Modelling and analysis of multi-agent systems using UPPAAL SMC

Christian Nigro, Libero Nigro* and Paolo F. Sciammarella

Laboratorio di Ingegneria del Software, Università della Calabria, DIMES, 87036 Rende (CS), Italy
Email: christian.nigro21@gmail.com
Email: l.nigro@unical.it
Email: p.sciammarella@dimes.unical.it
*Corresponding author

Abstract: This paper proposes a novel approach to modelling and analysis of complex multi-agent systems. The approach is based on actors and asynchronous message passing, and exploits the UPPAAL statistical model checker (SMC) for the experiments. UPPAAL SMC is interesting because it automates simulations by predicting the number of executions capable of ensuring a required output accuracy, it uses statistical techniques (Monte Carlo-like simulations and sequential hypothesis testing) for extracting quantitative measures from the simulation runs, and it offers a temporal logic query language to express property queries tailored to the application needs. The paper describes the approach, clarifies its structural translation on top of UPPAAL SMC, and demonstrates its practical usefulness through modelling and analysis of a large scale and adaptive version of the Iterated Prisoner’s Dilemma (IPD) problem. The case study confirms known properties, namely the emergence of cooperation under context preservation, that is when the player interaction links are preserved during the game, but it also suggests some new quantitative measures about the temporal behaviour which were not previously pointed out.

Keywords: modelling and simulation; multi-agent systems; MAS; actors; statistical model checking; UPPAAL; Iterated Prisoner’s Dilemma; IPD.


Biographical notes: Christian Nigro received his Specialistic Laurea degree in Computer Engineering from the Department of Informatics, Modelling, Electronics and Systems Science of the University of Calabria, 87036 Rende (CS) Italy. He works as a computer professional in the area of software engineering. His research interests include modelling, analysis and implementation of concurrent and distributed systems, and multi-agent systems in particular, using UPPAAL and Java.

Libero Nigro is a Full Professor of Computer Engineering in the Department of Informatics, Modelling, Electronics and Systems Science (DIMES) of the University of Calabria, 87036 Rende (CS) Italy. He heads the Software Engineering Laboratory (http://www.lis.dimes.unical.it) at DIMES, and teaches courses on Object-Oriented Programming and Software Engineering for Real-Time Systems and Multi-Agent Systems. His research interests include formal modelling (using Petri nets, timed automata, DEVS, etc.), analysis (by simulation and exhaustive verification by model checking) and final implementation of distributed actor-based systems with timing constraints, including cyber-physical systems. A major goal is ensuring model continuity during the development life cycle.

Paolo F. Sciammarella holds a Specialistic Laurea degree in Automation Engineering. Since 2015, he is a PhD student graduating in Information and Communication Technologies at the Department of Informatics, Modelling, Electronics and Systems Science (DIMES) of the University of Calabria, 87036 Rende (CS) Italy. His research interests are centred in the area of agent-based software engineering of distributed measurement systems and cyber-physical systems, covering formal modelling (e.g., by Petri nets), simulation and implementation aspects in Java, with a support to model continuity.
1 Introduction

Multi-agent systems (MAS) (Wooldridge, 2009) are widely recognised as an important paradigm for modelling and analysis through simulation (M&S) of complex and adaptive systems (North and Macal, 2007; Cicirelli and Nigro, 2016b; Yacoub et al., 2017). Power and flexibility of MAS derive from their ability of modelling both the individual behaviour of agents and their social interactions, i.e., the exchanges of information, and then the possibility of observing the emergence of properties at the society/population level. Agents are characterised by their basic abilities (Wooldridge, 2009) of autonomy, sociality, pro-activity, ‘intelligence’ for deliberative behaviour, learning and adaptation mechanisms, and so forth.

In this work a minimal yet efficient actor computation model (Agha, 1986; Cicirelli and Nigro, 2016a) is adopted for supporting MAS, which addresses complex models. A key feature of this framework is a light-weight notion of actors which:

1. are thread-less agents
2. hide an internal data status
3. communicate with each other by asynchronous message passing.

Message exchanges are ultimately regulated by a customisable control structure which can reason on time (simulated or real-time). The actor framework can effectively be hosted by popular languages like Java.

An original contribution of this paper is a structural translation of a MAS actor model into the terms of UPPAAL SMC (David et al., 2015). Challenging is an effective support of asynchronous timed message passing. Main motivations underlying the proposed work are the following:

1. using a formal modelling tool based on timed automata (Alur and Dill, 1994) to capture, in a natural way, actor behaviours and message exchanges
2. expressing in the associated temporal logic language, model specific properties to be checked on the MAS model by simulations
3. exploiting statistical model checking techniques (Younes et al., 2006; Larsen and Legay, 2016), that is automatising multiple executions of a MAS model, estimating the required number of simulation runs, and using statistical properties (Monte Carlo-like simulations and sequential hypothesis testing) to infer system properties from the observables of the various runs.

UPPAAL SMC was chosen among other competitive tools like PRISM (Hinton et al., 2006), PLASMA LAB (http://www.project.inria.fr/plasma-lab/), etc., because:

a. it is a popular and efficient toolbox based on a stochastic extension of Timed Automata (Alur and Dill, 1994; David et al., 2015)

b. it supports graphical, intuitive modelling
c. it offers high-level data structures and functions which improve the modelling of complex systems.

As for PRISM (see e.g., Jafari et al., 2014, 2016), UPPAAL SMC can also be exploited for modelling and quantitative assessment of timing constraints in probabilistic real-time systems (Nigro and Sciammarella, 2017b; Zeigler et al., 2017).

To the best of authors’ knowledge, this is a first attempt to support general actor-based MAS models and their quantitative evaluation using UPPAAL SMC, with the approach which can be concretely used by modelling and simulation practitioners and engineers.

The approach proposed in this paper is practically demonstrated through a case study concerned with a complex and adaptive model based on the Iterated Prisoner’s Dilemma (IPD) game (Axelrod et al., 2002). The model is challenging and aims to study the emergence of cooperation among competitive agents in the presence of different social interaction networks. SMC results confirm previous indications of Axelrod et al.’s work, that is that cooperation is possible among players when their interaction links are preserved during the game execution. In addition, the accomplished detailed probabilistic analysis permits to observe some new properties not previously revealed (details in Subsection 6.4).

This paper is an extended version of preliminary work published in ECMS 2017 conference (Nigro and Sciammarella, 2017a). With respect to this previous work, the approach was significantly improved from the point of view of both modelling and the execution performance. In addition, the IPD case study was completely re-designed and re-worked and new experimental results collected by studying, as in Axelrod et al. (2002), the scenario when the player last move only affects its next decision in the same game. These results are documented in Subsection 6.3 and were not covered in Nigro and Sciammarella (2017a). Results in Subsections 6.4 and 6.5, which refer to the more general scenario where the player last move always affects its next move throughout the IPD, were re-computed and statistically better quantified, with respect to preliminary experiments, by accurately characterising probability measures of model behaviour by their confidence intervals.

The paper is structured as follows. First, the adopted actor computational model is described. Then an actor-based model for the IPD problem is discussed. After that, major features of UPPAAL SMC are recapitulated. The paper goes on by detailing the proposed structural mapping of actors onto UPPAAL SMC using the IPD case-study as a reference. Then a thorough analysis of the properties of the IPD model is presented. Finally, conclusions are drawn with an indication of on-going and future work.

2 An agent and control-based framework

This section describes an agent framework which is based on actors (Agha, 1986). Actors represent a reference...
computational model for the development of concurrent and distributed systems. Actors communicate to one another by asynchronous message passing. In the basic model, actors are thread based entities which encapsulate a data status and expose an interface of messages, and have an input mailbox wherein incoming messages get buffered. The actor thread extracts, if there are any, one message at a time from the mailbox and processes it. Message processing (message reaction) is atomic and consists of actions for:

a. modifying the encapsulated data variables
b. sending messages to known actors (acquaintances) including itself
c. creating new actors.

The actors model is well suited for general untimed systems.

Some variants of the basic model, though, motivated by specific application domains, e.g., the needs for dealing with distributed and probabilistic time-dependent systems, have been proposed in the literature.

The RTSynchroniser abstraction mechanism proposed in Nielsen and Agha (1996) aims at capturing in a declarative way the interaction patterns existing in groups of actors. An RTSynchroniser tries to fulfil actors timing constraints by enforcing the `safe progress, unsafe block’ semantics upon sent messages.

The concept of RTSynchronisers was exploited in Beraldi and Nigro (1999) for designing and implementing a Time Warp mechanism enabling high performance distributed simulations in a networked context. The RTSynchroniser concept was also used in Nigro and Pupo (2001) for developing a schedulability analysis framework of distributed real-time actor systems modelled by coloured Petri nets.

The Rebeca modelling language (Varshosaz and Khosravi, 2012; Jafari et al., 2014, 2016) represents a recent project based on actors, addressing specifically probabilistic and timed distributed systems. Rebeca is not centred on the RTSynchroniser mechanism. Each message in timed Rebeca is timestamped with two quantities: an after and a deadline, which are relative to the instant in time the message is sent. A message cannot be delivered before after time units are elapsed, but it should be dispatched to its destination within its deadline. Otherwise the message is cancelled. The Rebeca project is characterised by its supporting tools, e.g., based on PRISM or McErlang. However, such tools were developed for real-time systems and do not support general modelling and analysis of MAS.

In the work described in this paper, a light-weight (thread-less) and control-based version of the actor model is adopted (Cicirelli and Nigro, 2013, 2016a), addressing particularly time-predictability and high-performance 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 an interface to a TCP-based transport layer. The design of a CM builds upon the RTSynchroniser concept: it transparently manages the cloud of sent messages and delivers them, one at a time, according to an application oriented control structure. Message processing is atomic (macro-step semantics) and constitutes the unit of scheduling and dispatching. A cooperative concurrency schema among the local agents of a theatre is ensured by message interleaving. True parallelism is instead possible among the actors allocated on to distinct theatres. The CMs of a system synchronise each other in order to guarantee a coherent global time notion. Details about control structure designs and global synchronisation algorithms based on a time server, e.g., distributed simulators and real-time control engines, can be found in Cicirelli and Nigro (2016a). In this paper the focus will be on a single theatre: instead of distributed simulation concerns, the goal is statistical model checking of an actor model by exploiting a temporal logic language for a thorough analysis of system properties.

The actor framework can be practically used according to different programming styles. In Cicirelli and Nigro (2013, 2016a) actors are supposed to be equipped by a single handler() method where the code for responding to all the admitted messages is confined. The handler captures the actor behaviour which, in general, consists of a finite state machine. In this paper, as in Rebeca (Jafari et al., 2016), a more intuitive programming style is assumed where each separate message is handled by a corresponding method (message server or msgsrv).

 Actors are instances of classes which inherit from the abstract Actor base class which exposes, among the others, the basic non-blocking message send operation.

Figure 1 exemplifies the actor programming style in Java 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 a first ping message to the ping actor. From then on a ping message is followed by a pong one and so forth endlessly.

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

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

The send operation can attach an optional relative timestamp to a message thus: target.send(delay, msg_name, args), where delay can be established by sampling a probability distribution function or it can be a deterministic value. When the timestamp is missing, it defaults to 0. 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 the current time (returned by the now () operation exported by the Actor class and implemented by a control machine). However, when a msgsrv gets actually invoked it depends ultimately on a
decisions of the control machine. No mailbox is provided per actor. Actors do not know the identity of the regulating control machine. A control machine acts as a plug-in component of a computational theatre.

3 A modelling example based on the Iterated Prisoner’s Dilemma

The following considers a case study modelling example based on the Iterated Prisoner’s Dilemma (PD) game (Axelrod, 1984; Wooldridge, 2009). The goal is to use a large and challenging model as a testbed for the modelling and simulation approach proposed in this paper.

PD 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) or to cooperate (C). The decision implies that each player gets a payoff as follows: (D, D) → (P, P), (D, C) → (T, S), (C, D) → (S, T), (C, C) → (R, R), where P means punishment for mutual defect, S temptation to defect, T 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 optimise their payoff, with (D, C) → (D, S). Therefore, a strategy must include the binary strategies: C → C, T → T, or D → D. The space of strategies includes the binary strategies: ALL → C, TFT → T, anti→ TFT or aTFT → T. The chosen model initially configures the population of shuffled agents by an even distribution of strategies where y = p and q can assume the sixteen probability values in the vector [0.16, 0.17, …, 0.31].

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, TFT or anti→ TFT or aTFT → T, and ALL → D. The chosen model initially configures the population of shuffled agents by an even distribution of strategies where y = p and q can assume the sixteen probability values in the vector [0.16, 0.17, …, 0.31].

At each period, each player plays four times the PD game separately with each of its neighbours, and the payoff is accumulated (and finally normalised) and the last move recorded, move by move, by both the player and its neighbours.

At the end of each period, following the PD moves, each player A adapts its behaviour by copying (imitation) the strategy of the best performing neighbour (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 neighbour, 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 that the comparison between A and B payoffs is wrongly performed and the best payoff misunderstood. Moreover, even in the case the strategy of A is not replaced by that of B, there is 10% chance that each ‘gene’ of the 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 2,500 steps or periods. First the average payoff per period is determined by...

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.

3.1 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 neighbours whose identity varies according to the adopted interaction network. Three cases are investigated:

- a persistent toroidal grid with neighbours established according to the Manhattan neighbourhood (NEWS – North, East, West and South)
- a persistent random network, where neighbours are established randomly once at the start of each run
- a temporary random network, where neighbours are re-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, TFT or anti→ TFT or aTFT → T, and ALL → D. The chosen model initially configures the population of shuffled agents by an even distribution of strategies where y = p and q can assume the sixteen probability values in the vector [0.16, 0.17, …, 0.31].

At each period, each player plays four times the PD game separately with each of its neighbours, and the payoff is accumulated (and finally normalised) and the last move recorded, move by move, by both the player and its neighbours.

At the end of each period, following the PD moves, each player A adapts its behaviour by copying (imitation) the strategy of the best performing neighbour (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 neighbour, 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 that the comparison between A and B payoffs is wrongly performed and the best payoff misunderstood. Moreover, even in the case the strategy of A is not replaced by that of B, there is 10% chance that each ‘gene’ of the 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 2,500 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 tendency to 3 (actually to a value greater than 2) testifies cooperation. The above described observables must be checked in all the possible model configurations.

3.2 An IPD actor-based model

The model (see Figure 2) consists of $N + 1$ actors: one Manager actor who directs the games, and $N$ players who actually do the game actions.

Figure 2  IPD actor model

A minimal Main program is responsible for creating the actors and sending an $\text{init}()$ message to initialise the Manager. The Manager continues by initialising all the players by sending to each of them an $\text{init}()$ message. After initialisation, the Manager enters its main cycle of operations. Each cycle begins by the Manager which sends to itself a $\text{next}()$ message (see also Figure 3) which lasts one time units, thus realising the basic step or period. On receiving $\text{next}()$ the Manager sends to all the players a $\text{play}()$ message. Each player reacts by executing the 4 moves with each of its neighbours and accumulates the period playoff. After that, a $\text{done}()$ message is sent to the Manager. When all the $\text{done}$ messages are received, the Manager broadcasts an $\text{adapt}()$ message to all the players so as to trigger the learning-and-adaptation phase. Finally, the players send a $\text{done}()$ message to the Manager. When all the $N$ done messages are collected, the Manager starts the next cycle and so forth. It should be noted that the initialisation messages are sent with a 0 delay. Since the $\text{next}()$ message is received after 1 time unit is elapsed, it is guaranteed the model has finished the initialisation phase before the first cycle begins. This explains why done messages are avoided during the IPD initialisation.

When the interaction network is persistent during the whole game, each player establishes its neighbourhood at the initialisation time. In the case of a temporary network, links are re-defined just before starting the next cycle of operations.

Figure 3  Message exchanges during IPD operation

4 An overview of UPPAAL SMC

In this paper an actor model is translated into the terms of UPPAAL SMC for a thorough analysis of its temporal behaviour. In this section, a brief overview of the concepts of UPPAAL (Behrmann et al., 2004) and of UPPAAL SMC (David et al., 2015) in particular, is given.

UPPAAL is a popular toolbox for modelling and analysis of real-time systems. A model is a network of timed automata (TA) (Alur and Dill, 1994), enhanced by integer data variables, high-level data structures (e.g., arrays) and other powerful constructs which improve modelling.

TA are designed as template processes, which can have parameters, can be instantiated, and consist of locations, edges and atomic actions. TA synchronise to one another by binary 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 is blocked. The synchronisation 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:

1. a non-deterministic selection
2. a guard
3. a synchronisation (? for input and ! for output) on a channel
4. an update consisting of a set of clock resets and a list of variable assignments.

An edge command models an atomic action, with the update of the sender of a synchronisation which is executed before that of the receiver. 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 benefit by the use of C-like functions which contribute to compact models.

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., when continuous time and stochastic behaviour 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 et al., 2006; David et al., 2015). UPPAAL SMC avoids the construction of the state graph, thus ensuring a linear memory consumption, and checks properties by performing a certain number of simulation runs. After that some statistics are collected and checks properties by performing a certain number of simulation runs. The meta-symbols (Pappas, 2007), can be issued to express properties over mostly derived from an extension of the metric interval [0, 1] instantiated and terminated at run time are supported too. Value of a clock, and floating point (double) variables which can be assigned the dynamic templates which can be expressed in the definition of a location invariant, broadcast synchronisations are allowed among stochastic systems which can imply an enormous (possible infinite) or a not decidable (e.g., when continuous time and stochastic behaviour are combined) state graph. Property checking in these cases can only be approximated or estimated.

UPPAAL SMC configuration options include:

a) The probability uncertainty \( \varepsilon \) of a confidence interval \([p – \varepsilon, p + \varepsilon]\) with significance level \( \alpha \), i.e., confidence degree \((1 – \alpha)\%\).

b) The half-width \( \delta \) of the indifference region and the probabilities of accepting a false negative, \( \alpha \), or false positive, \( \beta \), in hypothesis testing.

The accuracy probability parameters are used to estimate the number \( n \) of runs. In previous versions of the tool the number \( n \) was established as \( \ln(2/\alpha)/2\varepsilon^2 \) according to the Chernoff-Hoeffding bound (see, e.g., Bogsted, 2015). Actually, the ‘exact’ method of Clopper-Pearson (see, e.g., Pires and Amado, 2008) for binomial distribution is used which in many cases requires fewer runs than the Chernoff-Hoeffding bound. A sequential procedure is used which stops runs as the interval size is found smaller than \( 2\varepsilon \).

As a final remark, UPPAAL SMC enables visualisation of simulation output. Following a satisfied query, the modeller can right click on the executed query and select an available diagram (histogram, probability distribution density etc.) to be plotted. At the time of this writing, UPPAAL SMC is supported by the development version 4.1.19.

5 A structural translation from actors to UPPAAL SMC

Since actors are finite state machines, they can be naturally mapped onto UPPAAL SMC template automata. The subtle point of the translation is the achievement of dynamic asynchronous message passing and the control machine design which is in charge of collecting sent messages in a cloud of messages and dispatching them, one at a time, to the relevant receivers thus ensuring the macro-step semantics: only one message can be under processing at any
instant in time, and its execution is atomic. The next message will be selected and dispatched only at the end of the current message reaction. Concurrent messages, i.e., messages which are scheduled to occur at the same time, are selected and dispatched in a non-deterministic way. Details of the translation will be provided in the following using the IPD model as an example.

5.1 Actor and message naming

Actors and messages are supposed to be identified by unique integer names which ultimately depend from the model at hand. The idea is to introduce distinct integer subrange types for all the model actors and separately for the instances of each particular actor type. For the IPD model, the following global declarations can be introduced:

```c
const int N = 256; // player population
const int dim = 16; // sqrt(N)
```

typedef int[0, N-1] pid; // player ids subrange type

typedef int[N, N] mid; // manager id subrange type

typedef int[0, N] aid; // agent ids subrange type

Using a const pid a parameter for the Player automaton (see Figure 9), and a const mid m parameter for the Manager automaton (see Figure 8) guarantees N instances will be created for the Player template automaton, with the ids ranging from 0 to N - 1, and only one instance for the Manager template automaton whose id is N.

Messages of the IPD model can be classified as follows:

```c
// message ids
const int INIT = 0;
const int PLAY = 1;
const int ADAPT = 2;
const int NEXT = 3;
const int DONE = 4;
```

Then the following sub-range type can be introduced:

```c
const int MSG = 5;
```

typedef int[0, MSG-1]msg_id

To allow the control machine to dispatch a message to an actor automaton, the following matrix of channels (it should be recalled that UPPAAL SMC permits only broadcast synchronisations among automata) is used:

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

where the first index is the id of the recipient actor and the second index identifies the delivered message. Although `msgsrv[.,.]` channels are broadcast, the use of a specific actor id and message id restricts the potential actor receivers only to the message destination actor.

5.2 Dynamic messages

A straightforward and elegant solution for dealing with dynamic message instantiation (send operation) and scheduling can be directly based on the dynamic automata which UPPAAL SMC supports. A dynamic automaton must be announced in the global declarations thus:

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

and its behaviour is specified as for the regular timed automata. The dynamic automaton can then be instantiated in the update of a command with a `spawn` expression:

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

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

Two typical examples of message dynamic templates, respectively for an instantaneous (or immediate) message and a timed message, are shown respectively in Figures 4 and 5.

**Figure 4** Immediate message (see online version for colours)

scheduled

msgsrv[a][PLAY]!

delivered

exit()

**Figure 5** Timed message (see online version for colours)

scheduled

x = 1

x < 1

msgsrv[N][NEXT]!

delivered

x = 0

x = 0

exit()

send

Message automata start in an urgent location and admit two key locations: scheduled and delivered. The scheduled can be time-sensitive [possibly stochastic, see Section 7.2 of (David et al., 2015)]. In Figure 5 the message cannot be delivered before 1 time unit is elapsed from the send time. In Figure 4 the scheduled message must be immediately delivered. Delivering is achieved by sending a synchronisation over the `msgsrv` channel corresponding to the destination actor and the involved message id. Dynamic instantiation of a message automaton occurs in an actor automaton and it is supposed to transmit as parameter the identity of the recipient actor (see the a parameter in Figure 4).

Although dynamic messages are a flexible and elegant mechanism, a penalty in the execution performance can practically prohibit its exploitation, especially in large MAS models like the IPD model. In the preliminary work documented in Nigro and Sciammarella (2017a), the IPD model was completely prototyped using only dynamic message instantiations. The consequence was that a single simulation of 2,500 time steps required about six hours of wall-clock time (WCT) to complete.

In this paper a more efficient solution is adopted which is based on statically pre-allocated automata for all the
messages involved in the IPD model. Message automata are dynamically activated by specific synchronisation channels. When a message is eventually dispatched, the corresponding automaton resets its behaviour so as to be subsequently re-activated and so forth. After a careful examination of the IPD model, two static automata were designed as depicted in Figures 6 and 7, along with the following global declarations:

```
typedef int[INIT, ADAPT]player_msg;
//player message ids
broadcast chan next, done[pid], bSend[player_msg];
```

**Figure 6** Instantaneous Message automaton (see online version for colours)

![Diagram](image)

The instantaneously dispatched messages are re-activated in the corresponding automaton so that they may be subsequently re-used. After a careful examination of the IPD model, two static automata were designed as depicted in Figures 6 and 7, along with the following global declarations:

```
typedef int[INIT, ADAPT]player_msg;
//player message ids
broadcast chan next, done[pid], bSend[player_msg];
```

**Figure 7** Timed Message automaton (see online version for colours)

![Diagram](image)

The parameterless StopMessage template automaton exists in one instance only. It is activated by a synchronisation over the scalar next channel. The Message automaton, instead, is designed so as to pre-allocate N instances of immediate messages. Towards this it is sufficient to parameterise the Message template with the only parameter const pid p. The N instances of Message serve all the message purposes in the IPD model, and are continually re-used. The Message actor (see Figure 8) activates all the N instances of Message through a broadcast send (see the bSend[] channel array) which specifies the particular message to be sent to players among: INIT, PLAY and ADAPT. On the other hand, following a PLAY or ADAPT, a DONE message is sent to the Manager.

To give an idea of the practical impact of the new message design, a simulation of 2,500 time steps now lasts in less than 17 minutes of wall-clock time.

### 5.3 Actor automata

Figures 8 and 9 portray respectively the automaton of the Manager and that of the Player. An actor automaton 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.

An explicit Main automaton is avoided by having that initially the Manager starts its execution by initialising itself and then by broadcasting (through a bSend channel) an INIT message to all the players. Then a next synchronisation is sent which causes the StepMessage (see Figure 7) to be activated. On receiving a NEXT message, the Manager executes its basic cycle of operations as stated in Figure 3. As one can see from Figure 8, in the normal locations Waitdone1 and Waitdone2, the N done messages from the players are awaited and counted.

The finite state machine of the Player automaton in Figure 9, first awaits for an INIT message, then it expects a PLAY and then an ADAPT message. The response actions are purposely confined in the functions init_player(), d0_play() and do_adapt(). After a PLAY or ADAPT, a DONE message is sent to the Manager.

**Figure 8** The Manager actor automaton (see online version for colours)

![Diagram](image)

**Figure 9** The Player actor automaton (see online version for colours)

![Diagram](image)
Both automata in Figures 8 and 9 are finite state machines. In reality, only the Manager needs a finite state machine behaviour. Since the Manager asks the player reactions in the right order (first for INIT, then for PLAY and then for ADAPT repeatedly) the Player automaton can more easily be modelled according to the ‘pure reactive’ style of agents shown in Figure 10 (see also the ping/pong actors in Figure 1). This version of the Player actor illustrates the ‘input determinism’ principle required by UPPAAL SMC: from a location only one edge can exit with the next received message. As a general rule, a pure reactive actor model can be organised according to a couple of locations such as Receive and Select. To the edge outgoing Receive is attached a command with a non-deterministic selection of the received message id in an input msgsrv synchronisation. Then in the committed location, the particular message id is checked and a cascade of committed Select locations is entered, which realises the corresponding reaction. At the reaction end, the Receive location is re-entered.

Figure 10  A pure reactive Player automaton (see online version for colours)

As a consequence of the above message and actor design:

a The control structure of the cloud of sent messages gets automatically realised by UPPAAL SMC through the activation/deactivation of 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.

Point (b) is a consequence of the fact that message reactions in an actor automaton (see Figures from 8 to 10) are realised by one or more committed locations. On the other hand, an activated message involves only urgent (see Figure 6) or normal locations with a clock invariant (see Figure 7) which have lower priority with respect to committed locations.

5.5 Other global declarations

The following are some other global declarations useful for the operation of the IPD model.

Variables like totpayoff and period_payoff[] are examples of ‘decoration variables’, useful for extracting interesting properties of the model. At the start of each Manager cycle (see the reset() function in Figure 8), the payoff values of each player along its four links are set to

clock now; //current simulated time

Player links are conventionally named according to the Manhattan neighbourhood. However, only for the persistent toroidal grid (PTG) the four players stored in the neighbour matrix ngh[pid][link] effectively mirror the partners of a given player at north, east, south and west positions in the grid. In the case of a random network topology (PRN-Persistent Random Network, and TRN-Temporary Random Network), the four links are in reality established randomly, by ensuring the situations of duplicate neighbours and having the player as a neighbour of itself are avoided. The array last[pid][link] stores the last move taken by a player with respect to a certain link.

At the initialisation time (see the initialise() function of Manager in Figure 8) last is set to undefined (the NONE value is used). The matrix payoff[pid][link] stores the payoff of a player following the four moves with each partner at the four links. The period_payoff[pid] holds the average payoff of a given player along its four links. The scalar totpayoff accumulates the payoff averaged over the entire population. The constant PAYOFF matrix furnishes the earned payoff of each partner of a pair of competing players.
NONE. All of this is a key to handle ‘reciprocity’, i.e., the fact that if A is playing against B, both A and B have to store their achieved result. This in turn avoids to repeat unnecessary moves of B vs. A. The reset() function also updates the totpayoff variable by adding to it the average payoff computed over the entire population. At each time step, the ratio between the totpayoff with current time now (except for the case when now is 0.0) furnishes the instantaneous fitness value of the IPD model (see Figure 11).

Figure 11  Fitness value at current time step

double fitness(){
    if(now==0.0) return 0.0;
    return totpayoff/now;
} //fitness

6  Experimental work

The IPD model was thoroughly investigated, using the MITL temporal logic, about its functional and temporal behaviour. For simplicity, default statistical options of UPPAAL SMC were mostly used where, e.g., the probability uncertainty is $\varepsilon = 0.05$. A lower value for $\varepsilon$ would ensure a more accurate confidence interval estimation at the cost of increasing significantly the number of required runs and the corresponding wall clock time.

6.1 Debugging queries

Some queries were preliminarily issued to check specifically the functional behaviour of the model. In particular, the following query was used to check that effectively, after the Manager broadcasts a PLAY message to all the players, the Manager then will receive N DONE messages, stating that all the player actors have completed their moves at current time.

$$\Pr[\leq 5] Manager(N).WaitDone _L & & Manager(N), p = N - 1$$

This query asks about the probability that being at time 1 the state predicate Manager(N).WaitDone2 true, following this, and in 0 time (as required by the until time interval [0, 0]), it would occur that the p counter of the Manager reaches the value N – 1. UPPAAL SMC responds, after 738 runs, with a CI of [0.95, 1] 95% of confidence.

The following query was used to check that effectively, after a PLAY message, a player will receive, at the same time, an ADAPT message from the Manager. As an example, the query was directed to the process instance Player(0).

$$\Pr \left[ \exists [1, 1] \text{ Player(0).msg = PLAY U}[0, 0] \text{ Player(0).msg = ADAPT} \right]$$

Again, UPPAAL SMC proposes a CI of [0.95, 1] 95%, after 738 runs.

6.2 Transient behaviour

Some experiments were devoted to observing the shape of the average period-payoff in the first few time steps. The following query was used:

simulate $1[<=50]avg._payoff()$

Results for the PTG model in the presence of adaptation errors are shown in Figure 12. In reality, the basic behaviour holds with or without adaptation errors and also for PRN and TRN topologies. Figure 12 confirms the indications in Cohen et al. (1999, p.24, 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 12  Average payoff in the first 50 steps – with adaptation errors (see online version for colours)

The steady-state analysis was directed to studying the possible emergence of a cooperation regime in two cases:

a Scenario-1 – each game per period consists of the four moves and the last move is recorded and influences the next move in the same game. Of course, strategy adaptations are propagated from a period to the next one.

b Scenario-2 - the last move of a game affects the first move in the next period. In the scenario-1 the array last (see Section 5.5) is reset at each period. In the scenario-2 the array last is never reset.

6.3 Emergence of cooperation in the scenario-1

The six models PTG, PRN and TRN with and without adaptation errors, were repeatedly studied using the query:
where the average payoff at the population level, the fitness and the average values of the agent probabilities p and q are monitored. Figures from 13 to 15 refer to the case where no adaptation errors are introduced. Figures from 16 to 18 depict collected results when adaptation errors plus noise are possible at each period.

As expected, the evolution of IPD without errors tends to have small fluctuations and to stabilise soon depending on the initial random distribution of strategies to agents and ultimately on ‘who talks with whom’. As a consequence, both the attainment of a cooperative regime (see Figure 13) and of a defective regime (see Figure 14) are possible for a persistent interaction network. For a transitory network like TRN, instead, the defective regime was always observed.

In the case adaptation errors are admitted, both persistent interaction networks PTG (Figure 16) and PRN (Figure 17) testify the attainment of a cooperative regime with a steady state fitness() value of about 2.5. The TRN network, though, always confirms the achievement of a defective regime. As shown in Figures 16 to 18, the adaptation errors imply a greater fluctuation in the payoff values, caused by the dynamic creation of new strategies \((y, p, q)\) in the agent population.
The attainment of a cooperative regime in the presence of adaptation errors, was further checked by using the query:
\[
\Pr[\leq 2500]([\text{now} < 2000] \text{fitness} \geq 2.3)
\]
which estimates the probability that after 2,000 time steps a strictly cooperative \text{fitness()} value would emerge. The threshold of 2.3 was chosen according to Axelrod et al. (2002, p.343), as a minimal expectation following the value of 2.25 of an initial random population.

For both PTG and PRN networks, after 36 runs, there emerged a confidence interval of [0.902606, 1] with 95% of confidence, indicating a high occurrence probability. For the TRN network, instead, a CI of [0, 0.0973938] 95% was proposed, thus witnessing the event has a very low probability of occurrence.

Changing the probability uncertainty from \(\varepsilon = 0.05\) (default) to \(\varepsilon = 0.01\), implies UPPAAL SMC uses more runs and proposes the confidence intervals shown in Table 1, thus even better mirroring the event of achievement of a cooperative regime is almost guaranteed for persistent networks. The CI for TRN testifies the attainment of a defective regime.

In the case of absence of adaptation errors, the query \(\Pr[\leq 2500]([\text{now} < 2000] \text{fitness} \geq 2.3)\) proposes for PTG, after 376 runs, a CI of [0.300046, 0.399964] 95%, and for PRN, after 383 runs, a CI of [0.339927, 0.43987] 95%. All of this indicates that in the case of no errors, cooperation is possible although with a smaller probability of occurrence.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Observed CIs with adaptation errors, (\varepsilon = 0.01)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Confidence interval (95%)</td>
</tr>
<tr>
<td>PTG</td>
<td>[0.980044, 1]</td>
</tr>
<tr>
<td>PRN</td>
<td>[0.980044, 1]</td>
</tr>
<tr>
<td>TRN</td>
<td>[0, 0.019956]</td>
</tr>
</tbody>
</table>

It is worth noting that the above presented detailed probability estimations are not part of previous work, e.g., in Axelrod et al. (2002).

6.4 Emergence of cooperation in the scenario-2

The same experiments discussed in the previous Subsection 6.3 were repeated in the case of scenario-2,
where the last move of a player in a given period, is
exploited and always affects the next move of the player in
the next period. Such kind of exploration was not covered in
Axelrod et al. (2002). However, these experiments too are
capable of throwing some light about the emergence of
cooperation in a different context of application of IPD. For
simplicity, in Figures 19 to 21 only the fitness() behaviour,
with the adaptation errors allowed, is reported.

As one can see from Figures 19 and 20, also in the
scenario-2 a cooperation regime can ultimately be reached
for the PTG and PRN networks, although now with a more
modest cooperation level (about 2.1). The TRN network
continues to exhibit only the defective regime (Figure 21).
In particular, the query:

\[ \Pr([-2500 \leq \text{fitness()} < 2.0]) \]

proposes the confidence intervals reported in Table 2.

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Observed CIs in the presence of adaptation errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Confidence interval (95%)</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>PTG</td>
<td>[0.902606, 1]</td>
</tr>
<tr>
<td>PRN</td>
<td>[0.87865, 0.978536]</td>
</tr>
<tr>
<td>TRN</td>
<td>[0, 0.0973938]</td>
</tr>
</tbody>
</table>

The same query, launched on the models without adaptation
errors, suggested the confidence intervals shown in Table 3.

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Observed CIs without adaptation errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Confidence interval (95%)</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>PTG</td>
<td>[0.18312, 0.282947]</td>
</tr>
<tr>
<td>PRN</td>
<td>[0.222444, 0.322213]</td>
</tr>
<tr>
<td>TRN</td>
<td>[0, 0.0973938]</td>
</tr>
</tbody>
</table>

Table 2 confirms also in the scenario-2 a cooperation
regime can be finally attained, when adaptation errors are
admitted. Without errors (see Table 3) a cooperation regime
cannot be excluded, but has a lower occurrence probability.
Such probabilities are also smaller than the same
probabilities observed in the scenario-1 under the same
setting.

6.5 Model validation

The UPPAAL SMC models were also implemented in Java
as another source of validation. Figures 22 and 23 show the
observed fitness() from the Java models respectively in the
scenario-1 and in the scenario-2, with adaptation errors
included. The curves are the average of 100 runs.

It is beyond the scope of this paper to discuss the
meaning and interpretation of IPD results from a social or
economic point of view. For these concerns the reader is
referred to the Axelrod (1980a, 1980b, 1984) work.
However, the developed IPD models and the obtained
experimental results confirm the intuition that link
persistence, i.e., playing with the same partners throughout
the game, is the key for cooperation because it favours
player’s trustiness. All of this has an obvious interpretation
nowadays when one considers people’s 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.

7 Conclusions

This paper proposes a novel approach to modelling and
analysis of MAS which is based on a lightweight actor
model (Cicirelli and Nigro, 2013, 2016a, 2016b), statistical
model checking (SMC) (Larsen and Legay, 2016; Younes
et al., 2006) and the UPPAAL SMC tool in particular
(David et al., 2015). As a testbed the approach was applied
to modelling and analysis of a large and adaptive version of
the IPD game (Axelrod et al., 2002), thus demonstrating its practical usefulness.

Performance issues are currently limited by UPPAAL SMC which can exploit only one core of a multi-core machine, whereas multiple runs could in principle be spawned in parallel.

Extension of the research aims to:

- adapt the IPD model so as to investigate different player strategies proposed in the literature
- experiment with ‘mechanism design’ in general multi-agent-systems (Wooldridge, 2009)
- complete the development of a software tool which automates the translation of a Java actor model into the terms of UPPAAL SMC
- specialise the approach toward model checking temporal knowledge and commitments in MAS (Al-Saqqar et al., 2015)
- refine the approach for model checking (using both exhaustive verification and statistical model checking) timing constraints in distributed probabilistic real-time MAS. A preliminary experience is described in (Nigro and Sciammarella, 2017b).

References


