful computer. Another limitation is dependency from the initial marking: each possible initial marking may originate a different occurrence graph.

The OGs generated for the analysis of an actor system are always timed, i.e., each node represents a system state and contains a time value and a timed marking. Since the typical periodic behaviour of such an RT system, it is possible to reduce the size of an OG by choosing a suitable simulation time during which meaningful system activity (e.g., peak-load conditions) occurs. In addition, the adopted modelling techniques (i.e., colour sets as intervals or enumerated values, and the adopted net structure) purposely contribute to keep under control the state space explosion problem.

All standard dynamic properties for a CP-net can be derived from its OG, e.g., boundedness, home, liveness, and fairness properties. They all can be investigated by available functions in the OG Tool. Moreover ML functions can be written for issuing non-standard inquiries to OG.

For instance, the maximum number of simultaneously enabled transition instances can be found by the MaxTransEnabled function in Fig. 12.

```
fun MaxTransEnabled () : int
  = SearchNodes (EntireGraph,
                 fn _ => true,
                 NoLimit,
                 fn n => length( remdupl( map ArcToTI( OutArcs n ) ) ),
                 0,
                 max);
```

Figure 12: Predictable system behaviour check function.

When invoked on any generated OG for the Crane CPN model, the function returns 1:

MaxTransEnabled ();
1 : int;

thus confirming that the modelled system always evolves in a predictable manner. Indeed, when a system consists of a single subsystem there is always only one transition enabled (belonging either to an actor or to an internal component of the control machine).

Similarly, one such a function can capture the verification of a temporal property in positive or negative form, provided an OG is generated for a suitable simulation period.

To exemplify, it is possible to carry out the verification of properties like the following: “is it always true that for each instance of a given transition firing (representing, e.g., the beginning of a thread) there always (as the OG state space allows) exists an instance of the corresponding transition firing (modelling, e.g., the end of the thread) such that the time distance between them is less than a fixed time interval (e.g., deadline of thread execution) ?”

The negative form of a property can be more immediate. In this case the existence of a single occurrence of a searched event that contradicts the property is sufficient to assert that the property doesn't hold.

In the crane example it can be verified (see Fig. 13) that for each occurrence of the *Emergency* event it always (in the generated OG) follows a transition instance corresponding to turning off the magnet (*Magnet* action transition) such that the temporal distance between the two events is less than a deadline *d.*

```
fun EmerDeadln(): Node list =
  PredAllNodes(fn n =>
              let
                val Cr_Em = StripTime(Mark.Crane_System'CMPOut 1 n);
              in
                if Cr_Em == 1`Cr_Emergency then
                  null(PredArcs(EntireGraph,
                                fn a =>
                                  if ArcToTI(a) = TI.Magnet'Magnet 1 then
                                    ((CreationTime(DestNode(a))-CreationTime(n))<d)
                                    andalso(DestNode(a)>n)
                                  else false,
                                1))
                else false
              end)
```

Figure 13: Temporal property check function.

When invoked, the EmerDeadln() function always returns an empty list, proving that the list built with magnet transition instances which are causally connected to an Emergency event and temporally internal to the deadline interval, is never empty:

EmerDeadln();
val it = [] : Node list

The CP-net modelling of an actor system supports an *incremental development* through a modified spiral lifecycle model (Verber et al., 1997). Functional and temporal analysis can start as soon as a CPN specification has been produced and be based on the required obligations (periods, thread deadlines, ...) and a preliminary estimation of actor message processing times. As the temporal information about actors get more accurate, i.e., the project is tuned to a physical target architecture, the CPN model can be applied again to check the system remains schedulable. The key point is that the CPN model closely mirrors the design/implementation models of a system. Therefore, conclusions drawn from CPN analysis can directly be interpreted at the lower levels of development.

The experimental analysis of the Crane system was performed on a Sun Sparc 5 with 24 MB of physical RAM.

6 Modelling Issues of Distributed Systems

This section highlights how the CPN modelling techniques previously described can be adapted to support distributed systems. A simple example is sketched in Fig. 14 where two subsystems interact one to another by means of a network which introduces a constant transmission delay. In a more realistic case the network component could model a CAN priority bus (Nigro and Pupo, 1998).

The first subsystem (Site1) contains a pressure sensor and a valve. The second subsystem (Site2) contains a pressure controller which sends a message to open the valve when the pressure value exceeds a given limit. Site1 and Site2 share the basic Control Machine components which are instantiated for each subsystem.
The Network page is illustrated in Fig. 15. It models the message routing from a subsystem to another. The NBufOut place has a message list for each subsystem. A network message received into the NBufIn place is routed by the arc inscriptions to the correct message list in the NBufOut.

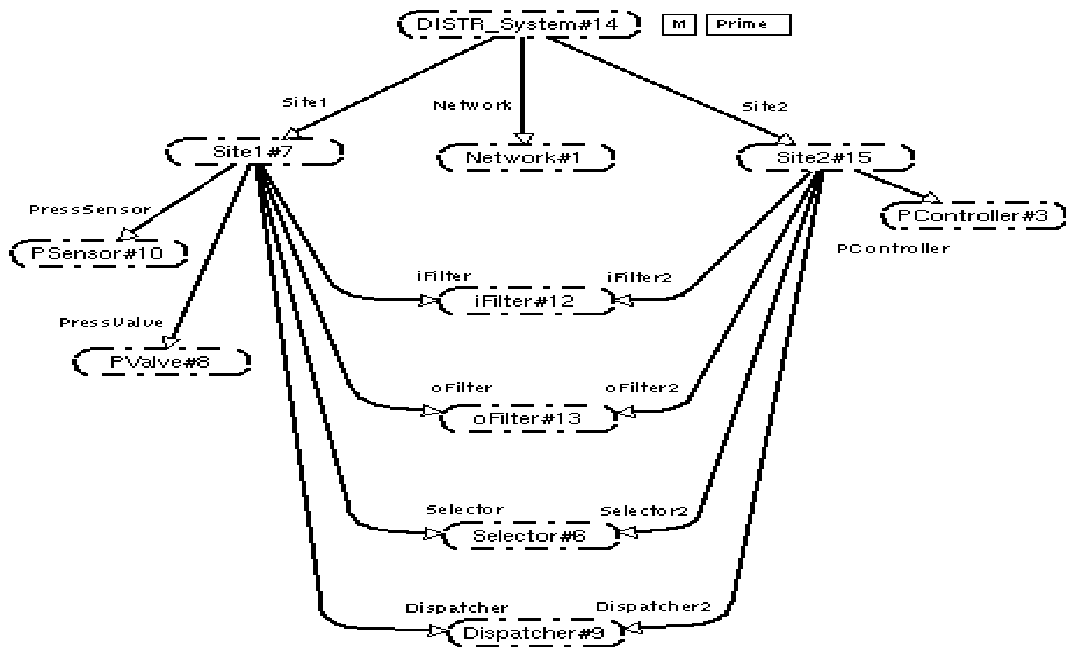

Figure 14: A simple distributed system.

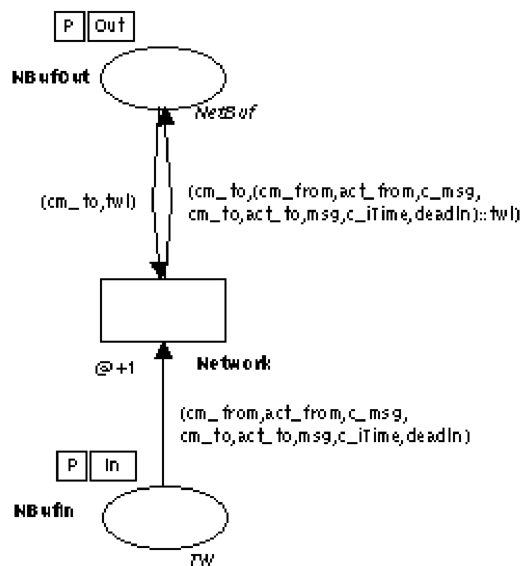

Figure 15: The Network page.

Fig. 16 shows the Site1 page. The identity of the control machine component instances is explicitly maintained by a token in the InstP11 and InstP12 places. Incoming network messages are scheduled on the plan along with the local generated messages. Outgoing network messages are deposited into the ToNetP place which corresponds to the NBufIn place of the Network page in Fig. 15.

It is worth noting that the two subsystems concurrently execute but the Design/CPN time notion ensures that the timing behaviour of each subsystem is always kept coherent with the time evolution of the whole system. For instance, a selected message by a control machine remains pending into the dispatch place if its timestamp is greater than the current CPN clock time. Only when no other transition in the model exists which can fire by incrementing the simulation time of a lower amount, the message is eventually delivered to its destination actor. As a consequence, the analysis of system-wide temporal properties, e.g., concerning threads of control which cross the boundaries of multiple subsystems, can be carried out according to the same techniques described in the section 5.

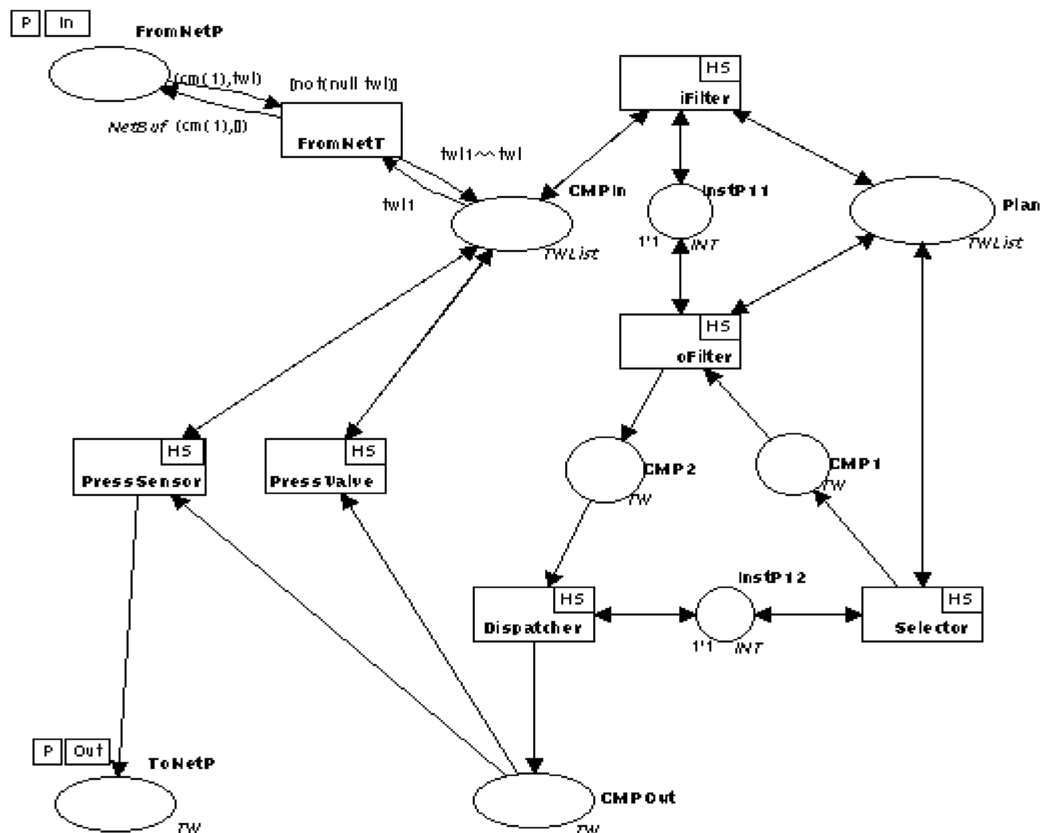

Figure 16: The Site1 page.

7 Conclusions

This paper describes an actor-based framework suited to the development of real-time distributed systems and shows a possible formalisation in terms of CP-nets. Benefits of the formalisation are the possibility of achieving visualisation and verification support in the context of the Design/CPN tool.

Like in (Verber et al., 1997) (Nigro and Pupo, 1996) both temporal and functional aspects can be considered since the early stages of a project using a modified spiral-model design life cycle where specifications can gradually be refined as the development becomes more and more specific. In particular, as the action execution time estimates become more precise, schedulability and timing behaviour can be checked that remain fulfilled. Temporal properties can be analysed by generating and exploring occurrence graphs.

The proposed approach facilitates transition to the final development phases, in the sense that a modelled and verified system can directly be transformed into the implementation terms of an object oriented language.

Directions of further work include

- experimenting with the proposed CPN modelling to real-life time-dependent systems
- improving the support of distributed systems by a better exploitation of page reuse and instance management mechanisms, in the presence of deterministic networks like CAN (Kirk, 1995) (Tindel et. al.,) which assign priority to inter-subsystem messages to control their transmission delay
- improving property analysis by using timed CTL (Cheng et al., 1996) for expressing queries on the state space of a generated occurrence graph.

The actor-based approach described in this paper is also in current use in the realisation of

- distributed measurement systems (Grimaldi et al., 1998)
- distributed simulation of cellular networks (Beraldi and Nigro, 1998)
- multimedia applications (Fortino and Nigro, 1998).

Acknowledgments

Work carried out under the financial support of the Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the Project "Methodologies and Tools of High Performance Systems for Distributed Applications".

The authors wish to thank L.M. Kristensen and K. H. Mortensen of the CPN group at the University of Aarhus for their support during the experimental work with the Design/CPN tools.

References

Agha G. (1986). *Actors: A model for concurrent computation in distributed systems*. MIT Press.
Agha G. (1996). Abstracting interaction patterns: a programming paradigm for open distributed systems. *Formal Methods for Open Object-based Distributed Systems*, Vol. **1**, Najm E. and Stefani J. B. (eds), Chapman & Hall.
Agha G., S. Miriyala and Y. Sami (1992). Visualizing Actor Programs using Predicate Transition Nets. *Journal of Visual Languages and Computation*, **3**(2), pp. 195-220, June.
Beraldi R. and L. Nigro (1998). Performance of a Time Warp based simulator of large scale PCS networks. *Simulation Practice and Theory*, **6**(2), pp. 149-163, February.
Bergmans L. and M. Aksit (1996). Composing synchronisation and real-time constraints, *J. of Parallel and Distributed Computing*, September .
Cheng A., Christensen S. and K.H. Mortensen (1996). Model checking Coloured Petri Nets: Exploting strongly connected components. WoDES'96, August 20. http://www.daimi.aau.dk/designCPN/libs/askctl.
Christensen S., K. Jensen and L. Kristensen (1996). The Design/CPN Occurrence Graph Tool. User's manual version 3.0. Computer Science Department, University of Aarhus. http://www.daimi.aau.dk/designCPN/.
Fortino G. and L. Nigro (1998). QoS centred Java and actor based framework for real/virtual teleconferences. *Proc. of SCS EuroMedia98*, Leicester (UK), Jan. 5-6, pp. 129-133.
Grimaldi D., L. Nigro and F. Pupo (1998). Java based distributed measurement systems. To appear on *IEEE Trans. on Instr. and Measurement*.
Genrich H. J. (1987). Predicate/transition nets. In *Advances in Petri Nets*, W. Brauer, W. Reisig and G. Rozenberg (eds.), New York, Springer Verlag.
Ghezzi C., D. Mandrioli, S. Morasca and M. Pezzè (1991). A unified high-level Petri net formalism for time-critical systems. *IEEE Trans. on Software Engineering*, **17**(2), pp. 160-172, February.
Kirk B. (1995). Real time protocol design for control area networks. *Proc. of Real Time'95*, Ostrava (Cz Rep.), pp. 251-268.
Kirk B., L. Nigro and F. Pupo (1997). Using real time constraints for modularisation. Springer-Verlag , LNCS 1204, pp. 236-251.
Jensen K. (1992). *Coloured Petri Nets - Basic concepts, analysis methods and practical use*. Vol. **1**: Basic concepts. EATCS Monographs on Theoretical Computer Science. Springer-Verlag.
Jensen K. (1994). *Coloured Petri Nets - Basic concepts, analysis methods and practical use*. Vol. **2**: Analysis methods. EATCS Monographs on Theoretical Computer Science. Springer-Verlag.
Jensen K. (1997). *Coloured Petri Nets - Basic concepts, analysis methods and practical use*. Vol. **3**: Practical use. EATCS Monographs on Theoretical Computer Science. Springer-Verlag.
Jensen K., S. Christensen, P. Huber and M. Holla. (1996). Design/CPN. A reference manual. *Computer Science Department*, University of Aaurus. Online: http://www.daimi.aau.dk/designCPN/.
Nigro L. and F. Pupo (1996). Modelling and analysing DART systems through high level Petri nets, Springer-Verlag, LNCS 1091, pp. 420-439.
Nigro L. and F. Tisato (1996). Timing as a programming in-the-large issue. *Microsystems and Microprocessors*, **20**, pp. 211-223, June.
Nigro L. and F. Pupo (1997). A modular approach to real time programming using actors and Java. *Proc. of 22nd IFAC/IFIP Workshop on Real Time Programming*, Lyon, 15-17 September, 83-88.
Nigro L. and F. Pupo (1998). Actors and Coloured Petri Nets in the development life cycle of distributed real time systems. To be presented at *IFAC Large Scale Systems'98*, Univ. of Patras Rio(Greece), 15-17 July.
Ren S. and G. Agha (1995). RTsynchronizer: language support for real-time specification in distributed systems. *ACM SIGPLAN Notices*, **30**, pp. 50-59.
Ren S., G. Agha and M. Saito (1996). A modular approach for programming distributed real-time systems. *J. of Parallel and Distributed Computing*, Special issue on Object-Oriented Real-Time Systems.
Saito M. and G. Agha (1995). A modular approach to real-time synchronisation. In *Object-Oriented Real-Time Systems Workshop*, 13-22, *OOPS Messenger*, *ACM SIGPLAN*.
Sami Y. and G. Vidal-Naquet (1991). Formalization of the behaviour of actors by coloured Petri nets and some applications. *PARLE '91*.
Tsai J.J.P., S.J. Yang and Y.-H. Chang (1995). Timing Constraints Petri Nets and their application to schedulability analysis of real-time system specification. *IEEE Trans. on Software Engineering*, **21**(1), pp. 32-49, January.
Tindel K., A. Burns and A.J. Wellings (1995). Analysis of hard real time communications. *Real Time Systems*, *9*, pp. 147-171.
Verber D., M. Colnaric, A.H. Frigeri and W.A. Halang (1997). Object orientation in the real-time system lifecycle. *Proc. of 22nd IFAC/IFIP Workshop on Real Time Programming*, Lyon, 15-17 September, pp. 77-82.