STATISTICAL MODEL CHECKING OF MULTI-AGENT SYSTEMS

Libero Nigro, Paolo F. Sciammarella

Software Engineering Laboratory
University of Calabria, DIMES - 87036 Rende (CS) – Italy
Email: l.nigro@unical.it, p.sciammarella@dimes.unical.it

KEYWORDS
Statistical Model Checking, Multi-agent systems, Actors, UPPAAL SMC, Iterated Prisoner's Dilemma.

ABSTRACT

This paper proposes an original approach to modelling and simulation of multi-agent systems which is based on statistical model checking (SMC). The approach is prototyped in the context of the popular UPPAAL SMC toolbox. Usefulness and validation of the approach are checked by applying it to a known complex and adaptive model of the Iterated Prisoner's Dilemma (IPD) game, by studying the emergence of cooperation in the presence of different social interaction structures.

INTRODUCTION

In the last years multi-agent systems (MAS) have proved to be a fundamental paradigm for modelling and simulation (M&S) of complex and adaptive systems (North & Macal, 2007)(Cicirelli & Nigro, 2016b). Power and flexibility of MAS stem from the ability of modeling individual behaviour of agents and their social interactions, and then to observe the emergence of properties at the population level.

In this work a minimal actor computation model (Cicirelli & Nigro, 2006a) is adopted for supporting MAS. A unique feature of this model is a light-weight notion of actors which (i) are threadless agents, (ii) hide an internal data status, and (iii) communicate to one another by asynchronous message passing. Message exchanges are ultimately regulated by a customizable *control structure* which can reason on time (simulated or real-time). The model can be effectively hosted by widespread languages like Java.

Novel in this paper is a mapping of the actor model onto UPPAAL SMC (David *et al*., 2015) so as to exploit statistical model checking techniques (Younes *et al*, 2006)(Larsen & Legay, 2016). SMC automatizes multiple executions, estimates a required number of simulation runs, uses statistical properties (e.g., Monte Carlo-like simulations and sequential hypothesis testing) to infer system properties from the observables of the various runs.

To the best of authors knowledge, no similar support in UPPAAL SMC of MAS with asynchronous messages was previously published. Originality is mainly tied to an exploitation of dynamic message template processes which are not supported by other tools.

The resultant approach is practically demonstrated through a case study concerned with a complex, adaptive and scalable model based on the Iterated Prisoner's Dilemma (IPD) game (Axelrod *et al*., 2002). The model is very challenging and it aims to study the emergence of cooperation among competitive agents when varying, e.g., the social interaction network. SMC results confirm previous indications of Axelrod et al. work, although with a smaller cooperation degree.

The paper is structured as follows. First, the actor computational model is summarized. Then major features of UPPAAL SMC are recapitulated. After that, a structural mapping of actors onto UPPAAL SMC is proposed. The paper goes on by detailing the developed IPD case study and the achieved experimental results. Conclusions are finally presented by stating on-going and future work.

ACTOR COMPUTATIONAL MODEL

Actors (Agha, 1986) with asynchronous message passing are a reference model for concurrent and distributed systems. Many variants of this model, though, motivated by specific uses and application contexts, are developed in the literature for both theory and/or practice reasons.

In this work a light-weight (threadless) and control-based version of the actor model is adopted (Cicirelli & Nigro, 2013)(Cicirelli & Nigro, 2016a), addressing specifically high-performance time-dependent applications. A system is a federation of theatres. A theatre (Logical Process or LP), allocated for execution on a computing node, hosts a collection of actors plus a control machine (CM) and interface to a transport layer. The goal of CM is to transparently manage the cloud of exchanged messages and to deliver them, one at time and in an interleaved way, according to a control structure. Message processing is atomic (*macro-step* semantics). As a consequence, a cooperative concurrency schema among the agents of a theatre is ensured. The CMs of a system synchronize one to another in order to ensure a coherent global time notion. Details about control structure design and global synchronization algorithms based on a time server are described in (Cicirelli *et al*., 2016a). In the following the focus will be on a single theatre and on the assumed actor programming style, e.g., in Java.

Actors are objects of classes which inherit from the abstract Actor base class which exposes, among others, the basic non blocking message *send* operation. An actor owns some local data variables and acquaintances (known actors) to whom messages can be sent (including itself for proactivity).

To simplify modelling/programming, message handling is split among separate methods which can have

Proceedings 31st European Conference on Modelling and
Simulation ©ECMS Zita Zoltay Paprika, Péter Horák, Kata Váradi,
Péter Tamás Zwierczyk, Ágnes Vidovics-Dancs, János Péter Rádics (Editors)
ISBN: 978-0-9932440-4-9/ ISBN: 978-0-9932440-5-6 (CD)

parameters. Each method represents a "message server" (*msgsrv*) which receives and handles the specific associated message. When a msgsrv gets actually invoked it depends ultimately on a decision of the control machine. No mailbox is provided per actor, rather all the sent messages get buffered in a cloud managed by the control machine.

Figure 1 clarifies the actor programming style through a simple ping-pong application. A main program (not shown for brevity) creates the two actors, sends them an *init* message carrying the identity of the partner (acquaintance) and then sends, e.g., a first *ping* message to the ping actor.

```
public class Ping extends Actor{
  protected Pong pong;
  public void init(Pong pong){
     this.pong=pong;
  }//init
  public void ping(){
     System.out.println("ping");
     pong.send( "pong" );
  }//ping
}//Ping
```

```
public class Pong extends Actor{
  protected Ping ping;
  public void init(Ping ping){
     this.ping=ping;
  }//init
  public void pong(){
     System.out.println("pong");
     ping.send( "ping" );
  }//pong
}//Pong
```

Figure 1 – Ping Pong actors

The send operation can attach a relative timestamp to a message: $target.send(\ delay, msg-name, args\ )$, where delay can be established by sampling a probability distribution function or it can be deterministic. For the timed send, the msgsrv $msg-name$ with $args$ (an array of objects) will be invoked on the target actor after delay time units are elapsed from current time (returned by the $now()$ service). When the timestamp is missing, it defaults to 0.

A library of control machines ranging from pure concurrency (time insensitive), to simulated time and real time was developed. Actors do not know the identity of the regulating control machine.

CONCEPTS OF UPPAAL SMC

UPPAAL (Behrmann *et al*., 2004) is a popular toolbox for modelling and analysis of real-time systems. A model is a network of timed automata (TA).

TA are modelled as *template processes*, which can have parameters, can be instantiated, and consist of *locations*, *edges* and *atomic actions*. TA synchronize to one another by CSP-like channels (*rendezvous*) which carry no data values. Asynchronous communication is provided by broadcast channels. The sender of a broadcast channel in no case blocks. The synchronization can be heard by 0, 1 or multiple receivers. Clock variables allow measuring the time elapsed from a given instant (clock reset). Locations of an automaton are linked by edges. Every edge can be annotated by a command with four (optional) elements: (i) a *non deterministic selection*, (ii) a *guard*, (iii) a *synchronization* (? for input and ! for output) on a channel, and (iv) an *update* consisting of a set of clock resets and a list of variable assignments. A clock *invariant* can be attached to a location as a *progress* condition. The automaton can stay in the location provided the invariant is not violated. *Committed* and *urgent* locations which must be exited without passage of time, are also supported. Committed locations have priority w.r.t. urgent locations. TA models can greatly benefits, in latest versions of the toolbox, by the use of C-like functions.

The *symbolic model checker* of UPPAAL handles the parallel composition of the TA of a model, i.e., all the possible action interleavings are considered. For exhaustive property assessment the verifier tries to build the model state graph.

The problem with symbolic model checking is that it could not be practically applied to realistic complex systems which can imply an enormous (possible infinite) or a not decidable (e.g., continuous time and stochastic behavior are combined) state graph. Property checking in these cases can only be approximated or estimated.

In recent years the UPPAAL toolbox was extended to support statistical model checking (Younes, 2006)(David *et al*., 2015). UPPAAL SMC avoids the construction of the state graph and checks properties by performing a certain number of simulation runs. After that some statistics techniques are used to infer results from the simulation runs. SMC refines and extends basic UPPAAL. Only broadcast synchronizations are allowed among stochastic TA (STA). Stopwatches, floating point (double) variables which can be assigned the value of a clock, and *dynamic templates* which can be instantiated and terminated at run time are supported too. On a stochastic TA model the following query types can be issued (meta-symbols **(** and | and **)** are in bold).

1. simulate N [**(**clock|#|void**)**<=bound] {Expr1,…, Exprk}
2. Pr[**(**clock|#|void**)**<=bound] (**(**<>|[]**)** Expression)
3. Pr[**(**clock|#|void**)**<=bound] (**(**<>|[]**)** Expression) **(**<=|>=**)** PROB
4. Pr[**(**clock|#|void**)**<=bound] (**(**<>|[]**)** Expression) **(**<=|>=**)** Pr[**(**clock|#|void**)**<=bound] (**(**<>|[]**)** Expression)
5. E[**(**clock|#|void**)**<=bound; N] (**(**min:|max:**)** Expression)

Expressions can specify an automaton is in a certain location, or some constraints on data variables or clocks etc. All the queries are evaluated according to a bound which can be related to the (implicit) global time or to a clock or to a number of simulation steps (#). Query 1 asks N simulations and collects information about the listed expressions. Query 2 evaluates the probability the given expression holds (<>) within the assigned bound or always holds within the bound ([]) with a confidence interval. Query 3 checks if the estimated probability is lesser/greater than a given probability value. Query 4 compares two probabilities. Query 5, finally, estimates the minimum/maximum value of an expression.

Quantitative estimation of a query of type 2 rests on Monte Carlo-like simulations. Qualitative queries of the type 3 and 4 use sequential hypothesis testing, and precompute a required number of runs. An important feature provided by UPPAAL SMC is *visualization* of simulation output. Following a satisfied query, the

modeler can right click on the executed query and choice an available diagram (histogram, probability distribution etc.) to be plotted. At the time of this writing, UPPAAL SMC is supported by the development version 4.1.19.

MAPPING ACTORS ONTO UPPAAL SMC

The translation of actors onto UPPAAL SMC relies on dynamic template processes. A dynamic automaton must be announced in the global declarations thus:

```
dynamic tName( params ); //only int params
```

and its behavior specified as for normal TA. The dynamic automaton can then be instantiated in the update of a command with a *spawn* expression:

```
spawn tName( args );
```

Similarly, it can be terminated by an *exit()* expression in the update of a command in the tName template process.

In the following, dynamic templates will be used only for messages, although they could also be exploited for creating/destroying actors dynamically. Let aid be an int type range for the agent unique identifiers, and msg_id a type range for the messages unique identifiers, in the MAS model. An array of broadcast channels corresponding to all the possible message servers in the model is then introduced:

```
broadcast chan msgsrv[aid][msg_id];
```

Two typical examples of message templates, respectively instantaneous and timed, are shown in the Figures 2 and 3 (see also the case study later in this paper):

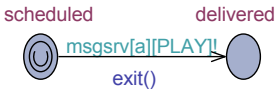

Fig. 2 – Immediate message

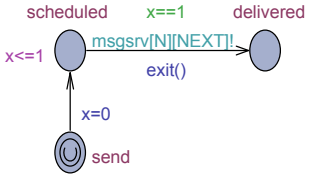

Fig. 3- Timed message

Message automata starts in a urgent location and admits two conceptual locations: *scheduled* and *delivered*. The *scheduled* can be time-sensitive. In Fig. 3 the message cannot be delivered before 1 time unit is elapsed from the send time. In Fig. 2 the scheduled message must be immediately delivered. Delivering is achieved by sending a synchronization on the msgsrv channel corresponding to the destination actor and the involved message id.

An actor automaton (see, e.g., Fig. 5) receives a message server invocation from a normal location, and then processes it through (in general) a cascade of committed locations which ends in a normal location too. Since UPPAAL SMC requires input determinism, that is, only one message an actor can receive at a time, Fig. 4 sketches a *message reception schema* which can be adopted. The actor a in the $receive$ location can get one message from $m1$, $m2$, …, mn. First a non deterministic selection is performed on the message id, immediately followed by a committed location which identifies the particular received message ID and starts its corresponding processing actions.

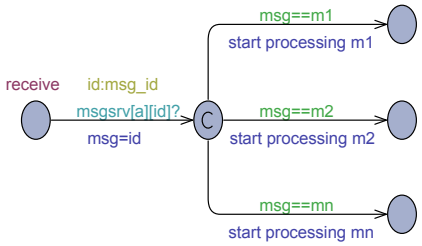

Fig. 4 – Input determinism during message reception

As a consequence of the above design: (a) the control structure of the cloud of sent messages is automatically realized by UPPAAL SMC through the activation/deactivation of dynamic message templates; (b) message processing in an actor automaton is guaranteed to terminate *before* any new message server can be delivered thus ensuring the macro step semantics.

A CASE STUDY USING THE ITERATED PRISONER'S DILEMMA

The Prisoner's Dilemma (PD) (Axelrod, 1984) is a binary game in which two players have to decide independently and without any form of communication, between two alternative choices: to defect (D, e.g., 0) or to cooperate (C, e.g., 1). The decision implies that each player gets a payoff as follows: $(D,D)\rightarrow(P,P)$, $(D,C)\rightarrow(T,S)$, $(C,D)\rightarrow(S,T)$, $(C,C)\rightarrow(R,R)$, where P means punishment for mutual defect, T temptation to defect, S sucker's payoff, and R reward for mutual cooperation. Classically $T > R > P >$S and $R > (S+T)/2$. Common adopted values are $S=0,\ P=1,\ R=3,\ T=5$.

Under the uncertainty of partner decision, players acting rationally direct themselves to defect in order to optimize their payoff, with (D,D) being the Nash equilibrium of the game. Indeed players spontaneously are driven by selfish behavior due to suspect about the opponent decision. In this situation it would be extremely risky to decide C, in fact if the partner choses D, the first player would achieve a 0 payoff and the partner the maximum reward of 5.

But if the one shot game admits the only outcome of (D,D), things are not determinate in the case the game is long iterated, with the number of iterations being unknown to players. The Axelrod book (Axelrod, 1984) triggered much interest in the social science toward studying conditions under which cooperation can emerge.

In the basic Iterated Prisoner's Dilemma (IPD) (see Axelrod tournaments (Axelrod, 1980a-b)), a certain number of players N, each equipped with a suitable strategy, repeatedly plays in turn with each of the other $N-1$ partners and the payoff is accumulated so as to detect some dominant strategy. Each player has memory

of what the opponent did in the previous move. The winner of the first tournament was the strategy Tit-for-Tat (TFT) proposed by Anatol Rapaport. TFT cooperates on the first move (i.e., it is a nice strategy) and then mimics the opponent decision taken in the previous move. Also in the second Axelrod tournament, with more competing strategies, TFT emerged as the winner strategy, but in addition the experiment revealed that "altruistic" strategies instead of "selfishness" and "greedy" behavior, in the long time can do better toward cooperation, particularly if strategies can evolve and adapt, thus learning from the experience, during the iterated game.

In (Cohen *et al*., 1999)(Axelrod *et al*., 2002) the IPD was studied from a different perspective, to investigate the role of a social interaction network upon player behavior. In particular, the goal was to check the influence of link persistence (also said context preservation) on the emergence of cooperation, in the presence of learning and evolution of the strategies. The study confirms cooperation is possible under link persistence.

The (Axelrod *et al*., 2002) complex and adaptive system was chosen as a challenging test bed for the approach described in this paper.

Case study description

The case study consists of a time step simulation of a large MAS of $N = 256$ players, where each agent plays PD with four neighbors whose identity varies with the adopted interaction network. Three cases are investigated:

- (PTG) a persistent toroidal $16x16$ grid with neighbors established according to the Manhattan neighborhood (NEWS – North, East, South and West);
- (PRN) a persistent random network, where neighbors are established randomly once at the start of the each run;
- (TRN) a temporary random network, where neighbors are defined at each step (also said period).

At the beginning of each simulation run, each player is assigned a strategy (y, p, q) of three probability values in [0,1], where y is the probability of choosing C at the first period, p is the probability of choosing C when the partner's last move was C, and q is the probability of choosing C when the partner's last move was D. The space of strategies includes the binary strategies $ALL-C$ $(y = p = q = 1)$, TFT $(y = p = 1, q = 0)$, anti TFT $(aTFT$: $y = p = 0, q = 1)$ and $ALL-D$ $(y = p = q = 0)$. However, the model initially configures the population of shuffled agents by an even distribution of strategies where $y = p$ and p and q can assume the sixteen probability values in the vector $[1/32, 3/32, \dots, 31/32]$.

At each period, each player plays 4 times the PD game separately with each of its neighbors, and the payoff is accumulated (and finally normalized) and the last move recorded, move by move, for both the player and its neighbors.

At the end of each period, following the PD moves, each player A adapts its behavior by copying (imitation) the strategy of the best performing neighbor (say it B), would the payoff of B be strictly greater than the period payoff of A. In addition, since the adaptation process can realistically be affected by errors (a comparison error can occur during the selection of the best performing neighbor, and a copying error can introduce a noise during the copying process) the following hypothesis are made. At each adaptation time, there exists a 10% chance the comparison between A and B payoffs is wrong performed and the best payoff misunderstood. Moreover, even in the case the strategy of A was not replaced with that of B, there is 10% chance that each "gene" of A strategy, i.e., the parameters y, p, q, be affected by a Gaussian noise with mean 0 and standard deviation 0.4.

The main goal of the case study is to monitor the fitness of the model vs. time, using a number T of 2500 steps or periods. First the average payoff per period is determined by adding all the period payoff of players and dividing the total period payoff by the population size N. Then the fitness is extracted by accumulating, at each time t, all the population average payoffs up to t, and dividing this sum by t. Other observables are the average values at each time of the probabilities p and q, averaged over all the population, so as to monitor the trend of strategy adaptation. Of course, a fitness value definitely moving toward 1 mirrors the emergence of defection, whereas its tendences to 3 (actually to a value greater than 2) testifies cooperation. The above described observables must be checked in all the possible model configurations.

Multi-agent model in UPPAAL SMC

Two basic agents were introduced: the $Manager$ (one instance) who is in charge of enforcing the time stepped simulation mode, and the $Player$ (N instances) who contains the details of the IPD game, along with global declarations which include the following:

```
//model global declarations
const int N=256;
const int dim=16; //sqrt(N)
typedef int[0,N-1] pid; //player id
typedef int[N,N] mid; //manager id
typedef int[0,N] aid; //agent id
const int MSG=5; //number of distinct message ids
typedef int[0,MSG-1] msg_id;
broadcast chan msgsrv[aid][msg_id];
clock now; //simulation time
```

Manager admits INIT, NEXT and DONE message ids, whereas Player can receive an INIT, PLAY or ADAPT message. NEWS links are randomly interpreted when a TRN or PRN topology is used instead of PTG. The following is the set of used dynamic messages. Each automaton (see Fig. 2 and Fig. 3) delivers to an actor a specific message id through the msgsrv array of channels. Except for the Next automaton (Fig. 3), all the others are instantaneous messages whose model follows that of Fig. 2.

```
//dynamic message declarations
dynamic InitManager();
dynamic Next();
```

```
dynamic Done();
dynamic InitPlayer( const pid a );
dynamic Play( const pid a );
dynamic Adapt( const pid a );
```

Fig. 5 depicts the Player automaton (whose only parameter is *const pid a*). A player first gets an INIT message then it waits for a PLAY message to which it responds with a Done to Manager. After that the player expects an ADAPT message to which it answers with another Done message to Manager. Functions *init_player*(), *do_play*() and *do_adapt*() respectively prepare and implement the game. For instance, in the *PTG* case, init_player() identifies the four neighbors NEWS which persist along all the simulation. The function also establishes the strategy assigned to the player. do_play() realizes the 4 moves with each of its neighbors, accumulates the payoff, and stores the last move for player *a* and for each of its neighbors. Two versions of do_adapt() were built: the first one ignores any error during the adaptation process; the second one considers probabilistic comparison and copy errors as stated in the case study description.

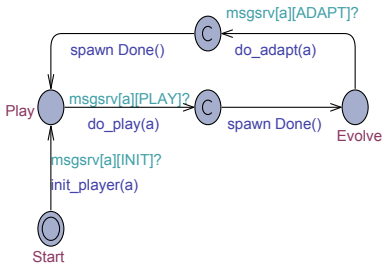

Figure 5 – *Player* automaton

Fig. 6 shows the Manager automaton (which admits the parameter *const mid m*). The Manager first receives an INIT from the Main automaton (see Fig. 7), which causes a INIT message to be sent to all the players, then it sends proactively to itself a NEXT message with constant delay of 1 time unit. On receiving the NEXT message, the Manager enters its basic cycle: first data structures are reset for the next period, then a PLAY message is sent to all players, followed by *N* DONE replies to be received. When all the DONE messages arrived, the Manager goes on by sending an ADAPT message to all the players, after that *N* DONE replies are awaited. Finally, a NEXT message is sent again to itself for triggering the next cycle and the story recommences.

It is important to note that scheduled groups of instantaneous (concurrent) messages, such as INIT, PLAY or ADAPT to players, are actually delivered in a non deterministic way. This is a direct consequence of the asynchronous spawning mechanism and the use of urgent locations in dynamic message templates, which are exited in an interleaved arbitrary way. This same non determinism is a key for achieving actor shuffling and even distribution of all the pairs of p,q probabilities.

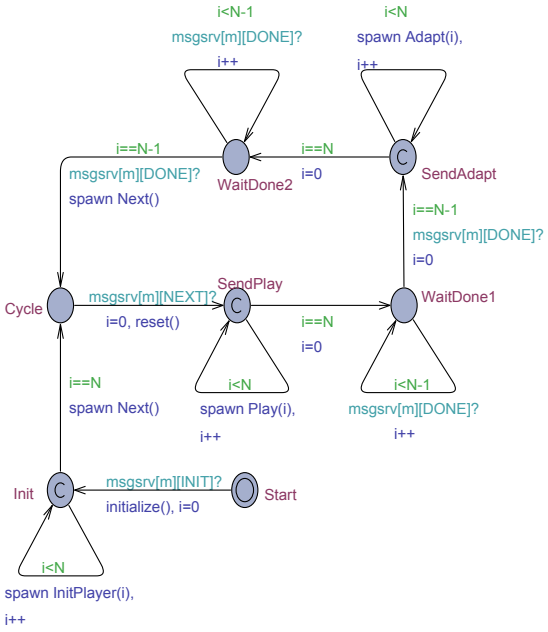

Figure 6 – *Manager* automaton

Model bootstrap is ensured through the *Main* automaton (see Fig. 7) which only sends the INIT message to the Manager by spawing the InitManager message template.

Figure 7 – *Main* automaton

System configuration relies on implicit template instantiation and is composed as follows:

system Main, Manager, Player;

One instance of Main (anonymous), one instance of Manager (with id *N*) and *N* instances of Player (with ids from 0 to $N-1$) are created.

Implicit system (simulated) time increases with timed NEXT message delivery. The decoration clock *now*, initialized to 0 (default) and automatically advanced, was introduced so as to make explicit the simulation time to statistic functions such as *avg_payoff*(), *fitness*(), *avg_p*(), *avg_q*() etc., which depend also on some further decoration variables like *totpayoff* (double).

EXPERIMENTAL RESULTS

Some preliminary experiments were carried out for observing the shape of the average period payoff in the first few time steps. The query:

simulate 1 [<= 50] { *avg_payoff*() }

was used. Results for the *PTG* model without adaptation errors are shown in Fig. 8. Basic behavior holds with or without adaptation errors and also for *PRN* and *TRN* topologies. Fig. 8 confirms the indications in (Cohen *et al.*, 1999) at page 24 and page 42. The average payoff

starts at 2.25 then sharply decreases, after which it will tend to a final possible regime. The initial value is due to an equivalent average strategy (y, p, q) of (.5,.5,.5) being randomly initially distributed. The sharp decline is due to the presence of akin $ALL - C$ strategies which play with akin $ALL - D$ strategies. As a consequence, $ALL - D$ tends to dominate, but as $ALL - D$ plays with other $ALL - D$ it causes a sudden decrease in the payoff.

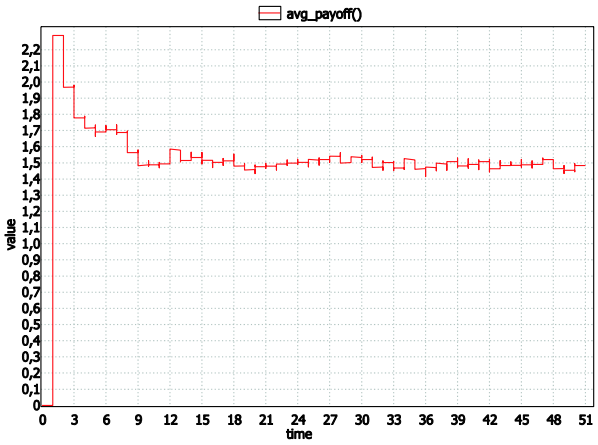

Figure 8 – Average payoff in the first 50 steps – no errors

The three models PTG, PRN and TRN were repeatedly studied by using the following query:

$$simulate\ 1\ [\leq 2500]$$
$$\{\ avg\_payoff(), fitness(), avg\_p(), avg\_q()\ \}$$

Figures from 9 to 11 depict the observed average period payoff, the temporal average $fitness$, and the average p value and q value for the three models, in the most general case when adaptation errors are involved. In Fig. 12 it is reported the watched TRN behavior without adaptation errors.

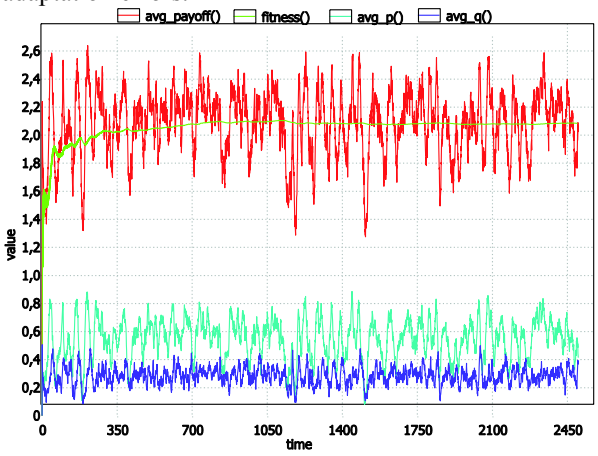

Figure 9 - PTG sampled behavior with adaptation errors

In the absence of adaptation errors, no new strategies can dynamically be introduced. Rather, some strategies can dominate whereas others can reduce its number or even disappear from the game. In these scenarios, fluctuations of the payoff tend soon to stabilize (see Fig. 12) and cooperation can or cannot possibly occur. All depends from the initial random assignment of strategies to shuffled players, i.e., who plays with who initially. However, in the TRN model, where neighbors are redefined at each time step, in no case cooperation was observed.

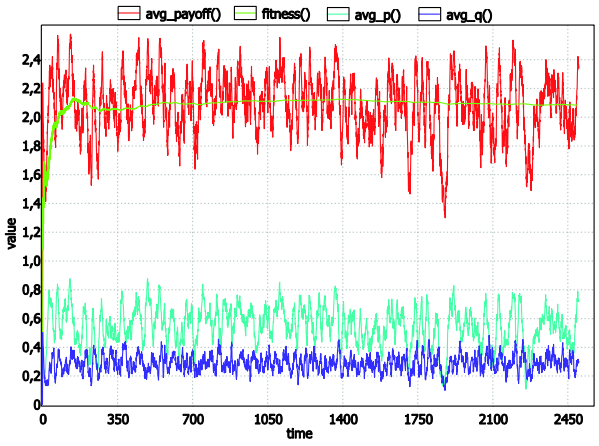

Figure 10 - PRN sampled behavior with adaptation errors

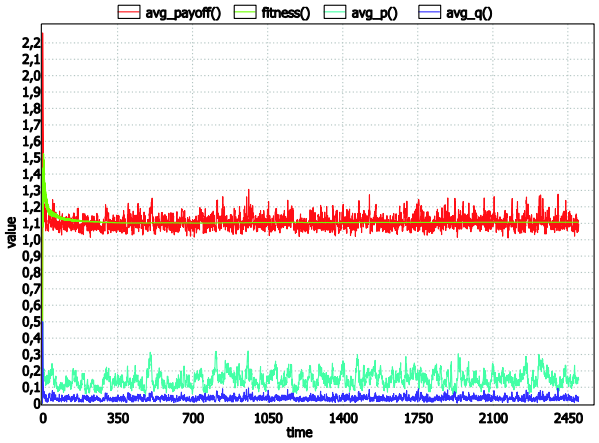

Figure 11 - TRN sampled behavior with adaptation errors

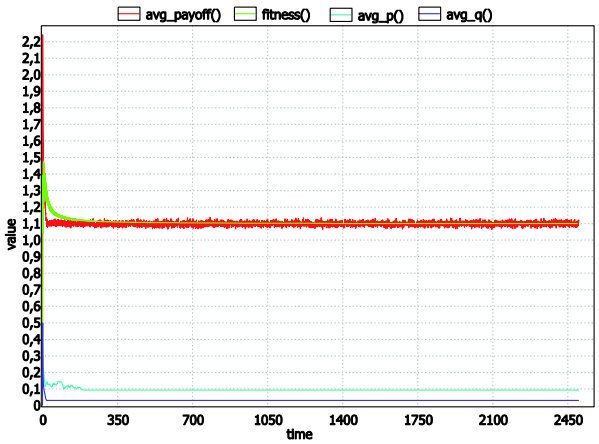

Figure 12 - TRN sampled behavior without adaptation errors

Under the presence of adaptation errors, new strategies can be created at run time and persistent topologies (PTG and PRN) manifest a more evident character to host a cooperative regime (Fig. 9 and Fig. 10). In the TRN model, instead, defection prevails (Fig. 12). The next step was to check the emergence and maintaining of a cooperative regime in the three models with errors, by a query like the following:

$$Pr[<= 800]\ (\ []\ (now < 500\ ||\ fitness() > 2.0)\ )$$

which asks to estimate through a number of runs the probability that after 500 steps a cooperation regime is eventually reached. Given the low number of time steps, it was inconclusive for PTG and PRN. For TRN, after 36 runs, UPPAAL SMC suggested Pr is in the interval $[0, 0.0973938]$ with a confidence degree of 95%, which testifies the attainement of a defection regime.

The achieved experimental results agree with the results documented in (Axelrod *et al.*, 2002), although with a lesser value of the cooperation level. Results were also validated by porting the IPD models in Java. Transportation was facilitated by the UPPAAL SMC formal approach. Fig. 13 depicts the observed payoff of the three topologies (with adaptation errors) averaged after 30 runs of the Java models.

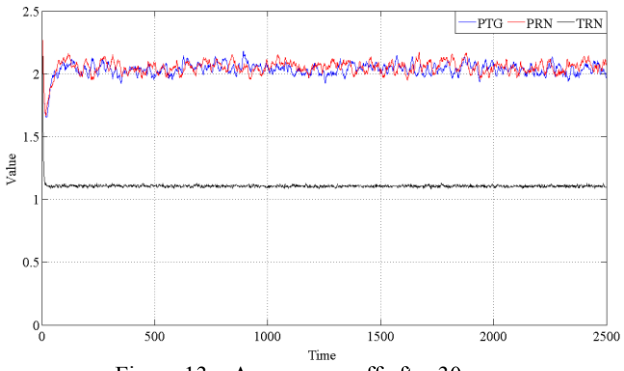

Figure 13 – Average payoff after 30 runs

From the Java models it emerged, after 100 runs, that beyond a threshold of 2000 time steps, both PTG and PRN reach and maintain the cooperative regime with a probability value of almost 1.

The experimental work confirms the intuition that link persistence (i.e., playing with the same partners during all the game) is a key for cooperation because it favours players trustiness. All of this has an obvious social interpretation nowadays when one considers people interactions through a social network.

Experiments were carried out on a Linux machine, Intel Xeon CPU E5-1603@2.80GHz, 32GB, using UPPAAL 4.1.19 64bit.

CONCLUSIONS

This work develops an original approach in UPPAAL SMC (David *et al.*, 2015), which enables modelling and analysis of complex adaptive asynchronous multi-agent systems (MAS). Benefits of the approach are formal modelling and the exploitation of statistical model checking techniques (Larsen & Legay, 2016).

For demonstration purposes, a scalable version of the Iterated Prisoner's Dilemma (IPD) game (Axelrod *et al.*, 2002) was modelled and thoroughly experimented. The goal was not to add to the theory of IPD, but only using a particular version of it as a benchmark. Practical limitations of the approach relate to the MAS model size, which can imply very long execution times. Prosecution of the research is directed to:

- Porting the approach in the Plasma Lab tool (Plasma Lab, on-line) so as to integrate scalable Java-based actor models for efficient SMC analysis;
- Adapting the IPD models toward an investigation of new player strategies;
- Extending the UPPAAL SMC approach to modelling and analysis of distributed probabilistic real-time actor systems.

REFERENCES

Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA.

Axelrod, R. (1980a) Effective choice in the prisoner's dilemma, *J. of Conflict Resolution*, **24**, pp. 3-25.

Axelrod, R. (1980b) More effective choice in the prisoner's dilemma, *J. of Conflict Resolution*, **24**, pp. 379-403.

Axelrod, R. (1984). *The evolution of cooperation*. Basic Books, New York.

Axelrod, R., R.L. Riolo, M.D. Cohen (2002). Beyond geography: cooperation with persistent links and in the absence of clustered neighborhoods. *Personality and Social Psycology Review*, **6**(2):341.346.

Behrmann, G., A. David, K.G. Larsen (2004). A tutorial on UPPAAL. In: *Formal Methods for the Design of Real-Time Systems*, Lecture Notes in Computer Science, Vol. 3185, Springer-Verlag, pp. 200-236.

Cicirelli, F., L. Nigro (2013). An agent framework for high performance simulations over multi-core clusters. *Communications in Computer and Information Science (CCIS)*, Volume 402, pp. 49-60, Springer.

Cicirelli, F., L. Nigro (2016a). Control centric framework for model continuity in time-dependent multi-agent systems. *Concurrency and Computation: Practice and Experience*, **28**(12):3333-3356, Wiley.

Cicirelli, F., L. Nigro (2016b). Exploiting social capabilities in the minority game. *ACM Trans. on Modeling and Computer Simulation*, **27**(1), article 6, DOI: http://dx.doi.org/10.1145/2996456.

Cohen, M.D., R.L. Riolo, R. Axelrod (1999). The emergence of social organization in the Prisoner's Dilemma: how context-preservation and other factors promote cooperation. *Santa Fe Institute Working Paper*: 1999-01-002.

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.

Larsen, K.G., A. Legay (2016). On the power of statistical model checking. In *7th Int. Symposium, ISoLA 2016*, pp. 843-862.

North, M.J., C.M. Macal (2007). *Managing business complexity-Discovering strategic solutions with Agent-based Modeling and Simulation*. Oxford University Press.

Plasma Lab, on-line, https://project.inria.fr/plasma-lab/.

Reynisson, A.H., M. Sirjani, L. Aceto, M. Cimini, A. Jafari, A. Ingolfsdottir, S.H. Sigurdarson (2014). Modelling and simulation of asynchronous real-time systems using timed Rebeca. *Science of Computer Progr.*, vol. 89, pp.41-68.

Varshosaz, M., R. Khosravi (2012). Modelling and verification of probabilistic actor systems using pRebeca. *LNCS 7635*, pp. 135-150, Springer.

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.