

AGENT-BASED MODEL CONTINUITY OF STOCHASTIC TIME PETRI NETS

Franco Cicirelli¹, Libero Nigro², Paolo F. Sciammarella²

¹CNR - National Research Council of Italy

Institute for High Performance Computing and Networking (ICAR) - 87036 Rende(CS) - Italy

²Software Engineering Laboratory

University of Calabria, DIMES - 87036 Rende (CS) – Italy

Email: f.cicirelli@dimes.unical.it, l.nigro@unical.it, p.sciammarella@dimes.unical.it

KEYWORDS

Multi-agent systems, model continuity, simulation, real-time, stochastic time Petri nets, Java, JADE.

ABSTRACT

Stochastic Time Petri Nets (sTPN) are a useful formalism for modelling and quantitative analysis of concurrent systems with timing constraints. This paper describes an implemented tool supporting sTPN, which was achieved on top of a control-centric agent-based framework which fosters model continuity. Model continuity means the same model can be used for property checking through simulation and for real-time execution. The paper demonstrates the effectiveness of the approach through a modelling example.

INTRODUCTION

Stochastic systems can be studied by either numerical or statistical solution techniques (Younes et al., 2006). Numerical methods enumerate the stochastic states of a model and can evaluate a probability measure over a path of state transitions by solving equations based on the state associated probability distribution functions. Numerical methods tend to be more accurate than statistical methods which are based on sampling and simulation. However, numerical methods can suffer of state explosion problems and can impose restrictions on the classes of modelled systems, e.g., based on timers which satisfy the Markov property or which admit regeneration points in more general systems. Stochastic Time Petri Nets (sTPN) (Paolieri et al, 2016) have been proposed for modelling and analysis of concurrent systems with timing constraints. They are supported by numerical techniques in the context of the ORIS tool (Bucci et al., 2010). An approach to statistical model checking of sTPN based on UPPAAL is described in (Cicirelli et al., 2015). This paper proposes an original agent-based tool supporting sTPN. Novel in the tool is a support to *model continuity* (Cicirelli&Nigro, 2016a-b) that is the possibility of using a same model for temporal analysis by simulation and for real-time execution. The paper first describes the definitions of sTPN. Then a summary of the underlying control-centric agent-based architecture is furnished. After that an overview of the tool implementation is provided. The developed approach is then demonstrated by a case study concerning a probabilistic formulation of

the Fisher's mutual exclusion algorithm (Lynch&Shavit, 1992)(Paolieri et al., 2016). Finally, conclusions are presented with an indication of on-going and future work.

STOCHASTIC TIME PETRI NETS

Syntax

An sTPN is a tuple

$$(P, T, B, F, M_0, I_{nh}, E, Uw, Ud, EFT^s, LFT^s, PDF, W)$$

where:

- P and T are disjoint finite nonempty set of places and transitions; $T = T_i \cup T_t$ where T_i are immediate transitions, and T_t are timed transitions;
- B is the backward incidence function, $B: P \times T \rightarrow \mathbb{N}$, where \mathbb{N} denotes the set of natural numbers;
- F is the forward incidence function, $F: P \times T \rightarrow \mathbb{N}$;
- M_0 is the initial marking function, $M_0: P \rightarrow \mathbb{N}$, which associates with each place a number of tokens;
- I_{nh} is the set of inhibitor arcs, $I_{nh} \subset P \times T$ where $(p, t) \in I_{nh} \Rightarrow B(p, t) = 0$;
- $E: T \rightarrow \{true, false\}$ is a boolean function which extends the enabling condition of a transition. If omitted, it defaults to *true*;
- Uw and Ud are two update functions which extend respectively the withdraw/deposit phase of a transition. If omitted, they default to *void*;
- $EFT^s: T_t \rightarrow R^+$ is a function which associates each timed transition with a (finite) earliest static firing time. R^+ denotes the set of non-negative real numbers;
- $LFT^s: T_t \rightarrow R^+ \cup \{\infty\}$ is a function which associates each timed transition with a (possibly infinite) latest static firing time. It must be $LFT^s \geq EFT^s$. An immediate transition logically has $EFT^s = LFT^s = 0$;
- PDF is a function which associates each timed transition with a probability distribution function constrained in the interval $[EFT^s, LFT^s]$;
- W is a function, $W: T_i \rightarrow R^+$, which associates each immediate transition with a weight.

Semantics

A transition t is *enabled* if each of its input places contains sufficient tokens and $E(t)$ evaluates to *true*, i.e., iff

$$\forall p \in P, (p, t) \in I_{nh} \Rightarrow M(p) = 0 \wedge B(p, t) > 0 \Rightarrow M(p) \geq B(p, t) \wedge E(t)$$

An enabled immediate transition t_i is *fireable*. Fireability of immediate transitions always has priority over that of timed transitions. Among the set of fireable immediate transitions, each t_i can *fire* with probability

$$Prob(t_i) = \frac{W(t_i)}{\sum_{t_j \in T_i \text{ and } t_j \text{ is enabled}} W(t_j)}$$

The *time-to-fire* $\tau(t_t)$ of a timed transition t_t is stochastically defined, at its enabling instant, by sampling its associated *PDF*(t_t) with the constraint:

$$EFT^s(t_t) \leq \tau(t_t) \leq LFT^s(t_t)$$

A timed transition is fireable at its absolute time-to-fire, i.e., *enabling time*(t_t) + $\tau(t_t)$, provided it is less than or equal to the absolute time-to-fire of all the other simultaneously enabled timed transitions. Timed transitions with the same absolute time-to-fire will fire non deterministically.

Let $m: P \rightarrow N$ be the net marking, which specifies the number of tokens of each place of the sTPN model at a certain instant of time. When the transition t fires, the marking m is replaced by a new marking m' which is derived from m by the withdrawal of tokens from the input places and the deposit of tokens in the output places. More precisely, the firing process consists of the two (atomic) phases:

$$m_{int}(p) = m(p) - B(p, t) - U_w(t) \text{ (withdraw phase)}$$

$$m'(p) = m_{int}(p) + F(p, t) + U_d(t) \text{ (deposit phase)}$$

Transitions which are enabled in m , in the intermediate marking m_{int} and in the final marking m' are said *persistent* to the t firing. Transitions which are enabled in m' but not in m_{int} are said *newly enabled*. Newly enabled timed transitions have their time-to-fire which is resampled.

A transition which is multiple enabled at a time instant is assumed to fire its enablings one at a time (*single server* semantics). Therefore, following its own firing, would t be still enabled, it is regarded as newly enabled.

As a final remark, it should be noted that the functions E , U_w and U_d are model-specific and can be exploited, e.g., for managing a high-level concept like a variable (see Fig. 7) or to avoid cluttering in complex topologies.

CONTROL SENSITIVE AGENT FRAMEWORK

The following highlights the control-based framework (Cicirelli&Nigro, 2016a-b) for building multi-agent systems which is at the basis of the sTPN tool described later in this paper.

The framework is founded on the notions of *actors* (agents) and *actions* (see Fig. 1)

Actors

Actors are modelled as finite state machines which communicate to one another by asynchronous message

passing. Actors are thread-less. They are at rest until a message arrives to be processed. The behavior of an actor (i.e., its state machine) is modelled in its associated *handler()* method. An incoming message causes local variables of the actor to be updated, possibly changes the current state of the state machine, can send new messages to known actors and can submit one or more actions.

A subsystem of actors (Logical Process or LP) is allocated for the execution on a computing node. All the actors of a same subsystem are regulated by a local *control machine* which transparently buffers exchanged messages into one or more message queues and ultimately delivery messages, one at a time, to recipient actors, according to a proper *control strategy*, e.g., based on a time notion (simulated or real-time). Message processing in a actor subsystem represents the *unit of scheduling*.

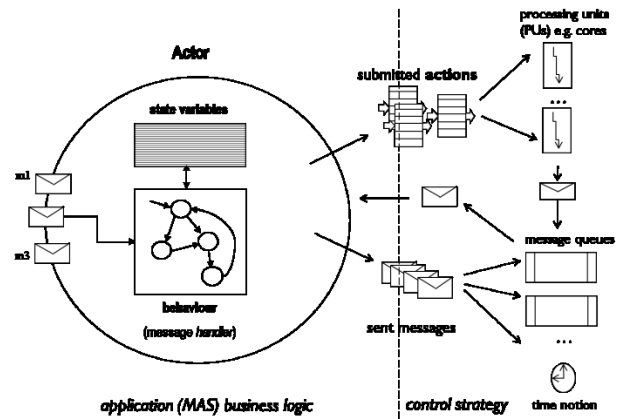


Figure 1 - Actor organization and orthogonal control aspects

In general, multiple actor subsystems can be federated to constitute a distributed system (see Fig. 2), using the services of a suitable transport layer and communication protocol. A *Time Server* can be in charge of maintaining a global time notion across the federated system.

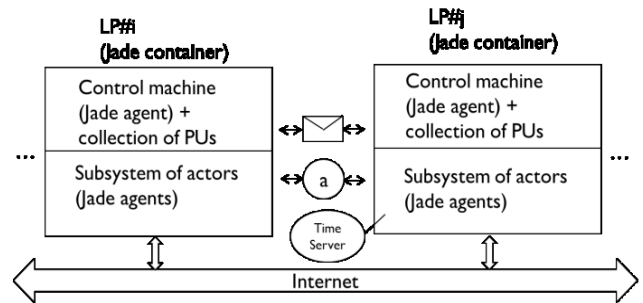


Figure 2 - Federated actor system based on JADE

Fig. 2 portrays a snapshot of a distributed actor system achieved on top of the open source JADE project (Bellifemine et al., 2007)(Cicirelli&Nigro, 2016a-b) which provides basic services for agent lifecycle, naming and message exchanges based on FIPA (Foundation for Intelligent Physical Agents). In addition it favours interoperability with legacy software FIPA compliant. Both actors and messages can be dynamically transferred from an LP (JADE container) to another.

A fundamental design issue of the actor-framework is related to the control machines which act as plug-ins tailored to the application needs.

Actions

Messages promote sociality among actors and capture the occurrence of events. They are handled sequentially in an interleaved way by the local control structure.

Besides messages, the actor framework relies also on *actions*, that are activities which consume time and require *processing units* (PUs) for them to be executed. Actions express computational needs associated to messages and can require the use of resources belonging to the external environment. Actions are executed in parallel, depending on the availability of PUs. In general an action, after its submission by an actor, can run to completion or it can be suspended/resumed or aborted.

An action is a black box with a list of input parameters and a list of output parameters. Actions have no visibility to the internal data variables of the submitter actor. As a consequence, no mutual exclusion mechanism is required and no interference can occur from the action parallel execution schema. When an action terminates, it can inform the submitter by an action completion message. The submitter can then access the output parameter list to get any result computed by the action.

Actions can be reified in different ways. Simulated actions consist of pure time consuming activities whose aim is to advance the simulated time. Real or effective actions have a concrete instruction body (algorithm) whose execution advances the real time. Pseudo real actions increases the real time but have no concrete algorithm to execute. They can be useful for *preliminary real-time execution* of a given model (see later in this paper) which is a key to check how the overhead introduced by message exchanges and message processing affect the system timing constraints.

As a further refinement, action execution can be atomic or it can be preempted. In addition, an action can express an imprecise computation which after a time threshold delivers a first result whose accuracy can be improved would more time be available, or it can be returned and the action execution interrupted.

The various notions of actions are handled by the corresponding *action schedulers* provided by the control machine. An action scheduler manages local processing units and stores actions which find no available PU in pending action queues, waiting for some specific or unspecific PU to be ready to accept a new action execution. A PU can be a physical core or it can be realized by a Java thread, or it can be a fake object in the case of simulated actions. The use of preemptive actions/PUs were used in (Cicarelli&Nigro,2016a) to enable schedulability analysis of real-time systems.

A key factor of the actor control framework is *model continuity*, that is transitioning a same model from property analysis to real time execution. Model continuity mainly depends on actions. Moving from simulation to real execution requires changing the control machine, the time notion and the nature of actions which are switched

from simulated actions to real actions and associated action schedulers. All the remaining part of the model, and particularly actor behaviors and message passing, remains exactly the same during the transition.

Control framework in Java/JADE

Fig. 3 recapitulates some of the fundamental classes of the control framework. Actors and control machines are mapped on JADE agents. Each control machine owns an action scheduler which administers a set of processing units. A control machine receives the submitted actions and forwards them to the action scheduler. Actions and messages are embodied as serialized objects within *ACLMessages* when exchanged between actors and control machines.

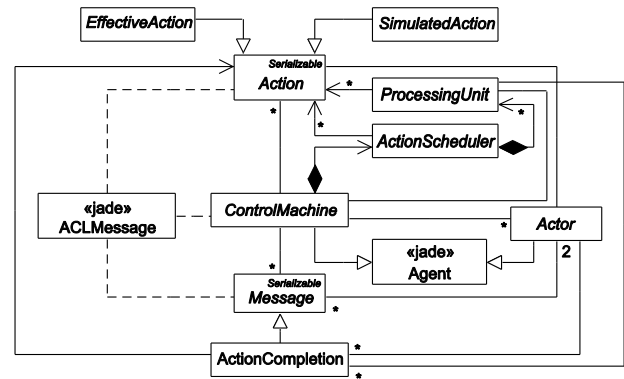


Figure 3 - Framework basic classes

The prototyped control machines are partitioned into three groups (see Fig. 4): (i) the untimed control machines, (ii) the time-aware control machines which operate in a sequential setting and (iii) the time-aware control machines which operate in a distributed context. A time server is required by the latter group in order to ensure a coherent time evolution among all the participating control machines (more details in (Cicarelli&Nigro, 2016a-b)).

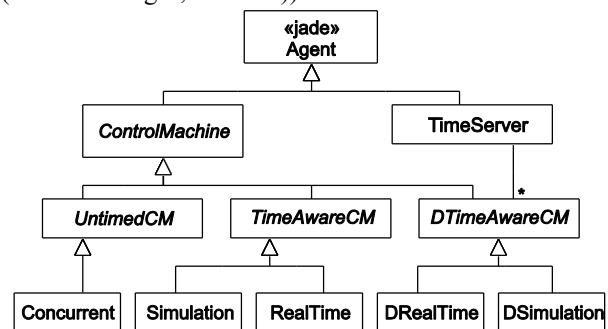


Figure 4 - Class hierarchy of control machines

Simulation, along with its parallel/distributed counterpart *DSimulation*, implements a classical discrete-event simulation schema. Messages are tagged with an absolute timestamp and are buffered into a time ranked queue where the head message holds the (or is one message with) minimum timestamp. The control machine can work with simulated actions. A simulated action carries the time duration of the associated activity. At its

submission, a simulated action is assigned to an exploitable PU which in this case simply means that an action completion message is scheduled with timestamp $now + duration$.

RealTime is a time-sensitive control machine using a real time notion built on top of the *System.currentTimeMillis()* *System* Java service. Only effective actions can be used. Messages have a timestamp and must be dispatched as soon as the current time exceeds their firing time. *RealTime* uses a configurable time tolerance *EPS*, so that a time-constrained message which should occur at absolute time t , is considered to be still in time if the current time is less than or equal to $t + EPS$. *RealTime* is useful for non-hard real-time applications. The *DRealTime* control machine replaces *RealTime* in the parallel/distributed context.

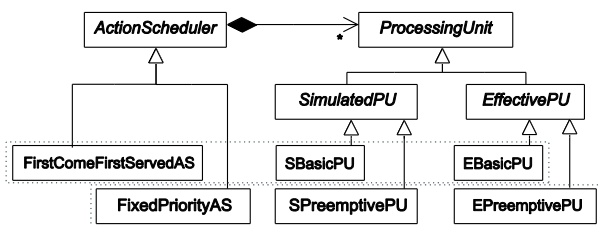


Figure 5 - Class hierarchies of action schedulers and PUs

As shown in Fig. 5, corresponding classes exist for action schedulers and PUs, which work together. Schedulers immediately put into execution a submitted action on an idle exploitable PU (if there are any), otherwise, different scheduling strategies can be adopted. In the case no such idle PUs exist, the scheduler *FirstComeFirstServedAS* organizes actions in a pending list. Actions will be executed according to their arrival time. The *FixedPriorityAS* uses instead an action priority to keep ordered the pending list. In this case, action execution is priority driven and preemptive.

For simulation purposes, the use of classes which are heirs of *SimulatedPU* is required. They are passive objects without inner threads. During real-time execution, heirs of *EffectivePU* should instead be used. They are thread-based objects able to execute effective actions (Cicirelli&Nigro, 2016a-b).

AN AGENT-BASED STPN TOOL

A class diagram of an *sTPN* tool built on top of the actor-based control framework is summarized in Fig. 6.

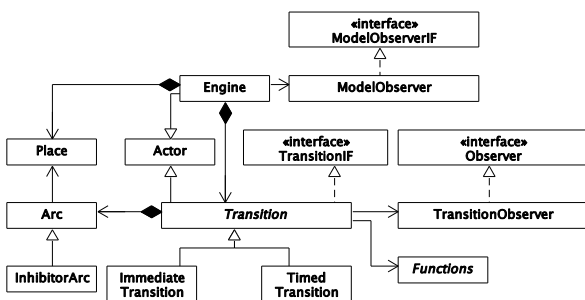


Figure 6 - Main classes in the *sTPN* tool

A model is feed through an XML file which is parsed into an internal representation consisting of a multi-agent system (MAS). Agents, i.e., actors, are associated with transitions which interact with the *Engine* (another agent) through messages and method calls. Each transition refers its input/output arcs which are linked to their input/output places. Arcs and places are realized as POJOs and provide their services by method invocations.

For analysis purposes, an *sTPN* model is simulated by associating to the generated MAS (single LP) the *Simulation* control machine (see Fig. 4) along with basic simulated actions paired with a first come first served scheduler (Fig. 5). Evolution of the MAS is triggered by transition firings and is controlled by the *Engine* which repeats a basic loop of simulation steps. At each step, the candidate set of enabled transitions is recomputed. The *Engine* owns the collection of transitions and the collection of places (marking) of the model. Transition enabling is checked by a method call on the transition agents.

A critical issue concerns the atomic firing process of a transition. Towards this the *Engine* separately executes the withdraw and the deposit phases of a transition firing. Each phase is immediately followed by a *retract* of disabled transitions which are removed from the candidate set and their firing messages invalidated in the simulation calendar. The two phases are required for correctly handling effective conflicts among transitions. An enabled transition can loss its enabling status in the intermediate marking following the withdraw phase or in the final marking reached after the deposit phase. Purposely, withdraw and deposit operations are realized by method calls (defined in the interface *TransitionIF* in Fig. 6) on the transition actor (recall that the computational status of an actor system is frozen between two consecutive message dispatches; therefore, method invocations, also with side-effects, are compliant with the actor lifecycle).

Immediate transitions, ranked according to their weights, are fired one at a time directly by the *Engine*. Timed transitions are instead fired by messages and actions. In particular, all the enabled timed transitions at a simulation step receive a *StartFiring* message from the *Engine* whose processing implies the next sample of the associated probability distribution function is obtained. The sample is passed to a submitted action which simulates the transition firing by scheduling the message completion message at the absolute time of $now + sample$. When the transition receives the *ActionExecutedMessage*, it informs the *Engine* about commitment of the transition firing through an *EndFiring* message. The *Engine* then executes the two phases *withdraw + retract*, *deposit + retract* on the transition. After that, the next step (iteration) of the *Engine* is started.

For property checking, a model evolution (see Fig. 6) is watched by suitable observers which collect statistical information about transitions or the entire model.

Since the *E*, *Uw* and *Ud* functions are model specific, the adopted solution consists in specifying, in the model

XML, the name of a Java class which provides an implementation of the above functions as methods. Such a class, subtype of the *Functions* abstract class (see Fig. 6), is then dynamically loaded, instantiated and exploited by transitions.

For preliminary execution of an *sTPN* model, the corresponding MAS is plugged with the *RealTime* control machine, and works with pure time consuming effective actions and the *FirstComeFirstServedAS* scheduler. All of this ensures the “effective actions” behave as in simulation but now advance the real-time.

A CASE STUDY USING MODEL CONTINUITY

The Fisher’s mutual exclusion protocol for N processes, having identifiers $1, 2, \dots, N$, competing for the access to some shared resource, was used as a case study for validating the obtained *sTPN* tool. The protocol is an example of a time-dependent mutual exclusion algorithm (Lynch&Shavit, 1992). Although the algorithm has been analyzed qualitatively by model checking, e.g., in the context of the UPPAAL toolbox (Behrmann et al., 2004), here it is used for quantitative evaluation using simulation and the results compared with those described in (Paolieri et al., 2016) which are based on probabilistic model checking and numerical approach.

The protocol assumes that basic read/write memory operations are atomic. A single global communication variable id is used, which stores the identifier of the process trying to enter its critical section, or it defaults to 0. Every process can try the protocol when $id = 0$. In this case the process executes the time-consuming operation $id \leftarrow i$. Since more processes can attempt the same operation simultaneously, it is required for a trying process to wait for a time (say it W^+) greater than the writing time W . After W^+ time units are elapsed, the process reads again id . In the case $id \neq i$ the process has to retract and to wait for the id to become again 0. If instead $id = i$ then the process can enter its critical section. At the exit from the critical section, the process sets id to 0.

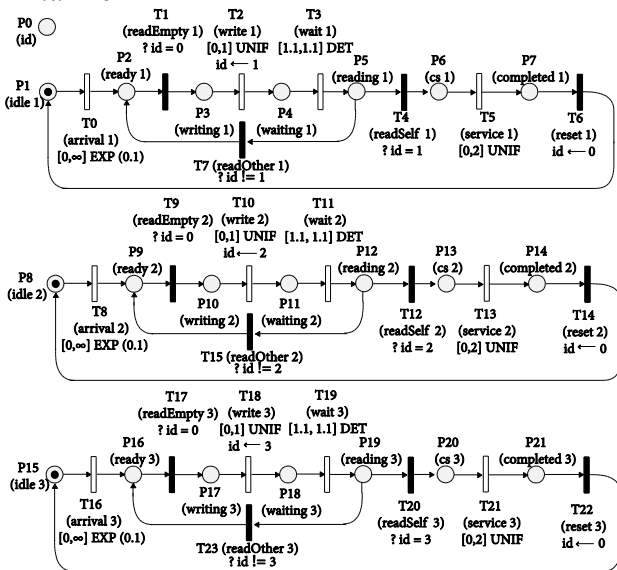


Figure 7 – A model of Fisher’s protocol (Paolieri et al., 2016)

The model in Fig. 7 with $N = 3$ processes, derived from (Paolieri et al., 2016), was used for the experiments. Immediate transitions are shown as black bars, whereas timed transition are depicted as white bars. The non critical section of each process is modelled by an *arrival* timed transition with interval $[0, \infty]$ and with an exponential pdf (*EXP*) with rate 0.1. W is supposed to be uniformly distributed (*UNIF*) within the interval $[0, 1]$, and W^+ is set to 1.1 time units. Other details of the model should be self-explanatory.

Property analysis

The Fisher’s *sTPN* model was studied in (Paolieri et al., 2016) using a probabilistic temporal logic built around an *interval until operator*:

$$\varphi U^{[\alpha, \beta]} \psi$$

which captures the event that a marking of the *sTPN* model is reached which satisfies a predicate ψ at some time in the interval $[\alpha, \beta]$ without violating a safety predicate φ . Of such event can be of interest finding the occurrence probability P , or bounding such probability against a given threshold value: $P \sim_p$ where $\sim \in \{<, >\}$. Predicate states are based on net markings. As usual, predicates and atomic propositions can be combined with boolean operators to form more complex formulas, but nesting of interval until operators is not allowed.

The interval until operator naturally can be used to assess transient behavior of a net model.

The following properties were studied using the Fisher’s protocol model upon the developed *sTPN* tool: (a) mutual exclusion (*safety*), i.e., no more than one process can enter its critical section at a time; (b) absence of starvation (*bounded liveness*), that is a trying process eventually enters its critical section. The latter property relates to estimating the overtaking factor, i.e., the maximum number of by-passes of other processes with respect to a waiting process, or equivalently to estimating the maximum waiting time of a trying process before entering its critical section; (c) some other examples of specific bounded liveness properties. In each case a proper decoration of the model observer was used. In the following, for simplicity, the notation, e.g., cs_1 is used instead of $m[cs_1]$.

Mutual exclusion

Mutual exclusion was checked by performing some simulation runs with $t_{End} = 3.5 \times 10^5$ time units, and by observing that the event

$$true U^{[0, t_{End}]} (cs_1 = 1 + cs_2 = 1 + cs_3 = 1 > 1)$$

has a 0 probability of occurrence. The model observer object was decorated to watch marking of cs_1 , cs_2 and cs_3 places. In no case it was found more than one process is in its critical section. As part of this assessment it was also checked that effectively it can happen that csi for any i assumes the value 1.

Absence of starvation

It was estimated the probability that a process can be affected by a certain number of by-passes (overtaking) from other competing processes. As one can see from Fig. 8, each process seems to suffer for no more than 6 by-passes. 5 simulations with $t_{End} = 3.5 \times 10^5$ time units were used to collect data behind Fig. 8 and Fig. 9.

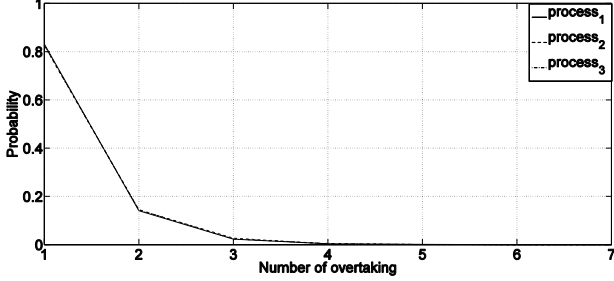


Figure 8 - Occurrence probability vs. number of by-passes

Another way to check the starvation-free behavior was estimating the worst case waiting time of a trying process. Results are shown in Fig. 9.

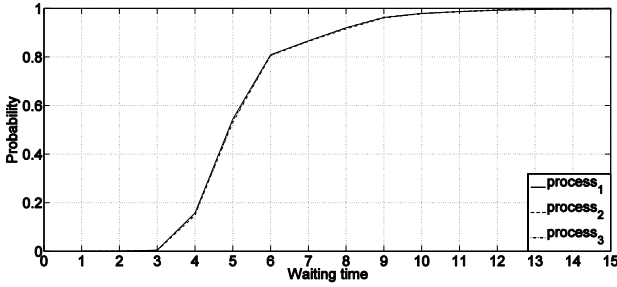


Figure 9 - Trying process waiting time

Examples of bounded liveness properties

As an example of a particular bounded liveness property it was measured the occurrence probability of the following event:

$$true U^{[0,\beta]}(cs1 = 1) (*)$$

for various values of β , starting separately from each of the following markings which describe possible execution states of the three processes:

$$\begin{aligned} m_A &\equiv ready_1 idle_2 idle_3 \\ m_B &\equiv id = 3 ready_1 idle_2 waiting_3 \\ m_C &\equiv id = 3 ready_1 waiting_2 waiting_3 \end{aligned}$$

The property addresses specifically a deadline requirement upon the delay $process_1$ experiments before entering its critical section.

A batch of simulation runs were carried out, terminating each of them as soon as the watched event occurs (that is the given number of by-passes happens). The proportion of the runs which satisfy the event divided by the total number of runs was then evaluated.

The number of required runs was empirically determined by watching the probability value which

almost stabilizes. 100 runs were used for building each curve in Fig. 10.

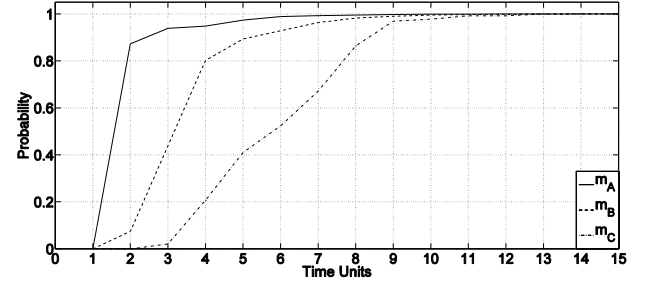


Figure 10 - Occurrence probability of the event (*) vs. time

For example, in the m_A scenario, as expected, the delay of $process_1$ can be short (best cases) with a probability of about 90% but in the worst case (probability 1) it amounts to known maximum waiting time (Fig. 10).

Another particular property concerned an evaluation of the occurrence probability of the following event:

$$!cs_1 U^{[0,\beta]}(completed_2 \vee completed_3) \wedge (ready_2 \vee ready_3) (**)$$

for various values of β , starting from the marking

$$ready_1 idle_2 idle_3$$

and separately for three different service time distributions: $UNIF[0,2]$, $UNIF[0,4]$, $UNIF[2,4]$. The event amounts to asking the following check: in the hypothesis that $process_1$ is not in its critical section, what is the worst case time (β) for each of the remaining processes so as to be ready to try or be capable of having completed an access to shared data? Respectively 235, 195 and 220 runs were used for generating the three curves in Fig. 11.

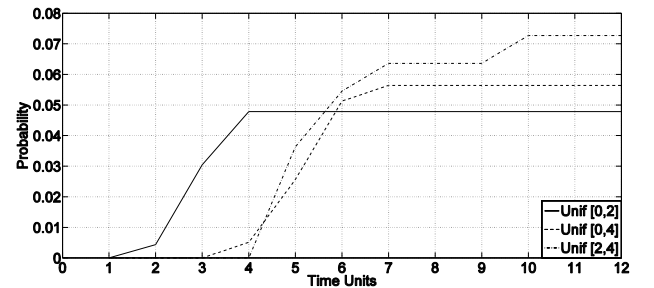


Figure 11 - Occurrence probability of event (**) vs. time

It is worth noting that the results portrayed in Fig. 10 and Fig. 11 are very close to the results reported in (Paolieri et al., 2016) for the same checked events.

Preliminary real-time execution

After property analysis, the sTPN Fisher's protocol was re-checked by executing it in real-time but with pure time consuming actions instead of effective instructions of a concrete programmed version of the process bodies.

Such *preliminary execution* is very important in the practical case for observing the overhead introduced by

scheduling and message exchanges on the fulfillment of model timing constrains. Configuring the model for preliminary execution only required: (a) interpreting the time unit as 1 sec; (b) changing the control machine from *Simulation* to *RealTime* and (c) using pure time consuming effective actions with the *FirstComeFirstServedAS* scheduler. No changes were introduced in the model. The Fisher's protocol was then executed with a time tolerance of $EPS = 200\text{ ms}$.

Basic properties of the mutual exclusion algorithm were watched during the execution and the time deviations, i.e., the latency with which messages and actions are actually executed with respect to their due time, measured.

Worst case results of 4 runs each of 7 hours of wall clock time are collected in the histogram of Fig. 11. As one can see, in almost all the cases, the time deviation is virtually 0 ms. The most frequent non zero deviation is 16 ms. The worst case deviation was found to be 155 ms which occurred just once at the execution start, i.e., at model bootstrapping.

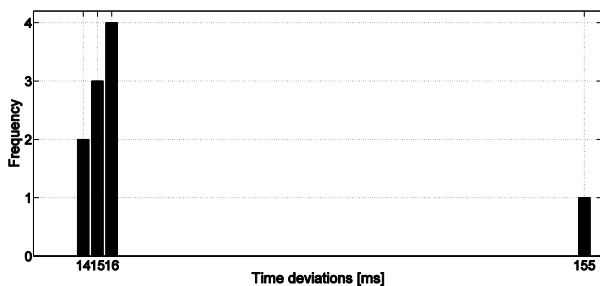


Figure 11 - Observed time deviations

During the whole real-time experiment, no more than one process was found in its critical section. In addition, in Fig. 12 is portrayed an histogram of registered overtaking factor and its occurrence probability of trying processes.

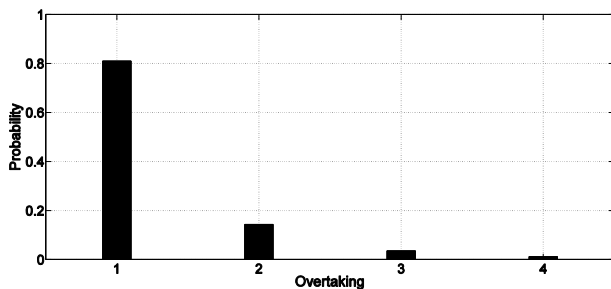


Figure 12 - Observed overtaking

All the experiments were carried out on a Win 7 workstation, 12GB, Intel Core i7, 3.50GHz, 4 cores, without an active Internet connection.

CONCLUSIONS

Although potentially less accurate of the probabilistic model checking approach based on numerical solutions proposed in (Paolieri et al., 2016), the agent-based simulation tool for Stochastic Time Petri Nets (sTPN) described in this paper proves effective in the practical case, as demonstrated by the reported case study.

A key factor of the approach is *model continuity*, i.e. the same model can be used for property checking by simulation and for real-time execution. The tool features derive from the adopted underlying agent-based control-centric framework (Cicirelli&Nigro, 2016a-b).

The proposed approach provides the abstraction mechanisms and the execution concerns suited, e.g., for modelling, analysis and execution of time-constrained workflow systems. The real-time preliminary execution, in particular, directly corresponds to workflow enactment.

Prosecution of the research work is geared at:

- optimizing the implementation of the sTPN tool;
- applying the approach to modelling, analysis and enactment of time-constrained workflow systems (Gonzales del Foyo&Silva, 2008);
- supporting a probabilistic temporal logic (Younes et al., 2006)(Paolieri et al., 2016)(David et al., 2015) for the expression of quantitative properties to check on an sTPN model, and automating the determination of the required simulation runs;
- extending the tool toward parallel/distributed simulation of large models.

REFERENCES

- Behrmann, G., A. David, K.G. Larsen (2004). A tutorial on UPPAAL. In: *Formal Methods for the Design of Real-Time Systems*, M. Bernardo and F. Corradini Eds., Lecture Notes in Computer Science, Vol. 3185, Springer-Verlag, pp. 200-236.
- Bellifemine, F., G. Caire, D. Greenwood (2007). *Developing multi-agent systems with JADE*. John Wiley & Sons.
- Bucci, G., L. Carnevali, L. Ridi, E. Vicario (2010). ORIS: a tool for modeling, verification and evaluation of real-time systems. *Int. J. on Software Tools for Technology Transfer*, Springer, vol. 12, pp. 391-403.
- Cicirelli, F., C. Nigro, L. Nigro (2015). Qualitative and quantitative evaluation of stochastic time Petri nets. Proc. of *2nd Int. Workshop on Cyber-Physical Systems (IWCPs'15)*, Lodz, Poland, pp. 775-784.
- Cicirelli, F., L. Nigro (2016a). Control aspects in multi-agent systems. In *Intelligent Agents in Data Intensive Computing*, Springer, Studies in Big Data, Kolodziej J., Correia L., Manuel Molina J. (Eds.), pp. 27-50.
- Cicirelli, F., L. Nigro (2016b). Control centric framework for model continuity in time-dependent multi-agent systems. *Concurrency and Computation: Practice and Experience*, Wiley, to appear.
- David, A., K.G. Larsen, A. Legay, M. Mikucionis, D.B. Poulsen (2015). UPPAAL SMS Tutorial, *Int. J. on Software Tools for Technology Transfer*, Springer, 17:1-19, 06.01.2015, DOI 10.1007/s10009-014-0361-y, 2015
- Gonzalez del Foyo, P.M., J.R. Silva (2008). Using Time Petri Nets for modelling and verification of timed constrained workflow systems. In *ABCM Symposium Series in Mechatronics*, vol. 3, pp.471-478.
- Lynch, N., N. Shavit (1992). Timing-based mutual exclusion. In *IEEE Real-time Systems Symp.*, pp. 2-11.
- Paolieri, M., A. Horváth, E. Vicario (2016). Probabilistic model checking of regenerative concurrent systems. *IEEE Trans. Soft. Eng.*, to appear (available on-line).
- Younes, H.L.S., M. Kwiatkowska, G. Normaln, D. Parker (2006). Numerical vs. statistical probabilistic model checking. *Int. J. on Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 216-228.