A DEVELOPMENT METHODOLOGY FOR CYBER-PHYSICAL SYSTEMS BASED ON DETERMINISTIC THEATRE WITH HYBRID ACTORS

FRANCO CICIRELLI[1] AND LIBERO NIGRO[2]

[1] *CNR - National Research Council of Italy,*
Institute for High Performance Computing and Networking (ICAR),
87036 Rende (CS), Italy
f.cicirelli@icar.cnr.it

[2] *DIMES – Engineering Department of Informatics,*
Modelling Electronics and Systems Science,
University of Calabria,
87036 Rende (CS) – Italy
l.nigro@unical.it

(received: 31 July 2021; revised: 30 August 2021;
accepted: 1 September 2021; published online: 30 October 2021)

Abstract: The goal of the work described in this paper is to propose a development approach for cyber-physical systems (CPS) which relies on actors as the fundamental modelling blocks. The approach is characterized by its capability to deal with the discrete aspects of the cyber part of a CPS, as well as the continuous behaviour of the physical part. More in particular, the approach is based on the Theatre actor system which fosters determinism in model behaviour, and favours model continuity when switching from system modelling and analysis down to prototype and synthesis phases. A key factor of Theatre is the possibility to combine both discrete-event actors, which operate on a discrete timeline, with continuous-time actors which reproduce, in general by using Ordinary Differential Equations (ODEs), the dynamical evolution of physical components. For formal property assessment, Theatre actors (both discrete and continuous) can be reduced to Timed Automata (TA) in the context of the Uppaal toolbox, where the exhaustive and/or the statistical model checkers can be exploited. This paper first d escribes t he p roposed approach, then it demonstrates its suitability to CPS modelling and analysis through examples. The paper also discusses how abstract and formal modelling actor concepts can be naturally transitioned to implementation concepts in Java.

Keywords: Cyber-physical systems, model-driven development, timing models reconciliation, hybrid actors, model continuity, determinism, Theatre actor system, Uppaal, Java.

DOI: https://doi.org/10.34808/tq2021/25.2/e

1. Introduction

Cyber-physical systems (CPS) (e.g. [1–4]) are exploited in modern society to provide critical services in such application domains as healthcare, smart environments, new industry standards, automotive, avionics and so forth. They are very challenging to develop because they require the fulfilment of timing, reliability and resilience constraints, in a context where a continuous operating physical part, interfaced by sensors and actuators, has to be controlled by a discrete event/discrete time cyber/software part, the two parts being interconnected through the services of a network and associated protocols. Design difficulties can be retrieved in the necessity of developing and integrating multiple models for the physical and the cyber components, and in the need to reconciliate Newtonian time with discrete time of the software controlling part.

Several approaches and associated modelling languages and development tools have been proposed in last years for CPS. A notable formal approach is represented by Ptomely [5] modelling and supporting tools. Ptolemy rests on the adoption of a special actor model which addresses composability by means of typed input/output ports. The adopted concurrent actor model purposely avoids the pitfalls of classical multi-threaded programs [6], and the dependencies from the hidden services of an underlying, difficult to control, Operating System. Ptolemy emphasises the usefulness of controlling actors through high-level, application-tailored mechanisms. An analysed model can, finally, be directly translated, in a case, in C code and implemented with the use of PLC.

Recently, the Ptolemy community has advocated the use of *deterministic actors* [7] for CPS and for the development of Industrial Internet-of-Things applications [8]. Determinism is motivated in [9] for modelling and development to ensure *repeatability* in model behaviours, that is guaranteeing that a model, starting from a given input and initial state, always generates the same behaviour and output. Determinism is felt as a fundamental design issue to help *reproducibility* during synthesis, that is enabling an engineered model to be "faithfully" reproduced in physical terms.

More concretely, determinism was experimented in a *synchronous* modelling language named Reactors [10], abstracted through the universal formalism of Lingua Franca (LF) [10–12]. LF favors the definition of CPS closed models, which explicitly include the *external input/output actions*, in general modelled with the help of Ordinary Differential Equations (ODE), which are points where continuous physical time requires integration with the cyber discrete time. LF allows to define reactors where the body of message reactions can possibly be specified according to different programming languages (e.g., C).

Deterministic actors are supposed to operate in a computational framework where events occur at their *due* times. Simultaneous events, though, that is events occurring at a *same* time point, are delivered and ultimately processed in a deterministic way, established either by a *precedence graph* or, more practically, by modelling concepts (e.g., unique ID acting as priorities) directly associated to reactors and to their message handlers *(reaction methods)*.

In [12] deterministic actors were exploited to model check Lingua Franca models preliminarily transformed into Timed Rebeca [13] and analyzed by the Afra model checker tool.

The work described in this paper adheres to the same development guidelines of Ptolemy and Lingua Franca. However, the proposed approach is original because it is based on the Theatre actor system implemented in Java [14], which has proved its effectiveness in supporting time-sensitive applications [15, 16] both in a standalone, parallel [17] or distributed context [14]. Theatre is characterized by its volition of being *control-based*. Lightweight thread-less actors are used, which are transparently regulated by an application-dependent reflective control layer which supervises the exchange of asynchronous messages and settles its ultimate delivery. The control layer can be specialized to ensure determinism in the actor operation. As a modelling language, Theatre is provided of formal operational semantics [15] which was used to define a reduction of an actor model onto Uppaal [18, 19] which enables property assessment by both exhaustive and/or statistical model checking [20]. A strength of Theatre is its support to *model continuity* [21, 22, 4] in the system lifecycle, meaning that a model can be transitioned in a seamless way from its analysis down to the final implementation phase.

Flexibility of the Theatre design was exploited specifically for CPS development. Toward this, the pre-existing discrete-event and discrete-time actors were paired with a notion of continuous time (or hybrid) actors [23, 24], which are devoted, during modelling and analysis, to reproduce, possibly through ODEs, the dynamical laws of variation of continuous external environment variables.

The work described in this paper improves previous authors' work and provides the following new contributions.

- A more clear and compact characterization of the behaviour of continuous actors is adopted. The new framework unifies, to a large extent, the modelling of continuous actors to that of discrete actors, with a positive impact on the model checking activities carried out with Uppaal, and to model checking.
- A support to deterministic actors was achieved through a novel control structure (*scheduler*) for message scheduling and delivery, which takes into account the inevitable non-determinism caused by messages generated by continuous actors.
- A development methodology for CPS is proposed which centres on (a) determinism of message delivery to actors according to the timing of the cyber part; (b) the integration of discrete and continuous actors; (c) model continuity being extended to cope also with continuous actors.

The rest of this paper is organized as follows. Section 2 provides related work and the essential background information about Theatre. Section 3 describes the proposed CPS development methodology based on Theatre. The approach is demonstrated through two realistic modelling examples, their formal reduction onto Uppaal and their property assessment by exhaustive or statistical model checking. The design of a scheduler component which enforces model determinism is presented. The actor programming style in Java is also clarified. Section 4 discusses how a CPS Theatre-based model can be transitioned toward the implementation. Finally, Section 5 concludes the paper with an indication of on-going and future work.

2. Background and Related Work

In last years, the Actors computational model [25–27] emerged as a more safe and scalable concurrent programming paradigm w.r.t classic multi-threaded programs based on shared data with locks which are used to prevent data race problems [6]. Actors encapsulate a local data status, exposes a mailbox upon which actor naming is based, and interact one to another through the exchange of asynchronous messages. In the basic actor model, which is implemented in many nowadays actor systems like Scala/Akka [28], ActorFoundry [29], CAF [30] and so forth, each actor hosts an internal thread which, repeatedly, extracts one message at a time, if there are any, from the mailbox (queue) and processes it by executing atomically a corresponding message reaction method. A fundamental semantic aspect of actors is non-deterministic message delivery: no particular order is observed when delivering messages to their recipients. This in turn can enhance concurrency and distribution issues, although it can complicate the modeller activity. Classical actors are best suited to untimed distributed systems.

Some extensions to actors were proposed in the literature to adapt their application to real-time systems. Besides the Ptolemy concepts which were recapitulated in the Introduction section, a significant modern time-sensitive extension is represented by Timed Rebeca [13] which has shown its usefulness in formal modelling and analysis of time-dependent systems. Timed Rebeca maintains a thread per actor (said a rebec) but refines the non-blocking send operation by (possibly) attaching two (relative) time information to each message: an $after$ time (which defaults to 0) and a *deadline* (whose default is ∞). The meaning is that the message cannot be consigned before $after$ time units are elapsed since the sending time, and that it should be delivered before *deadline* time units are passed. Going beyond the deadline, causes the message to become invalid and to be discarded. A rebec reactive class specifies the local variables and acquaintances (known actors to which messages can be sent) and the message reaction methods called *message servers*. Message servers can admit a *delay* operation to suspend its execution (and the actor) for an amount of time units. No further messages can be managed by a rebec during its suspension period. Timed Rebeca constrains non-determinism on message delivery by ensuring, pragmatically, that messages sent, e.g. over a TCP network, by a given sender to a given receiver are received and processed in the sending order.

To address specifically the needs of CPS modelling, the Ptolemy II framework [5] makes it possible to specify and compose hierarchically distinct model-of-computations (MoC), which define rules for components to concurrently execute and communicate. Examples of MoCs include process networks, discrete events, data flows and continuous time. Properties of a complex composition can be analysed by simulation.

Hybrid Rebeca [23] is a recent extension of Timed Rebeca with Hybrid Automata concepts [31] which were added through a notion of physical actors which can be used in combination with software actors (normal rebecs). Physical actors are hybrid automata which model continuous behaviour of an external environment. A physical actor is made up of continuous modes (states), among which the actor can move dynamically. A mode consists of an initialization, an invariant, one or more flows (ODEs), a guard and a final action. Typically, the invariant establishes a time duration during which the flows operate to advance the values of continuous variables. When the guard is satisfied, the final action is executed and some event is generated toward, e.g., a software actor. Formal semantics is provided in [23] to a Hybrid Rebeca model by preliminarily transforming it into a monolithic hybrid automaton which is then model checked by the SpaceEx tool.

Timed Rebeca and Hybrid Rebeca represent an important research work concerning the use of actors for modelling and analysis of discrete and hybrid systems. However, both modelling languages have a lack when coming to transforming an engineered model into a compliant concrete realization.

The Theatre actor system [14, 15] was designed to act both as a formal modelling language for distributed, possibly probabilistic, real-time systems and cyber-physical systems [32, 33], and as a concrete implementation tool in Java [14, 17] useful to experiment filling the gap between modelling and analysis and final synthesis phases, according to model continuity [21, 22, 4].

Theatre can work with different timing models and programming styles. As an example, Theatre was adapted to wearing the syntax of Timed Rebeca. However, the following are some fundamental semantic differences from Timed/Hybrid Rebeca:

- Theatre actors have no internal thread and pay no context-switch overhead during message dispatching.
- Message delivery is regulated by a reflective control layer which is a key for achieving deterministic software actors.
- An actor is at rest until a message arrives. The execution of a message server (reaction) is truly atomic and cannot be suspended nor pre-empted. This in turn contributes to time predictability. When used, the *delay* operation must be the last instruction of a message server method.
- Theatre is based on global time.

- An application is a federation of multiple and interacting theatre nodes. Each theatre hosts [17] a control-layer, a transport layer and a collection of business actors. Actors can dynamically migrate from a theatre to another for reconfiguration purposes.
- A *timeserver* component is used in a parallel/distributed system, to keep temporally aligned the various theatres.
- A CPS Theatre model can exploit both regular (software) actors and continuous actors (modes). Each continuous mode is modelled as a separate actor. A software actor (*accessor* [32]) can manage a collection of continuous modes thus achieving an equivalent hybrid automaton like in Hybrid Rebeca. Also considering that hybrid automata are in many cases undecidable, hybrid Theatre models are mainly analysed by simulation. A reduction of Theatre onto Uppaal timed automata makes it possible to use the Statistical Model Checker for property prediction. In some cases [32, 33] a hybrid actor model (see also later in this paper) can exhaustively be model checked.

3. A CPS Methodology based on Theatre

CPS modelling introduces some specific challenges which are not shared with the development of other complex systems. As a first concern, the design and the analysis of a CPS requires the modelling and the analysis of both the cyber and the physical part of the system and, when moving toward the implementation, the modelled physical part has to be replaced by its physical counterpart which is often constituted by third-party acquired components. In addition, the modeller has also the task of (i) identifying the boundary between the two parts of a CPS by keeping clear which components will be replaced by software elements and which by real components whose behaviour has to match with the behaviour of the previously modelled entities, (ii) defining in which way the cyber and the physical parts have to interact when the system is put into real execution. A further challenge is guaranteeing that the properties assessed on the CPS model are maintained in the engineered system in a faithful way. Model continuity [21, 22, 4] is a key in the fulfilment of all the above issues, although the manner model continuity is supported determines the effectiveness and the exploitability of a given development process.

The proposed CPS methodology fosters model continuity, and rests on the use of actors as the basic development tools. Discrete actors are used for modelling the cyber part. Continuous actors (*modes*) are instead used to interface the physical part.

Formal modelling depends on a reduction onto the (possibly hybrid/stochastic) timed automata (TA) of Uppaal [18, 19]. Since Theatre has a natural distributed formulation, during the modelling stage theatre nodes are abstracted as *processing units* (PU). Each theatre/PU hosts a disjoint set of local actors. Application partitioning assigns actors to PU and can range from maximal parallelism (every actor runs on a separate PU) to minimal parallelism (all actors are allocated to a same PU). Other intermediate configurations can be adopted as well. A *scheduler* component [33] is responsible to providing deterministic message delivery to actors. The use of Uppaal for modelling and analysis makes it possible to exploit the flexibility and formality of a temporal logic language to express queries for property assessment, which can be either (a subset of) TCTL queries [18] for the symbolic/exhaustive model checker, or MITL queries [19] for the statistical model checker. A simulation control layer was also developed which can be used for checking a CPS Theatre model directly expressed in Java.

A fundamental aspect of the CPS Theatre-based methodology concerns the way actors are handled during the phases of the system lifecycle (see also Section 4). Basically, an actor model with its message passing remains unchanged when moving from a development phase to the next one. Only the control layer needs to be adapted/replaced to cope with the time requirements (which can be simulated-time, real-time or a mix of the two).

More challenging is the management of continuous actors, which can be logically located in the cyber or the physical part model. From this point of view, the approach relies on a mediator component, an *envGateway* [22], which acts as a *boundary* between the cyber and physical parts. The envGateway abstracts away the concerns about the used network and protocols in the physical part. Three common scenarios can be identified as described in the following.

First Scenario. The continuous modes, logically belonging to the cyber part, are exploited during modelling and remain in the synthesis phase, although with a necessary adaptation of their behaviour. Whereas during analysis external messages are generated toward discrete actors according to timing constraints (e.g., a period or a non-deterministic time interval), at the implementation time messages are related to the occurrence of physical events of the external environment. Such an occurrence can be sensed by the *envGateway* and ultimately transmitted to accessor actors through a message interaction. Message interaction can be based on *polling* (the continuous mode actor periodically checks the *envGateway* for the event occurrence) or (better) a *publish/subscribe* pattern is used so that when the event occurrence is sensed by the *engGateway*, it gets propagated to the interested subscriber(s) continuous actor(s).

Second Scenario. Continuous actors logically belong to the physical part model and get replaced by physical devices during synthesis. In this case, the cyber part model explicitly interacts with the physical part, during both the analysis and synthesis, through messages exchanged with the *envGateway*, which must be explicitly represented during modelling and analysis.

Third Scenario. It refers to a more complex scenario where continuous modes, e.g., equipped of ODEs, are part of the cyber model and play the role, e.g., of *predictors* which act in simulation and help the decision-making process of the real-time controlling part. Such modes are exploited during analysis and must be reified also during synthesis. It is worth noting that building in Java continuous actors with ODEs can be achieved on top of the Apache commons math3 library as demonstrated in [32].

Two examples will next be presented to show the application of the proposed methodology.

3.1. A Train-Door Controller

The following considers the real-time behaviour of a train-door controller modelled using Theatre. The example is a completely reworked version of the model presented in [33], redesigned according to the methodological guidelines adopted in this paper. The example naturally adheres to the above mentioned *First Scenario.*

The train-door understands the commands for closing/locking and unlocking/opening. However, the door can be locked provided it is closed. The train is allowed to move when the door is locked (and then necessarily closed). Following a train stop, the door first is unlocked then opened. Pairs of consecutive messages close/lock and unlock/open are assumed to be time separated to ensure, e.g., a lock command will be heard *after* a close and so forth. The train door is supposed to be also equipped of an open button which a passenger can press to ask, abruptly, the train to stop moving and to open the door. The button signal, though, is ignored if the door is already locked.

The modelling example is dependable and hard real-time. A failure in model behaviour can have catastrophic consequences in the practical case. Model analysis must ensure (safety), e.g., that never a state can be entered where the train is moving and the door is opened.

The system is modelled by the following actors: `Controller, Train` (mode), `Door` (mode), `Button` (mode), `Main. Controller` and `Main` are normal discrete actors. The other are continuous mode actors. `Train` and `Button` are input modes: they generate (in a non-deterministic way) respectively the external events `EXT_MOVE` and `EXT_OPEN` which are received and processed by the `Controller`. The `Door` is an input/output mode which understands the `OPEN, CLOSE, LOCK, UNLOCK` messages. The effects of these commands is simply to update local state variables and to confirm the `Controller` about the unlocked/opened, closed/locked door state. During reification, the commands are propagated to the physical door through actuators. The `Main` actor configures the system by initializing all the actors and by moving them to processing units according to a desired partitioning, e.g., `Main` and `Controller` allocated to PU 0, `Train` to PU 1, `Door` to PU 2, `Button` to PU 3. It is worth noting that actor initialization is accomplished by an explicit `INIT` message with (possibly) associated arguments. The `INIT` message can also transmit the acquaintances. For example, the `Controller` receives the identity of the `Train`, the `Door` and the `Button. Train` and `Button` must know the `Controller` to send it an external event.

3.1.1. Reducing actors onto Uppaal

A Theatre model like the train-door controller, can be reduced onto Uppaal by associating an automaton to each distinct actor (discrete or continuous), plus a `Scheduler` automaton (see Fig. 6) which hides the control layer and handles, with the help of data structures, the scheduling of sent messages and their deterministic delivery (see later in this paper for details). The reduction depends on the assignment of unique identifiers to messages and to actors. Actor uid are used to define type ranges which control the generation of instances at system configuration time. Each actor template process has one parameter of the associated type range, conventionally named `self`. Sending a message to an acquaintance is realized by the `send` broadcast channel and by the use of a few global variables: `S` (for the sender), `R` (for the receiver), `M` (for the message uid), `A` (for the *after* relative time, 0 if omitted), `D` (for the *deadline* relative time, ∞ if omitted). Message arguments, if there are any, are transmitted through the global array `arg`[].

The automaton of a discrete actor (see for example Fig. 5) is built around two basic locations: `Receive` and `Select`. In the `Receive` location a message is awaited through the broadcast channel `msgsrv[self]?`. The identity of the message is held into the `M` global variable. In the Select location the actor automaton decodes the value of M and executes the corresponding message server (reaction). A message server body is realized as a cascade of committed locations which are traversed without time passage. This realization complies with the atomicity and instantaneity of message reactions in normal Theatre actors [15], which in turn corresponds to the *macro-step semantics* (a message must be completely processed for the actor to accept a new message) [34].

The automaton of a continuous actor (mode) is similar to that of a normal actor. The major difference is that now a message server reaction can have a duration during which some ODE can be applied, or simply a given time is allowed to pass. At the end of the time duration, the mode actor typically concludes its behaviour by sending a message to an accessor actor (e.g. the `Controller` for the `Train` and `Button` mode actors). Due to the continuous behaviour, a mode actor naturally restricts the kind of messages it can handle during its activity. In general (see also later in this paper) an activated mode could be suspended (or stopped) so as to be subsequently resumed from the state it was left off. No other messages can be admitted. For demonstration purposes, the `Train` mode (see Fig. 1), once initialized, always remains active and can send and `EXT_MOVE` to the `Controller` at any time in the interval [`MIN_EMD,MAX_EMD`]. The `Button` mode (see Fig. 2), instead, is activated by the `Controller` as soon as the `Train` receives the command to start moving and its current status is stopped. Being active, the `Button` can issue an `EXT_OPEN` message to the `Controller` at any time in the interval [`MIN_EOD,MAX_EOD`]. The following are the scenario timing parameters for the distributed train model (1 time unit is the network delay for a message).

```
//Scenario parameters
const int MIN_EMD=10;//Minimum External Moving Delay
const int MAX_EMD=20;//Maximum External Moving Delay
const int MIN_EOD=0;//Minimum External Open Delay
const int MAX_EOD=5;//Maximum External Open Delay
const int NET_DELAY=1;
```

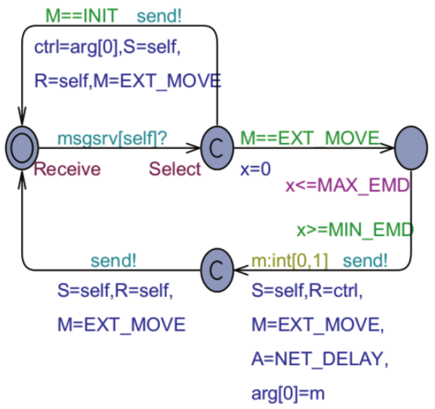

Figure 1. The `Train` mode automation

The `Door` actor mode (see Fig. 3) "actuates" a door command and replies to the Controller about the new reached state). It should be noted, in the Figures from 1 to 3 and Fig. 5, the use of `NET_DELAY` as the *after* time of any networked message. The `Main` actor (see Fig. 4), first initializes the model through the `setup`() function which takes care of actor partitioning to PUs, then sends instantaneous `INIT` messages to all the model actors, along with any initialization data (e.g., acquaintances). The `Controller` actor (see Fig. 5) stores the moving status of the Train and the `door_opened`/`door_locked` status of the `Door`. Such state variables dictate in which way the `Controller` react to an external event. For example, if `moving` is false (the `Train` is stopped) and a command to start moving arrives (`arg`[0] is true), the `Controller` first activates the `Button` then sends a `CLOSE` message to the `Door` which will be followed by a `LOCK` message. It should be noted that the `OPEN` message of the `Door` actor has a lower ID (greater priority) than e.g. a `LOCK` message. Other details of Fig. 5 should be self-explanatory.

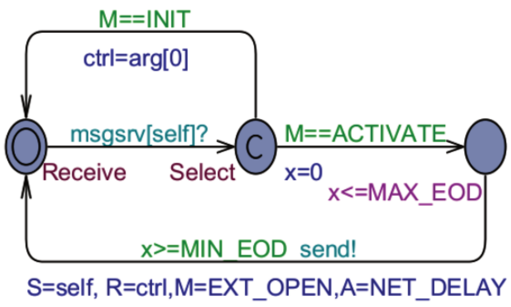

Figure 2. The `Button` mode automation

3.1.2. A Scheduler for deterministic message delivery

Fig. 6 shows the `Scheduler` automaton which is a more general and enhanced design w.r.t. the preliminary design reported in [33]. The `Scheduler` operates in discrete time and takes into account the non-deterministic scheduling of messages originated in continuous mode actors. The `Scheduler` behaviour basically collects into hidden data structures the attributes of a message `send` (or of a `delay` operation which can only be used in discrete actors). Timing information is held in relative form. Normally the `Scheduler` finds itself into the `Schedule`

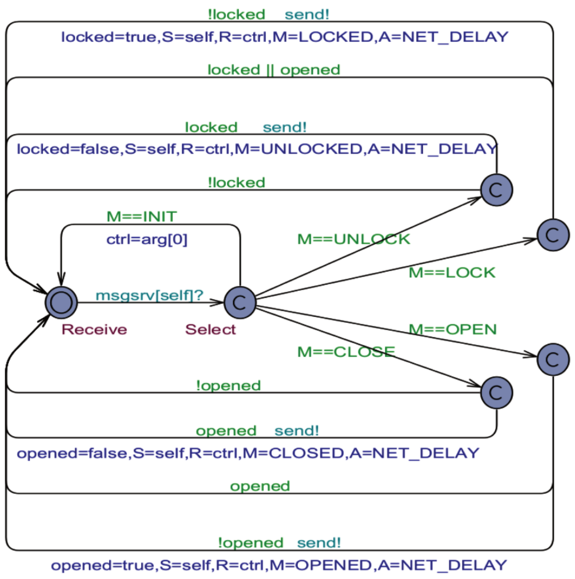
!locked send!
locked=true,S=self,R=ctrl,M=LOCKED,A=NET_DELAY
locked || opened
locked send!
locked=false,S=self,R=ctrl,M=UNLOCKED,A=NET_DELAY
!locked
M==INIT
ctrl=arg[0]
M==UNLOCK
M==LOCK
msgsrv[self]?
Receive
Select
M==OPEN
M==CLOSE
!opened
opened send!
opened=false,S=self,R=ctrl,M=CLOSED,A=NET_DELAY
opened
!opened send!
opened=true,S=self,R=ctrl,M=OPENED,A=NET_DELAY

Figure 3. The `Door` mode automation

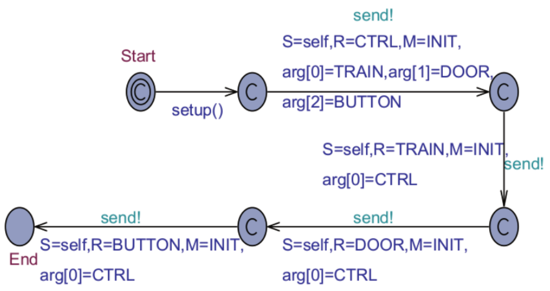

Figure 4. The `Main` automation

normal location, from which it can exit at the end of an actor message server reaction (it is recalled that a message server body, in a normal actor, is realized as a cascade of committed locations, which surely terminates *before* the `Scheduler` can abandon the `Schedule` location). Exiting from the normal location `Schedule` can actually occur provided the current actor finished the execution of its message server, and there are some pending scheduled events in the scheduler (the function `pending`() returns true). The fictitious synchronization on the `decision` broadcast and urgent channel, which is sent (!) but received by no one, forces abandoning the `Schedule` location as soon as it is possible. When exiting `Schedule`, the (or one of) most imminent event (message or delay) is determined and let t be its minimum *after* time determined by function `mt`(). Such a relative time is allowed to pass, by increments of 1 time unit, in the `TimeAdvance` location with the help of the local clock `x`. When the t time units are elapsed, `TimeAdvance` is exited and: (a) all the scheduled events are updated by decrementing their occurrence time by t through the `tup(t)` function; (b) the identity of the next event to deliver is determined by the function `dopc()` (see below for details). After that, the event is allowed to occur, e.g., a message is dispatched to its destination actor or the delay is fired.

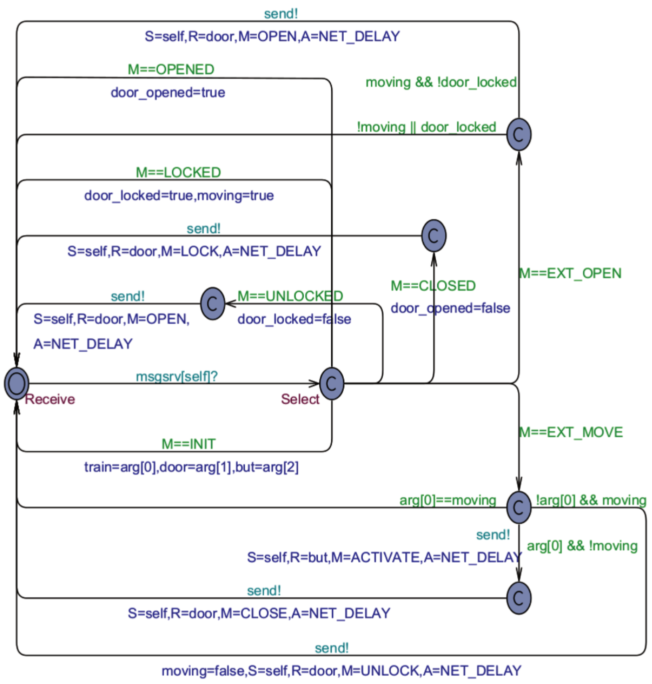

Figure 5. The `Controller` automation

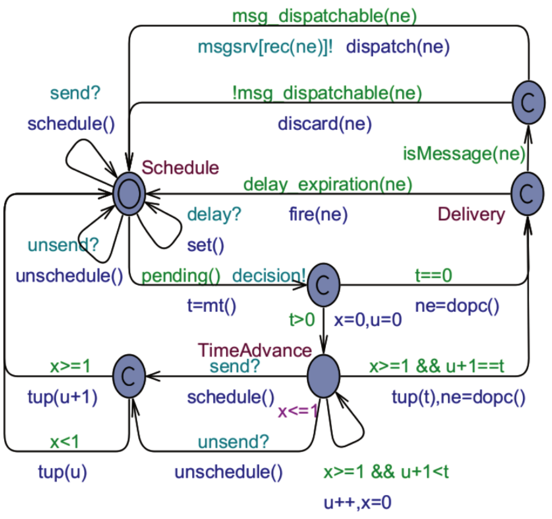

Figure 6. The `Scheduler` automaton with deterministic message delivery

A subtle point in Fig. 6 is the handling of a non-deterministic message sent by a mode actor while the `Scheduler` stays in `Schedule` or in `TimeAdvance`. A send heard in `TimeAdvance` implies a new message is added to the Scheduler data structures. Then the (rounded) discrete time elapsed from the entrance into `TimeAdvance` to the arrival of the non-deterministic message, is evaluated and its amount subtracted from all the after time fields of *previously* scheduled messages (the just scheduled message is ignored).

The `Scheduler` also allows an actor to request a previously scheduled message to be anticipately removed, through the use of the `unsend` channel and `unschedule()` function. The unsend operation assumes a unique message is identified through its attributes (sender, receiver, message id and so forth).

The `Scheduler` behaviour ensures messages are delivered in timestamp order (timestamps are established by the *after* time attributes). Simultaneous messages, though, are delivered using the *precedence constraint* rules which underlie the definition of the `dopc()` function (**d**eterministic **o**rder **p**recedence **c**onstraints). Such rules, similarly to [12], guarantee a deterministic delivery order as follows. First, actors are supposed to be assigned a unique identifier which acts as priority. Secondly, each message server in an actor interface is associated with a unique identifier which, for example, mirrors the textual appearance of message servers (see also Fig. 8). When multiple simultaneous messages exist to be delivered, the `dopc()` determines the next message to dispatch by first applying the priority of the receiving actors. In the case multiple messages exist which are directed to a same highest priority actor, the message having the highest priority (i.e., with lowest unique identifier) is selected. Finally, when multiple actors (senders) have sent a same message to a same receiver (with highest priority), the priority of the senders decides which message is delivered first.

3.1.3. System composition

The Train-Door Controller model was configured by the following system command-line:

```
system Train,Door,Button,Controller,Scheduler,Main
```

which specifies the instances of the timed automata (actor template processes) which are parallel composed for the analysis. All the possible action interleavings of the component automata are investigated during the exhaustive model checking activity. The Uppaal model checker [18] builds the timed transition system (*state graph*) of the model, which contains all the possible execution states of the model, and makes it possible to inquire properties which could hold on all the reachable states (see the invariantly query A[] in the Table 1) or which can hold on some reachable state (see the existential query E<> in the Table 1).

3.1.4. Analyzing the Train-Door Controller Model

Despite the continuous time in the Train and Button mode actors, the absence of ODEs and of double variables makes it possible for the train-door model to be exhaustively model checked. It is worthy of note that the proposed reduction of Theatre actors onto Uppaal timed automata purposely uses only broadcast channels for communication/synchronization. This design is compatible with the needs of both model checking and/or statistical model checking activities.

Properties of the train-door reduced model were assessed by TCTL queries, some of which are collected in Table 1. For brevity, queries which were issued for debugging purposes, e.g., assuring that each actor receives only the expected messages of its interface, are not reported.

Query 1 (satisfied) guarantees that in all the states of the model state graph, there is no deadlock. In other terms, the model can always make some progress (*liveness*). Queries 2 and 3 ensure that there is no state where the train is moving and the door is opened or unlocked (a fundamental *safety* property for the model). Query 4 is a functional check assessing that it is not possible for the door to be simultaneously opened and locked. Instead (query 5) it is perfectly possible for the door to be closed *but* not yet locked. Query 6 ensures that there is at least one state where the door can be closed *and* locked. Query 7 reinforces the fact that in all the states where the door is locked it is necessarily also closed. Query 8 checks if there is a state where the `Controller` receives an `EXT_OPEN` message from the `Button`, and the door is still unlocked. The query is satisfied. Query 9 guarantees that it is possible that the Controller receives an EXT_OPEN and the door is locked (in this case the button will be ignored, because it causes no state change in the door). Queries 10 and 11 are based on the leads-to operator (–>) which checks if invariantly, starting from a given state, necessarily or inevitably a state can be reached where the condition after –> will hold. Query 10, in particular, confirms the expectance that on the arrival of an EXT_OPEN not always the train will be stopped and the door opened. On the other hand, query 11 says that starting from a state where an EXT_OPEN is received by the Controller, the train is moving but the door is still unlocked, it effectively happens that the train will be stopped and the door opened.

Table 1. TCTL queries for model checking the Train-Door Controller model

#	Query	Result
1	A[]!deadlock	satisfied
2	E<> Controller(CTRL).moving && Door(DOOR).opened	not satisfied
3	E<> Controller(CTRL).moving && !Door(DOOR).locked	not satisfied
4	E<> Door(DOOR).locked && Door(DOOR).opened	not satisfied
5	E<>!Door(DOOR).opened && !Door(DOOR).locked	satisfied
6	E<> !Door(DOOR).opened && Door(DOOR).locked	satisfied
7	A[] Door(DOOR).locked imply !Door(DOOR).opened	satisfied
8	E<> Controller(CTRL).Select && M= =EXT_OPEN && !Controller(CTRL).door_locked	satisfied
9	E<> Controller(CTRL).Select && M= =EXT_OPEN && Controller(CTRL).door_locked	satisfied
10	Controller(CTRL).Select && M==EXT_OPEN –> Controller(CTRL).door_opened && !Controller(CTRL).moving	not satisfied
10	Controller(CTRL).Select && M==EXT_OPEN && Controller(CTRL).moving && !Controller(CTRL).door_locked –> Controller(CTRL).door_opened && !Controller(CTRL).moving	satisfied

On the light of the query results in Table 1, the train-door model was found correct from both the functional and the temporal behaviour. A side-benefit of the use of the deterministic Scheduler, is a reduction of the partial-order on the model state graph. In fact, due to the deterministic message delivery, in many cases the exiting from a node of the state graph can happen in a single manner, that is there is one only exiting transition. All of this improves the performance of the model checker. For example, query 1, which checks the absence of deadlocks, terminates in 0.015s on an Asus ZenBook Win10 laptop.

3.1.5. Java Programming Style

Theatre is currently implemented in Java [14] with the help of Java reflection and annotations. To give an idea of the actor programming style, Figures 7 and 8 reproduce respectively the Button mode and the Controller actor. The Button version is supposed to operate during analysis. When activated, the button evaluates a non-deterministic after time in the time window [MIN_EOD,MAX_EOD] and schedules a corresponding timed "ext_open" message to the controller. A normal actor must be programmed as a derived class of the Actor abstract base class which exposes all the fundamental services (the non-blocking send, the move operation, the value of current time now() which refers to the time notion provided by a control layer, and so forth), and Mode which is a specialization of Actor for continuous time actors. Scenario parameters are supposed to be defined into a G class as static entities which are directly imported by the application actor classes.

```
public class Button extends Mode{
    private Controller ctrl; //acquaintance
    @Msggsrv public void init( Controller ctrl ) { this.ctrl=ctrl; }
    @Msggsrv public void activate() {
        double after=Math.random()*(MAX_EOD-MIN_EOD)+MIN_EOD;
        ctrl.send( after, "ext_open" );
    }//activate
}//Button
```

Figure 7. The `Button` mode in Java

```
public class Controller extends Actor{
    //acquaintances
    private Train train;
    private Door door;
    private Button but;
    //state variables
    private boolean moving=false, door_opened=true, door_locked=false;
    @Msgsrv public void init( Train train, Door door, Button but ) {
        this.train=train; this.door=door; this.but=but;
    }//init
    @Msgsrv public void ext_move( Boolean move ) {
        if( move==moving ) return;
        if( !move && moving ) { //stop request for the moving train
            door.send( NET_DELAY, "unlock" ); moving=false;
        }
        else {// move && !moving - move request for the stopped train
            but.send( NET_DELAY, "activate" ); door.send( NET_DELAY, "close" );
        }
    }//ext_move
    @Msgsrv public void ext_open() {
        if( !moving || door_locked ) return;
        door.send( NET_DELAY, "open" );
    }//ext_open
    @Msgsrv public void unlocked() { door_locked=false; door.send( NET_DELAY, "open" ); }//unlocked
    @Msgsrv public void locked() { door_locked=true; moving=true; }//locked
    @Msgsrv public void closed() { door_opened=false; door.send( NET_DELAY, "lock" ); }//closed
    @Msgsrv public void opened() { door_opened=true; }//opened
}//Controller
```

Figure 8. The `Controller` actor in Java

3.2. Admission control system for home appliances

A not trivial admission control system (ACS) for the electrical appliances in a smart home is considered. The modelling example is an original, enhanced version of the design reported in [24]. In the new model, deterministic actors are used along with the Scheduler automaton shown in Fig. 6. In addition, continuous mode actors, like in the Train-Door Controller example of Section 3.1, follow a behavioural modelling close to that of discrete actors. The possibility offered by the Scheduler automaton is exploited to possibly un-schedule and remove a no longer useful scheduled message.

A model for the ACS was developed according to the logical architecture shown in Fig. 9. Rounded rectangles represent discrete actors. Polygonal boxes denote continuous mode actors. As one can see in Fig. 9, mode actors come in pairs: a "normal" continuous mode (possibly with ODEs in its behaviour) and a specialized version of it devoted to *prediction* purposes. As usual, mode actors abstract physical components of the system, and the overall model is analysed in the context of the deterministic cyber part integrated with non-deterministic messages generated by the continuous mode actors. The generation of such messages and their handling in the cyber model, represent the time points where continuous time is reconciliated with discrete time.

Two kinds of electrical appliances are distinguished depending on the associated power curve: *tabular loads* and *continuous loads*. The consumption curve of a tabular load consists of a level-based curve which can be described numerically by two tables: one for the duration of each horizontal power segment, and a second one for the power value of each segment. The consumption power curve of a continuous load is instead represented by ODEs. In particular, in the model of Fig. 9, three tabular power loads are considered: a *washing machine* (WM), a *boiler* (BL) and a *hair dryer* (HD), and one continuous load in the form of an *HVAC* (HV) for climatization needs. All the electrical appliances are supposed to be operated (for activation/deactivation) through a smart plug. The mission of the ACS model is to orchestrate the power loads as they announce their arrival, so as to ensure, at each moment, the total consumed power does not exceed an assigned *threshold* (for example, contracted with the provider so as to reduce the electricity costs). The arrival requests of the loads are specified in the behaviour of the `LoadManager`. Of course, the worst case occurs when all the power loads request to be simultaneously active. The `Controller` evaluates the admission of a load by temporarily suspending the operation of all the active loads, and by examining the remaining power curve of each partially executed load together with that of the newly arrived load. The new load will be admitted only in the case the new power requests plus those of the currently active loads in no case will be beyond the threshold until load terminations. If a new load can't be accepted at current time, its request is placed in a *deferred buffer* so as to be analysed again after a *defer time* (DT) which is one of the fundamental behavioural parameters of the ACS model.

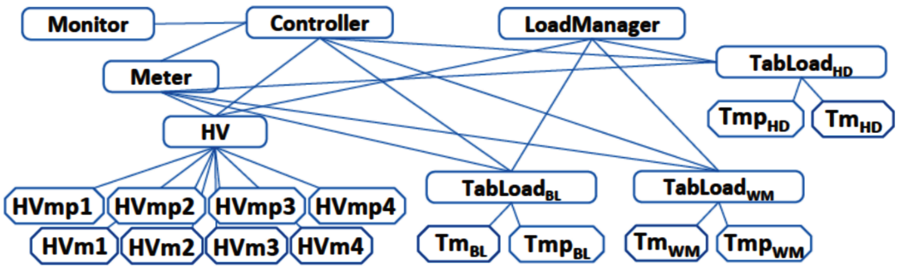

Figure 9. Model architecture for the Admission Control System (ACS)

Realizing the mission of the ACS model is challenging, because the model must alternate, each time is required, an *operating phase* with a *prediction phase*, followed by an *operating phase* and so forth until all the power loads terminate. During a prediction phase, not only the active loads must be suspended, but the prediction of future power requirements, starting from the suspension time instant, must be properly studied. Toward this, the normal load modes are suspended (e.g., by setting to 0 the first derivative of a double variable) and the prediction mode versions are activated by the `Controller`. A prediction mode first executes a *coasting forward* phase on the appliance power curve (starting from its beginning), so as to reach the suspension time point. Then a *prediction* is accomplished by inspecting future power requests of the load until its termination. As soon as all the involved prediction modes inform the `Controller` they have finished, the `Controller` can decide if or not to accept the new load. After that, all the suspended power loads are re-activated from the point they were last suspended or, would the newly arrived load be accepted, from its beginning.

Fig. 10 shows the power consumption curve of the washing machine tabular load. Fig. 11 depicts the power curve of the HVAC (HV) continuous load. Both pictures were generated by the Uppaal statistical model checker tool [19] (see also later in this paper), by selecting the chosen load as the only one activated by the `LoadManager` and by choosing a threshold value of 7.0 which ensures the load is activated and runs to completion without deferments. In both the Figures 10 and 11 it is clearly visible the prediction time which in any case has to be spent before the load is put into execution. The prediction curve is shown with a negative sign for clarity. In addition, only one active load exists which finally terminates.

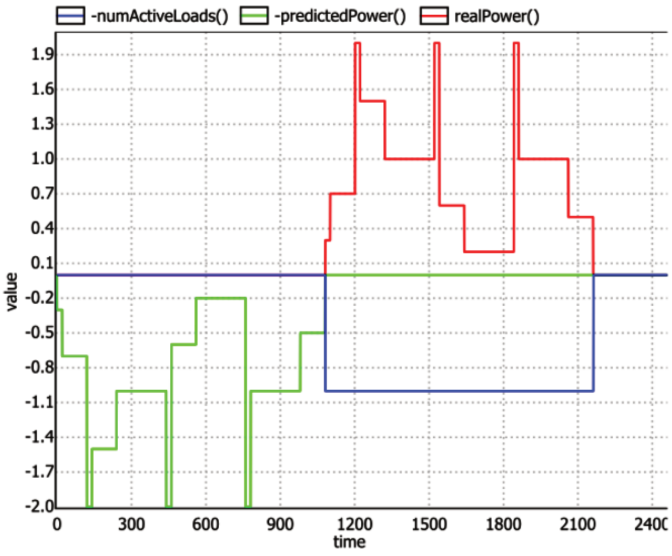

Figure 10. Power curve of tabular washing machine

From Fig. 11 it emerges that the HVAC, as implied by the use of an internal inverter, admits four behavioural regions which correspond to *rising*, up to keep the high reached power level, *falling* and then *down* where it is kept constant the lower last value until termination. Each region is achieved by a first order differential equation (ODE) solved over a given time interval.

To give an idea of the Uppaal modelling details of a power load, Fig. 12 illustrates the HVAC actor automaton, which depends on four continuous mode actors each associated with a mode prediction actor. Each edge in Fig. 12 is annotated (as in Section 3.1) with a *guard* (green), a *synchronization* (azure) and an *update* (blue). Mode actors have respectively the ids `M1, M2, M3` and `M4`, and their predictor counterparts have the ids `M1P, M2P, M3P` and `M4P`. `HV` receives at the initialization time (from the `Main`) the ids of the initial mode and of its predictor mode which are respectively held in the local variables `cm` and `pm`. The `HV` modes are organized according to a finite state machine. When a mode terminates, it sends to the HVAC a `SAMPLE` message which carries as an argument the id of the next mode. Similarly, a `SAMPLEP` message is received by the predictor mode with the id of the next predictor mode as an argument. The other messages of the HVAC are received from the `Controller` during the operational or prediction phases. Fig. 13 depicts the model of the `Hvm1` automaton. Fig. 14 shows the associated `Hvm1p` predictor mode.

Once started, `Hvm1` will be in the `Hold` location where the *rising* part of the HVAC behaviour (see Fig. 11) is achieved through the ODE $p' == K1*(h-p)$ which applies for $T1$ time units. Global clock variables t and p denote respectively the effective operational elapsed time, and the current value of the generated power. When suspended, the `Hvm1` mode reaches the

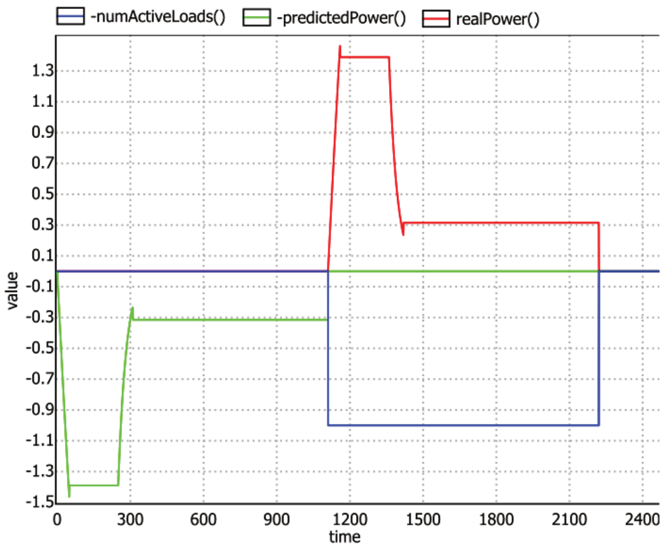

Figure 11. Power curve of the continuous HVAC

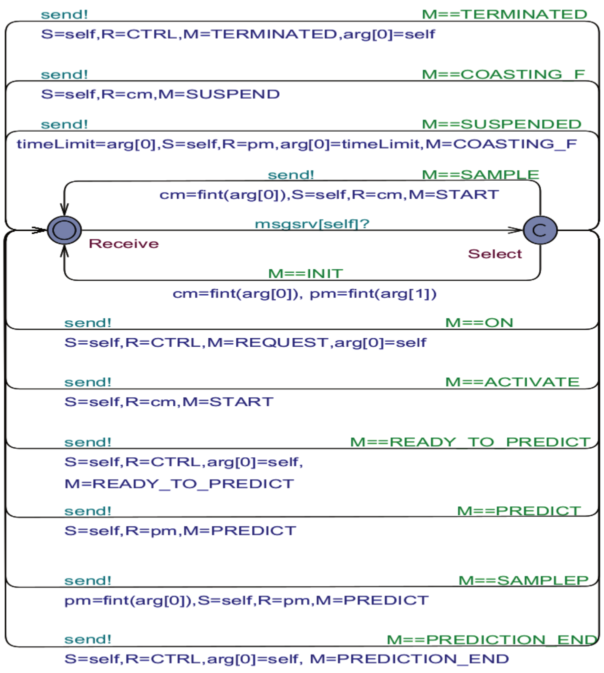
send!
M==TERMINATED
S=self,R=CTRL,M=TERMINATED,arg[0]=self
send!
M==COASTING_F
S=self,R=cm,M=SUSPEND
send!
M==SUSPENDED
timeLimit=arg[0],S=self,R=pm,arg[0]=timeLimit,M=COASTING_F
send!
M==SAMPLE
cm=fint(arg[0]),S=self,R=cm,M=START
msgsrv[self]?
Receive
Select
C
M==INIT
cm=fint(arg[0]), pm=fint(arg[1])
send!
M==ON
S=self,R=CTRL,M=REQUEST,arg[0]=self
send!
M==ACTIVATE
S=self,R=cm,M=START
send!
M==READY_TO_PREDICT
S=self,R=CTRL,arg[0]=self,
M=READY_TO_PREDICT
send!
M==PREDICT
S=self,R=pm,M=PREDICT
send!
M==SAMPLEP
pm=fint(arg[0]),S=self,R=pm,M=PREDICT
send!
M==PREDICTION_END
S=self,R=CTRL,arg[0]=self, M=PREDICTION_END

Figure 12. The HVAC (HV) model automation

`Suspend` location where the values of t and p are frozen by putting to 0 their first derivative. The `END` message is self-sent by `Hvm1` at each (re)starting time, with an $after$ time set to the remaining time for mode termination. Would a `SUSPEND` message be received, the pending scheduled `END` is removed and the suspension state of the `Hvm1` mode is entered. The `END` message is also removed

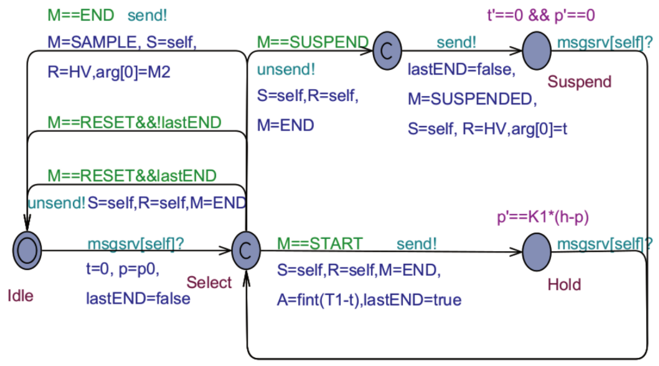

Figure 13. The automaton model of Hvm1 actor mode

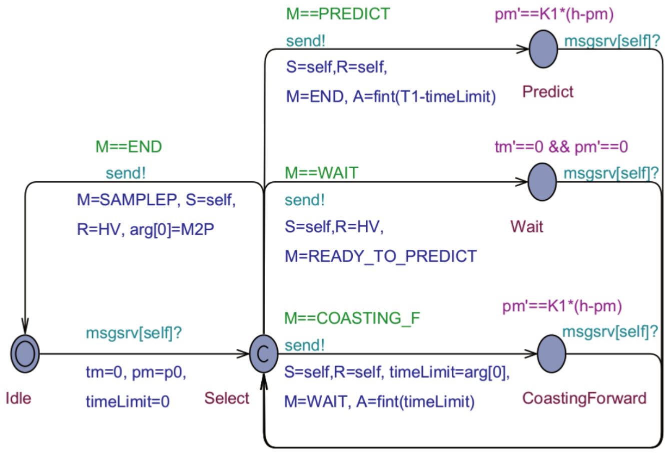

Figure 14. The automaton model of the Hvm1p actor mode

would the `Hvm1` be reset. On the arrival of `END, Hvm1` sends to the hvac a `SAMPLE` message with the argument `M2` as the id of the next mode. Of course, the current value of the computed power can always be accessed through the p global clock. The prediction mode `Hvm1p` exploits the same `ODE` of `Hvm1` in two cases: when it is in the `CoastingForward` or the `Predict` locations. Clock variables tm and pm are used by `Hvmp1` for predictive time and predictive power. Coasting forward is performed by `Hvmp1` by receiving the time limit at which the normal actor mode was last suspended. When such time limit is elapsed, coasting forward finishes and the `Hvmp1` receives the self-sent `WAIT` message which puts the predictor mode in a state waiting to start prediction, with tm and pm frozen. During prediction, the `Hvm1p` mode virtually generates the future values of the HVAC consumption curve whose final value is held in pm. The prediction phase ends when `Hvm1p` receives the `END` self-sent message, which implies a `SAMPLEP` message is sent the HVAC whose argument contains `M2P` as the id of the next predictor mode. In a similar way were modelled the other modes of the HVAC, not shown for brevity. The following are the ODEs used respectively by `Hvm2, Hvm3` and `Hvm4`: $p^{'} == 0$; $p^{'} == K2 * p$; $\mathrm{p}^{'} == 0$. The same modelling logic was applied to tabular loads where instead of using ODEs, the next power samples (during the operational or prediction phases) are achieved by modes and predictor modes by consulting tabular values. Fig. 15 and Fig. 16 show respectively the `Meter` and the `Monitor` components of the ACS model (see Fig. 9).

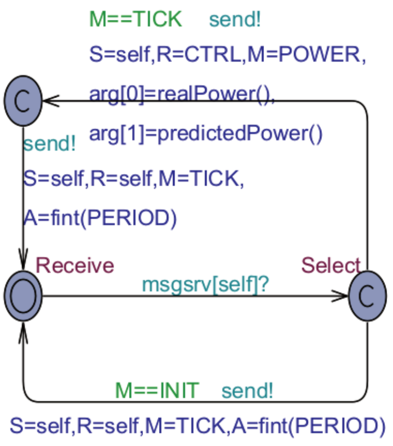

Figure 15. The Meter actor

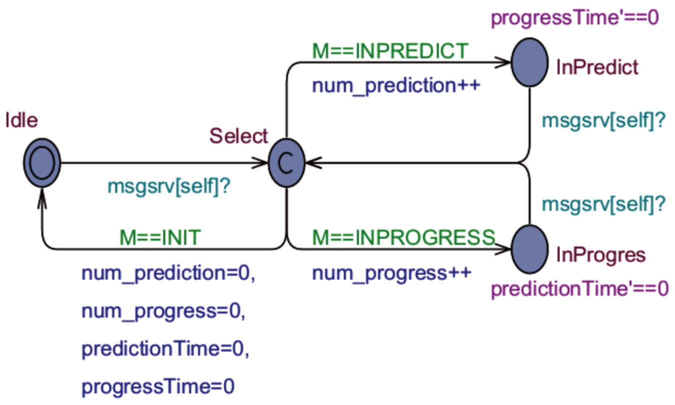

Figure 16. The Monitor mode actor

`Meter` is a periodic actor that at each `TICK` message sends a `POWER` message to the `Controller` with arguments the sampled values of the real and the predicted power. For the experiments, the `PERIOD` of the `Meter` was set to 1.0. The `Monitor` mode actor simply accumulates information about the number and timing of the operational and the prediction phases.

The `Meter` defines the "discrete time view" of the cyber model about the continuous physical part model. Whereas the cyber part model relies on the latest sampled values of the real and the predicted power, such quantities evolve in a "continuous" way in the physical power loads. In reality, in the Uppaal model under the Statistical Model Checker (SMC) [19], the advancement of the consumption power also proceeds discretely, by the discretization step used by the underlying ODE solver, which by default is set to 0.01 time units.

3.2.1. *Experimental Analysis*

Since the use of ODEs in mode actors, the behaviour of the ACS model was assessed by using the Uppaal Statistical Model checker (SMC) [19] and then through simulations. In the considered scenario all the appliances make altogether the requests for their admission at system start-up. The goal is to evaluate how the system evolves when different values for the `threshold` and the defer time `DT` are chosen. Fig. 17 refers to the case where the `threshold` is set to 6 and `DT` is set to 50 tu. The picture was generated by Uppaal SMC following the query:

```
simulate [<=9000] { realPower(), -predictedPower(), threshold,
                                 -threshold, -numActiveLoads() }
```

From the figure, it emerges that system is capable to maintain the real power consumption under the specified threshold. By looking at the predicted power consumption, it follows that the system can admit three loads without violating the threshold. In fact, as soon as the admission request of the fourth load is considered, a violation is detected (see the spike of the blue curve which goes below the threshold line at about 4000 tu) and the load admission is postponed. This is witnessed by the fact the number of active loads remains equal to three up to about 7000 tu. From the figure it also emerges that the ACS model performs five predictions during its operation which correspond to the time windows in which the curve of the real power consumption goes to zero. The spikes of the real power consumption curves occurring between these windows are directly related to the value **DT**. In fact, following an admission request which is deferred, either because the admission would violate the threshold or because another admission request is under evaluation, a `DT` time interval elapses during which the system can evolve. The model requires about 9000 tu to schedule and to operate to completion all the appliances.

Fig. 18 shows instead the time spent by the ACS model in the prediction and in the progress (real execution) phases. Up to 7200 tu, the system evolves by almost exclusively making predictions. Beyond this time limit, however, the system is able to really evolve and prediction time stops to grow. Fig. 18 was drawn by Uppaal SMC through the query:

```
simulate [<=9000] { Monitor(MONITOR).progressTime,
                    Monitor(MONITOR).predictionTime }
```

It is worth noting that although during the Uppaal analysis the prediction time can be virtually preponderant w.r.t. real execution time, in a physical implementation it is expected the opposite behaviour, with prediction which should be a very small part of the real system operation time.

The correct behaviour of the model was also checked by the following MITL [19] query:

```
Pr[<=9000](<>realPower()>threshold)
```

which asks to quantify the probability of the event *"does it exist an instant where the total consumed real power exceeds the threshold?"*. Uppaal SMC, after 29 runs, proposes a confidence interval of [0,0.0981446] with confidence degree 95%, which testifies, with the adopted default statistical parameters, e.g., an uncertainty in confidence intervals of ε=0.05, the event is (practically) impossible.

Another set of experiments were carried out by adopting a lower value for the `threshold`. Specifically, a `threshold` of 5 (KW) was considered, with an unchanged value of `DT`=50 tu. In this scenario, the ACS model has to defer more admission requests of loads in order to meet the new constraint on the `threshold` with respect to the previous case. In Fig. 19 it is shown that, in this case, the overall time needed to schedule all the loads grows at about 20000 tu, and twelve prediction phases are required to guarantee the proper scheduling of the loads. The figure also confirms the `Controller` behaviour is fine because in no case the real power goes over the `threshold`. During the prediction phase, instead, some cases occur in which the threshold is violated, and these are the cases where deferring the loads is mandatory. Fig. 20 depicts the time spent in the prediction phase with respect to the time in which the model really progresses. Obviously, the prediction time is now much higher than the progress time.

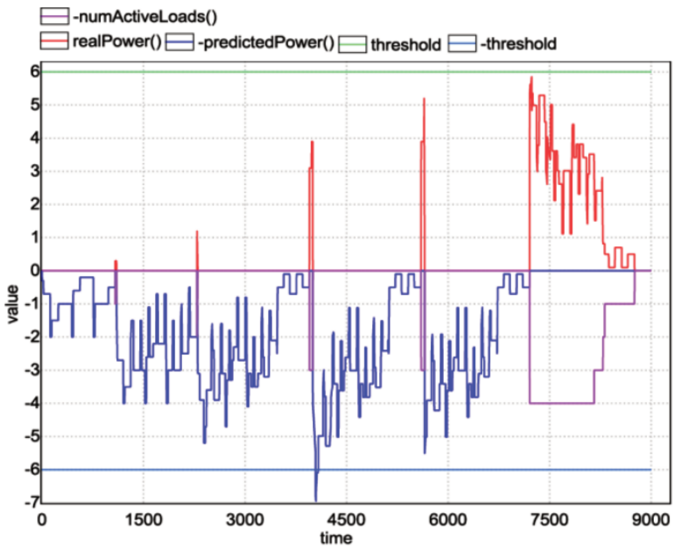

Figure 17. The real and predicted power consumption (in KW) and the number of active loads during ACS operation with threshold=6 KW and DT=50 tu

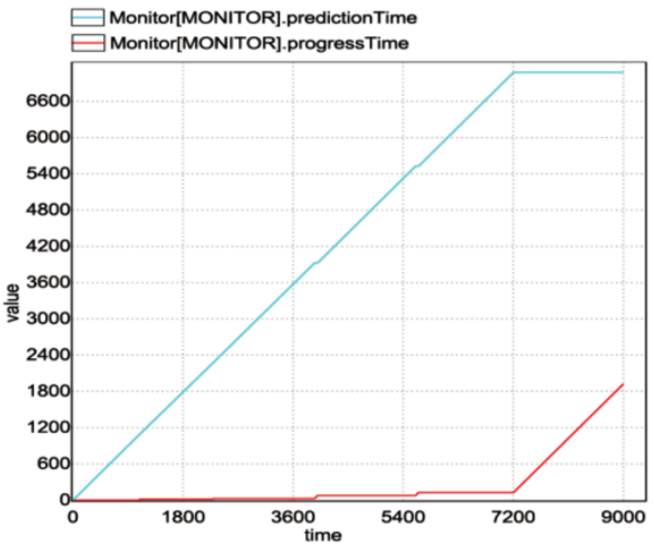

Figure 18. The time spent by the ACS system in the prediction phase and in the real execution (progress) with threshold=6 KW and DT=50 tu

In order to reduce the number of the prediction phases required to schedule all the arriving loads, a good choice is that of augmenting the `DT` value. Increasing this value permits the model to evolve between two consecutive prediction phases, because more chunks of the active loads get consumed. In other terms, the already admitted loads can conclude their execution quickly, thus permitting the admission of new loads. This behaviour was confirmed by another set of

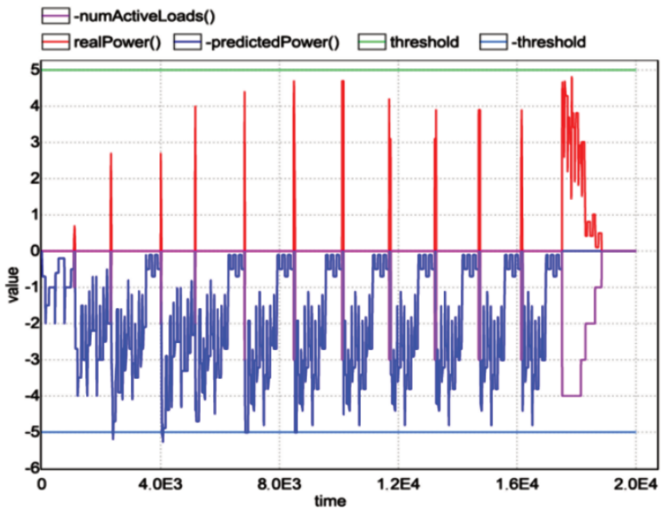

Figure 19. The real and predicted power consumption (in KW) and the number of active loads during ACS operation with threshold=5 KW and DT=50 tu

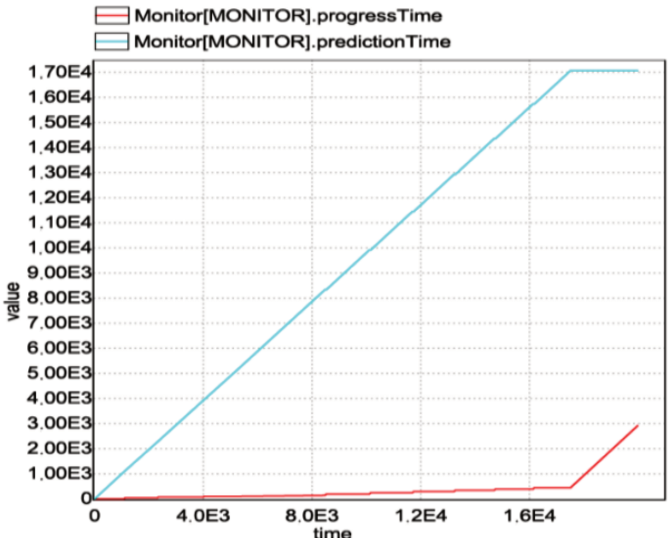

Figure 20. The time spent by the ACS system in the prediction phase and in the real execution (progress) with threshold=5 KW and DT=50 tu

experiments which were carried by setting `DT`=200. This new scenario is not here reported for brevity, but it confirms that a time of 9000 tu is enough for scheduling all the loads with a threshold of 5 KW, and the number of prediction phases reduces to five.

As a final remark, it is important to point out that due to the use of deterministic message delivery, any simulation started from the same scenario parameters always generate the identical model behaviour witnessed by pictures like Fig. 17 and Fig. 18.

The experimental results confirmed the ACS model is correct from both the functional and the temporal viewpoint.

All the experiments were executed on a laptop Win10 Asus ZenBook 14, Intel Core i7-8565U, CPU@1.80GHz, 16GB RAM, using the latest 64bit development version of Uppaal 4.1.25-5.

4. Transitioning a Theatre Model toward Implementation

The following outlines problems and suggests a possible development guideline when transforming a Theatre-based CPS model to implementation. A fundamental issue is the management of continuous mode actors in the context of model continuity whose goal is the achievement of a physical system compliant or as "faithful" as possible with respect to the realized abstract model and its analysed properties. Discrete actors and their message exchanges remain basically unaltered when moving to the synthesis phase. The control layer of the application is required to operate in real-time and not in simulated time. Theatre has a library of control forms [14], [32] tailored to the system lifecycle needs. Whereas determinism in message delivery can be easily established in the control layer for a standalone or parallel setting, it is more difficult to achieve in a distributed context. The problem is how to reach certainty, in a remote theatre and at a given time, that all the messages directed to its actors to be processed at current time were actually received and then it is possible to apply the precedence constraint rules for the ordered delivery of messages. The concept adopted when distributing Reactors [9] is borrowed from Ptides [35] and Google Spanner [36] and consists in estimating the maximum time [10] (including the maximum latency in network connections, and the bound on the clock synchronization error among distributed computing nodes) which could be allowed to pass, from current time, for an external message to be received. Determinism in a distributed Theatre system can be based on the same concept as in Reactors. However, work is in progress for evaluating different solutions, e.g., using in combination a distributed simulator with the real-time control form, to anticipate, time to time, the possible message arrivals at the various theatre nodes.

According to the development methodology proposed in this paper, mode actors often remain in the system implementation by assuming the shape of normal discrete actors, although with a more specific behaviour. As has been anticipated in Section 3, Theatre advocates the use of an `envGateway` as a boundary component between the cyber and the physical part. Similar provisions are adopted in [10]. A realization of the `envGateway` necessarily has to be strongly connected to the control layer of a Theatre and should possess two interfaces: one toward the physical part (sensor and actuators) and one toward the cyber part. Interfacing physical devices can be achieved, in a case, by inexpensive peripheral hardware [22] like Arduino Uno, Raspberry Pi and so forth, to which the external devices are physically attached. The peripheral hardware components are then linked to the `envGateway` by a network infrastructure and protocols (e.g., MQTT) which can reduce in some cases to the use of a serial communication line or a wireless connection. Such an infrastructure serves the purposes to grab sensor data and carry it to the `envGateway`, or to execute an external command directed to an actuator. As a concrete design (see also [22]), multiple input/output threads for reading/writing from/to communication lines and finally with selected input/output devices can be introduced. Then suitable concurrent data structures in the `envGateway` can be used to safely store the last value read from a sensor or the command to be forwarded to an actuator.

The interface side vs. the cyber part is actor-based with interactions realized by message passing. Obviously, since the generation of a message in the `envGateway` toward a cyber actor is intrinsically non-deterministic, the control layer to which an instance of the **envGateway** is tied, can receive any external generated message through a lock-free buffer like a ConcurrentLinkedQueue object [17] of the Java collection framework. The event-loop of the control layer then will sense, at each iteration, the input buffer for an external message and, if there are any, extract it and schedule it on to the internal scheduling message data structure. The above described design configures the `envGateway` as a special actor. Knowledge of the `envGateway` actor can purposely be restricted to mode actors which remain in a final implementation. To simplify the exchange of messages between the `envGateway` and mode actors a publish/subscribe interaction scheme can be used. This way, each time a new sensor value is available, the subscriber actor will receive a message with the data as an argument from the `envGateway`.

The Train-Door Controller model can be moved to synthesis by maintaining the mode actors, now turned into normal actors, and reifying their behaviour so as to interact with the `envGateway`. The physical button device can transmit its pressing signal to the `Button` actor by a publish/subscribe interaction.

Fig. 21 represents the architecture of the ACS model described in Section 3.2, adapted to implementation purposes.

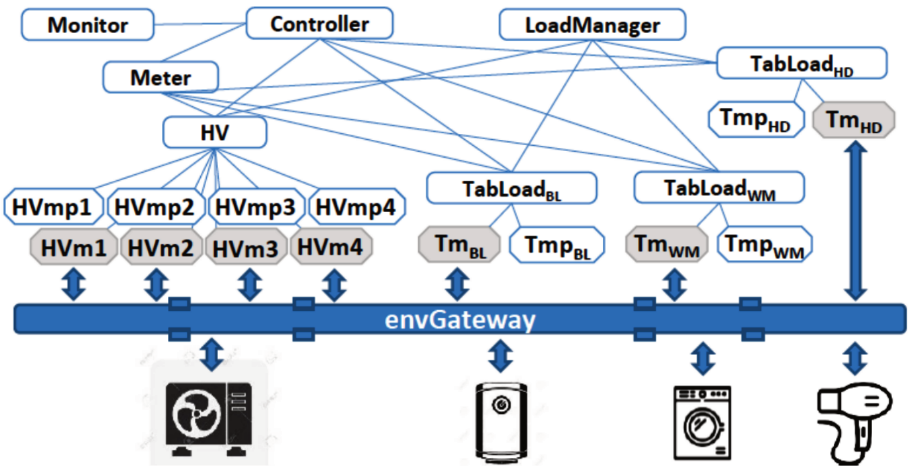

Figure 21. Transitioning architecture of the ACS model to synthesis mediated by the `envGateway`

Grey boxes denote normal mode actors with ODEs which remain with a redefined behaviour (no more based on ODEs) when moving the model to a synthesis. Such mode actors, in fact, represent concrete devices in the physical system with which the interactions are mediated by the `envGateway`. Predictor mode actors, instead, remain with their ODEs and internal behaviour because the prediction process is still required during the real operation of the ACS system. It is worthy to note that the above described development guideline represents just one way for the reification of a CPS model. As an interesting alternative, a modeller could represent explicitly the `envGateway` already in the early modelling phase of a CPS, so as to improve specifically the model continuity activities.

5. Conclusions

This paper proposes a methodology for the development of cyber-physical systems (CPS) which is based on the Theatre actor framework [15], [16] and on model continuity during the system lifecycle [21, 22]. A key factor of Theatre is the fact that it is control-based. The evolution of actors can be made deterministic [9] by customizing the control layer which reflectively regulates the message delivery order. Theatre can manage both normal discrete actors, and continuous mode actors whose behaviour can be specified by ODEs. A reduction was defined which allows one to translate a Theatre model into the terms of the timed automata of the Uppaal toolbox [18], [19], for property assessment using either the exhaustive model checker and/or the statistical model checker. An analysed model can then be implemented e.g. into Java.

The paper introduces the proposed methodology and demonstrates its practical application by two CPS models: a train-door controller system and an admission control system (ACS) which orchestrates (through dynamic (re)activation and suspension operations) the electrical appliances in a smart home so as to never exceed a fixed power threshold. The paper then discusses problems existing when transitioning an analysed model into a “faithful” engineered system.

The described work is being continued in the following directions.

- Extending the ACS model so as to cope with (a) the dynamic arrival of power loads, (b) the handling of static/dynamic schemes of priority-based loads, (c) obtaining optimal load scheduling, e.g. by using backtracking, thus minimizing the time for serving all the loads.
- Experimenting with mechanisms for supporting determinism of a Theatre system over a distributed or parallel execution context.
- Applying the methodology to complex industrial-size CPS applications.
- Completing a porting of Theatre in the object-oriented Rust [37] programming language, which promises better execution performance and time-predictability.

References

[1] Lee E A and Seshia A S 2017 *Introduction to embedded systems-A cyber-physical systems approach*, 2^{nd} Edition

[2] Derler P, Lee E A and Sangiovanni-Vincentelli A January 2012 *Modeling Cyber-Physical Systems*, Proc. of the IEEE, **100** (1) 13

[3] Lee E A 2015 *The Past, Present and Future of Cyber-Physical Systems: A Focus on Models.*, Sensors 2015 **15** 4837-4869

[4] Castro R, Marcosig E P and Giribet J I 2020 *Simulation model continuity for efficient development of embedded controllers in cyber-physical systems. In Complexity Challenges in Cyber Physical Systems, Using Modelling and Simulation (M&S) to support Intelligence, Adaptation and Autonomy*, 1st Edition, S. Mittal & A. Tolk (Eds), John Wiley and Sons

[5] Ptolemaeus C (ed.) 2014 *System Design, Modeling, and Simulation using Ptolemy II.*, Ptolemy.org

[6] Lee E A 2006 *The problem with threads*, Comput. **39** 33–42
doi: https://doi.org/10.1109/MC.2006.180

[7] Lohstroh M and Lee E A 2019 *Deterministic actors*, Forum on Specification and Design Languages, Southampton, UK

[8] Jerad C and Lee E A 2018 *Deterministic timing for the Industrial Internet of Things*, IEEE Int. Conf. on Industrial Internet (ICII) 13-22
doi: DOI 10.1109/ICII.2018.00010

[9] Lee E A May 2021 *Determinism*, ACM Transactions on Embedded Computing Systems **20** (5) Article 38 doi: https://doi.org/10.1145/3453652

[10] Lohstroh M, Menard C, Bateni S and Lee E A May 2021 *Toward a Lingua Franca for deterministic concurrent systems*, ACM Transactions on Embedded Computing Systems **20** (4) Article 36 1-27

[11] Lohstroh M, Romeo I I, Goens A, Derler P, Castrillon G, Lee E A and Sangiovanni-Vincentelli A 2019 *Reactors: A deterministic model for composable reactive systems*, Model-Based Design of Cyber Physical Systems (CyPhy’19)

[12] Sirjani M, Lee E A and Khamespanah E 2020 *Verification of cyberphysical systems*, Mathematics **8** (7) 1068

[13] Jafari A, Khamespanah E, Sirjani M, Hermanns H and M. Cimini M 2016 *PTRebeca: modeling and analysis of distributed and asynchronous systems*, Science of Compututer Programming **128** 22–50 doi: https://doi.org/10.1016/j.scico.2016.03.004

[14] Cicirelli F, Nigro L and Sciammarella P F 2020 *Seamless development in Java of distributed real-time systems using actors*, Int. J. Simulation and Process Modelling **15** (1/2) 13-29

[15] Nigro L and Sciammarella P F 2018 *Qualitative and quantitative model checking of distributed probabilistic timed actors*, Simulation Modelling Practice and Theory **87** 343-368 doi: 10.1016/j.simpat.2018.07.011

[16] Nigro L and Sciammarella P F 2018 *Time synchronization in wireless sensor networks: A modelling and analysis experience using Theatre*, The 22nd International Symposium on Distributed Simulation and Real-Time Applications (IEEE/ACM DS-RT 2018), October 15-17, Madrid, Spain

[17] Nigro L 2020 *Parallel Theatre: An actor framework in Java for high performance computing*, Simulation Modelling Practice and Theory doi: doi:10.1016/j.simpat.2020.102189

[18] Behrmann G, David A and Larsen G K 2004, A tutorial on UPPAAL. In Formal Methods for the Design of Real-Time Systems, M. Bernardo and F. Corradini Eds., Lecture Notes in Computer Science **3185** Springer-Verlag 200-236

[19] David A, Larsen G K, Legay A, Mikucionis M and Poulsen B D 2015, Uppaal SMC tutorial. Int. J. Softw. Tools Technol. Transf. **17** (4) 397-415 doi: https://doi.org/10.1007/s10009-014-0361-y

[20] Agha G and Palmskog K 2018 *A survey of statistical model checking*, ACM Transactions on Modelling and Computer Simulation **28** (1) 6:1–6:39 doi: 10.1145/3158668

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

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

[23] Jahandideh I, Ghassemi F and Sirjani M 2021 *An actor-based framework for asynchronous event-based cyber-physical systems*, Software and Systems Modeling. Apr 3 1-25

[24] Cicirelli F and Nigro L *Admission control in home energy management systems using Theatre and hybrid actors*, MDPI Modelling **2** 288–307 doi: https://doi.org/10.3390/modelling2020015

[25] Hewitt C, Bishop P and Steiger R 1973 *A universal modular Actor formalism for artificial intelligence*, In 3rd International Joint Conference on Artificial Intelligence (IJCAI) 235-245

[26] Agha G 1986 *Actors: A model of concurrent computation in distributed systems*, MIT Press, Cambridge, MA, USA

[27] Agha G and Hewitt C 1987 *Actors: A conceptual foundation for concurrent object-oriented programming*, Research directions in object-oriented programming 49-74

[28] Haller P and Odersky M 2007 *Actors that unify threads and events*, In 9th International Conference on Coordination Models and Languages **4467** of Lecture Notes in Computer Science, Springer

[29] Astley M 1998 *The ActorFoundry: A Java-based actor programming environment*, Open Systems Laboratory, University of Illinois at Urbana-Champaign

[30] Charousset D, Hiesgen R and Schmidt C T 2014 *CAF-The C++ actor framework for scalable and resource-efficient applications*, Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control

[31] Hensinger A T 2000 *The theory of hybrid automata*, In Verification of Digital and Hybrid Systems. Springer, Berlin, Heidelberg 265-292

[32] Cicirelli F and Nigro L 14-16 September 2020 *Model checking actor-based cyber-physical systems*, 24th IEEE/ACM Int. Symp. on Distributed Simulation and Real Time Applications (DSRT 2020), Prague

[33] Nigro L July 2020 *Modelling and analysis of cyber-physical systems using deterministic Theatre*, Fourth IEEE World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4 2020); London (UK), IEEE Xplore 27-28

[34] Karmani K R and Agha G 2011, Actors. Springer US, Boston, MA 1–11 doi: https://doi.org/10.1007/978-0-387-09766-4$_1$25

[35] Zhao Y, E.A. Lee E a and Liu J 2007 *A Programming model for time-synchronized distributed real-time systems*, Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 259-268

[36] Corbett C J et al 2012 *Spanner: Google's Globally-Distributed Database*, OSDI

[37] The Rust programming language, on-line (accessed on July 2021) *https://www.rust-lang.org/*

Libero Nigro is a full professor of Computer Engineering in the Engineering Department of Informatics, Modelling, Electronics and Systems Science (DIMES) of University of Calabria, 87036 Rende (CS) Italy. He currently teaches Object Oriented Programming and Systems Programming (covering modelling, simulation, real-time and multi-agent systems) courses. He heads the Software Engineering Laboratory at DIMES whose main goal is formal modelling by Petri nets, DEVS, actors, statecharts, timed automata etc., and tool development for analysis, e.g. by distributed/parallel simulation, or by exhaustive model checking, and concrete implementation of complex timed systems. Libero was the tutor of several PhD students at DIMES. He is currently an editor of Simulation Modelling Practice and Theory (SIMPAT) and of Int. J. of Simulation and Process Modelling (IJSPM). In addition, from several years, he is serving in the program committee of well-known international conferences and symposia on modelling, simulation and real time applications, and as a referee of journals including Science of Computer Programming, Software and Systems Modeling, J. of Systems and Software, SIMPAT, Simulation Trans. of SCS, J. of Cellular Automata, Discrete Event Dynamic Systems etc..

Franco Cicirelli is a researcher at ICAR-CNR (Italy) since December 2015. He earned a Ph.D. in System Engineering and Computer Science at the University of Calabria (Italy). His research work mainly focuses on Software Engineering tools and methodologies for the modeling, analysis and implementation of complex time-dependent systems. Research topics are agent-based systems, distributed simulation, parallel and distributed systems, real-time systems, workflow management systems, Internet of Things and cyber–physical systems.