Qualitative and Quantitative Model Checking of Distributed Probabilistic Timed Actors

Libero Nigro, Paolo F. Sciammarella

DIMES - Engineering Department of Informatics Modelling Electronics and Systems Science University of Calabria, 87036 Rende (CS) - Italy
l.nigro@unical.it, p.sciammarella@dimes.unical.it

Abstract

This article proposes an approach to modelling and analysis of distributed probabilistic timed actors. The approach rests on a lightweight infrastructure of actors -THEATRE- whose design aims to favoring the development of predictable time-dependent applications. Adopted actors are thread-less and their evolution is transparently regulated by a customizable control layer which has a reflective link with the application. A THEATRE system consists of a collection of interacting computing nodes (theatres) each one hosting a sub system of local actors. The control layers of the various theatre components coordinate each other so as to enforce a common global time notion (real or simulated time). The abstract THEATRE modelling language can be reduced in a case to UPPAAL, which opens to the analysis of the functional/non-functional aspects of a distributed system. A key factor of the reduction process concerns the possibility of making both a non-deterministic analysis of an actor model (checking that something, e.g., an event, *can* occur), and a quantitative evaluation of system behavior by statistical model checking of the same model (e.g., estimating the probability for an event to occur). The paper describes the THEATRE architecture and introduces a real-time case study which is used as a running example throughout the paper. The operational semantics of THEATRE is provided and the proposed reduction of THEATRE actors on top of UPPAAL detailed through the chosen example. Some experimental results are reported about qualitative and quantitative analysis of the case study. Finally, conclusions are presented with an indication of further work.

Keywords: Modelling and verification, model checking, statistical model checking, actors, asynchronous message passing, timing constraints, probabilistic behavior, Uppaal.

1. Introduction

The work described in this paper is concerned with a model-driven methodology for the development of distributed real-time systems such as cyber-physical systems [1, 2]. Ensuring the correctness of such systems is challenging and strongly depends on the use of formal tools for modelling and analyzing the system behavior earlier in a design, so as to assess the fulfillment of functional and temporal requirements.
The starting point of the methodology is the Actors computational model [3] which is a well-known formal framework [4] suited for modeling and implementation of untimed distributed systems based on asynchronous message passing. Each actor exposes a message interface and hides an internal data status which can only be modified by responding to messages. Incoming messages get buffered into a local mailbox of the actor, from where they are extracted, one at a time, by an underlying control thread of the actor, and eventually processed. Being not based on shared variables and associated lock mechanisms for excluding data races, the actor concurrent model is intrinsically less incline to common pitfalls of classical multi-threaded programming [5]. However, problems are tied to message delivery to actors which can follow complex interleaving, whose consequences on system behavior are required to be predicted before of an implementation. Non-deterministic behavior of threaded actors makes them less suited to a time-sensitive context such as real-time, whose essence is predictability [6], or discrete-event simulation which requires high-performance execution [7]. For example, achieving a simulation control engine with threaded actors typically implies the simulation engine (that is a specialized actor) delivers a message to an applicative actor and needs an explicit message back from the activated actor to witness message processing was completed thus the engine can possibly advance the simulated time and proceed with the next message delivery and so forth. All of this introduces an obvious overhead at each message (event) occurrence.
In the last years many efforts and tools have emerged addressing specifically the modelling and analysis of distributed timed, possibly probabilistic, actors. A significant state-of-the-art example is represented by the Rebeca modelling language [8] along with its probabilistic and timed extension (PTRebeca) [9]. Rebeca represents an interpretation of the classical actors model [3], formally

defined through a structural operational semantics [9]. Different tools were implemented for the analysis of both functional and temporal behavior of a system. Analysis tools are based on a preliminary translation of a PTRebeca model into the terms, e.g., of a Timed Markov Decision Process (TMDP) and its properties studied using the PRISM model checker [10] or the IMCA (Interactive Markov Chain Analyzer) [11], or targeting the model to Erlang with the timed McErlang tool [12] used for model checking or, to avoid state explosion problems, by statistical model checking activities [13]. Despite their value, these efforts lack, in our opinion, of an effective link to the implementation phase of a system.

The work described in this paper claims that a full model-driven methodology can be established by using lightweight (thread-less) actors in a reflective control-sensitive framework [14] which clearly separates application concerns from crosscutting control aspects affecting message scheduling and dispatching. The adopted actor framework is named THEATRE. A system consists of a collection of computing nodes (logical processes, LPs, or theatres) where each theatre hosts a subset of applicative actors plus a (transparent) control structure. The control structure can be tailored to the application needs and can manage a specific time notion (simulated or real-time). The entire system life-cycle is addressed: a same model can be transitioned without distortions (*model continuity* [2]) from the analysis phase based on model checking and/or simulation, down to design and real implementation in Java. The overall process mainly depends, at each phase, on a different concretization of message processing and on the replacement of the regulating control structure.

The THEATRE actor framework was recently successfully applied, e.g., to the performance evaluation of a new version of the minority game [15], to the real-time control of power management in a smart micro grid [2], to the support of modelling and analysis of general complex multi-agent systems [16, 17]. A library of prototyped control forms can be found in [14, 18], and includes controls for distributed simulation and distributed real-time operation where a time server is used to orchestrate the various theatres thus homogenizing the local times through the achievement of a common global reference time for the theatres.

A side benefit of the adopted control-based actor framework relates to the possibility of customizing also the programming style. For example, in [14] the actor behavior, which in general follows the pattern of a finite state machine, is captured in one single *handler()* method which receives the next message and updates the actor status and possibly sends new messages to acquaintances.

In this paper, following the PTRebeca [9] approach, a more intuitive and readable programming style is advocated, where messages are handled by corresponding *message server* methods, which are reflectively activated by the control engine. All of this corresponds to the design and realization of new specific control forms (see Section 7) inspired by the timing model of PTRebeca. The goal is to capture the PTRebeca programming and timing model into the terms of our control-based and time-predictable actor framework.

The original contribution of this paper consists in tailoring the abstract modelling language of THEATRE according to PTRebeca, providing its formal operational semantics, and defining a reduction of a THEATRE model into the terms of the Timed Automata [19] of the UPPAAL popular toolbox [20, 21] so as to support, for a same model, both qualitative non deterministic analysis through the exhaustive model checker, and quantitative simulation-based analysis through the Statistical Model Checker [22] [21]. Current paper significantly extends the preliminary experience described in [23] where only statistical model checking activities were enabled. A major difference from [23] consists in the replacement of *dynamic message templates* [21] which were used to model message exchanges among actors, with a statically dimensioned pool of message automata, which are dynamically activated and, after their dispatch, are reset so as to be reused again. In addition, the new message TA more faithfully reproduces the timing model of PTRebeca (see later in this paper).

The paper is structured as follows. The next section describes some related work. Then the basic concepts of THEATRE, related to both the "in-the-large" (architectural view) and the "in-the-small" (application view) aspects are discussed. After that the abstract modelling syntax of THEATRE is furnished together with a real-time modelling example which is used as a case study throughout the paper. The paper goes on by presenting a formal structural operational semantics for THEATRE. After that the proposed reduction process of THEATRE onto the timed automata of UPPAAL is presented, using the modelling example to clarify the transformation details. Then the paper illustrates the analysis activities which can be carried out on a reduced model, by focusing both on the non-deterministic analysis (model checking) and the quantitative analysis (statistical model checking) of the chosen case study, also considering the partitioning concerns. After that the paper discusses the implementation status of THEATRE and some methodological guidelines. Finally, the conclusions are presented together with direction of further work.

2. Related Work

The actors paradigm suited to general untimed distributed and asynchronous systems was originally developed by C. Hewitt in [24] as an agent-based language. The computational model was then refined by Agha in [3] as a formal language where the actor is the fundamental unit to express a concurrent/distributed computation. Examples of popular actor frameworks include Scala [25] and Erlang [26] which can be used for the development of general-purpose distributed applications.
Actors were first extended toward real-time system requirements through the *RTSynchronizer* abstraction mechanism [27] which declaratively captures the interaction patterns existing in group of actors. An RTSynchronizer filters exchanged messages in a group of actors and manages them according to the "*safe progress, unsafe block*" semantics: if a message cannot be delivered for a timing issue, it will be kept pending in the synchronizer; otherwise the message is allowed to be consigned to its relevant actor. The aim of an RT-Synchronizer is to fulfill actor timing constraints.
A framework for the schedulability analysis of distributed real-time actor systems modelled by Colored Petri Nets was proposed in [28] which is based on the RTSynchronizer concept. RTSynchronizers were also exploited in [29] for designing and implementing a Time Warp synchronization mechanism [7] adequate for high performance networked simulations.
The RTSynchronizer mechanism was specialized in [30] where QoS synchronizers are introduced for controlling and enforcing quality of service to a multimedia application. A similar experience was pursed in [31, 32] using the actor model described in [28].
A real-time actor language was proposed in [33] where each message send can have a *release time* and a *deadline* which are relative times measured from the activation time of the method raising the send operation. The two time quantities respectively express the *earliest* and the *latest* times constraining the delivery of the message to its recipient actor. Timed Rebeca [9] uses similar time requirements (see also later in this paper), called *after* and *deadline*. It is intended that the message cannot be delivered before *after* time units are elapsed, and should be dispatched before *deadline* time units are elapsed. Timed Rebeca also allows to specify an expected duration of a

message processing by a *delay* operation.
The control-sensitive actor-based architecture described in [14] enables messages to be time-stamped by their *due* delivery time. Assessing/enforcing timing constraints rests the responsibility of a customizable control structure whose operation follows the semantics of the RTSynchronizer mechanism.
The above mentioned literature of timed actors is exploited in the work described in this paper where the Rebeca timing model and programming style [9] is embedded in the THEATRE architecture which favors time predictability in distributed timed actors. THEATRE is described in the next section. The goal is supporting in a seamless way modelling, analysis and implementation (in Java) of a distributed (possibly probabilistic) real-time actor system. Similar goals are advocated, in the context of Rebeca, by Marjan Sirjani in a recent paper [34] where the concept of *friendliness* as a synthesis of *usability* and *faithfulness* is recommended. Usability refers to the easy of use of a modelling language for the modeller working on a domain specific application. Faithfulness, instead, is concerned with the degree with which a design/implementation strictly obeys to the analyzed model.
The analysis phase of a THEATRE model can conveniently exploit the Timed Automata [19] in the context of the popular UPPAAL toolbox [20]. First the model checker of UPPAAL can be used for exhaustive, qualitative evaluation of an actor model. All of this is challenging due, e.g., to the asynchronous message passing communication model which implies buffering of messages and then the risk for the state graph to easily end into a state explosion. State explosions are avoided by the use of the statistical model checker (SMC) of UPPAAL [21] which does not build the model state graph and uses instead simulation runs whose memory demand are linear with the model size. The SMC permits a quantitative property checking of system behavior, which in any case is of great value from the engineering practical point of view.
The reflective control architecture of THEATRE can be related to the Ptolemy approach [35]. Ptolemy is characterized by its flexibility in supporting different models of computation in actor-based applications. In a case, Ptolemy offers a lightweight, not thread-based, notion of actors where the exchange of messages is asynchronous and where a *Director* (similarly to the concept of control machine in THEATRE as described in this paper) can supervise the actual delivery of messages to actors. The Ptolemy toolbox features an extensive set of tools for the analysis and synthesis of a modelled system.

3. THEATRE concepts

3.1. Architectural view

A THEATRE system (see also Fig. 1) consists of a collection of interacting theatres (logical processes or LPs) each allocated for execution onto a computing node (a core or a JVM instance). A theatre hosts an *application layer* populated by a subsystem of local actors, a *control layer* which provides to local actors the basic services of *message scheduling* and *dispatching*, and a *network layer* which interfaces the theatre with its peers using a communication network and a reliable transport layer.

The control layer is realized by a *control machine* component which has

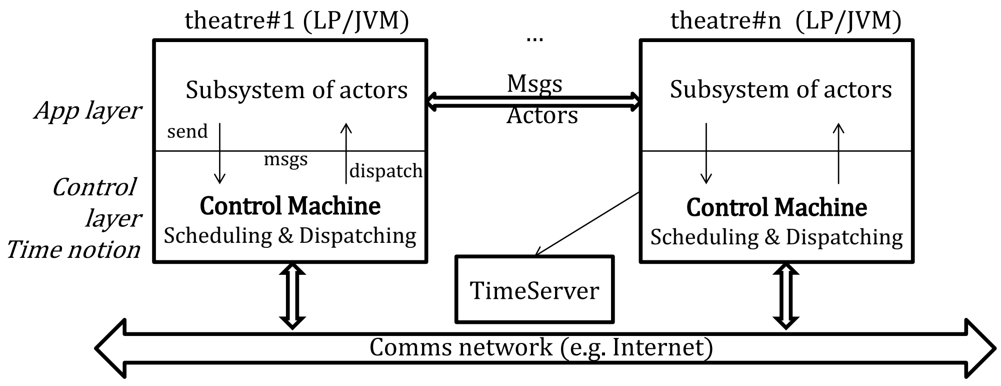


Figure 1: A THEATRE system

a reflective link with the application layer and regulates its behaviour in a transparent manner: each message send is first captured by the control machine and put in a *cloud* of *sent but not yet dispatched messages*. Major responsibilities of a control machine are the management of the cloud of sent messages and of a particular *time notion* (real time or simulated time). A control machine repeats a basic *control loop*. At each iteration, a message is selected, if there are any, in the cloud of messages, possibly according to an application dependent strategy, and dispatched to its destination actor. The control machine is founded on the *macro-step semantics* of messages [36]. Only one message at a time, in a theatre, can be dispatched and processed by its recipient actor. When the message processing is completed, the control loop is re-entered and the story continues. Therefore, a *cooperative concurrency* schema, determined by message interleaving, is ensured within

a same theatre. Actors can instead be executed in parallel if they belong to distinct theatres allocated, e.g., on different physical CPUs.
The transport layer can be directly based on TCP sockets (see section 7). However, other solutions were experimented as well. For example, in [14, 2] the JADE agent infrastructure [37] was used as a middleware providing naming, messaging and network services; in [38] the Terracotta services were exploited, in [39] the Globus middleware was used etc.
A Theatre system (see Fig. 1) can admit a *time server* (allocated to a given theatre) which is in charge of managing a *global time notion*, e.g., the common "real time" in a cyber-physical system, of the global simulation time in a distributed simulation. A suitable protocol is defined among the theatres and the time server for the exchange of control information.
As a final remark, it should be noted that the Theatre architecture can logically reproduce the classical Actors model [3] by allocating one actor per theatre.

3.2. Abstract modelling language

In the following, the "in-the-small" modelling aspects of Theatre actors are tailored according to the PTRebeca modelling style [9]. In Fig. 2 it is shown the assumed abstract modelling language.
Theatres are abstracted as processing units $pu_1, pu_2, \ldots, pu_N$ each one hosting a disjoint set of actors. The meta symbols $< \cdots >$ embody a block of elements, $|$ denotes alternatives, $[\ldots]$ envelop an optional text, superscripts $+$ and $*$ respectively mean repetition of the left symbol one or more times, and zero or more times. Furthermore, the notation $< e >^*$ or $< e >^+$ subsumes a comma separated list of elements e. T is a primitive type (int or boolean); C is a class name; v is a variable or value; a denotes an actor instance; pu denotes a processing unit; e means either an arithmetic or boolean expression; m is a method name.
A Theatre model can admit environmental declarations which introduce scenario parameters. For the rest it consists of a collection of actor classes. An actor class, besides encapsulated local variables, including acquaintances, specifies the message servers (*msgsrv*) which provide reactions to corresponding messages. An actor can be a main actor if it defines the main method which is used for bootstrapping purposes. The main instantiates actors and puts a model into operation.
For generality, the initialization of actors does not rely on a built-in constructor but is delegated to a first message like *init*, which carries the initialization

Model ::= Env* Class*
Env ::= **env** T v = literal;
Class ::= **actor** C{VarDcl* MsgSrv*[Main]}
VarDcl ::= T < v[=literal] >$^+$; |C <a>$^+$;
Msgsrv ::= **msgsrv** m(<T v|C a>*){Stmt*}
Main ::= **main**() { InstanceDcl* Stmt*}
InstanceDcl ::= a = C();
Stmt ::= v = e; |v =?(e <,e>$^+$; |v= ? (ep:e <, ep:e >$^+$); |
    if(e){Stmt*}[**else** {Stmt*}] | Send | **delay**(v) | **move**(a,pu)
Send ::= a.m(<e>*)[**after**(v)][**deadline**(v)]

Figure 2: THEATRE abstract modelling language

data (both acquaintance actors and primitive data values). The main actor can receive an acknowledgment message (e.g., a *done* message) from actors to state the initialization is terminated. In a typical setting, the main is launched on a default processing unit (pu) which is then inherited by created actors. Following the initialization, actors can be moved to different pus, using the *move* operation.
The asynchronous *send* operation can optionally be tagged by an *after* and a *deadline* time. Such values are relative to the instant in time the send was issued. When missing, *after* evaluates to 0, whereas *deadline* defaults to ∞. In a message server, *self* identifies the executing actor. The predefined function *now*() returns the current time. As in PTRebeca, all the time quantities are assumed to be int.
Statements include a *delay* operation which expresses a duration of (a code segment of) a message server. Also a *delay* time parameter is an int and is relative to the current time value.
Both a non-deterministic $v =?(e_1, e_2, \ldots, e_n)$ and a probabilistic $v =?(p_1 : e_1, p_2 : e_2, \ldots, p_n : e_n)$ assignment are available. p_i are probabilistic weights with the constraint $\sum p_i = 1$. The result of expression e_i is assigned to v with probability p_i. In the non-deterministic assignment, the probabilistic weights are implicitly equal to $1/n$.
It is worth noting that a THEATRE model can be straightforwardly be expressed in Java syntax, where actors are programmed as classes inheriting, directly or indirectly, from an Actor base class (see Section 7) which exposes all the fundamental services: send, now(), etc. Actor classes rely only of the

default language constructor implicitly used at each actor creation.

3.3. A modelling example

Fig. 3 depicts a Theatre model of a dependable real-time toxic gas sensing system (TGSS), adapted from [9]. The system is devoted to controlling a lab environment wherein there is a working scientist. In the environment a toxic gas level, changing with time, can assume a critical level thus putting the life of the scientist to a severe risk. One or more sensors in the lab periodically measure the gas toxicity, and transmit the gas level to a controller for a decision. Periodically, the controller checks if the scientist life can have a danger, in which case the scientist is asked to immediately abandon the lab. The scientist must acknowledge in a timely manner a danger signaling message. Not receiving the expected ack, the controller requires the intervention of a rescue team. If the rescue reaches in time the lab, it informs the controller that the scientist was saved. If the controller does not receive this notification, it means the scientist is dead. The model consists of 6 types of actors: Environment, Sensor, Scientist, Rescue, Controller and Main, together with some scenario parameters (see Fig. 3) which affect actor operation. Fig. 4 summarizes the message exchanges among the actors.
The TGSS model is configured by the Main which creates the remaining actors and sends them an init message with initialization parameters (e.g., the acquaintances for each actor). After that, each actor is moved to a specific theatre (processing unit) thus establishing, as an example, a maximum parallelism setup. Every actor replies to the Main with a done message. When all the replies are received (see the done() msgsrv in Fig. 3), the main actor starts model execution by sending a changeGasLevel() to the environment actor with CHANGING_PERIOD as the *after* time, a checkGasLevel() to each sensor (Fig. 3 considers only one sensor) and a checkSensors() to the controller.
The periodic arrival of a checkGasLevel() message causes the environment to randomly change the gas level (with probability 0.98 the normal level 2 is kept, but in the 2% of the cases the abnormal value 4 is established). If a dangerous level occurs, a die() message is sent to the scientist with an *after* time of SCIENTIST_DEADLINE, which is the assumed amount of time within which the scientist should be saved. The controller gets regularly informed of the gas level by the sensor(s) which periodically ask the environment for the current gas level through a giveGas() message.

```
//scenario parameters
env int SCIENTIST_DEADLINE=14;
env int SCI_ACK_DEADLINE=3;
env int RESCUE_DEADLINE=5
env int NET_DELAY=1;
env int CONTROLLER_CHECK_DELAY=3;
env int SENSOR_PERIOD=2;
env int CHANGING_PERIOD=5;
env int RESCUE_DELAY=2;
env int NR_SENSORS=1;

actor Environment{
    //acquaintances
    Scientist sc;
    //state vars
    int gasLevel=2; //4 is dangerous
    bool meetDangerousLevel=false;
    msgsrv init(Main m, Scientist s){
        sc=s; m.done();
    }//init
    msgsrv changeGasLevel(){
        if(gasLevel==2)
            gasLevel=?(0.98:2,0.02:4);
        if(gasLevel>2 &&
           !meetDangerousLevel){
            sc.die()
            after(SCIENTIST_DEADLINE);
            meetDangerousLevel=true;
        }
        self.changeGasLevel()
            after(CHANGING_PERIOD);
    }//changeGasLevel
    msgsrv giveGas(Sensor sender){
        sender.doReport(gasLevel);
    }//giveGas
}//Environment

actor Sensor{
    //acquaintances
    Environment en;
    Controller co;
    msgsrv  init(Main m, Environment e
     Controller c){
        en=e; co=c; m.done();
    }//init
    msgsrv checkGasLevel(){
        en.giveGas(self);
    }//checkGasLevel
    msgsrv doReport(int gasL){
        if(?(0.99:true, 0.01:false)){
            co.report(gasL)
                after(NET_DELAY);
            self.checkGasLevel()
                after(SENSOR_PERIOD);
        }
    }//doReport
}//Sensor

actor Scientist{
    //acquaintances
    Controller co;
    //state vars
    bool isDead=false;
    bool isOutEnv=false;
    msgsrv init( Main m, Controller c ){
        co=c; m.done();}
    msgsrv leftEnv(){
        if(!isDead) isOutEnv=true;
        else isOutEnv=false;
    }//leftEnv
    msgsrv abortPlan(){
        if(?(0.90:true, 0.10:false)){
            if(!isOutEnv && !isDead){
                isOutEnv=true;
                co.ack() after(NET_DELAY);
            }
        }
    }//abortPlan
    msgsrv die(){
    if(!isOutEnv) isDead=true;
    else isDead=false; }//die
}//Scientist

actor Rescue{
    //acquaintance
    Controller co;
    msgsrv init(Main m, Controller c){
        co=c; m.done(); }//init
    msgsrv go(){
        delay(RESCUE_DELAY);
        co.rescueReach()
        after(NET_DELAY+RESCUE_DELAY)
        deadline(RESCUE_DEADLINE-NET_DELAY
                +RESCUE_DELAY)
    }//go
}//Rescue

actor Controller{
    //acquaintances
    Scientist sc;
    Rescue re;
    //state vars
    bool danger=false;
    bool abortSent=false;
    bool sciAlive=false;
    msgsrv init(Main m,
       Scientist s, Rescue r){
        sc=s; re=r; m.done();
    }//init
    msgsrv report(int value){
        if(value>2)
            danger=true;
    }//report
    msgsrv rescueReach(){
        sciAlive=true;
        sc.leftEnv();
    }//rescueReach

    msgsrv checkSensors(){
      if(!sciAlive){
       if(danger){
        if(!abortSent){
        sc.abortPlan()
          after(NET_DELAY);
        self.checkScientistAck()
          after(SCI_ACK_DEADLINE);
        abortSent=true;
        }
       }
       self.checkSensors() after(
       CONTROLLER_CHECK_DELAY);
      }
    }//checkSensors
    msgsrv ack(){ sciAlive=true; }
    msgsrv checkScientistAck(){
        if(!sciAlive)
        re.go() after(NET_DELAY);
    }//checkScientistAck
}//Controller

actor Main{
    Environment en;
    Scientist sc;
    Rescue re;
    Controller co;
    Sensor se1;
    int cnt=0;
    msgsrv done(){
        cnt++;
        if(cnt==1){
            move(en,1);
            sc.init(self,co); }
        else if(cnt==2){
            move(sc,2);
            re.init(self,co); }
        else if(cnt==3){
            move(re,3);
            co.init(self,sc,re); }
        else if(cnt==4){
            move(co,4);
            se1(self,en,co); }
        else if(cnt==5){
            move(se1,5);
            en.changeGasLevel() after(
            CHANGING_PERIOD);
            se1.checkGasLevel();
            co.checkSensors();
        }
    }//done

    msgsrv main(){
        move(self,0);
        //create actors
        en=Environment();
        sc=Scientist();
        re=Rescue();
        co=Controller();
        se1=Sensor();
        en.init(self, sc);
    }//main
}//Main
```

Figure 3: A Theatre model for the toxic gas sensing system, adapted from [9]

After that the sensor receives a doReport() message from the environment with the gas level and transmits it to the controller through a report() message. The controller periodically checks the sensor(s) and in the case a dangerous situation is sensed, it sends an abortPlan() message to the scientist with NET_DELAY as the *after* time so as to "immediately" ask the scientist to abandon the lab. The scientist is expected to send back to the

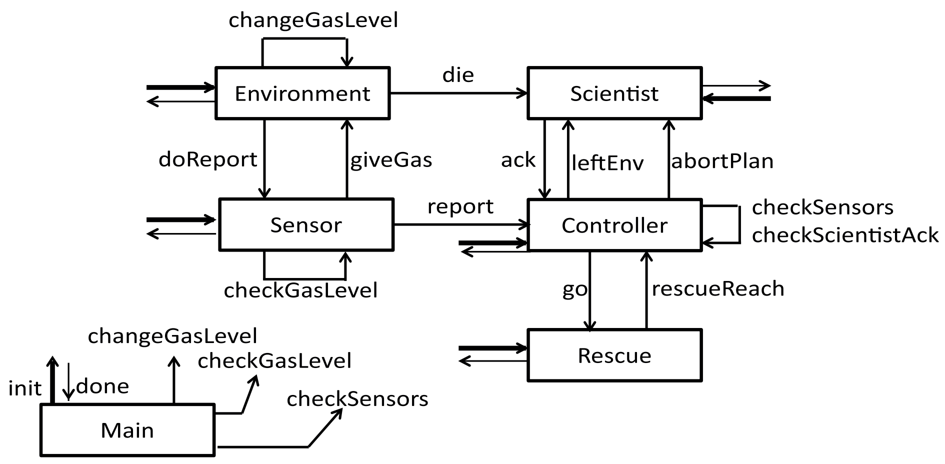


Figure 4: Message exchanges in the toxic gas sensing system model

controller an ack() message. The controller checks the arrival of the scientist ack by sending to itself a checkScientistAck() message whose *after* time is SCI_ACK_DEADLINE, i.e., the maximum time allowed to the scientist for replying. In the case the ack() message is not received in time, the controller delegates the rescue team to go to the lab to try to save the scientist. The sciAlive variable in the controller is put to true as soon as an ack from the scientist is received or when the rescue communicates it reached the scientist. A true value of sciAlive only indicates that *possibly* the scientist is saved. The problem is that message delivery times and non-deterministic/random aspects of the model can imply, e.g., the rescue team arrives late and find the scientist already dead. For example, a sensor which receives a doReport() message from the environment, can be found working in 99% of the cases, but in 1% of the cases the sensor is not working and then it cannot inform the controller about a dangerous gas level. The scientist model includes a probabilistic behavior when it receives an abortPlan() message. The abortPlan() can be perceived with a probability of 0.90. This in turn can force the controller to activate the rescue team because it raises the probability of violating the SCI_ACK_DEADLINE.

Besides any similarity with the PTRebeca modelling syntax [9], significant semantics differences are due to the operational and timing model of THEATRE (more details in the next section) where, e.g., global time is assumed and message servers cannot be preempted nor suspended. In PTRebeca each actor owns its local notion of time. The existence of global time simplifies and makes it uniform the interpretation of message time constraints across actors. In addition, whereas a delay(d) statement blocks the actor thread for d time units to express, during modelling and analysis, the duration of a code segment, in THEATRE a delay request is just another asynchronous operation like the non-blocking send. A delay(d) operation causes the corresponding processing unit of the requesting actor to become occupied for d time units. No messages can be delivered to actors sharing an occupied processing unit. The processing unit becomes again free at the end of the delay duration. Therefore, the delay duration parameter has to be added, during modeling and analysis phases, to any following send operation in the message server (see the go() msgsrv in the Rescue actor in Fig. 3). Further details about the semantics of THEATRE are given in the next section.

4. An operational semantics of THEATRE

As in [9], a structural operational semantics of THEATRE in the style of the transition rules of Plotkin [40] and Kahn [41] is provided in the following. First some basic data structures are introduced.

- E, a set of environments (an environment maps variable names to their values);
- M, an unordered bag of messages (cloud of sent but not yet dispatched messages);
- D, an unordered bag of delays (cloud of set but not yet expired delays);
- C, configuration, a set of N theatres abstracted as a set of processing units $pu_1, pu_2, \ldots, pu_N$, paired with their associated *free* or *occupied* (delayed) status. Each pu consists of a set of actors which share the pu for the execution: $pu_i \cap pu_j = \varnothing$, $i \neq j$. The function $pu(a)$ returns the processing unit of the actor a. A particular configuration associates one pu (theatre) to each distinct actor (*maximal parallelism*);
- *now*, a variable holding the current global time.

Typically, E is the union of all the local stores of the actors Ai: $\cup \rho_{Ai}$. A local store also holds the predefined name (noun) *self* which denotes the currently executing actor.
A *sent message*, i.e., an item of M, is a tuple: $< receiver, m, \overline{args}, AF, DL >$ where

- *receiver* is an actor name;
- m is a message name of the receiver, which identifies a method/msgsrv which handles the message;
- $\overline{args}$ is the list of arguments of m;
- AF and DL are the absolutized values of the *after* and *deadline* timing attributes of the message, that is: $AF = now + after$, and $DL = now + deadline$. It is recalled that *after* and *deadline* are relative to the send time. When omitted, *after* amounts to 0, and *deadline* to ∞.

It is worth noting that in the case a message server needs to know the identity of the sender actor, the sender information is assumed to be explicitly transmitted as an argument.
A *delay object*, i.e., an item of D, is a tuple $< receiver, ET >$ where

- *receiver* is the actor name who is delaying;
- ET is the absolutized expire time of the delay, that is: $ET = now + duration$ where *duration* is the amount of the delay.

The configuration C maps processing units to their statuses: $C[pu \rhd free]$, $C[pu \rhd occupied]$. In the default configuration, at the creation of a new actor bound to *varname* in current store of actor *self*, it occurs: $C[pu(self) = pu(self) \cup \{varname\}]$, that is the new actor is grouped together with the *self* actor. A *move* operation such as $move(a, pu')$ is equivalent (see also later in this section) to $pu(self) = pu(self) \setminus \{a\} \bigwedge pu' = pu' \cup \{a\}$.
A system state is a tuple: $< E, M, D, C, now >$. The stepwise evolution of a theatre system is characterized by a relation $\rightarrow$ thus: $< E, M, D, C, now > \rightarrow < E', M', D', C', now' >$. Basic steps correspond to a message dispatch or to a delay expiration, both of which are executed by the scheduler (control machine). Each step is then realized by an atomic block of micro-steps which correspond, e.g., to the statements which compose a msgsrv method, and which consume no time.
Due to the use of probabilistic constructs in a Theatre model (see the assignment operations in Fig. 2 which can affect the temporal behavior of an actor by probabilistically defining the time duration of an *after* or *deadline* or of a *delay* clause) the transition relation evolves in general a system state according to a probability distribution which assigns probability values to

reachable states. In the following, though, such a probability distribution is only handled implicitly, i.e., the probability weight of transitions is not specified but left implicitly defined by the executing steps. Reasons for doing this are simplicity and the chosen goal of analyzing a THEATRE model through the UPPAAL SMC [21] Statistical Model Checker, hence through simulations, and not by a Probabilistic Model Checker which would require, e.g., a Timed Markov Decision Processes model, as advocated in PTRebeca using the IMCA (Interactive Markov Chain Analyzer) tool [9]. On the other hand, probabilities are ignored and turned into non-determinism when a THEATRE model is analyzed through an exhaustive symbolic model checker like UPPAAL [21, 42]. Anyway, the provided semantic rules could be extended to specify the probability distribution of transitions using an approach similar to that shown in [9].
The representation of the step-wise evolution of a THEATRE model can concretely be based on two specific relations: $\xrightarrow{i}$ and $\xrightarrow{d}$, the first one being concerned with an instantaneous *pure-action* state change, the second one with a *pure time-advancement* operation, needed to reach the time of the (or one of the) next most imminent event in the system, that is either a msg-dispatch or a delay-expiration.

4.1. Transition rules $\xrightarrow{i}$ and $\xrightarrow{d}$

The transition relation $\xrightarrow{i}$ specifies an instantaneous action transition (it consumes no time). Two important occurrences of this transition are the selection and dispatching of an eligible message (see Fig. 5) and the execution of the associated message server in the receiver actor, or the processing of an expired delay which makes again free a given processing unit (Fig. 6).

A message dispatch is *eligible* as soon as its processing unit (pu) becomes free and *now* has reached its AF but it is not beyond its DL. When multiple messages are eligible for dispatch and/or multiple delays are ready to expire, one event is chosen non-deterministically, therefore executing the message-dispatch or delay-expiration rule. As a consequence of a message-dispatch, E', M', D' and C' are the result of the following changes: (i) modification to the local store ρ_{Ai}, as an effect of the execution of assignment statements in the message server body, (ii) new sent messages scheduled in M, (iii) some delays scheduled in D, (iv) new actors created whose local store is added to E and whose configuration (execution location or processing unit) is reflected in C. Moreover, in the new C the processing unit of A_i is occupied.

$(msg - dispatch)$

$$\frac{\rho_{Ai} \notin E \quad (A_i, m, \overline{args}, AF, DL) \notin M \quad now \leq DL \quad now \geq AF \quad C[pu(A_i) \rhd free] \qquad \rho_{Ai}(m), \rho_{Ai}[\overline{par} = \overline{args}], E, M, D, C, now) \xrightarrow{i} (\rho'_{Ai}, E', M', D', C'[pu(A_i) \rhd occupied], now)}{(\{\rho_{A_i}\} \cup E, (A_i, m, \overline{args}, AF, DL) \cup M, D, C, now) \xrightarrow{i} (\{\rho'_{A_i}\} \cup E', M', C', now)}$$

the local store ρ_{Ai} and the dispatch message $(A_i, m, \overline{args}, AF, DL)$ are supposed to be extracted respectively from E and from M for clarity of presentation

Figure 5: Message dispatch rule

$(delay - expiration)$

$$\frac{(A_i, ET) \notin D \quad now == ET \quad C[pu(A_i) \rhd occupied] \qquad (E, M, D, C, now) \xrightarrow{i} (E, M, D, C'[pu(A_i) \rhd free], now)}{(E, M, (A_i, ET) \cup D, C, now) \xrightarrow{i} (E, M, D, C', now)}$$

for clarity the delay object (A_i, ET) is supposed extracted from D

Figure 6: Delay expiration rule

$(time - progress)$

$$\frac{d_1 = min_M\{AF - now\} \quad d_2 = min_D\{ET - now\} \quad d = min\{d_1, d_2\} \qquad now < min_M\{AF\} \quad now < min_D\{ET\} \qquad (E, M, D, C, now) \xrightarrow{d} (E, M, D, C, now' = now + d)}{(E, M, D, C, now) \xrightarrow{d} (E, M, D, C, now')}$$

Figure 7: Time progress rule

$(send)$

$$(varname.m(\overline{args})\ after(a)\ deadline(d), \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho_{self}, E, \{(\rho_{self}(varname), m, eval(\overline{args}, \rho_{self}), AF = a + now, DL = d + now)\} \cup M, D, C, now)$$

$(delay)$

$$(delay(d), \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho_{self}, E, M, \{(self, ET = now + d)\} \cup D, C[pu(self) \triangleright occupied)]$$

$(assignment)$

$$(x = e, \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho'_{self}[x = eval(e, \rho_{self})] \cup E, M, D, C, now)$$

$(non\ deterministic - assignment)$

$$(x = ?(e_1, e_2, \ldots, e_n), \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho'_{self}[x = eval(e_l, \rho_{self})\ l \in [1, n]] \cup E, M, D, C, now)$$

$(probabilistic - assignment)$

$$(x = ?(p_1 : e_1, p_2 : e_2, \ldots, p_n : e_n), \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho'_{self}[\alpha \in [0, 1),\ x = eval(e_l, \rho_{self},\ \alpha \in [\textstyle\sum_{j=0}^{l-1} p_j, \sum_{j=0}^{l} p_j), p_0 = 0, \sum_{j=0}^{n} p_j = 1] \cup E, M, D, C, now)$$

$(create)$

$$(varname = A(), \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho_{self}[varname = a] \cup E, M, D, C[pu(self) = pu(self) \cup \{a\}], now)$$

$(move)$

$$(move(a, pu_t), E, M, D, C, now) \xrightarrow{i} (E, M, D, C[pu(a) = pu(a) \setminus \{a\}, pu_t = pu_t \cup \{a\}], now)$$

Figure 8: Statement rules - 1st part

As a consequence of the $delay - expiration$ rule in Fig. 6, the configuration C will be changed to $C' = C[pu(Ai) \rhd free]$ thus (possibly) enabling the dispatch of messages in M whose receiver is A_i or any other actor belonging to $pu(Ai)$. It should be noted that, for a selected expiring delay, the value of now is certainly $now == ET$. This is a consequence of the fact that the

($cond_1$)

$$\frac{eval(e, \rho_{self}) = true \; (S_1, \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho'_{self}, E', M', D', C', now)}{(if(e) \; then \; S_1 \; else \; S_2, \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho'_{self}, E', M', D', C', now)}$$

($cond_2$)

$$\frac{eval(e, \rho_{self}) = false \; (S_2, \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho'_{self}, E', M', D', C', now)}{(if(e) \; then \; S_1 \; else \; S_2, \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho'_{self}, E', M', D', C', now)}$$

($sequence$)

$$\frac{\begin{array}{c}(S_1, \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho'_{self}, E', M', D', C', now), \\ (S_2, \rho'_{self}, E', M', D', C', now) \xrightarrow{i} (\rho''_{self}, E'', M'', D'', C'', now)\end{array}}{(S_1; S_2, \rho_{self}, E, M, D, C, now) \xrightarrow{i} (\rho''_{self}, E'', M'', D'', C'', now)}$$

($msgsrv - end$)

$$\frac{\begin{array}{c}(self, ?) \notin D \\ < E, M, D, C, now > \xrightarrow{i} < E, M, D, C[pu(self) \rhd free], now >\end{array}}{< E, M, D, C, now > \xrightarrow{i} < E, M, D, C', now >}$$

Figure 9: Statement rules - 2nd part

execution of a msgsrv is supposed (during analysis) to be instantaneous and that the duration of a delay is established as an asynchronous event.
The $time - progress$ rule in Fig. 7 is responsible for the time advancement (now update). It is ensured that the advancement of the global time can occur only when no eligible event exists. Therefore, the minimum occurrence time of the (or one of) most imminent message dispatch or delay expiration

is evaluated and *now* is advanced of that minimum.
The message dispatch rule implies the atomic and instantaneous execution of the message server body which is effectively carried through multiple $\xrightarrow{i}$ relations. Each such a transition is devoted to the execution of a single statement (micro-step) of the msgsrv body. As a consequence, different relations are provided in Fig. 8 and Fig. 9, each corresponding to a distinct basic action admitted by the THEATRE modelling language (see Fig. 2).
The $probabilistic - assignment$ rule in Fig. 8 deserves some further comment. The probability interval $[0, 1)$ is first split into n sub-intervals (slots): $[0, p_1)$,$[p_1, p_1 + p_2)$, $[p_1 + p_2, p_1 + p_2 + p_3)$, ..., $[p_1 + p_2 + ... + p_{n-1}, 1)$, which are respectively associated to the expressions e_1, e_2, ..., e_n. Then a random value α in $[0, 1)$ is generated using a common uniform random generator. The slot to which α belongs selects the expression whose value is assigned to the left-hand variable.
In the $non\ deterministic - assignment$ rule (see Fig. 8) any expression e_1, e_2, $\ldots, e_n$ has the same chance to be selected for the assignment. Its meaning is equivalent to that of the $probabilistic - assignment$ rule where the probabilistic weights are all equal to $1/n$.
It should be noted that a processing unit is occupied at the time a message dispatch occurs, directed to an actor assigned to the processing unit (see the $msg - dispatch$ rule in Fig. 5), and it is freed at the message server end (see the $msgsrv - end$ rule in Fig. 9) provided no delay operation was raised during the message server. All of this ensures the macro-step semantics of messages within a same theatre.

5. A reduction of THEATRE onto UPPAAL

The structural operational semantics of the previous section was interpreted in a case in the context of the UPPAAL toolbox [20, 21] using timed automata (TA) [19]. UPPAAL was chosen because its powerful modelling language provides clocks to measure relative times (durations), atomic actions, normal locations where an automaton can stay an arbitrary time or a limited time constrained by an invariant, urgent/committed locations which have to abandoned without passage of time (with the committed which have priority over the urgent locations), (possibly urgent) unicast/broadcast channels, integer and boolean data variables, double variables (recognized only by the SMC), C-like data structures and functions, etc., which facilitate the translation of actors. Table 1 recapitulates basic correspondences.

The micro-step statements of a message server can easily be achieved by atomic actions attached to the edges outgoing from a committed location (see, e.g., Fig. 12). For example, a probabilistic assignment is reproduced by a branch point whose dashed exiting edges (Fig. 12) are labelled by the probabilistic weights which drive the selection. The mapping of actors, messages and delays on (possibly stochastic) timed automata (TA) could be based on the use of *dynamic timed automata* as permitted by latest version of Uppaal [21]. Dynamic automata were experimented, e.g., in [23]. However, they are only supported by the statistical model checker. As a consequence, a different and more efficient solution is proposed in this paper which consists in the use of a static configured pool of TA, where each automaton instance can dynamically be activated by a channel synchronization and, after its termination, it is reset so as to be reused again. In the toxic gas sensing system, a fixed number of non-terminating actors is considered, and dynamic "creation/consumption" operations are tied respectively to the non-blocking message send and message delivery to actors, and to the setup and expiration of a delay.
A critical point concerns the attainment of the control-based message scheduling and dispatching capable of ensuring the macro-step semantics of messages on a processing unit (see section 3.1).
Concrete steps of the reduction process from Theatre to Uppaal will be detailed by considering the translation of the toxic gas sensing system modelled in section 3.3.

5.1. Scenario parameters

The environmental scenario parameters are easily handled by corresponding global constants of the Uppaal model.

```
//scenario parameters
const int SCIENTIST_DEADLINE=14, SCI_ACK_DEADLINE=3, RESCUE_DEADLINE=5, NET_DELAY=1,
CONTROLLER_CHECK_DELAY=3, SENSOR_PERIOD=2, CHANGING_PERIOD=5, RESCUE_DELAY=2,
NR_SENSORS=1;
```

5.2. Entity naming

A fundamental step is to assign a unique identifier to each existing actor, message, delay and processing unit. All of this can be achieved by introducing some sub-range integer types as follows. Sub-range types are also a key for implicit instantiation of actors, messages and delays at system configuration time.

Table 1: Main mappings from Theatre to Uppaal

Theatre	**Uppaal**
actor	(stochastic) timed automaton
message	timed automaton
delay	timed automaton
timing constraint (in a message or delay)	clock invariant on a normal location
message reception (in an actor)	normal location without an invariant
message delivery	broadcast channel synchronization
asynchronous message send	broadcast channel synchronization
delay setup	broadcast channel synchronization
message server	cascade of committed locations
control machine	established by timed and non-deterministic behaviour of sent message and set delay TA

```
const int EN=0,SC=1,RE=2,CO=3,MAIN=4;
const int N=5+NR_SENSORS; //number of actors
//actor subrange types
typedef int[EN,EN] env_id;
typedef int[SC,SC] scie_id;
typedef int[RE,RE] resc_id;
typedef int[CO,CO] cntr_id;
typedef int[MAIN,MAIN] main_id;
typedef int[MAIN+1,N-1] sens_id;
typedef int[0,N-1] aid;
//message identifiers
const int INIT=0, CHANGE_GAS_LEVEL=1, GIVE_GAS=2, CHECK_GAS_LEVEL=3, DO_REPORT=4,
DIE=5, ABORT_PLAN=6, LEFT_ENV=7, GO=8, REPORT=9, RESCUE_REACH=10, CHECK_SENSORS=11,
ACK=12, CHECK_SCIENTIST_ACK=13, DONE=14;
const int MSG=15; //number of distinct messages
typedef int[0,MSG-1] msg_id; //possible message ids
//delay identifiers
const int DELAY=1; //number of distinct delays
typedef int[0,DELAY-1] did; //delay ids
//PU resources
const int NPU=N; //number of PUs - maximal parallelism
typedef int[0,NPU-1] pu_id; //pu identifiers
bool avail[pu_id]; //availability status of pus
pu_id pu[aid]; //actors to pus mapping
```

The bool $avail[pu\_id]$ array stores the status (true $\rightarrow$free, false $\rightarrow$ occupied) of each processing unit. The array $pu[aid]$ specifies the processing unit upon which a given actor is allocated. The configuration is established by *move*() operations.

5.3. Message and delay pools

Depending on the fact if messages carry or not arguments, the two classes of Message and VoidMessage are distinguished. A corresponding pool must then be introduced with a statically defined dimension. For the case study translated model the following declarations hold. It should be noted that the main actor purposely initializes every actor and expects its done message before proceeding with the next initialization.

```
const int MI=NR_SENSORS; //number of Message instantiations
typedef int[0,MI-1] mid; //msg instance ids
const int VMI=N; //number of VoidMessage instantiations
typedef int[0,VMI-1] vmid; //vmsg instance ids
const int DI=1; //number of delay instantiations
typedef int [0,DI-1] did; //delay instance ids
bool avVM[VMI]; //pool of void messages
bool avM[MI]; //pool of messages
bool avD[DI]; //pool of delays
```

Pools are assumed to be initialized to all true. When a (void)message (or a delay) is requested, the first available message (or delay) in the relevant pool is returned. Functions nVM() and nM() respectively return the index of the first available message in the corresponding pool. Similarly, the function nD() returns the index of the next available delay instance. Would a pool be exhausted, -1 is returned instead, which causes an obvious runtime array access error which stops the operation of the model checker. The occurrence of such one error clearly indicates a pool was insufficiently dimensioned.

5.4. Asynchronous message passing and delay setting

The following broadcast channel arrays make it possible to send (schedule) a message directed to a given target actor, carrying (*send*[]) or not (*vSend*[]) arguments. The channel synchronization has the effect of activating a corresponding automaton instance.

```
broadcast chan send[mid];
broadcast chan vSend[vmid];
meta aid A; //actor id
meta msg_id M; //message id
meta int AFTER, DEADLINE, DELAY; //timing attributes of a message send
```

A send operation needs an index in the relevant pool of messages, the name (aid) of the destination actor, the specification of the involved message id, and the *after* and *deadline* relative times of the message send. The message

index in the pool is typically achieved by invoking either the nM() (for a Message instance) or nVM() (for a VoidMessage instance) function. The remaining information are provided to the send operation by using the meta variables [20] A (for the actor id) M (for the message id) and AFTER and DEADLINE. Meta variables do not take part to the data component of the states of the model state graph. They thus can contribute to the efficiency of the model checking process. However, values of meta variables are significant *only* during a channel synchronization. After the synchronization, they are undefined. The two functions lb(after), ub(after,deadline) are provided for defining respectively the value of AFTER when DEADLINE is missing (infinite), or the values of both variables.
In a similar way, the asynchronous setting of a delay can be achieved by synchronization on a delay channel:

```
broadcast chan delay[did];
```

A delay channel is identified by the index of an available delay instance (typically provided by the nD() function). Setting a delay instance requires also the identity of the actor requesting the delay, and the duration of the delay. The A meta variable is used for the actor id, the DELAY meta variable provides the amount of the delay. The function d(delay) can be used to assign a value to the DELAY variable.

5.5. Message delivery and arguments

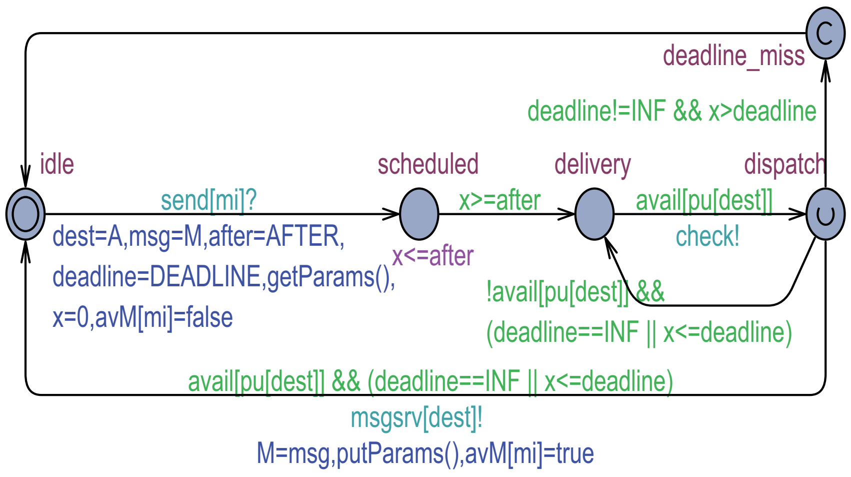
The following global declarations support the delivery of a message to an actor, along with some possible carried arguments.

```
broadcast chan msgsrv[aid];
const int MAX_ARGS=3;
int args[MAX_ARGS]; //buffer of msg arguments
```

An output synchronization on a channel like msgsrv[a]! causes the delivery of the message specified by meta variable M to actor a, thus activating the corresponding message server. Carried arguments of message M can be retrieved from the args[] buffer.

5.6. The Message automaton

Fig. 10 shows the Message automaton (parameterized as: const mid mi) which is provided of arguments transmitted through the args[] buffer. Message uses a local clock x. A Message instance is activated through a send operation. As a consequence, the automaton passes from the idle location to the scheduled location, and it is flagged as unavailable into its belonging pool. The function getParams() copies the global args[] buffer onto a local params[] buffer. Function putParams(), at final dispatching time, copies back the local params[] on to the args[] buffer from which they are finally retrieved by the target actor. From the AFTER and DEADLINE variables the Message instance gets the *after* and *deadline* times of the message. The message cannot be delivered before *after* time units are elapsed from the sending time. Such a time is awaited in the scheduled location through an invariant based on the after time. When the *after* time is elapsed, the automaton moves to the delivery location. However, for the dispatching to occur it is necessary that the message becomes eligible (see section 4), that is the processing unit of the destination actor is free and the time is not greater than the deadline time. As soon as the message automaton finds the processing



deadline_miss
deadline!=INF && x>deadline
idle
scheduled
delivery
dispatch
send[mi]?
dest=A,msg=M,after=AFTER,
deadline=DEADLINE,getParams(),
x=0,avM[mi]=false
x>=after
x<=after
avail[pu[dest]]
check!
!avail[pu[dest]] &&
(deadline==INF || x<=deadline)
avail[pu[dest]] && (deadline==INF || x<=deadline)
msgsrv[dest]!
M=msg,putParams(),avM[mi]=true

Figure 10: The Message timed automaton

unit is available, it abandons the delivery location and moves to the dispatch location where one of three events can occur: (a) the current time is found beyond the message deadline and therefore the message is no longer valid and must be discarded (the deadline_miss location is reached); (b) for non-determinism, a different message is dispatched whose processing occupies the processing unit, and the automaton must come back to the delivery location; (c) the message is found effectively eligible and a synchronization over the *msgsrv*[*dest*] channel is generated toward the destination actor (the identity of the message msg is assigned to the meta variable M). Message dispatching

causes the "consumed" message instance to be returned to its pool and the idle location is re-entered.
A subtle point in Fig. 10 concerns the transfer from delivery to dispatch. In order to ensure the dispatch location is immediately entered as the processing unit becomes free, the following urgent and broadcast channel *check* is used:

```
urgent broadcast chan check;
```

The synchronization signal check! obliges, due to the urgent character of the channel, the automaton to immediately exit from delivery.
Another subtle point in the design of Fig. 10 regards the dispatch location which was made urgent but not committed. This way, the automaton can remain in delivery (without passage of time) would a message server of a different actor be triggered into execution on the same processing unit. Recall (see also Table 1) that a message server is realized by a cascade of committed locations which have priority on urgent locations, and consume no time. At the end of this alternate message server, the message instance in dispatch can still proceed with its own dispatching or it is forced to come back to delivery if the processing unit was just occupied by a delay operation.
As a final remark, the timed automaton of Fig. 10, rests on the relative time model of UPPAAL: the *after* and *deadline* times of a message are directly used as relative times. This is due to the use of clock x which measures the time elapsed since its last reset (see the edge from idle to scheduled in Fig. 10).
The VoidMessage automaton, not shown for brevity, is identical to Message except that it does not manage any arguments, so it does not use the getParams()/putParams() functions and does not have a local *params*[] array.

5.7. Delay automaton

A delay is scheduled through a synchronization on a *delay*[*did*] channel which activates a Delay instance. The Delay timed automaton in Fig. 11 admits the parameter: const did di. It uses a local clock x which is reset when the delay is set, and measures the elapsed time until expiration. During the delay, the processing unit of the requesting actor is kept occupied. Time is awaited in the scheduled location in Fig. 11 through an invariant on the delay amount. When the delay expires, the automaton must move from scheduled to idle. It should be observed, though, that at the last time of the delay expiration the scheduled location becomes equivalent to an urgent location: it must be exited before time can go on, but it has no priority with respect to another urgent location.

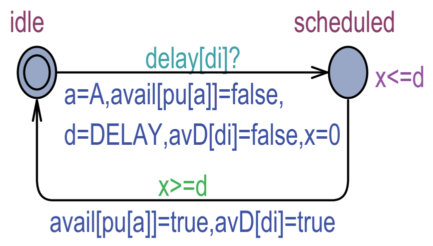


Figure 11: The Delay automaton

5.8. An actor automaton

The model of an actor (see, for example, the Environment automaton of the toxic gas sensing system in Fig. 12 which is parameterized as: const env_id self) can easily be built in UPPAAL around two basic locations: Receive and Select. Receive is often also the initial location. It is a normal location, meaning the actor can stay in Receive an arbitrary amount of time until the reception of the next message.

When a message arrives, that is, a synchronization over the channel $msgsrv[self]$ is received, with the message id being communicated through the M meta variable, the actor moves to the Select location which is committed. From Select, the particular arrived message is checked, and its processing (message server) launched through, in general, a cascade of committed locations. When the processing of the message server is complete, the automaton comes back to Receive for it to be ready for a next message to be received, and so forth. As one can see from Fig. 12, the execution of a message server (message reaction) can exploit branch points for a probabilistic behaviour. For instance, when a CHANGE_GAS_LEVEL message is received, and the gasLevel is currently equals to 2 (normal level), with 98% of probability it remains to 2, but with 2% of probability it is raised to 4 (abnormal level). INIT and GIVE_GAS are two examples of messages carrying some arguments. When INIT is received, in args[0] is transmitted the identity of the Main actor, and in args[1] the identity of the scientist actor (acquaintance) is specified. The Environment actor directly sends to args[0] a reply DONE message, and stores into its local variable sc the identity of the scientist. Similarly, when receiving a GIVE_GAS message, sent by a sensor, in args[0] is contained the sender identity. The message server replies by sending a DO_REPORT message to the sender sensor and puts into args[0] the current value of the gas level.

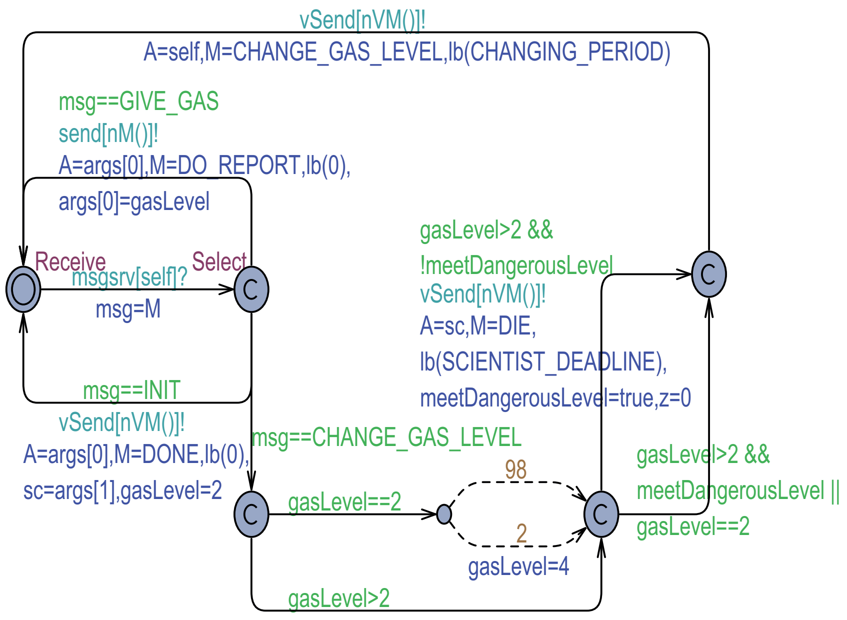


Figure 12: The Environment actor automaton

Each message server (i.e., message response or reaction) can directly be achieved from the abstract model of the actor. Since a message server is atomic and consumes no time, multiple data updates can be put on a same edge of the message server. The exact composition of the message server depends on the number of messages which are sent within it. Indeed, only one message send (channel synchronization) can be specified per edge.

5.9. Preservation of THEATRE semantics

Into a reduced Uppaal model of a Theatre system, the cloud of sent messages (see the M data structure in section 4) is represented, at any instant in time, by all the activated message automata. Similarly, the cloud of delays (see the D data structure in section 4) is composed by all the delay instances which were activated but are not yet expired. The reduction process, and in particular the design of message, actor and delay timed automata, directly complies with the structural operational semantics of Theatre. In fact:

- it there exists a notion of global time, implicitly advanced by Uppaal;
- at each instant in time, the most imminent event (message dispatch or delay expiration) occurs;

- when multiple events exist which can occur at the same time, one of them is chosen non-deterministically.

A key point of the reduction process is concerned with the attainment of the macro-step semantics (see section 3.1), i.e., no new message can be dispatched in a theatre (i.e., processing unit) before the current dispatch is completely processed. Toward this it should be noted that:

- an event occurrence (message dispatch or delay expiration) is always provided by an urgent location of an automaton;
- a message server, in an actor, is achieved by a cascade of committed locations.

As a consequence, a message server is atomic and instantaneous. In addition, the use of committed locations guarantees message server termination *before* any new event (message dispatch or delay expiration) can occur. Whereas this result does not impede message server parallelism (i.e., message dispatches occurring at the same time, although they are executed one at a time) into distinct processing units, it genuinely ensures, in a same theatre, the macro-step semantics of messages.
As a further remark, there is no need to explicitly occupy the processing unit during a message server execution (see the $msg-dispatch$ rule in section 4). The processing unit, in fact, remains unavailable during the message server as a consequence of the use of committed locations. The processing unit needs to be occupied explicitly only in response to a delay operation.
In the light of the above observations, it emerges that the proposed reduction process automatically realizes the behaviour of the control machine components of a THEATRE model.

5.10. Translated UPPAAL model of the toxic gas sensing system

Fig. 13 to Fig. 17 show all the remaining UPPAAL actor timed automata for the case study (the Environment automaton is shown in Fig. 12). Each actor model is parameterized with only one parameter of its corresponding sub_range type (see section 5.2).
In the Sensor model, the arrival of a DO_REPORT message from the Environment, is accompanied by the gas level as an argument in args[0]. Such a value is not stored locally. In the case the sensor is correctly working, the args[0] value is then transmitted as part of a REPORT message sent to the

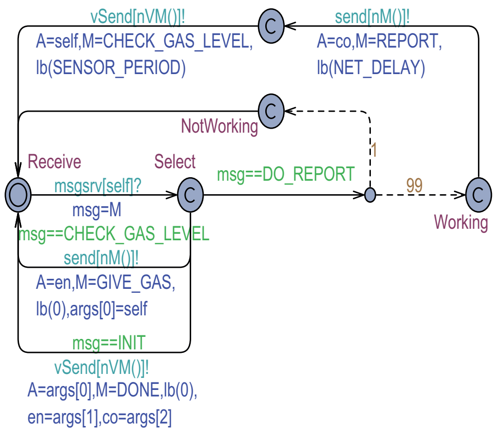


Figure 13: The Sensor actor automaton

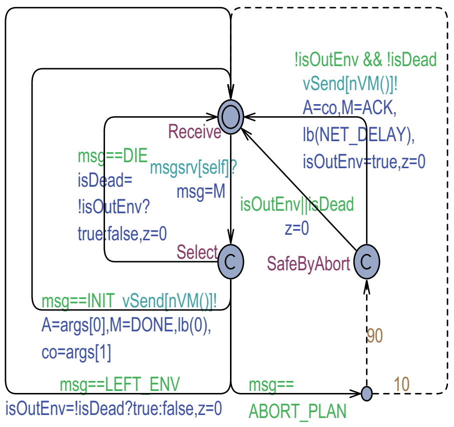


Figure 14: The Scientist actor automaton

controller. Care was taken in the model in Fig 3 and its UPPAAL reduction toward avoiding the introduction of unnecessary data variables which would complicate the model checker exhaustive verification activities.

It is worth noting, in the Main automaton, that sensor ids range from $MAIN + 1$ to $MAIN + NR\_SENSORS$.
The following is the system configuration line:

```
system Environment,Sensor,Scientist,Rescue,Controller,Main,Message,
       VoidMessage,Delay;
```

Of each template automaton a number of instances is automatically created as determined by the corresponding sub-range type (see section 5.2). The Main automaton can easily be adapted to work with a different number of sensors, or with a different grouping of actors to processing units.

6. Analysis of a THEATRE model reduced into UPPAAL

In general, the verification of a real-time THEATRE model aims to check *safety* properties (i.e., a bad state is never reached), *liveness* properties (i.e., a good state is eventually reached, possibly within a timing constraint), and *reachability* properties (i.e., assessing that a certain state is reachable in the model behaviour). In the following, the toxic gas sensing system (TGSS) model reduced in to UPPAAL will be thoroughly analysed by both non deterministic analysis, that is qualitative evaluation of system properties through exhaustive model checking, and by simulation, that is quantitative evaluation of system properties through statistical model checking. Each kind of

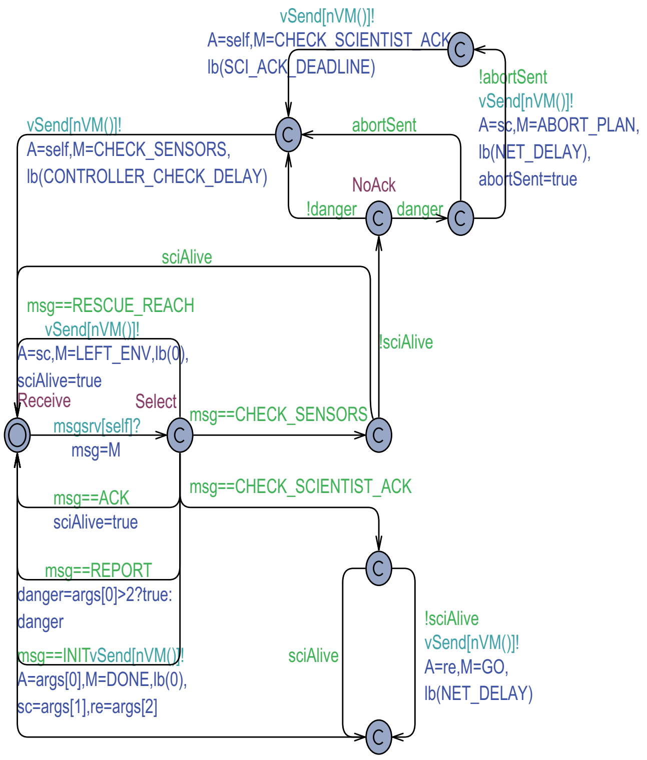


Figure 15: The Controller actor automaton

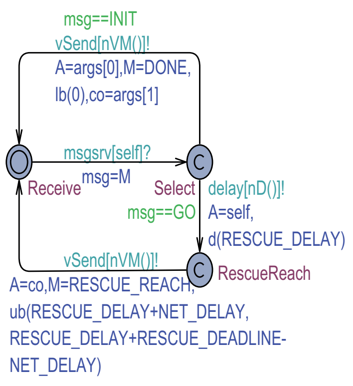


Figure 16: The Rescue actor automaton

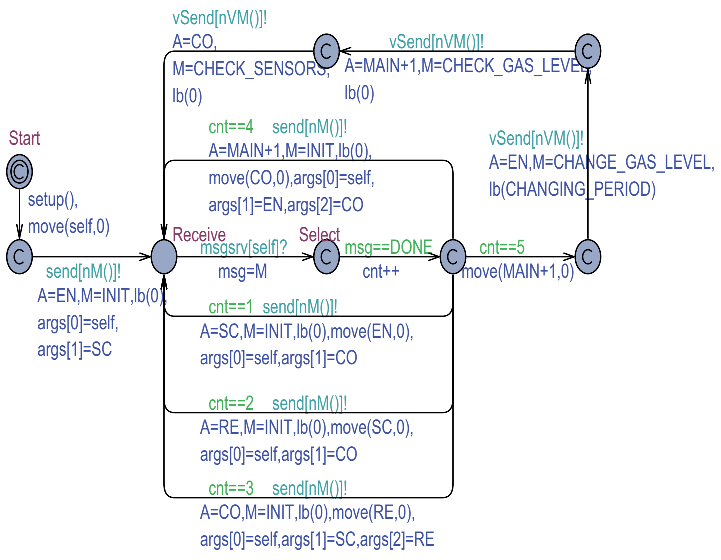


Figure 17: The Main actor automaton

Table 2: Space/time demands of the A[] !deadlock query on the TGSS model

Number of sensors	WCT (sec)	RAM Peak (MB)
1	4	38
2	2244	10842

analysis exploits a corresponding temporal logic language for the formal expression of system properties (*specification*) to check. The exhaustive model checker of Uppaal relies on a subset of the Timed Computation Tree Logic (TCTL) [20] which does not admit formula nesting. The statistical model checker of Uppaal uses, instead, an extended version of the Metric Interval Temporal Logic (MITL) [43]. All the experiments were carried out on a Linux machine, Intel Xeon $CPU E5-1603@2.80GHz$, 32GB, using Uppaal 4.1.19 64bit.

6.1. Qualitative non-deterministic model checking

In this case the complete state graph of a reduced Theatre model is built and queries are verified on the state graph. A preliminary concern was to check the absence of deadlocks in all the states of the TGSS model, under maximal parallelism, through the query:

$$A[]\ !deadlock$$

which is satisfied. This in turn guarantees the number of generated message and delay instances (of the Message, VoidMessage and Delay template automata, see section 5.3) is sufficient to cope with the model needs. An insufficient number of such instances would cause the model checker to immediately stop its operation for an illegal access to a non-existent array position. Another critical issue is concerned with the possible loss of a message when the time goes beyond the message deadline, which can be unacceptable in the context of a real-time application. The following queries were used.

$$E <>\ exists(m : mid)Message(m).deadline\_miss$$

$$E <>\ exists(m : vmid)VoidMessage(m).deadline\_miss$$

They respectively check for the existence of at least one state of the state graph where a message instance can be found in the deadline_miss location (see Fig. 10). Both queries terminate by saying they are not satisfied.

Due to the asynchronous message exchanges among actors, a reduced THEATRE model can easily suffer of scalability problems for state explosions. Table 2 collects some space/time demands of the TGSS model when one or two sensors are used. The wall clock time (WCT) and peak memory usage observed when checking deadlock absence are shown.

In the following, the TGSS model with one sensor will be used for the remaining verification work. A fundamental time parameter is the SCIENTIST_DEADLINE (see section 3.3) which constrains, following a detected dangerous level of the toxic gas, the end-to-end delay (*response time*) within which the scientist could be saved. Such a value mainly depends on the sensor period and the sensor correct behavior. Since during the exhaustive verification, any probabilistic behavior is turned into a non-deterministic one, to properly check the SCIENTIST_DEADLINE, the TGSS model was slightly modified to ensure the sensor is always correctly working (if the sensor could not work, it would there exists a path in the model state graph in which the sensor is always not working and then there would be no upper bound for the SCIENTIST_DEADLINE capable of saving the scientist), and the scientist was observed both in the case it always perceives an ABORT_PLAN message sent by the controller, and in the case it, non-deterministically, *can* perceive or not this message thus (possibly) triggering the intervention of the rescue team. More in particular, during this verification phase, the SCIENTIST_DEADLINE was set to an over estimated value (e.g., 50), the SENSOR_PERIOD was varied from 1 to 20, and the remaining scenario parameters set as shown in Fig. 3. In addition, a decoration clock z was introduced which is reset when the environment detects a dangerous gas level, and then checked when a scientist critical event occurs, that is an ABORT_PLAN, a LEFT_ENV or a DIE message is received. The following query, based on the *leads-to* operator, was used to determine the best case value (lower bound lb) of the response time:

$$\begin{aligned}
&Environment(EN).gasLevel > 2 \;\&\& \\
&!Environment(EN).meetDangerousLevel \rightarrow \\
&(!Scientist(SC).isOutEnv \;\&\&\; !Scientist(SC).isDead \\
&\&\&\; (Scientist(SC).SafeByAbort || (Scientist(SC).Select \\
&\&\&\; msg == LEFT\_ENV))) \;\&\&\; z \geq lb
\end{aligned}$$

where the value *lb* is the highest value which satisfies the query, and represents the minimum time required for saving the scientist. The query permits to check the time amount which elapses from the time instant the environment changes the gas level to a toxic value (*starting* or *premise state*), to the

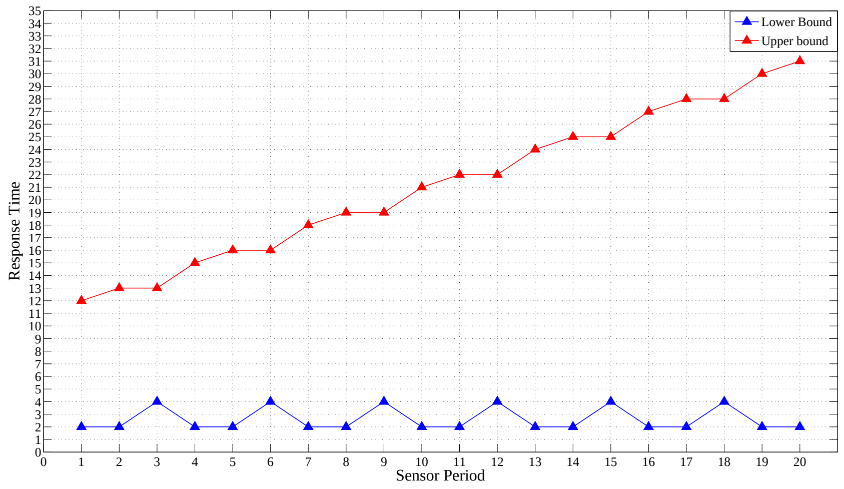


Figure 18: Response time windows when the scientist *can* perceive an ABORT_PLAN

time instant the scientist is alerted about the dangerous situation (inevitable *consequent state*). Changing the clock constraint to $z \leq ub$ where ub is the lowest value which satisfies the query, allows one to find the upper bound of the response time.
Fig. 18 shows the $[lb, ub]$ emerged time windows for the scientist to be saved in the case an ABORT_PLAN message would not be perceived. As expected, as the sensor period increases, the upper bound of the response time augments because the controller gets late informed about a dangerous gas level.

It is interesting to note, that under the assumed operating conditions, the scientist is always saved, either by an ABORT_PLAN message or through a LEFT_ENV message. In fact, the following query

$$A[]\ !Scientist(SC).isDead$$

which checks that in no case the scientist dies, is satisfied.
Fig. 19 depicts the observed time windows when the scientist, optimistically, is always capable of perceiving an ABORT_PLAN (in Fig. 14 the two dashed arcs are pointed to the SaveByAbort location). In this case, the scientist is always saved and the rescue team is never contacted as witnessed by the (not satisfied) query:

$$E <> \ Rescue(RE).RescueReach$$

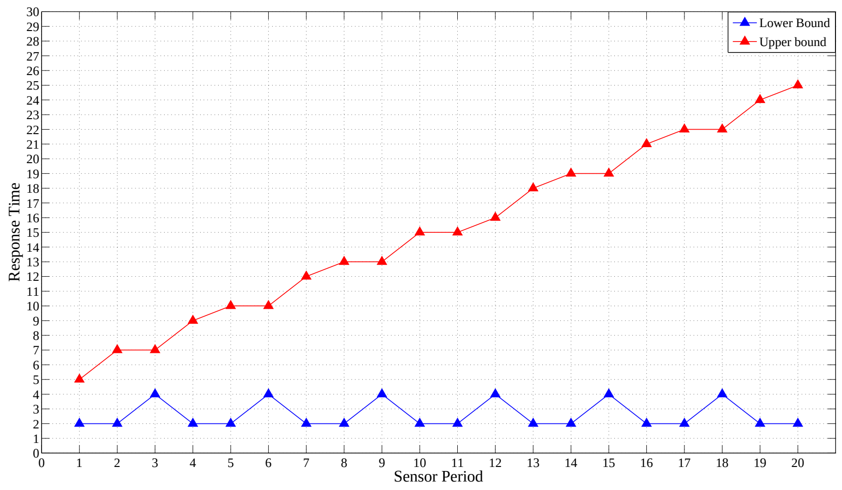


Figure 19: Response time windows when the scientist *always* perceives an ABORT_PLAN

With respect to Fig. 18, the lower bounds of the response time in Fig. 19 are obviously the same, and the upper bounds are lower because an ABORT_PLAN message gets heard soon by the scientist, and there is no rescue team involvement.
By enabling the full model behavior, that is the sensor can fail and the scientist can or not perceive an ABORT_PLAN, the following query:

$$E <> \; Scientist(SC).isDead$$

is satisfied thus testifying, as expected, that the scientist *can* die.

6.2. Quantitative statistical model checking

The importance of this second analysis phase can derive, in general, from an impossibility of making an exhaustive verification on a given complex model, and from the necessity to quantify the rate or likelihood with which selected events can occur in the system. From the latter point of view, whereas the qualitative non-deterministic analysis can suggest that "*something can occur*", e.g., the "*scientist can die*" following a dangerous gas level, it is of great interest from the practical point of view knowing how is the probability of the event to happen. Therefore, qualitative and quantitative analysis are synergic to each other and both contribute to a full characterization of the system behavior. On the other hand, a Statistical Model Checker (SMC)

does not build the model state graph but relies on simulation runs and statistical techniques, such as Monte Carlo-methods and sequential hypothesis testing [22], for estimating properties of the simulated model. As a consequence, the memory usage during SMC is linear with the model.
A series of experiments were carried out on the toxic gas sensing system (TGSS) model using Uppaal SMC, under the general behavior of the scientist which can or not perceive an ABORT_PLAN message, and of the sensor which can or not be working thus possibly not transmitting to the controller the gasLevel. No changes are required by the model except for some new decoration variables which, although unnecessary under exhaustive model checking, can be useful to gather information from the simulations. As a preliminary test, Fig. 20 shows 30 simulation traces of the TGSS model, using 2 as the sensor period, 14 as the SCIENTIST_DEADLINE (1 more of the upper bound of the response time emerged during exhaustive verification) and keeping the values of the other scenario parameters as in Fig. 3. In Fig. 20 the monitored values of the gasLevel managed by the environment and of the isOutEnv and isDead variables of the scientist, are depicted. The picture is directly achieved from Uppaal SMC as part of the query:

$$\begin{aligned}&\mathit{simulate}\ 30\ [<= 1000]\ \{\mathit{Environment}(\mathit{EN}).\mathit{gasLevel},\\&\mathit{Scientist}(\mathit{SC}).\mathit{isOutEnv}, \mathit{Scientist}(\mathit{SC}).\mathit{isDead}\}\end{aligned}$$

As one can see from Fig. 20, 1000 time units enable the environment to generate, in many cases, a toxic gas level. Furthermore, the picture confirms that there are cases where the scientist is rescued and others where he dies. For example, it was observed that the dangerous gas level generated at time 635 is followed by the scientist which gets saved at 640, thus strictly within its deadline. But at time 785 the toxic gas level is followed by the scientist which dies at time 800, i.e., one time beyond the allowed deadline.

Fig. 21 shows the estimated probability with which the scientist can die following a toxic gas level, when the sensor period is varied from 1 to 20. For each sensor period, the SCIENTIST_DEADLINE parameter is set to the corresponding value determined during exhaustive verification, augmented by 1 for safety reasons. In particular, Fig. 21 depicts the bounds of the confidence intervals proposed after launching the query:

$$\mathit{Pr}[<= 5000](<> (\mathit{Scientist}(\mathit{SC}).\mathit{isDead}))$$

The default values of Uppaal SMC statistical parameters were used, e.g., 95% of confidence degree with a confidence interval error $\epsilon = 0.05$. Each confidence interval emerges after a number of simulation runs which ranges

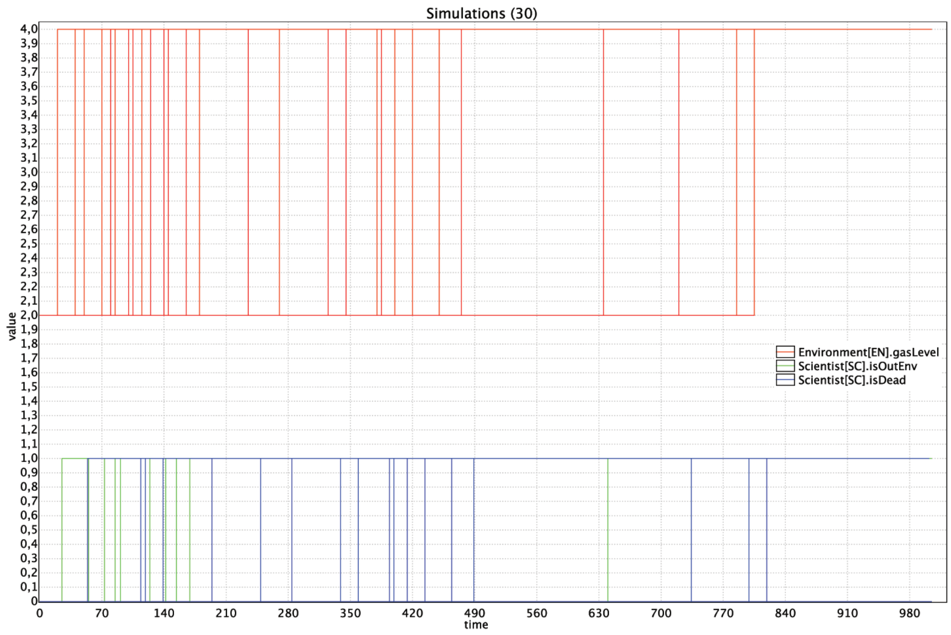


Figure 20: Traces of 30 simulations monitoring the gasLevel, isOutEnv and isDead variables

between 300 and 400. The probability decreases as the sensor period augments. Indeed, low values of the sensor period imply a high value of the sensor activation frequency and then the higher is the probability for the sensor to become not working. Vice versa, a high value of the sensor period diminishes the number of times the sensor is activated and also the probability of being not working. Therefore, in these cases the controller can be informed late about a dangerous gas level. However, the use of a larger SCIENTIST_DEADLINE value (see Fig. 18) ensures in many cases the scientist can be rescued.

The time bound of 5000 proved sufficient for injecting in the system the event of a dangerous gas level. In fact, the query:

$$Pr[<= 5000](<> Environment(EN).gasLevel > 2$$
$$\&\& \; !Environment(EN).meetDangerousLevel)$$

proposes, after 29 runs, a confidence interval of [0.901855,1], thus confirming the event has a great occurrence probability.
The results in Fig. 21 are also a consequence of the transformation of a non-deterministic behavior into a true probabilistic one guided by the adopted

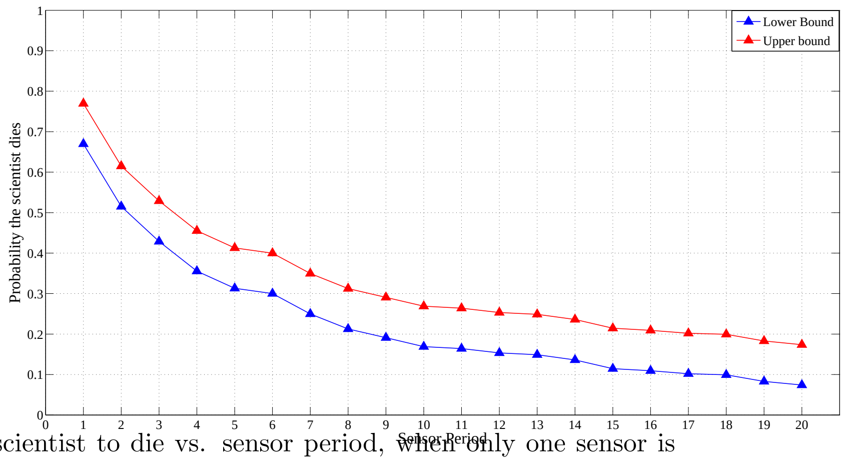


Figure 21: Probability of the scientist to die vs. sensor period, when only one sensor is used

probabilistic weights. For example, whereas during non-deterministic analysis perceiving or not an ABORT_PLAN message by the scientist is a matter of a non-deterministic choice (see Fig. 14), during SMC analysis perception is an event whose occurrence probability is 0.90, thus in many cases the scientist is saved by an ABORT_PLAN message. All of this was controlled by the following queries, in extended MITL, based on the until operator U [43]. First it was checked that the probability of saving the scientist (with SENSOR_PERIOD=2 and SCIENTIST_DEADLINE=14) is almost the complement of that of the scientist dying in the same operating conditions (Fig. 21) thus:

$$Pr(<>[0,5000](Environment(EN).gasLevel > 2\ U[0,14]\ Scientist(SC).isOutEnv))$$

The query asks to quantify the event occurrence: *"assuming that at an instant in time in [0,5000] a dangerous gas level occurs, what is the probability that within the next 14 time units the scientist is saved?"*. UPPAAL SMC uses 738 runs and suggests a probability confidence interval of [0.384959,0,484959] with confidence 95%.
The following query estimates, in particular, the probability of saving the scientist through an ABORT_PLAN message:

$$Pr(<>[0,5000](Environment(EN).gasLevel > 2\ \ U[0,14]\ Scientist.SafeByAbort))$$

In this case, always by 738 runs, a confidence interval is proposed of [0.355149,0.455149].

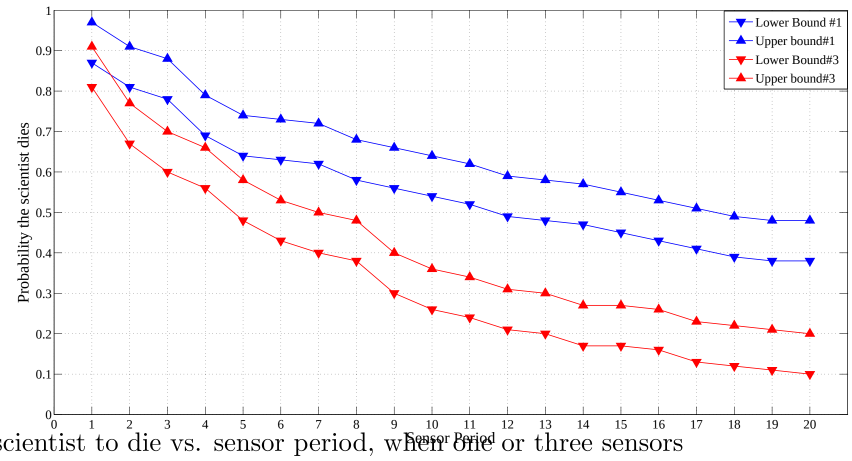


Figure 22: Probability of the scientist to die vs. sensor period, when one or three sensors are used and the probability for a sensor to be working is 95%, and not be working is 5%

Finally, the query

$$Pr(<>[0,5000](Environment(EN).gasLevel > 2\ \ U[0,14]\ msg == LEFT\_ENV))$$

quantifies the probability of saving the scientist through the rescue team (LEFT_ENV message). Uppaal SMC proposes, after 738 runs, a confidence interval of [0,0.0920054] to testify the event has a very low occurrence probability. The stochastic behavior of the TGSS model was also checked when more sensors are used. In these cases it is expected that whereas a sensor can possibly be not working, another one can be working so as to guarantee the controller gets informed of a dangerous gas level. However, as expected and confirmed experimentally, the use of probability weights 99 and 1 (see Fig. 13) for the sensor to be respectively working or not, would imply a not real benefit can be gained by the use of multiple sensors. Therefore, some experiments were performed by changing the probabilistic weights to 95 and 5.

Fig. 22 shows the probability for the scientist to die when one or three sensors are used. The sensor period is varied from 1 to 20, the SCIENTIST_DEADLINE is set to 1 more of the value detected during model checking for the same sensor period, and the other scenario parameters are left unmodified. As one can see from Fig. 22, the probability is greater than that shown in Fig. 21 when only one sensor is used and the probability weight for the sensor to be worked is diminished from 99 to 95. Moreover, it clearly

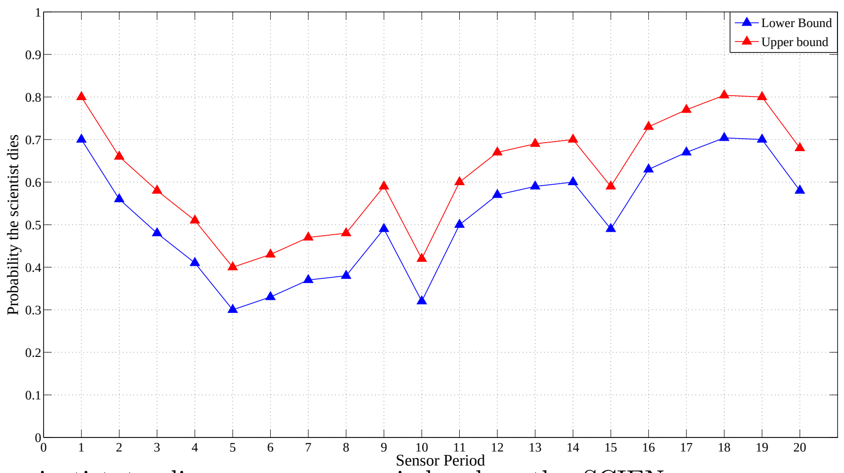


Figure 23: Probability of the scientist to die vs. sensor period, when the SCIENTIST_DEADLINE is 10

emerges from Fig. 22, that the probability value significantly decreases when 3 sensors are used.
Finally, in Fig. 23 and Fig. 24 the probability of the scientist to die is shown when the model is that of Fig. 3, only one sensor is used, the sensor period is varied from 1 to 20, and the SCIENTIST_DEADLINE is kept fixed in all the experiments (a safety requirement). In particular, Fig. 23 refers to the case the SCIENTIST_DEADLINE is 10 and in Fig. 24 it is 12. The following query was used for Fig. 23. Each point is the result of 738 runs. The until interval is turned to [0,12] for Fig. 24.

$$Pr(<>[0,5000](Environment(EN).gasLevel > 2\ \ U[0,10]\ Scientist(SC).isDead))$$

It emerges that for small values of the sensor period, the probability to die is high because the sensor more likely can be not working. Similarly, for high values of the sensor period the probability is again high because the sensor, although now is more likely to be found working, could detect late the change in the environment, causing a delay in the start of the rescue operations.
Both figures 18 and 19 confirm there is a value for the sensor period where the probability gets to a minimum. In the case of Fig. 23 this occurs at abscissa 5, whereas it shifts to 10 in Fig. 24, due to the greater value of the SCIENTIST_DEADLINE. In reality, in fig. 23 there are more local minima. This is due to a parameters alignments configuration in the proposed

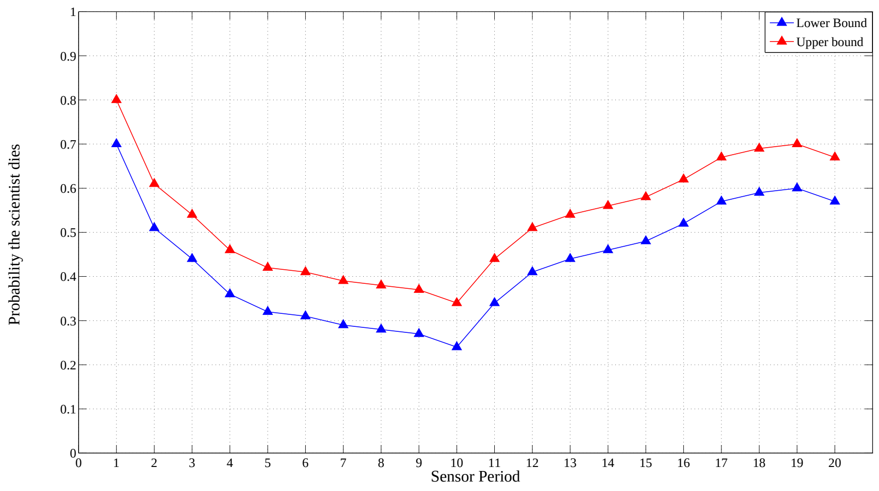


Figure 24: Probability of the scientist to die vs. sensor period, when the SCIENTIST_DEADLINE is 12

scenario. Setting the SCIENTIST_DEADLINE=10, it happens that when the sensor period moves to multiples of the CHANGING_PERIOD=5, it can perceive a dangerous gas level contextually with the environment change or with a delay that, in these cases, leaves more time to complete the rescue operations. In Fig. 24 there is only one minimum, because the widest SCIENTIST_DEADLINE already offers time units sufficient to act the saving operations, and the die probability curve has lower levels.

6.3. Partitioning

A key factor of a Theatre model is the possibility of modulating the number of processing units/theatres (i.e., the parallelism degree) upon which the application actors can be partitioned. From this point of view, although the TGSS example was modelled and analyzed using the maximal parallelism hypothesis (i.e., by using 6 processing units (PUs) when only one sensor is involved), it can as well be studied and implemented using a different partitioning schema which uses less resources. In particular, from the model interactions (see Fig. 3) one can infer that a configuration with 3 PUs is sufficient to cope with the distribution requirements of the system. In fact, the Environment, Sensor, Scientist and Main can be grouped together onto one PU, and the Controller and the Rescue assigned each to a distinct PU.

The TGSS model can easily be adapted to work with 3 PUs by varying the number of available processing units (constant NPU in the section 5.2), (possibly) adapting the required number of Message instances and, finally, adjusting the move() operations in the main automaton (see Fig. 17).
It is worth noting that the TGSS temporal behavior, e.g., the scientist deadline values when the sensor period is varied from 1 to 20, emerged unchanged also when only three PUs are used. For demonstration purposes it was also verified that the timeliness of the system rests confirmed even when all the actors are put on to one PU.

7. Implementation issues

A full implementation of THEATRE was achieved in Java together with some methodological guidelines which assist transitioning a verified model to final implementation and real-time execution.
The framework structure of THEATRE is a consequence of being realized by a hierarchy of classes (see Fig. 25) together with a *flow of events*. The flow of events glues together, transparently, the user-defined actor classes with the operation of a control machine. A control machine iterates its *control loop* until a termination condition is possibly met. At each iteration one event is selected (i.e., a message delivery or a delay expiration) and actuated. In the case of a message, a corresponding message server is activated in the destination actor. At the termination of the message server, control is returned to the control machine which proceeds with the next iteration of the control loop and so forth.

The following briefly summarizes some key points of the Java realization. More details can be found in [18].
Actors are programmed as classes which inherit from the *Actor* base class which exposes the fundamental services like the non-blocking send operation. Actor *universal names* are strings. The programming style naturally follows the abstract modelling style shown in Fig. 3.
Distributed functionalities are split between the theatre, which interfaces to network and transport layer services (see Fig. 1), and the control machine which regulates the evolution of local actors. A standalone and a distributed version of many control machines are available (see Fig. 25). *Concurrent/DConcurrent* are control machines which support the execution of general untimed applications. *Simulation/DSimulation* enable simulation and performance prediction of general stochastic systems. *DSimulation* de-

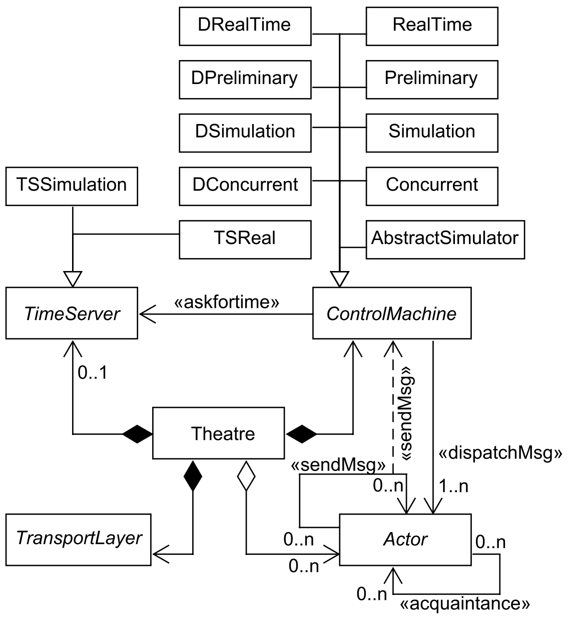


Figure 25: UML class diagram of THEATRE framework, taken from [18]

pends on a conservative synchronization algorithm [7]. *AbstractSimulator* implements in Java the abstract semantics of a THEATRE model, and corresponds to the same control structure realized by the reduction process into UPPAAL proposed in this paper. However, whereas *AbstractSimulator* represents an *ad hoc* simulator, which requires the addition (*decoration*) to a model of suitable observer/monitor classes for collecting output data and extracting statistical information, the use of a temporal logic in UPPAAL like MITL [21] provides greater flexibility for property specification and assessment. *Preliminary/DPreliminary* are control machines which use real-time but still operate on an abstract THEATRE model. They are useful to control the overhead introduced by message scheduling and dispatching, message server execution and network communications delay and their influence on the message timing constraints. *RealTime/DRealTime* manage real-time and regulate the execution of a THEATRE model with final concrete implementation of message servers. The model is supposed to be partitioned so as to run over one or multiple theatres.
It is useful to observe that whereas the virtual clock of a simulation con-

trol machine is immediately advanced to the time of the next most imminent event, in a preliminary/real-time control machine, the real-time clock advances at its own rate and the control loop can, sometimes, have some no-operation cycles just to allow time to increase.
The adopted transport layer depends on Java sockets. The network of theatre sockets (in general a full mesh structure) required by a system is initially set up on the basis of an xml configuration file (*config.xml*). A theatre coincides with a JVM instance. The main program of each theatre plays alternatively the role of server or client for each distinct socket instantiation, according to the configuration file. After socket creation, actors are created and allocated (moved). From this point of view, in a typical configuration, a main theatre can create actors, initialize them and then move them to their destination theatres. Control messages are exchanged among the theatres of a system, to start/stop a distributed execution and to regularly keep updated a global time notion with the help of a time server (Fig. 1).
The implementation ensures actors and messages can be remotely transmitted (in a serialized format). When an actor moves from a theatre A to a theatre B, it leaves on A a proxy version of itself which acts as a message *forwarder*. Would an actor come back to a theatre where a proxy version of itself exists, the proxy version is replaced by the actual version. Sending a message to a local proxy actor causes (transparently) a remote message transmission to occur.
Some methodological guidelines based on the concept of *model continuity* are defined in [2, 14, 18]. They can be naturally exploited with the THEATRE version described in this paper and are assisted by specific control machines (see Fig. 25). Following modelling and analysis a *preliminary execution* phase, based on *Preliminary/DPreliminary* control machine, can be actuated where the model is still abstract (that is message servers still use delay constructs instead of final concrete code) but time is real.
Whereas during modelling and analysis message scheduling and dispatching activities are assumed to consume a negligible time, during preliminary execution their timing overhead can be monitored and its influence upon message timing constraints and the overall temporal behavior of a system evaluated. The *Preliminary/DPreliminary* control machines represent processing units (theatres) as Java threads and delays are achieved by sleep operations on such threads.
After preliminary execution, a model can be finally implemented with the actual version of message servers provided. Such a final implementation de-

pends on the services of a control machine like *RealTime/DRealTime* in Fig. 25.

8. Conclusions

This paper introduces THEATRE, a variant of the Actors model [3] designed specifically to address modelling, analysis and implementation of time-dependent probabilistic distributed systems. In its current version THEATRE adopts the timing model and programming style of PTRebeca [9]. THEATRE is formally defined through a structural operational semantics. THEATRE models can straightforwardly be expressed in Java. The paper contribution consists in proposing a reduction of an abstract THEATRE model onto the timed automata of UPPAAL [20, 21]. The reduction process is novel because it enables both the exhaustive verification through model checking (MC) and the statistical model checking (SMC) of a system functional and temporal behavior. A same model can be used unchanged for MC and SMC purposes. In general, though, due to the asynchronous character of actors, MC can be difficult to apply to complex and scalable models for state explosion problems. In these cases, SMC makes it possible to quantitatively evaluate system behavior through simulation runs.
The practical aspects of THEATRE modelling and analysis are demonstrated through a real-time distributed and dependable case study. Finally, the paper gives a summary of an implementation which was achieved in Java along with some methodological guidelines which practitioners and engineers can follow to transform, without distortions, an abstract model into a final implementation for real-time execution.
Prosecution of the research mainly aims to use Java as the modelling, analysis and implementation language for THEATRE systems. Preliminary experience is reported in [18]. To help formalization and assessment that a final Java program correctly corresponds to an early analyzed model, the use of such tools as Java Modelling Language [44] can be a key. Starting from the developed control structures of message scheduling and dispatching, the goal is to experiment with the Java Path Finder (JPF) tool [45] which permits, in general, the exhaustive verification of Java multi-threaded programs.

References

[1] E. A. Lee, Cyber physical systems: Design challenges, in: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, ISORC '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 363–369. doi:10.1109/ISORC.2008.25.

[2] F. Cicirelli, L. Nigro, P. F. Sciammarella, Model continuity in cyber-physical systems: A control-centred methodology based on agents, Simulation Modelling Practice and Theory 83 (4) (2018) 93–107.

[3] G. Agha, Actors: a model of concurrent computation in distributed systems, MIT Press, Cambridge, MA, USA, 1986.

[4] G. A. Agha, I. A. Mason, S. F. Smith, C. L. Talcott, Foundation for actor computation, J. of Functional Programming 7 (1) (1997) 1–72.

[5] E. A. Lee, The problem with threads, Computer 39 (2006) 33–42. doi:doi.ieeecomputersociety.org/10.1109/MC.2006.180.

[6] J. A. Stankovic, K. Ramamritham, What is predictability for real-time systems?, Real-Time Systems 2 (1990) 247–254.

[7] R. M. Fujimoto, Parallel and Distribution Simulation Systems, 1st Edition, John Wiley & Sons, Inc., New York, NY, USA, 1999.

[8] M. Sirjani, Rebeca: Theory, applications, and tools, in: Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures, 2006, pp. 102–126.
URL `https://doi.org/10.1007/978-3-540-74792-5_5`

[9] A. Jafari, E. Khamespanah, M. Sirjani, H. Hermanns, M. Cimini, PTRebeca: Modeling and analysis of distributed and asynchronous systems, Sci. Comput. Program. 128 (2016) 22–50.
URL `https://doi.org/10.1016/j.scico.2016.03.004`

[10] M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proc. 23rd International Conference on Computer Aided Verification (CAV'11), Vol. 6806 of LNCS, Springer, 2011, pp. 585–591.

[11] D. Guck, T. Han, J.-P. Katoen, M. Neuhuer, Quantitative timed analysis of interactive Markov chains, Lecture Notes in Computer Science, Springer, 2012, pp. 8–23. doi:10.1007/978-3-642-28891-3_4.

[12] A. H. Reynisson, M. Sirjani, L. Aceto, M. Cimini, A. Jafari, A. Ingólfsdóttir, S. H. Sigurdarson, Modelling and simulation of asynchronous real-time systems using timed rebeca, Sci. Comput. Program. 89 (2014) 41–68.
URL `https://doi.org/10.1016/j.scico.2014.01.008`

[13] A. Jafari, E. Khamespanah, H. Kristinsson, M. Sirjani, B. Magnusson, Statistical model checking of timed rebeca models, Computer Languages, Systems & Structures 45 (2016) 53–79.
URL `https://doi.org/10.1016/j.cl.2016.01.004`

[14] F. Cicirelli, L. Nigro, Control centric framework for model continuity in time-dependent multi-agent systems, Concurrency and Computation: Practice and Experience 28 (12) (2016) 3333–3356, cpe.3802. doi:10.1002/cpe.3802.

[15] F. Cicirelli, L. Nigro, Exploiting social capabilities in the minority game, ACM Trans. Model. Comput. Simul. 27 (1) (2016) 6:1–6:21.
URL `http://doi.acm.org/10.1145/2996456`

[16] C. Nigro, L. Nigro, P. F. Sciammarella, Modelling and analysis of multi-agent systems using Uppaal SMC, Int. J. of Simulation and Process Modelling 13 (1) 73–87.

[17] C. Nigro, L. Nigro, P. F. Sciammarella, Model-checking knowledge and commitments in multi-agent systems using actors and uppaal, in: 32nd European Conference on Modelling and Simulation (ECMS 2018), Wilhelmshaven, Germany, 2018, pp. 136–142.

[18] F. Cicirelli, L. Nigro, P. F. Sciammarella, Seamless development in Java of distributed real-time systems using actors, in: Int. Symposium Simulation and Process Modelling (ISSPM 2018), 21-22 July, Shenyang, China, 2018.

[19] R. Alur, D. Dill, A theory of timed automata, Theoretical Computer Science 126 (2) (1994) 183–235.

[20] G. Behrmann, A. David, K. Larsen, A tutorial on Uppaal, in: M. Bernardo, F. Corradini (Eds.), Formal Methods for the Design of Real-Time Systems, LNCS 3185, Springer, 2004, pp. 200–236.

[21] A. David, K. G. Larsen, A. Legay, M. Mikuăionis, D. B. Poulsen, Uppaal smc tutorial, Int. J. Softw. Tools Technol. Transf. 17 (4) (2015) 397–415. URL http://dx.doi.org/10.1007/s10009-014-0361-y

[22] G. Agha, K. Palmskog, A Survey of Statistical Model Checking, ACM Trans. Model. Comput. Simul. 28 (1) (2018) 6:1–6:39. URL http://doi.acm.org/10.1145/3158668

[23] L. Nigro, P. F. Sciammarella, Statistical model checking of distributed real-time actor systems, in: 21st IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2017, Rome, Italy, October 18-20, 2017, 2017, pp. 188–195. URL https://doi.org/10.1109/DISTRA.2017.8167684

[24] C. Hewitt, Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot, Tech. rep., Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab (1972).

[25] P. Haller, M. Odersky, Actors that unify threads and events, in: International Conference on Coordination Languages and Models, Springer, 2007, pp. 171–190.

[26] Erlang, on-line, https://www.erlang.org.

[27] R. Shangping, G. A. Agha, RTSynchronizer: language support for real-time specifications in distributed systems, ACM SIGPLAN Not. 30 (11) (1995) 50–59.

[28] L. Nigro, F. Pupo, Schedulability analysis of real time actor systems using Coloured Petri Nets, Lecture Notes in Computer Science (2001) 493–513.

[29] R. Beraldi, L. Nigro, Distributed simulation of timed Petri nets. a modular approach using actors and Time Warp, IEEE Concurrency 7 (4) (1999) 52–62.

[30] A. G. Ren S., Venkatasubramanian N., Formalizing Multimedia QoS Constraints Using Actors, IFIP Advances in Information and Communication Technology, Springer, Boston, MA, 1997, pp. 139–153.

[31] A. Furfaro, L. Nigro, F. Pupo, Aspect oriented programming using actors, in: Proc. of 2nd Int. Workshop on Aspect Oriented Programming for Distributed Computing Systems (AOSDCS 2002), IEEE CS, 2002, pp. 493–502.

[32] A. Furfaro, L. Nigro, F. Pupo, Multimedia synchronization based on aspect oriented programming, Microprocessors and Microsystems 28 (2) (2004) 47–56.

[33] B. Nielsen, G. Agha, Semantics for an actor-based real-time language, in: Fourth International Workshop on Parallel and Distributed Real-Time Systems (WPDRS96), IEEE Computer Society Press, 1996.

[34] M. Sirjani, Power is overrated, go for Friendliness! Expressiveness, Faithfulness and Usability in Modeling-The Actor Experience, Principles of Modeling-Essays dedicated to Edward A. Lee on the Occasion of his 60th Birthday. Available at http://rebeca-lang. org/assets/papers/2017/Friendliness. pdf.

[35] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, S. Neuendorffer, Taming heterogeneity - the Ptolemy approach, Proceedings of the IEEE 91 (1) (2003) 127–144.
URL `http://chess.eecs.berkeley.edu/pubs/488.html`

[36] R. K. Karmani, G. Agha, Actors, Springer US, Boston, MA, 2011, pp. 1–11.
URL `https://doi.org/10.1007/978-0-387-09766-4_125`

[37] F. L. Bellifemine, G. Caire, D. Greenwood, Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology), John Wiley & Sons, 2007.

[38] F. Cicirelli, A. Furfaro, A. Giordano, L. Nigro, Performance of a multi-agent system over a multi-core cluster managed by terracotta, in: Proc. of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, 2011, pp. 125–133.
URL `http://dl.acm.org/citation.cfm?id=2048476.2048492`

[39] F. Cicirelli, A. Furfaro, L. Nigro, F. Pupo, Agents over the grid: An experience using the globus toolkit 4, in: Proc. of the 26th European Conference on Modelling and Simulation (ECMS'2012), 2012.

[40] G. Plotkin, A structural approach to operational semantics, Tech. rep., Computer Science Department Aarhus University, Tech. Report DAIMI FN-19 (1981).

[41] G. Kahn, Natural semantics, Tech. rep., INRIA Research Report RR-0601, also Springer LNCS, vol. 247, pp. 22-39 (1987).

[42] G. Behrmann, A. David, K. Larsen, A tutorial on Uppaal, in: M. Bernardo, F. Corradini (Eds.), Formal Methods for the Design of Real-Time Systems, LNCS 3185, Springer, 2004, pp. 200–236.

[43] R. Alur, T. Feder, T. A. Henzinger, The benefits of relaxing punctuality, J. ACM 43 (1) (1996) 116–146.
URL `http://doi.acm.org/10.1145/227595.227602`

[44] Java Modelling Language, on-line, `http://www.eecs.ucf.edu/~leavens/JML//index.shtml`, accessed on April 2018.

[45] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda, Model checking programs, Automated Software Engineering 10 (2) (2003) 203–232.