

MODULAR MODELLING AND ANALYSIS OF TIME-DEPENDENT SYSTEMS

Franco Cicirelli, Angelo Furfaro, Libero Nigro, Francesco Pupo
Laboratorio di Ingegneria del Software
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria, 87036 Rende (CS) – Italy
E-mail: *{f.cicirelli,a.furfaro}@deis.unical.it {l.nigro,f.pupo}@unical.it*

KEYWORDS

Modular modelling, real-time systems, time Petri nets, discrete-event simulation, model checking, UPPAAL.

ABSTRACT

This paper describes an approach to the analysis of time-dependent systems which combines discrete-event simulation and model-checking techniques. The approach rests on Merlin and Farber's Time Petri Nets (TimePNs) and is supported by a Java toolbox –TPN Designer– which enables graphical modelling, simulation and translation into UPPAAL/Timed Automata, for exhaustive state space verification, of modular TimePN models. The paper discusses the potential of the proposed approach through its application to a real-time system model.

INTRODUCTION

The design of a time-dependent system (e.g. an embedded real-time system, a communication protocol etc.) must fulfil a set of functional and non-functional requirements. Non functional requirements include *timeliness*, that is the ability of the software system to provide responses to events (e.g., originated in an external controlled environment) which must be functionally correct *and* furnished within precise and predictable timing constraints (e.g. deadlines). To ensure a system development compliant with timing requirements, the use of formal tools is mandatory. A formal language can favour unambiguous specification of system behaviour and, most importantly, can allow reasoning on and assessment of system properties.

In the work described in this paper, Merlin and Farber's Time Petri Nets –TimePNs– (Merlin and Farber, 1976) are chosen as the modelling language. TimePNs permit rigorous specification of a time-dependent system by associating a time interval to transitions (activities) which constrains transition firing. More precisely, let t be a transition and $[a,b]$, $0 \leq a \leq b$, b can be ∞ , the time interval associated with t (a is said the *earliest firing time*, b is the *latest firing time* of t). Let τ be an instant in time when t gets enabled in the usual sense of Petri nets (Murata, 1989). Provided t is continuously enabled, it cannot fire before $\tau+a$ but should fire before or at

$\tau+b$ unless t is disabled by the firing of a conflicting transition.

TimePNs were recently added to a Java toolbox –TPN DESIGNER (Carullo *et al.*, 2003)– which supports modelling and discrete-event simulation of complex and modular systems. TPN DESIGNER consists of two integrated sub environments: CAD and ENGINE. CAD is concerned with model editing and project management. ENGINE supports compilation, debugging, simulation and statistical data collection. A large system can be split and step-wise refined into modular units called *pages*. A page has an interface of *input/output ports* and hides an internal behaviour expressed by a GSPN-like (Marsan *et al.*, 1984)(Marsan *et al.*, 1987) subnet which in turn can be organized in sub pages and so forth recursively to any arbitrary depth. Pages can be spatially replicated to create linear (pipeline) or bi-dimensional (grid) topologies.

Model configuration requires port interconnections and net parameters (marking of places and attributes of transitions) to be established. Ports can be linked to one another in CAD through the mouse. More in general, a simple but powerful *scripting language* can be used to “program” port interconnections. Scripting can occur at *page level* or at *model* (or *root*) *level*. Page level scripting can be exploited to establish port bindings among instances (*siblings*) of a same page model. Root level scripting can be used to define connections between arbitrary but compatible subjects (places or transitions) possibly belonging to different pages, and to furnish values for model parameters. Root level scripting is essential for model *scalability*. A properly configured model can be scaled (e.g. page multiplicities changed and model parameters redefined) simply by modifying a few values in the root level script. A hierarchical model is compiled into a flat representation which is then managed by ENGINE.

For statistics data collection a flexible *watch system* can be used. Common statistics like gathering minimum, maximum and average number of tokens/firings for selected places/transitions can be achieved by adding one or more watches to selected entities in the graphical ENGINE sub-environment. To cope with general statistics calculations, aspect-oriented *monitors*, i.e.

external Java classes, can be prepared and transparently weaved to TPN DESIGNER control engine. A monitor is event-driven. Specific event occurrences can be observed and application-dependent actions correspondingly executed. Multiple monitors can possibly be jointly used to observe model execution.

General TPN DESIGNER features can be exploited for managing TimePN models. In addition, the ENGINE sub-environment was extended so as to compile a TimePN model in terms of timed automata (Alur & Dill, 1994) in the context of the UPPAAL toolbox (Behrmann *et al.*, 2004), using a template process corresponding to basic TimePN transition. In this way, a TimePN model can be analyzed using simulation and (possibly) model checking (Clarke *et al.*, 2000) i.e. exhaustive verification of system properties against the whole state space of the model. UPPAAL was chosen because of its compact data structures and efficient model checker. This paper argues that simulation can help model checking by suggesting estimates of response time bounds which can then be refined by using a few queries in the context of the verifier. Very often, in fact, the modeller has to tentatively issue many times a given query to the verifier, each time with a different parameter value, in order to determine a worst case response time. All of this can be highly expensive in the wallclock time for complex models. On the other hand, for a large system model, exhaustive verification can be practically prohibitive. In these cases discrete-event simulation, although it can never replace model checking, can anyway furnish important indications about quantitative temporal behaviour of the system.

This paper describes the proposed approach to systems modelling and analysis centred on TPN DESIGNER and UPPAAL, by reporting its application to a real-time modelling example.

A ROBOT CONTROLLER MODEL

The following considers a distributed robot controller system (Gerber & Lee, 1992) whose purpose is to periodically collect positional data from four tactile sensors, to convert the data into real world coordinates, to integrate the data, to determine the robot next position and to send the new position to the robot. The system consists of four tactile sensors, four conversion processes, four single buffered channels, an integrator and the robot itself.

Communications between interacting components are assumed to be synchronous (rendezvous). Each communication and every internal action are supposed to consume 1 time unit (e.g., 1 ms). Each sensor has a period of 6 time units within which it takes a sample reading from the environment and then attempts to communicate the result to the corresponding converter. Since the environment is supposed to be subject to very rapid changes, the communication between a sensor and

its converter must be made within a hard deadline of 4 time units, able to ensure that each sensor terminates its internal and i/o activities before the next period is commenced. Another deadline of 1 time unit exists between a converter which is ready to transmit its converted data and its associated buffered channel. The integrator must first receive the four communications from the buffered channels, in any order, and then prepares the next positional data for the robot. From when the robot is ready to accept its next command, it must do so within 11 time units (hard deadline). Fig. 1 shows a TPN DESIGNER modular model for the system, under the hypothesis of maximal parallelism, i.e. that each component is mapped onto a distinct physical processor.

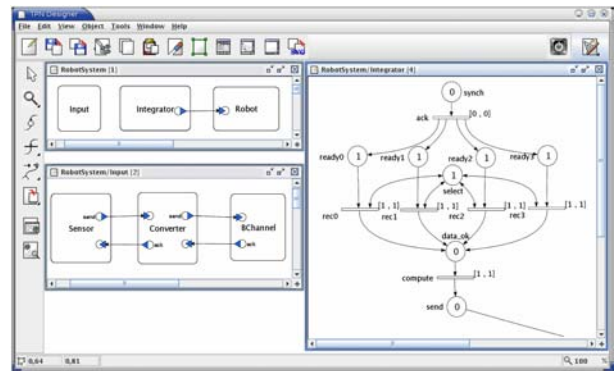


Figure 1: A modular TPN DESIGNER model for the robot controller system

The **Input** page encapsulates **Sensor**, **Converter** and **BChannel** sub pages. To obtain the required system topology, the **Integrator** and **Robot** pages are instantiated only once (default), whereas the **Input** page (and implicitly its contained sub pages) has a monodimensional 4-multiplicity. Interacting page instances within **Input** have input/output ports directly linked by the mouse. Connections between the four instances of the **BChannel** page and the single instance of the **Integrator** are accomplished by the root level script shown in Fig. 2.

```
(1) set Input={4, 1};
(2) link /Input[0,0]/BChannel[0,0]/send to /Integrator[0,0]/rec0 with 1;
(3) link /Input[1,0]/BChannel[0,0]/send to /Integrator[0,0]/rec1 with 1;
(4) link /Input[2,0]/BChannel[0,0]/send to /Integrator[0,0]/rec2 with 1;
(5) link /Input[3,0]/BChannel[0,0]/send to /Integrator[0,0]/rec3 with 1;
(6) link /Integrator[0,0]/rec0 to /Input[0,0]/BChannel[0,0]/reply with 1;
(7) link /Integrator[0,0]/rec1 to /Input[1,0]/BChannel[0,0]/reply with 1;
(8) link /Integrator[0,0]/rec2 to /Input[2,0]/BChannel[0,0]/reply with 1;
(9) link /Integrator[0,0]/rec3 to /Input[3,0]/BChannel[0,0]/reply with 1;
(10) link /Robot[0,0]/receive to /Integrator[0,0]/synch with 1;
```

Figure 2: Root-level script for the Robot model

Line (1) declares the multiplicity of the **Input** page, i.e. 4x1 (4 rows and 1 column). Lines (2) to (5) specify that the **send** place in the various **BChannel** instances (see Fig. 5) has to be linked with the **rec_j**, $j=0, \dots, 3$ transition of the **Integrator** (Fig. 6). Lines from (6) to (9) serve to

connect the rec_j transitions of the Integrator with the reply place of the corresponding BChannel instance. Line (10) states that the receive transition of the Robot page (Fig. 7) must be linked to the synch place of the Integrator. All the arcs corresponding to the link instructions have unitary weight (clause with 1).

Fig. from 3 to 7 depict the internal sub net of each model page together with its initial marking. As a rule, a communication is followed by an ack event sent back to the sender at the end of the synchronization. In addition, the communication delay is spent in the receiver so that the ack event is instantaneous (time interval $[0,0]$).

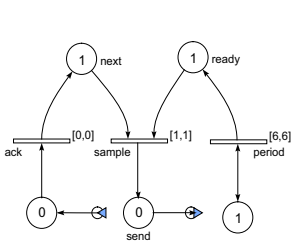


Figure 3: Sensor subnet

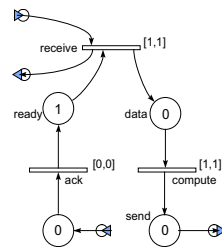


Figure 4: Converter subnet

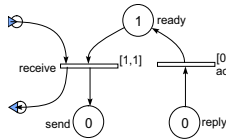


Figure 5: BChannel subnet

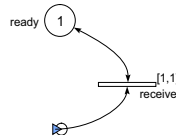


Figure 6: Robot subnet

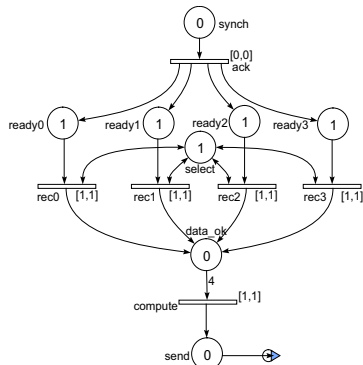


Figure 7: Integrator subnet

Fig. from 3 to 7 depict the internal sub net of each model page together with its initial marking. As a rule, a communication is followed by an ack event sent back to the sender at the end of the synchronization. In addition, the communication delay is spent in the receiver so that the ack event is instantaneous (time interval $[0,0]$).

In the **Sensor** subnet, one token in the **ready** place indicates the sensor is ready to sample the environment. However, reading the sample effectively requires one token in the **next** place too, which mirrors the fact, in general, that previous data has already been communicated to the converter. Satisfaction of sensor deadline can easily be related to having no more than

one token at all times in the **ready** place. **Converter** sub net ensures that first a data is received from a sensor (transition **receive**), then the conversion process is accomplished (**compute** transition) after that a communication is attempted with the corresponding buffered channel, whose completion is witnessed by the firing of the **ack** transition. **BChannel** sub net first receives a converted data from a converter then tries to transmit it to the integrator. The **ack** message is generated as soon as the **send** place empties. The **Integrator** sub net is able to receive data from the four buffered channels through the rec_j transitions.

However, only one communication can take place at each time (due to the **select** place). When all the four data has been achieved, the integrator computes next robot position (**compute** transition) and, finally, tries to communicate it to the robot. At the end of this communication, a token is generated in the **synch** place and then the integrator repeats its behaviour. The **Robot** sub net simply receives next positional information from the integrator (**receive** transition in Fig. 7) and then is again available for the next communication. At each firing of the **receive** transition of the robot, a token is generated in the **synch** place of the integrator through the link established by the root level script (see line (6) in Fig. 2).

A modular and hierarchical TPN DESIGNER model is flattened into a representation made up of concrete subjects (places, transitions and arcs). Meta data like ports are eliminated and port inter-connections and links introduced by scripts are replaced by effective arcs.

SIMULATION EXPERIMENTS

The ENGINE sub environment of TPN DESIGNER simulates a TimePN model by generating the occurrence time of a transition as uniformly distributed in the transition time interval. Toward this, an unbounded interval like $[a, \infty]$ is actually managed as $[a, Tend+1]$ where $Tend$ is the simulation time limit. Moreover, according to TimePN semantics of transition firing, the race policy is implemented. All the enabled transitions at a given time are scheduled to occur to their proposed firing time. Firing of a transition, though, due to conflicts, can disable some pending scheduled transitions which are then removed from the event list. At each enabling, and also after its own firing, a transition is always scheduled by re-sampling its occurrence time from the associated time interval. Transition firing is regulated by *single-server* policy, meaning that a multiple enabled transition will fire its enablings one at a time sequentially.

The robot system model was simulated for 10^6 time units and simulation data were collected to test “satisfaction” of timing constraints (deadlines). For demonstration purposes, both the *watch system* and the

monitor facility were used. In particular, the *between-watch* (see Fig. 8) was employed for estimating the maximum and minimum temporal distance existing between two causally connected events, one of which acts as *cause* (or source) and the other as *effect* (or destination).

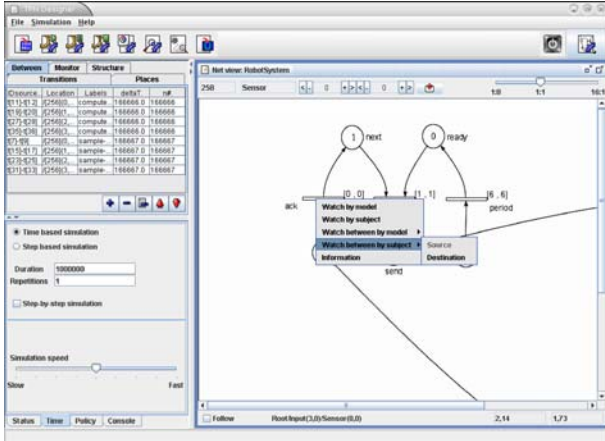


Figure 8. Screenshot of ENGINE showing between-watches in action

A distinct between-watch was associated to the *sample* (cause) and *ack* (effect) pair of events within each *Sensor* page instance, and to the *compute* (cause) and *ack* (effect) pair of events within each *Converter* page instance. The hard deadline of robot was instead estimated by programming a *RobotMonitor* Java class which is dynamically loaded in the context of ENGINE. A monitor class extends the *Monitor* base class, introduces suitable variables for collecting simulation output and redefines the three basic methods: *actionPerformed()*, *calculateStatistics()* and *showResults()*. *actionPerformed* is launched at each transition firing, *calculateStatistics* extracts simulation results at the end of a simulation run and accumulates them during a simulation batch, and *showResults* displays simulation outputs at the end of a simulation batch. Fig. 9 portrays an excerpt of the *actionPerformed* method of the *RobotMonitor* class. *lastFire* variable (initialized to zero) remembers the occurrence time of latest firing of the *receive* transition of the *Robot*. Other details should be self-explanatory.

```

public void actionPerformed( Event e ){
    if( e is Robot receive event ){
        double delay=e.timestamp-lastFire;
        if( delay<minDelay ) minDelay=delay;
        if( delay>maxDelay ) maxDelay=delay;
        lastFire=e.timestamp;
    }
}
//actionPerformed

```

Figure 9. An excerpt of *actionPerformed* method of *RobotMonitor*

After 5 simulation runs of the robot model, it emerged that the between-time in each sensor and in each

converter is bounded to 1 time unit, i.e. when a sensor or a converter is ready to communicate, the partner component is also found ready to engage in the synchronization.

In addition, the inter-receive time window for the robot was estimated to be [6,10], that is two consecutive occurrences of the receive transition of the *Robot* occur in the best case after 6 time units, and in the worst case after 10 time units. As a consequence, from the perspective of simulation, the system “fulfils” all its deadlines.

UPPAAL TRANSLATION

The UPPAAL toolbox (Behrmann *et al.*, 2004)(UPPAAL on-line) allows to verify systems modelled as networks of timed automata (Alur & Dill, 1994) extended with integer variables, structured data types and channel synchronization (*rendezvous*). To permit model checking activities, TPN DESIGNER is able to translate automatically a TimePN model into UPPAAL by using the template automaton shown in Fig. 10 which reproduces semantics of general TimePN transition, according to the UPPAAL 3.5.3 version which supports C syntax and in particular C function declarations and loop constructs. Let T and P denote respectively the set of transitions and the set of places in a model, PRE and $POST$ the (maximal) set of input places and output places of any transition. The translation generates the backward $B_{|T| \times |PRE|}$ and forward $F_{|T| \times |POST|}$ incidence matrices of a TimePN model, its marking vector $M_{|P|}$ and the time interval vector $I_{|T| \times 2}$. An *Info struct* was defined which contains an *index* in M which selects a place, and the *weight* of an arc linking the place to a transition or vice versa. Each element of B or F is an *Info* value. Inhibitor arcs are allowed. An unrestricted time interval like $[a, \infty]$ is represented in I as $[a, -1]$. Functions *enabled()*, *withdraw()* and *deposit()* which respectively check transition enabling, withdraw tokens from input places and generate tokens to output places, are declared locally to the *Transition* template and refer to the particular transition through its ID.

A disabled transition stays in the *Disabled* location. An enabled transition starts firing by resetting its clock x and moving to *Firing* if its time interval is strict, or to *U_Firing* if its latest firing time is ∞ . An enabled transition can complete its firing as soon as its clock goes beyond its earliest firing time. A time strict transition is obliged to complete its firing at its latest firing time, provided it is still enabled, by the invariant $x \leq I[ID][1]$ attached to *Firing*. Transition firing, though, is non deterministic among transitions which can complete their firing at a given time.

Firing completion is an instantaneous and atomic process in two steps which involve the committed locations *Withdraw* and *Deposit* in which time is not

allowed to pass. The transition first removes tokens from its input places, then moves to the **Withdraw** location where it sends (*first step*) a broadcast synchronization signal over the **end_fire** urgent and broadcast channel. This signal forces all the remaining transitions, disabled or under firing, to re-evaluate its enabling status. This is crucial for proper management of conflicts. A no longer enabled transition moves from **Firing/U_Firing** to **Disabled** and resets its clock. The firing process continues by generating tokens in the output places and then by sending (see **Deposit** outgoing edges in Fig. 10) a second broadcast synchronization through **end_fire** (*second step*) which permits detection of newly enabled transitions and non persistent transitions which lose their enabling due to inhibitor arcs.

The **Transition** automaton follows the *single server* firing policy. After its own firing, a still enabled transition is always regarded as a newly enabled one and its clock reset.

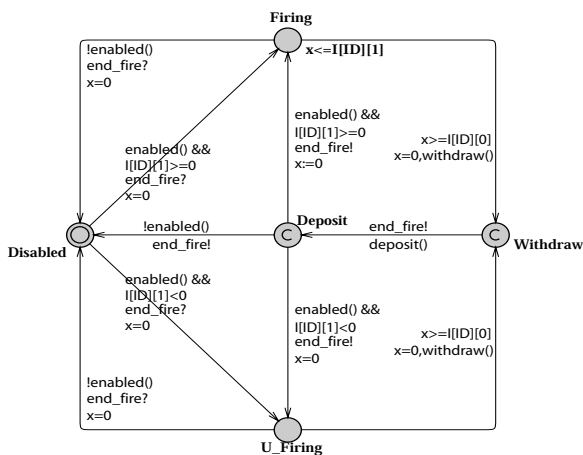


Figure 10: The Transition automaton

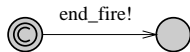


Figure 11. The Starter automaton

A final remark concerns model bootstrapping. A transition starts in its **Disabled** location. An initial broadcast synchronization ensured by the **Starter** automaton shown in Fig. 11 allows transitions enabled in the initial marking of the TimePN model to reach **Firing/U_Firing** location. After that, **Starter** will take no part in the subsequent evolution of the model.

About Translation Correctness

Correctness of the **Transition** template follows intuitively from its construction. A TimePN model translates into a collection of **|T|** instances of the automaton in Fig. 10, i.e. the corresponding UPPAAL model is the product of **|T|** automata corresponding to

the **|T|** transitions of the TimePN model, and the **Starter** automaton.

Translation correctness can be formally proved by showing that the semantics of the achieved UPPAAL model is *timed bisimilar* to the original TimePN model. An example of such a formal proof was described in (Cassez & Roux, 2004) which suggested a translation from TimePNs to timed automata based on a transition automaton very close to the automaton in Fig. 10. Cassez and Roux used a separate supervisor automaton to ensure the two broadcast synchronization steps used during the firing completion process. The equivalence of a TimePN model and its corresponding UPPAAL model enables TCTL model checking activities (Alur *et al.*, 1993)(Clarke *et al.*, 2000)(Behrmann *et al.*, 2004). In particular *safety* and *bounded-liveness* properties verified on the UPPAAL model can directly be interpreted on the original TimePN model. Actually, bounded TimePN models can be efficiently model checked using the translation proposed in this paper. A screenshot of TPN DESIGNER/ENGINE during UPPAAL translation of the Robot system is shown in Fig. 12.

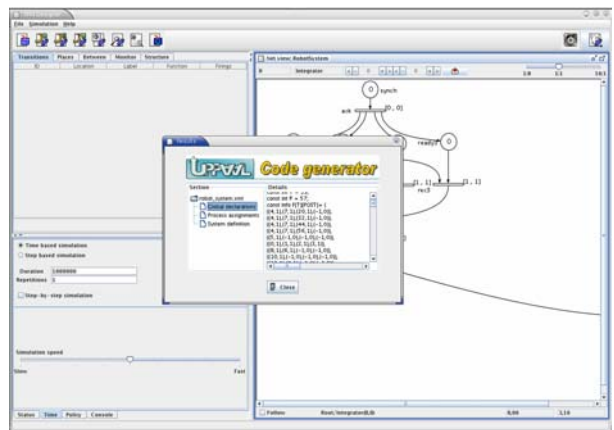


Figure 12. A screenshot of ENGINE during UPPAAL translation of the Robot system

MODEL CHECKING THE ROBOT SYSTEM

The translation process associates by default a distinct clock to each transition of a TimePN model. However, different clocks are effectively required by concurrent or conflicting transitions. The UPPAAL version of the robot model was verified by using 21 clocks: 2 clocks for each **Sensor** instance (see Fig. 3)(one clock is needed by the **period** transition, the other clock is shared by **sample** and **ack** transitions which operates sequentially), 1 clock for each **Converter** instance, 1 clock for each **BChannel** instance, 4 clocks (one for each **rec_j** conflicting transitions) for the **Integrator**, 1 clock for the **Robot**.

A first concern in model checking the achieved UPPAAL model of the robot system, was verification of basic safety property, i.e. that the system is live or free of deadlocks. The property, which cannot be demonstrated

in simulation, was verified by launching the following query to the model checker:

$A[] \text{!deadlock}$

The query asks if in *all* the states of the model state graph there is no deadlock. It is worth noting that a deadlock at the TimePN level, i.e. a marking where no transition is enabled, is identically reflected by a deadlock in the state graph of UPPAAL verifier where all the transition automata stay in the Disabled location (Fig. 10). All of this can be observed by telling the verifier to generate a diagnostic trace which leads to the deadlock situation. There is another cause of deadlock, though, for a translated TimePN model which is related to the model being unbounded, a property which cannot be known in advance and which was shown to be undecidable for TimePNs in the general case. The translation process proposed in this paper uses knowledge arising from preliminary simulation to estimate the boundedness degree k of the model. Accordingly, the translator generates a bounded declaration for the marking vector like this: $\text{int } [0,k] \text{ M}=\{\dots\}$. In latest versions of UPPAAL, when a limited integer variable is in the position of being assigned a value out of its admitted range, the model checker generates again a deadlock. Here too the modeller can understand the specific cause of deadlock by consulting a diagnostic trace.

Temporal properties (bounded liveness) of each sensor instance was separately checked by the queries:

$A[] \text{M}[\text{Input_i_j_Sensor_ready}] \leq 1$

$i=0, \dots, 3, j=0$, which asks if the marking of the *ready* place of each *Sensor* instance is always less than or equal to 1, meaning that never a new period is commenced before the activities of latest period are completed. All these queries were found satisfied.

Verification of converters and robot deadlines was accomplished by *model decoration* (Lindhal *et al.*, 2000). For example, to check a converter deadline, a global clock z was introduced which is reset at each firing of the *compute* transition (see Fig. 4) and analyzed at subsequent firing of the *ack* transition. Model decoration was simply achieved by adapting the *deposit()* function so that when the transition ID corresponds to that of the *compute* transition, clock z is reset. The following queries were separately launched:

$A[] \text{tInit_i_j_Converter_ack.Withdraw imply } z \leq 1$

$i=0, \dots, 3, j=0$. Each query verifies if at each complete firing of *Converter ack* transition, clock z is always found less than or equal to 1 (as suggested by preliminary simulation study). The queries were found

satisfied. Similarly, for the robot deadline the *deposit()* function was adjusted so as to reset clock z each time the *Robot receive* transition completes its firing, and the following query was issued:

$A[] \text{tRobot_receive.Withdraw imply } (z \leq 10 \ \&\& \ z \geq 6)$

This query verifies that at each firing of *Robot receive*, clock z is always in the interval $[6,10]$ proposed by simulation. The query was found satisfied. However, the query was found not satisfied e.g. in the cases $(z \leq 9 \ \&\& \ z \geq 6)$ and $(z \leq 10 \ \&\& \ z \geq 7)$. In other terms, effectively 6 is the upper bound of best case inter-*receive* time of robot, and 10 is the lower bound of worst case inter-*receive* time of robot.

Model checking the robot model demonstrated that the system *is* temporally correct. In addition, verification exactly confirmed the indications provided by the simulation study. Model checking activities were carried out on a Windows XP platform (Pentium IV, 3.4GHz).

ANALYSIS OF OTHER SCENARIOS

The robot system was also analysed according to other configurations, when the hypothesis of allocating components to distinct physical processors no longer holds. Alternative system configurations, though, were found temporally unfeasible except for the case the four converters are split into two pairs with each converter within a same pair which runs to completion before the partner converter can engage in synchronization with its corresponding sensor. In this scenario, the worst case time a sensor waits for a communication with its converter was found (both in simulation and model checking) to be 4 time units, the worst case time a converter waits for a communication with its associated buffered channel remains 1 time unit, and the time window of the inter-*receive* time of the robot changes from $[6,10]$ to $[6,11]$.

CONCLUSIONS

The approach described in this paper is currently being extended in the following directions:

- experimenting with timed Petri nets (Ramchandani, 1974) and the *pre-selection* policy. Under *pre-selection*, conflicts are resolved earlier during the enabling process by a non deterministic choice of the transition to fire. A chosen transition fires in three stages: in the first one (*start-firing*) tokens are immediately withdrawn from input places; in the second step (*firing-in-progress*) tokens are “frozen” and made unavailable to other transitions; in the third step (*end-firing*) new tokens are generated in the output places. TPN DESIGNER already supports the *pre-selection* policy for simulation of TimePN models. Model checking activities rest on a

straightforward adaptation of the Transition template in Fig. 10

- enabling the analysis of very large systems by distributed simulation techniques (Beraldi *et al.*, 2003). A major goal is a systematic exploitation of the temporal uncertainty which comes with a TimePN model through transition time intervals, so as to speed-up the simulator performance (Cicirelli *et al.*, 2005)
- enhancing TPN DESIGNER and the translation into UPPAAL by supporting also Pre-emptive Time Petri Nets (Bucci *et al.*, 2004)(Furfaro *et al.*, 2004) which permit modelling and analysis of real-time tasking sets under application dependent control structures (mutual exclusion on shared data, message passing and so forth) and common practiced priority-based scheduling strategies.

REFERENCES

- Alur R. and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, **126**(2), 1994, 183-235.
- Alur R., C. Courcoubetis & D.L. Dill. Model checking in dense real-time. *Information and Computation*, **104**(1), 1993, pp. 2-34.
- Behrmann G., A. David and K.G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*, M. Bernardo and F. Corradini Eds., *LNCS 3185*, Springer, 2004, pp. 200-236.
- Beraldi R., L. Nigro and A. Orlando. Temporal Uncertainty Time Warp: An implementation based on Java and ActorFoundry. *Simulation-Transactions of the Society for Modelling and Simulation International*, **79**(10), 2003, pp. 581-597.
- Bucci G., A. Fedeli, L. Sassoli and E. Vicario. Timed state space analysis of real-time pre-emptive systems. *IEEE Transactions on Software Engineering*, **30**(2), 2004, pp. 97-111.
- Carullo L., A. Furfaro, L. Nigro and F. Pupo. Modelling and simulation of complex systems using TPN DESIGNER, *Simulation Modelling Practice and Theory*, **11**, 2003, pp. 503-532.
- Cassez F. and O.-H. Roux. From time Petri nets to timed automata. *Fourth International Workshop on Automated Verification of Critical Systems (AVoCS'04)*, Electronic Notes in Theoretical Computer Science, Elsevier, 2004.
- Cicirelli F., A. Furfaro and L. Nigro. Exploiting temporal uncertainty in the distributed simulation of Time Petri Nets. *Proc. of SCS 38th Annual Simulation Symposium*, 4-6 April, San Diego (CA), USA, pp. 233-240.
- Clarke E.M., O. Grumberg and D.A. Peled. *Model checking* (MIT Press, 2000).
- Furfaro A., L. Nigro and F. Pupo. Model checking pre-emptive tasking sets using Time Petri Nets and UPPAAL. *Proc. of 28th IFIP/IFAC Workshop on Real-Time Programming*, 6-8 September, Istanbul (Turkey), 2004.
- Gerber R. and I. Lee. A layered approach to automating the verification of real-time systems. *IEEE Transactions on Software Engineering*, **18**(9), 1992, pp. 768-784.
- Lindhal M., P. Petterson and W. Yi. Formal design and analysis of a gear controller. *Software Tools for Technology Transfer*, **3**(3), 2001, pp. 353-368.
- Marsan M.A., G. Balbo and G. Conte. A class of generalized stochastic Petri nets for the performance evaluation of systems, *ACM Transactions on Computer Systems*, **2**(2), 1984, pp. 93-122.
- Marsan M.A., G. Balbo, G. Chiola and G. Conte. Generalised Stochastic Petri Nets revisited: random switches and priorities, *Proc. of the 2nd Int. Workshop on Petri Nets and Performance Models*, IEEE-CS Press, 1987, pp. 44-53.
- Merlin P. and D.J. Farber. Recoverability of communication protocols. *IEEE Trans. Commun.*, **24**(9), 1976, pp. 1036-1043.
- Murata T. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, **77**(4), 1989, 541-580.
- Ramchandani C. Analysis of asynchronous concurrent systems by timed Petri nets. *Technical Report TR-120*, MIT, February 1974.
- UPPAAL on-line, <http://www.uppaal.com>

AUTHOR BIOGRAPHIES

FRANCO CICIRELLI is a computer science PHD student at University of Calabria (Unical), DEIS, making research on agent and service paradigms for the development of distributed systems, parallel simulation, Petri nets, distributed measurement systems. He holds a student membership with ACM.

ANGELO FURFARO, PHD, is a computer science assistant prof. at Unical, DEIS, teaching object-oriented programming. His research interests include: multi-agent systems, Petri nets, parallel simulation, verification of time-dependent systems, distributed measurement systems. He is a member of ACM.

LIBERO NIGRO is a prof. of computer science at Unical (www.lis.deis.unical.it/~nigro), DEIS, where he teaches object-oriented programming, software engineering and real-time systems courses. His current research interests include: software engineering of time-dependent and distributed systems, real-time systems, Petri nets, modeling and parallel simulation of complex systems, distributed measurement systems. Prof. Nigro is a member of ACM and IEEE.

FRANCESCO PUPO, PHD, is a computer science assistant prof. at Unical, DEIS, teaching introductory programming and computer architecture courses. His research interests include: Petri nets, discrete-event simulation, real-time systems, distributed measurement systems.