Universitá della Calabria

DIPARTIMENTO DI INGEGNERIA INFORMATICA, MODELLISTICA, ELETTRONICA E SISTEMISTICA

Dottorato di Ricerca in
Information and Communication Technologies
XXXI ciclo

Modelling Analysis and Implementation of Distributed Probabilistic Timed Actors using Theatre

Candidato:
Paolo Francesco Sciammarella

Relatore:
Prof. Libero Nigro

Correlatore:
Prof. Domenico Grimaldi

Anno Accademico 2018-2019

"That Every long lost dream led me to where you are
Others who broke my heart they were like northern stars
Pointing me on my way into your loving arms
This much I know is true
That God blessed the broken road
That led me straight to you"

Melodie Crittenden - Broken Road

Contents

Preface 7

I Formal Modelling and Verification **9**

1 Concepts of Model Verification **10**
1.1 Model Checking 11
1.2 Timed Automata Theory 13
1.2.1 Introduction 13
1.2.2 Formal syntax 14
1.2.3 Labelled Transition System semantics 14
1.2.4 Bisimulation 16
1.2.5 Symbolic semantics and Verification 17
1.3 Some Available Model Checking Environments 21
1.3.1 Java PathFinder 21
1.3.2 SPIN and PROMELA 22

2 The Uppaal Symbolic Model Checker **24**
2.1 Modelling language 25
2.1.1 Normal, urgent and committed locations 25
2.1.2 Guarded commands 26
2.1.3 Progress conditions 27
2.1.4 Global time and clocks 27
2.2 Query language 28
2.3 Advanced features 29

3 Probabilistic and Statistical Model Checking **32**
3.1 Probabilistic Models 33
3.1.1 Discrete-Time Markov Chains (DTMC) 33
3.1.2 Markov Decision Processes (MDPs) 35
3.1.3 Probabilistic Automata (PAs) 36
3.1.4 Continuous-Time Markov Chains (CTMCs) 36
3.1.5 Probabilistic Timed Automata (PTAs) 37
3.2 The Prism probabilistic model checker 38
3.3 Statistical Model Checking 39
3.3.1 SMC Operation 39

3.3.2 Monte Carlo simulations . . . 40
3.3.3 Hypothesis testing . . . 41
3.4 Plasma Lab statistical model checker . . . 46
3.5 VeStA and PVeStA . . . 47

4 The Uppaal Statistical Model Checker 48
4.1 Network of Stochastic Timed Automata . . . 48
4.2 Query language . . . 49
4.2.1 Bound and number of runs . . . 50
4.2.2 Simulation . . . 50
4.2.3 Statistical algorithms . . . 50
4.2.4 Probability Estimation . . . 51
4.2.5 Hypothesis testing . . . 51
4.2.6 Probability comparison . . . 51
4.2.7 Value bound determination . . . 51
4.2.8 Support for WMITL . . . 52
4.2.9 Additional queries . . . 52
4.3 SMC options . . . 53
4.4 Dynamic template processes . . . 53
4.5 Custom probability distribution functions . . . 54
4.6 Non-deterministic vs. stochastic interpretation . . . 55

II Distributed Probabilistic Timed Actors 56

5 Actor-based Development of Distributed Probabilistic Timed Systems 57
5.1 Untimed actors . . . 57
5.2 Timed actors . . . 59
5.3 Actor Extensions for Real Time Modelling and Analysis . . . 61
5.4 The Theatre infrastructure . . . 65
5.4.1 Basic Concepts . . . 65
5.4.2 Programming in-the-small concepts . . . 66
5.4.3 Programming in-the-large concepts . . . 67
5.4.4 Simulation applications . . . 68
5.4.5 Development methodology and model-continuity . . . 68
5.4.6 Implementation status . . . 69
5.4.7 Contributions of this dissertation . . . 69

III Theatre in Action 71

6 Model Continuity in Cyber-Physical Systems 72
6.1 Introduction . . . 72
6.2 Related Work . . . 74
6.3 From Modelling to Implementation of a CPS . . . 77
6.3.1 The proposed methodology . . . 77
6.3.2 Control machines and time management . . . 80
6.3.3 Actions and processing units . . . 81
6.3.4 envGateway and environment control . . . 81

6.3.5 Specializing the *envGateway* to work with Arduino 82
6.4 A case study using power management 83
6.4.1 Modelling the system 85
6.4.2 Data configuration 85
6.4.3 Analysis phase 86
6.4.4 Preliminary execution 87
6.4.5 Prototype implementation and real execution 89

7 Qualitative and Quantitative Model Checking of Distributed Probabilistic Timed Actors 93
7.1 Introduction 93
7.2 THEATRE concepts 96
7.2.1 Architectural view 96
7.2.2 Abstract modelling language 97
7.2.3 A modelling example 98
7.3 An operational semantics of THEATRE 102
7.3.1 Transition rules $\xrightarrow{i}$ and $\xrightarrow{d}$ 103
7.4 A reduction of THEATRE onto UPPAAL 107
7.4.1 Scenario parameters 107
7.4.2 Entity naming 108
7.4.3 Message and delay pools 109
7.4.4 Asynchronous message passing and delay setting 109
7.4.5 Message delivery and arguments 110
7.4.6 The Message automaton 110
7.4.7 Delay automaton 111
7.4.8 An actor automaton 112
7.4.9 Preservation of THEATRE semantics 113
7.4.10 Translated UPPAAL model of the toxic gas sensing system 114
7.5 Analysis of a THEATRE model reduced into UPPAAL 116
7.5.1 Qualitative non-deterministic model checking 116
7.5.2 Quantitative statistical model checking 118
7.5.3 Partitioning 124

8 Seamless Development in Java of Distributed Real-Time Systems using Actors 125
8.1 Introduction 125
8.2 An overview of Theatre 125
8.2.1 Basic Java Framework 126
8.2.2 Development Phases And Control Machines 127
8.3 A modelling example 129
8.4 Analysis of the toxic gas system 135
8.4.1 Qualitative Experiments 135
8.4.2 First Scenario: 2 Sensors and Scientist Deadline set to 10 136
8.4.3 Second Scenario: 2 Sensors and Scientist Deadline set to 13 137
8.4.4 Third Scenario: 3 Sensors vs. 1 Sensor, 95% Working . . 137
8.4.5 Fourth Scenario: Scientist Die Probability vs. Number of Sensors 138
8.5 Preliminary Execution of the TGS Model 138
8.6 Real time execution 141
8.7 Theatre Implementation Status 142

8.7.1 Configuration and Bootstrap of a Distributed System . . 143
8.7.2 Actor Migration 145

9 Case Studies **146**
9.1 Asynchronous Leader Election 146
9.1.1 Experimental results 148
9.2 A Time Synchronization Algorithm 148
9.2.1 Modelling the Time-Synch algorithm using Theatre . . 151
9.2.2 Experimental results 153
9.3 Actor-based NetBill Protocol 159
9.3.1 Modelling the NetBill protocol into Uppaal SMC 161
9.3.2 Experimental results 162

10 Modelling and Analysis of Multi-Agent Systems Using Uppaal 168
10.1 Introduction 168
10.2 Modelling the Iterated Prisoner's Dilemma 169
10.2.1 Case study description 170
10.2.2 An IPD actor-based model 171
10.3 A structural translation from actors to Uppaal SMC 172
10.3.1 Actor and message naming 173
10.3.2 Dynamic messages 173
10.3.3 Actor automata 175
10.3.4 Other global declaration 176
10.4 Experimental work 178
10.4.1 Debugging queries 178
10.4.2 Transient behavior 179
10.4.3 Emergence of cooperation in the scenario-1 180
10.4.4 Emergence of cooperation in the scenario-2 181
10.4.5 Model validation 184

11 Conclusions **186**

List of publications and bibliography **188**

IV Appendixes 212

A Formal Modelling and Analysis of Probabilistic Real-Time Systems **213**
A.1 Introduction 213
A.2 The formalism of Stochastic Time Petri Nets 214
A.2.1 Basic Concepts 214
A.2.2 Syntax 214
A.2.3 Semantics 215
A.3 Mapping sTPN onto Uppaal 216
A.3.1 Timed Automata For Transitions 217
A.4 First Example 219
A.5 Experimental analysis 221
A.5.1 Non-deterministic analysis 221
A.5.2 Quantitative analysis 223

A.6 Second Example . . . 226
A.7 Analysis of the Fisher's protocol . . . 227
A.7.1 Non-deterministic analysis . . . 229
A.7.2 Stochastic Analysis . . . 230
A.8 Conclusions . . . 232

B A layered IoT-based architecture for distributed SHM 234
B.1 Introduction . . . 234
B.2 SHM Based On Multi-Agent IoT . . . 235
B.2.1 Sensing layer . . . 237
B.2.2 Signal Processing Layer . . . 239
B.2.3 Event Detection Layer . . . 239
B.2.4 Application Layer . . . 240
B.2.5 Remote Transmission Protocol . . . 241
B.2.6 Experimentation validation . . . 242

Acknowledgements 246

Preface

Nowadays, the technological advances introduced e.g. by the Internet-of-Things, motivate the development of complex software systems, which can exploit the growing connection flexibility among different computational units and devices. Smart objects, embedded systems and cyber-physical systems are just a few examples of design techniques and applications fueled by this ongoing technological revolution.

From a software engineering point of view, systems that follow these design methods and operation logic, are best known as *reactive systems*. Reactive systems are computer systems that continuously react to the external stimuli generated by the user or by a controlled environment, producing an answer as soon as the request arrives. Such systems cover different safety and business critical areas such as e-commerce, e-banking, air traffic control, mechanical devices control for train operation or nuclear reactors and many others.

Correctness and reliability are project requirements of paramount importance in the design of many of the above mentioned systems. In fact, an application failure or malfunction can have severe consequences from the practical point of view.

Traditional software validation techniques are *simulation* and *testing*. Unfortunately, they are insufficient because not exhaustive. They are often manually constructed and typically can explore only a small fraction of all the possible computational paths. Therefore, they cannot guarantee the absence of bugs in the left unexplored state trajectories, particularly in the presence of complex concurrent interleavings.

In the last few decades, the alternative approach of *formal verification* emerged, which is based on the exhaustive exploration of all the possible system behaviors. Using e.g. the *model checking* technique, it is possible to determine if a (not large) system satisfies a specific behavioral property, by examining all its reachable states. However, with not trivial systems, model checking can incur into the *state explosion* problem which limits its practical applicability. A more approximate solution to infer properties and ensure system correctness, is based on the concept of *statistical model checking*. It uses a finite number of system executions and hypothesis testing, to judge if collected samples provide a statistical evidence for the specification satisfaction or violation.

This thesis is devoted to developing methods and tools of software engineering, supporting modelling, analysis and implementation of reactive systems.

What makes the development of such systems very challenging is the necessity of handling in combination such issues as distribution, non-determinism, probabilistic/stochastic behavior, concurrency, timing constraints, heterogeneity, synchronicity and asynchronicity, which can render systems undecidable.
The Actors computational model is widely recognized as a reference point for building reactive systems. It relies on highly modular encapsulated entities which interact to one another by asynchronous message passing. Extensions have been proposed to actors so as for them to be more adapt for distributed probabilistic real-time systems. A notable project on timed and probabilistic actors is represented by the family of Rebeca modelling languages and related analysis tools based on McErlang, PRISM and recently IMCA.
Starting from some previous research carried out in the Software Engineering Laboratory (www.lis.dimes.unical.it), directed by Prof. Libero Nigro, the proposed work consists in enhancing a lightweight actor system called THEATRE, so as to provide a formal modelling language in the light of the inspiring Rebeca work, together with some supporting tools for property assessment which in a case are based on the Uppaal model checkers (qualitative analysis - detecting that an event *can* occur - based on the non-deterministic exhaustive model checker, and quantitative analysis - finding a quantitative measure of the probability for an event to actually occur- based on the statistical model checker). An important part of the work consists in experimenting the use of Theatre according to the concept of *model continuity*, namely transitioning, without distortions, a same model from early analysis down to design, prototyping and final implementation in Java. The goal is trying to ensure *faithfulness* of a synthesized system to the analyzed model.
The accomplished work on Theatre is accompanied by several concrete applications and case studies aimed at demonstrating the flexibility and potential of Theatre features.
The thesis is structured in three parts. In the first part (chapters from 1 to 4), the concepts of formal modelling and verification are summarized. In particular, the automated techniques of exhaustive symbolic model checking, based on timed automata, and of probabilistic and statistical model checking are discussed. As a notable example, the UPPAAL model checkers, used in this thesis, are outlined. In the second part (chapter 5), the development issues of distributed probabilistic and timed actor systems are reviewed, together with some relevant related works. All of this provides context for the THEATRE actor system which is the goal of this thesis. Moreover, the contributions of this work are highlighted. In the third part (chapters from 6 to 10), the main developments and achievements concerning the extended version of THEATRE, its implementation in Java and its application to modelling and verification of selected case studies are described. The thesis terminates by presenting conclusions and by furnishing directions of future works. Two appendixes complete the dissertation which refer to some additional accomplished work: that of experimenting with modelling and analysis of Stochastic Time Petri Nets and showing a further application of our actors to a prototypical structural health monitoring and control system.

Part I

Formal Modelling and Verification

Chapter 1

Concepts of Model Verification

The development of reliable hardware and software systems is a well-known critical and crucial issue [83]. In the last decades, several tools have been developed to help to identify defects and implementation problems. Although very often used, *simulation* and *testing* techniques [28, 133, 169] prove inadequate for correctness assessment [94]. As shown in Fig. 1.1, testing normally works by comparing results obtained at run-time, with the expected ones computed on a finite set of test-cases [176]. Only a small fraction of all the computational paths is investigated. Moreover, by using a few runs, problems such as deadlocks or concurrency errors due to complex thread interleavings, are not easily reproducible or detectable [261].

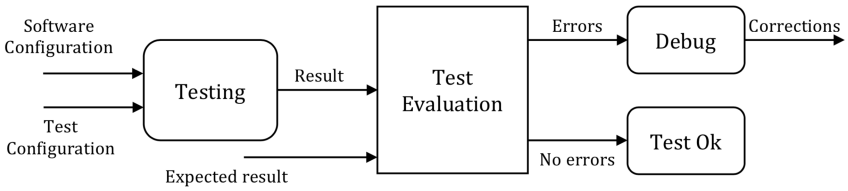


Figure 1.1: Schematic view of the testing workflow

Formal verifications methods [125], based on mathematical techniques, aim to prove the correctness of a system by reasoning on an abstract model and carrying an *exhaustive exploration* of all the possible reachable configurations [166](see Fig. 1.2. Such methods combine three main components [171], that are:

- an *abstract model*, that is in charge of emulating the system behaviour;
- a *specification language*, used to formally express system requirements and properties of interest;
- an *analysis method*, that is in charge to verify the fulfilment of the requirements.

Modelling is a creative yet difficult activity [162] which involves experience and abstraction, that is introducing only relevant details while omitting non-essential

others. This work claims, though, that a model should facilitate subsequent transitioning of a model toward the design and final implementation phases. In other words, modelling should be not so much abstract but it should admit such issues as distribution/concurrency, probabilistic and timing aspects. By reducing the *semantic gap* between modelling and synthesis, the goal is to ensure to the large extent *"faithfulness"* of an implementation to its analyzed model [235].
Several approaches to formal verification are nowadays available such as *model checking* [25], that is a widespread and mostly studied technique to automate properties verification. It differs, e.g., from *theorem provers* [40], which require first a (possibly complex) mathematical formalization of the problem but not always offer the possibility of checking timing properties.

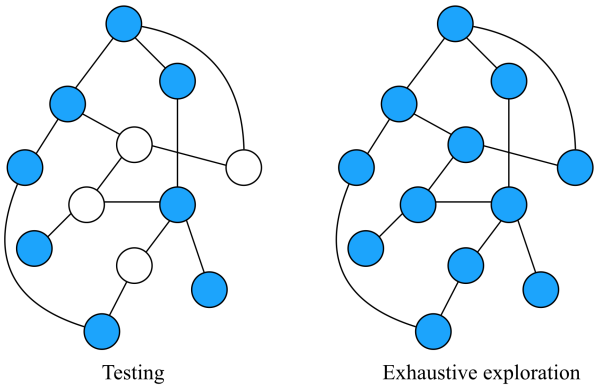


Figure 1.2: Correctness verification by using testing or exhaustive exploration, taken from [172]

1.1 Model Checking

Model checking was independently developed by Clarke and Emerson [81] and Queille and Sifakis [214]. As shown in Fig 1.3 the starting points to perform this procedure relies on two components:

- a *formal description* $\mathcal{M}$ of the model under analysis;
- a *logical property* φ associated with a requirement that has to be evaluated, written in a *specification language*, like classical computational tree logic (CTL) [81], linear temporal logic (LTL) [211] and their extensions.

If the given model satisfies the logical property, we write that $\mathcal{M} \models \varphi$, on the contrary we write $\mathcal{M} \not\models \varphi$, and the model checker suggests a diagnostic trace (counterexample), practically a path of event occurrences, where the violation takes place.

During verification the model checker compiles the model under investigation into a directed state-transition graph, called *Kripke structure* [139], consisting of:

- nodes, each of which represents a system execution state;

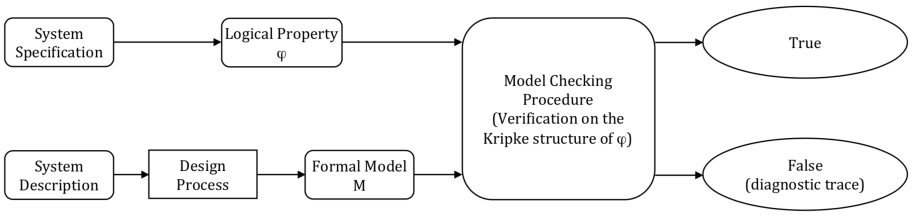


Figure 1.3: Schematic view of the model checking workflow

- edges, which represent transitions between states, i.e., events that are in charge of changing the system status.

More formally, a Kripke structure over a set A of atomic propositions (i.e. boolean expressions), is a triple $\mathcal{K} = \langle S, R, L \rangle$ where:

- S is the state-space that includes all the possible states as a set of *true* atomic propositions, where s_0 is the initial state;
- $R \subseteq S \times S$ is the transitions set;
- $L : S \to 2^A$ is the labelling function that associates a set of valid atomic propositions to each state. In particular, the notation $L(s)$ groups together the true propositions for the system, when it is in the state s [84].

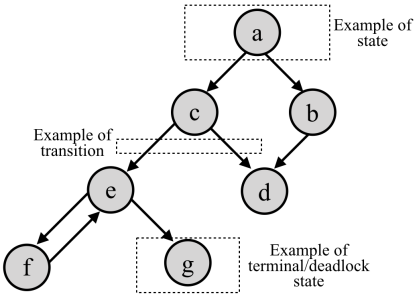


Figure 1.4: Example of a state graph

A model checker can explore the entire state space of a system, evaluating all the possible computational paths in order to verify if a property is satisfied. The model checking theory, though, has a computational limit due to resources needed in terms of time and memory for verification. When analyzing non-trivial medium-large sized systems, in fact, the number of computational paths tends to increase exponentially, even after adding a single variable. This leads to the construction of a huge state graph, which cannot possibly be completed. The problem, known as the *state-space explosion*, limits the use of model checkers and has committed researchers to an attempt to identify alternatives or optimizations, such as symmetry reduction [99] or partial order reduction [83].

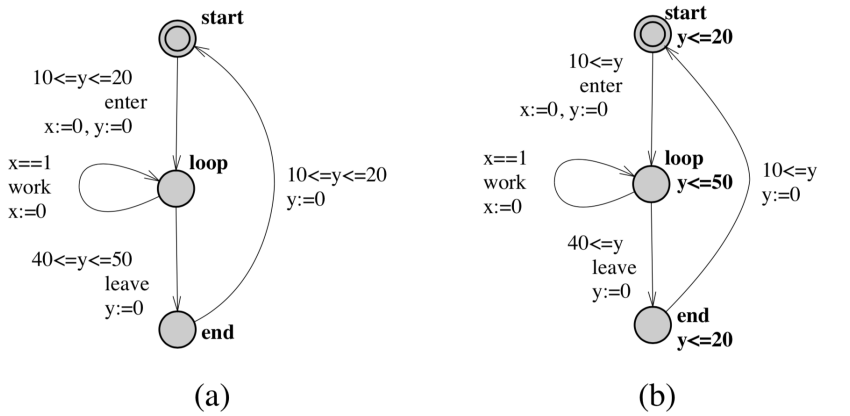


Figure 1.5: Timed Automata and Location Invariants, taken from [31]

1.2 Timed Automata Theory

Among the different modelling languages available to design systems, such as Time Petri Nets [35], timed process algebras [219] [182], and real-time logics [262], in this work the *timed automata* formalism was chosen, because it constitutes the core of the input language used by UPPAAL, that is the symbolic model checker exploited for the verification.

1.2.1 Introduction

Timed automata (TA) [16] are an extension of the finite-state Büchi automata, enriched with a set C of real number variables called *clocks* [27]. Intuitively, TA are finite-state machines with clocks used to model the time elapse, according to a dense-time notion. All the clocks values are initialized to zero, and increased synchronously with the same rate. They and may be reset during an execution path. A transition edge may be labelled with a *guard*, consisting of a time-constraint belonging to the set $B(C)$ of all the possible constraints built over the clocks C, and a set of *actions* ranging over a finite alphabet Σ. A guard has the form $x \sim k$ or $x - y \sim k$, where $x, y \in C$ are clock variables, k is a non-negative integer and the operator $\sim \in \{\leq, <, ==, >, \geq\}$, and its fulfilment enables the edge and, if the transition is taken, the corresponding action is carried out.
Figure 1.5 a) depicts an example of timed automaton, taken from [31]. Its timed behavior depends on the value of two clocks: x, that is used to control the self-loop in the location *loop*, and y, that is used to control the time elapse of the overall model. The dynamic progress is as follow. At the beginning, the automaton may leave the location *start* at any time when the y value is in the interval $[10, 20]$. Then it may evolve towards the *loop* place, along with the self-loop if *x==1*, and finally it can move to *end* when y is in $[40, 50]$. After that, since the y value is reset, the automaton may return to start when the y is in $[10, 20]$. In the TA, satisfaction of a guard on an edge is a *necessary* but *not sufficient* condition for moving from one location to another. In the example in figure 1.5 a), the automaton may stay forever in any location.

To ensure progress, in [16] the *Büchi acceptance condition* was introduced, that is marking a set of locations as *accepting*, and having that the only admissible automaton executions are those that include an infinite transit from them. Marking in figure 1.5 a) the location *end* as accepting, it would imply that all the automaton executions *must* visit that location infinitely times, thus ensuring progress. As a consequence, the location start will be left at most when $y == 20$; would this not happen, the acceptance condition would be violated, because after 20 the guard on the edge becomes invalid and the automaton will be blocked in that location. A similar argument could be applied to the loop location.
A simpler and more intuitive notion of progress was introduced in the *Timed Safety Automata* [31]. Clock constraints in the form $x \sim k$, called *invariants*, are used to label locations and give them a local view of time behaviour. An automaton can remain in a location as long as its invariant is satisfied, and forced to exit when the invariant is up to be violated. Would no output transition be enabled, a deadlock will occur. In figure 1.5 b) an example is shown.
In literature, Timed Automata and Timed Safety Automata most often denote the same concept.

1.2.2 Formal syntax

Formally, a timed automaton is a tuple (L, l_0, C, A, E, I), where:

- L is a finite set of locations;
- $l_0 \in L$ is the initial location;
- C is the set of clocks;
- A is a set of actions (i.e. the alphabet Σ);
- $E \subseteq L \times A \times B(C) \times 2^c \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset;
- $I : L \longrightarrow B(C)$ is a function that assigns invariants to locations.

An edge is also indicated as $l \xrightarrow{g,a,r} l' \in E$ where g is a guard, a is an action, r is a set of clocks to reset.
To analyze concurrentcy systems modelled as a *network of interacting TA*, TA theory rests on the *product automaton* built using the CCS parallel composition operator [177]. Although this operation is entirely syntactical and allows interleaving of actions as well as hand-shake synchronizations, it has a computationally expensive cost. For this reason tools such as Uppaal, generate on-the-fly the product automaton during verification. Figure 1.6 depicts some examples of timed automata composition. The notation associated to the case c), i.e., the symbols ! and ? modelling hand-shake synchronization, will be clarified below.

1.2.3 Labelled Transition System semantics

For verification of a single timed automaton, the model checker works on a state-graph which is a *labelled transition system* (LTS), directly corresponding

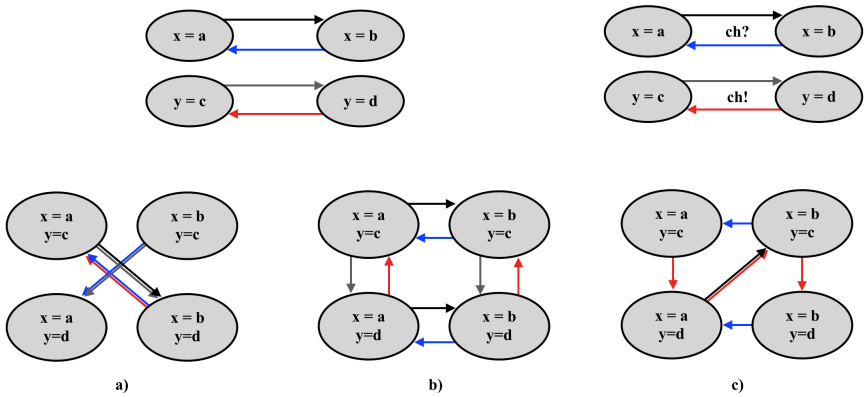


Figure 1.6: Examples of product automaton: a) sychronous composition, b) asynchronous composition, c) explicit synchronization composition

to the operational semantics of the automaton. A LTS consists of a collection of *states* and *transitions*. A state is a pair $(l, u) \in L \times \mathbb{R}^C$, where l is a location id and u is a clock valuation ($u \in \mathbb{R}_+$). A state represents a specific configuration of the model in which a set of predicates $p_i \in P$ hold. The initial state s_0 of a LTS is $(l_0, 0_{|C|})$. Transitions denote all the possible ways an LTS state has to evolve. They are labelled with actions $a_i \in A$, executed when the transition is taken [247].
Formally, let A and P be sets of actions and predicates respectively. A LTS over A and P is a triple $(S, \rightarrow, \models)$, where:

- S is a set of states;
- $\rightarrow$ is a set of transitions $\xrightarrow{a} \subseteq S \times \Sigma \times S$, where $\Sigma = \mathbb{R}^+ \cup \{d\}$ is the set of moves [213];
- $\models \subseteq S \times P$, indicates that a predicate $p \in P$ holds in state $s \in S$.

To infer properties, all the possible LTS reachable paths need to be explored and evaluated.
The transitions of a LTS can be of two types:

- a *delay transition*, labelled with a real number, which expresses the possibility for an automaton to stay for a certain time d in a state, as long as d does not violate the invariant:
  $(l, u) \xrightarrow{d} (l, u + d)$ if $(u + d) \in I(l)$ for any $d \in \mathbb{R}_+$
- an *action transition*, labelled with elements of Σ, which expresses the possibility for an automaton to evolve towards another state, following an enabled edge and possibly resetting a subset of clocks:
  $(l, u) \xrightarrow{a} (l', u')$ if $l \xrightarrow{g,a,r} l'$, $u \in g$, $u' = [r \mapsto 0]u$ and $u' \in I(l')$

A *timed action* is a pair (t, a), representing an action $a \in \Sigma$ executed by an automaton after $t \in \mathbb{R}_+$ time units since the beginning of the run, where t is the *absolute time*, or *time-stamp* of the action a. A sequence (possibly infinite) of timed actions of an automaton $\xi = (t_1, a_1)(t_2, a_2) \ldots (t_i, a_i) \ldots$, where $t_i \leq t_{i+1}$

for all $i \geq 1$, is a *timed trace*. A *run* of an automaton in the corresponding LTS, with initial state (l_0, u_0) and over a timed trace $\xi = (t_1, a_1)(t_2, a_2)(t_3, a_3)\ldots$, is defined as a sequence of transitions:

$$(l_0, u_0) \xrightarrow{d_1} \xrightarrow{a_1} (l_1, u_1) \xrightarrow{d_2} \xrightarrow{a_2} (l_2, u_2) \xrightarrow{d_3} \xrightarrow{a_3} \ldots$$

where $t_i = t_{i-1} + d_i$ for all $i \geq 1$.
The semantics of a network of TA is the associated LTS generated from the product automaton, and it is similar to that a single automaton. To model hand-shake synchronizations (see Figure 1.6), the action alphabet Σ is assumed to consist of symbols for input actions denoted $a?$, output actions denoted $a!$, and internal actions represented by the distinct symbol a.
The state of a LTS associated to a network of TA is a pair (l, u), where $l = l_1, l_2, \ldots, n$ is a vector constituted by the current locations of each automaton of the network and u is a clock array that summarizes the current values of clocks of the system. The rule for delay transitions is similar to the case of a single automaton, where the invariant of a location vector is the conjunction of the location invariants of the processes.
Let l_i stand for the $i-$th element of a location vector l and $l[l'_i/l_i]$ for the vector l with l_i being substituted with l'_i. The transition rules are defined as follows:

- $(l, u) \xrightarrow{d} (l, u + d)$ if $u \in I(l)$ and $(u + d) \in I(l)$, where $I(l) = \wedge I(l_i)$
- $(l, u) \xrightarrow{a} (l[l'_i/l_i], u')$ if $l_i \xrightarrow{g,a,r} l'_i$,$u \in g$, $u' = [r \mapsto 0]u$, $u' \in I(l[l'_i/l_i])$
- $(l, u) \xrightarrow{a} (l[l'_i/l_i][l'_j/l_j], u')$ if there exist $i \neq j$ such that:
  - $l_i \xrightarrow{g_i,a?,r_i} l'_i$, $l_j \xrightarrow{g_j,a!,r_j} l'_j$ and $u \in g_i \wedge g_j$, and
  - $u' = [r_i \cup r_j \mapsto 0]u$ and $u' \in I(l[l'_i/l_i][l'_j/l_j])$

where u' is obtained by resetting a subset of clocks.
It is important to note that a network is a closed system, which may not perform any external action [31].

1.2.4 Bisimulation

The set of the all finite timed traces of a LTS $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$ and is called the *timed language of* $\mathcal{A}$. Two different LTSs $\mathcal{A}_1$ and $\mathcal{A}_2$ are timed-language equivalent if and only if $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$. In order to check behavioural equality among processes, language equivalence is not the most suitable notion to consider. *Bisimulation* equality, called *bisimilarity*, among different states within a LTS or different LTSs, is the most used technique to determine and check behavioural similarities [226].
Let $(S, \rightarrow, \models)$ be an LTS over A and P. A bisimulation is a binary relation $\mathcal{R} \subseteq S \times S$ over the set of states, satisfying:

- if $s_1 \mathcal{R} s_2$ then $s_1 \models p \Leftrightarrow s_2 \models p$ for all $p \in P$,
- if $s_1 \mathcal{R} s_2$ and $s_1 \xrightarrow{a} s'_1$ with $a \in A$, then there exists a transition $s_2 \xrightarrow{a} s'_2$ such that $s'_1 \mathcal{R} s'_2$,

- if $s_1 \mathcal{R} s_2$ and $s_2 \xrightarrow{a} s_2'$ with $a \in A$, then there exists a transition $s_1 \xrightarrow{a} s_1'$ such that $s_1' \mathcal{R} s_2'$.

Two states s and s' are timed bisimilar, written as $s \sim s'$, if and only if there is a timed bisimulation that relates them.
Let $g = (S, I, \rightarrow, \models)$ and $h = (S', I', \rightarrow', \models')$ be process graphs over A and P. A *bisimulation* between g and h is a binary relation $R \subseteq S \times S'$, satisfying $I \mathcal{R} I'$ and the same three clauses above. g and h are bisimilar, written as $g \sim h$ if there exists a bisimulation between them [247].

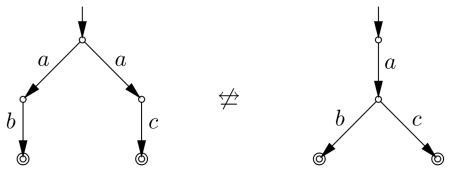


Figure 1.7: Example of two processes not bisimulation equivalent, taken from [247]

Figure 1.7 shows an example of two not bisimulation equivalent processes, taken from [247]. They accept the same language, but the choice between b and c is made at different moment, because the two systems have different branching structure.
Intuitively two systems are considered bisimilar if they match each other's moves, becoming indistinguishable from an observer. More details can be found in [208] and [167].

1.2.5 Symbolic semantics and Verification

The reachability analysis, performed on the LTS generated from a network of TA, is a key problem in verification, because the correctness of a model can be verified starting from a question: a given desirable or undesirable state of the model is reachable from its initial state [181]? Two class of algorithms can be applied to answer this question, performing:

- the *forward analysis*, that consists of computing iteratively the successors of the initial states and checking if the state we want to reach is eventually computed or not;
- the *backward analysis*, that consists of computing iteratively the predecessors of the states we want to reach and of checking that the initial state is eventually computed or not.

Since to medium-sized network of TA, can be associated LTSs containing an uncountable set of reachable states, such analysis can be difficult or not decidable. This is due to the fact that in a state (l, u), u represents an instantaneous evaluation clocks. Such time granularity originates a potentially infinite state-space, because locations with a same data part, will be included in several states which are distinguished each other only by the potentially different infinite instantaneous clock values. This limit can be overcome by partitioning the clocks evaluations by a finitely equivalence relation. Exploiting the concept of *time abstract bisimulation*, by which two configurations s and s' are *equivalent* when

any action transition a (or any delay transition d) enabled by s, can be simulated from s' by an action transition a (or a delay transition d'), it is possible to compact the representation [181]. In a LTS two configuration states (l, u) and (l', u'), can be considered equivalent if $l = l'$ and if $u \equiv_M u'$, i.e., if:

- they exactly satisfy the same clock constraints of the TA;
- in both automata the time elapse, will lead to same integer values for the clocks, in the same order [181].

where M is the maximal constant value appearing in the guards of the automaton associated to the specific clock. Using the notation $u \equiv_M u'$ holds whenever for each clock $x \in X$,

- $u(x) > M \Leftrightarrow u'(x) > M$,
- if $u(x) \leq M$, then $\lfloor u(x) \rfloor = \lfloor u'(x) \rfloor$, and $(\{u(x)\} = 0 \Leftrightarrow \{u'(x)\} = 0)$, where the operator $\lfloor \alpha \rfloor$ indicates the integer part of α, while $\{\alpha\}$ the fractional part,

and for each pair of clocks (x, y),

- if $u(x) \leq M$ and $u(y) \leq M$, then $\{u(x)\} \leq \{u(y)\} \Leftrightarrow \{u'(x)\} \leq \{u'(y)\}$.

The relation $\equiv_M$ is called *region equivalence* and the associated equivalence class is called a *region* [15]. States placed within the same region will always evolve into states belonging to the same region and this feature enables to characterize (and reduce) the state-space of a LTS in terms of a finite *region automaton* or *region graph* $\mathcal{R}_\mathcal{A}$. The $\mathcal{R}_\mathcal{A}$ can be built starting from the initial LTS $\mathcal{A}$ and the equivalence relation, following this formal semantics:

- states of the graph are pairs (l, R) where l is a location of LTS and R is a region;
- the transitions are expressed by $(l, R) \xrightarrow{a} (l', R')$ if there exists a transition $l \xrightarrow{g,a,r} l'$ in $\mathcal{A}$, a valuation $v \in R$ and $d \geq 0$ such that $u + d \models I(l)$, $u + d \models g$, $[r \leftarrow 0](u + d) \models I(l')$ and $[r \leftarrow 0](u + d) \in R'$.

The region graph size depends exponentially from the number of the involved clocks and the maximal constant parameter value appearing in the guards. This means that these parameters may lead to *state explosions*, because a large number of configurations (arising from all the possible regions), needs to be checked. As reported in [31], figure 1.8 depicts an example of all the admissible regions for an automaton with two clocks x and y, where the x maximum value is 3, while the y maximum value is 2. All corner points (intersections), line segments, and open areas are regions. Doing maths, the number of possible regions in each location is 60. The fundamental property of $\mathcal{R}_\mathcal{A}$ is that it recognizes exactly the set of words $a_1 a_2 \ldots$ if exists a corresponding timed word $(a_1, t_1)(a_2, t_2) \ldots$ recognized by $\mathcal{A}$. This allows to check a reachability property of a location in $\mathcal{A}$, by resolving a reachability problem in its $\mathcal{R}_\mathcal{A}$.
In figure 1.9 it is showed an example of a LTS $\mathcal{A}$ and of its own region automaton $\mathcal{R}_\mathcal{A}$ taken from [17]. As one can see, the location l_3 of $\mathcal{A}$ is reachable if and only if one of the states (l_3, R) with a region R is also reachable in $\mathcal{R}_\mathcal{A}$. By analyzing $\mathcal{R}_\mathcal{A}$, we note that the path $(l_0, x = y = 0) \xrightarrow{a} (l_1, 0 = y < x < 1) \xrightarrow{c}$

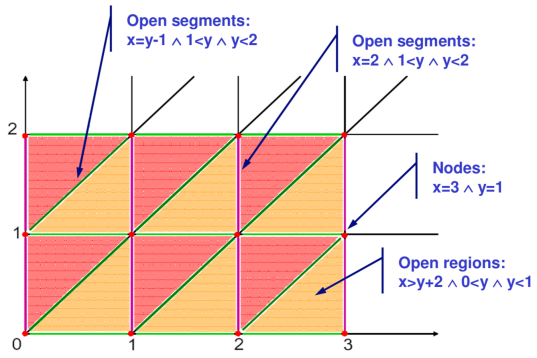


Figure 1.8: Example of regions

$(l_3, 0 < y < x < 1)$ leads to l_3. Consequently, this implies that in $\mathcal{A}$ exists an execution, given by $(l_0, u_0) \xrightarrow{t_1} (l_0, u_0 + t_1) \xrightarrow{a} (l_1, u_1) \xrightarrow{t_2} (l_1, u_1 + t_2) \xrightarrow{c} (l_3, u_2)$, that leads to l_3, identifiable for some value of t_1 and t_2.

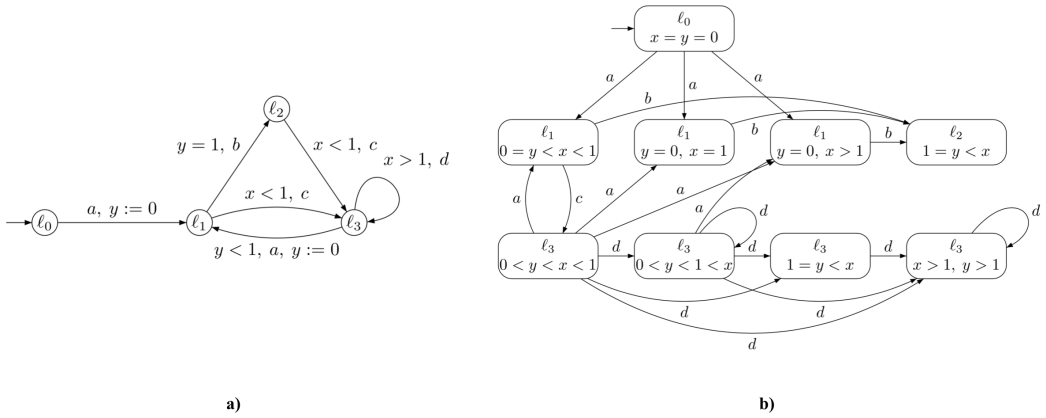
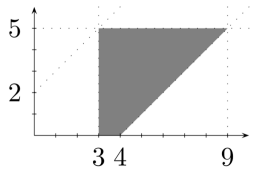


Figure 1.9: a) A labelled transition system $\mathcal{A}$ b) The corresponding region graph $\mathcal{R}_{\mathcal{A}}$. Example taken from [17]

In practice, model checker tools avoid the construction of the region automaton, because the region partition is too refined and not efficient to be manipulated. To provide a more efficient representation of the symbolic state-space, in [41] the concept of *zone* and *zone-graph* was introduced, that rely on the concept of on-the-fly algorithms. A *zone* is defined as the solution set of a conjunction of atomic clock constraints, among inequalities of the form:

$$x_i - x_j \leq u_{ij}, \qquad l_i \leq x_i \leq u_i$$

where $i, j \in \{1, \ldots, n\}$ and $u_{ij}, l_i, u_i \in \mathbb{R}$. Compared with regions, zones offer a more coarse and reduced representation of the state-space [96]. In fact, the symbolic states of a *zone graph* that would be manipulated by the forward and backward analysis algorithms, are defined by (l, Z), where $Z \in B(C)$ is a zone defined through the union of clock constraints, that incorporates several



$$M = \begin{array}{c} \\ x_0 \\ x_1 \\ x_2 \end{array} \begin{array}{c} \begin{array}{ccc} x_0 & x_1 & x_2 \end{array} \\ \begin{pmatrix} +\infty & -3 & +\infty \\ +\infty & +\infty & 4 \\ 5 & +\infty & +\infty \end{pmatrix} \end{array}$$

Figure 1.10: Example DBM taken from chapter 4 of [181]

symbolic states of the region graph, having the same data part and satisfying the same time constraints.
Many operations can be performed using this representation [181], that simplify the implementation of algorithms useful for testing reachability.
Zones are efficiently represented and stored in memory by Difference Bound Matrices (DBMs) [30], that are $(n+1)$-square matrices, where n is the number of clocks. Coefficients $(m_{i,j})_{0 \leq i,j \leq n}$ of a DBM M are pair of values $(k, \prec)$, where k is an integer and $\prec$ is either $<$ or $\leq$, representing the constraint $x_i - x_j \leq k_{i,j}$, involving the set of clocks $\{x_i \mid 1 \leq i \leq n\}$. Moreover, to represent a constraint in the form $x_i \leq k$ a fictitious clock x_0 is introduced, whose value is always 0; so it is possible to write $m_{i,0} = k$ with the associated constraint $x_i - x_0 \leq k$. Using an example taken from chapter 4 of [181], in figure 1.10 the zone defined by the constraint $(x_1 \geq 3) \wedge (x_2 \leq 5) \wedge (x_1 - x_2 \leq 4)$ can be represented through the matrix of the coefficients M. The presence in a cell of the $+\infty$ value indicates that there is no constraint between the clocks associated to the specified row and column, while a negative coefficient is used to model a $\geq$ constraint, in fact, considering $m_{0,1} = -3$, the original constraint $x_1 \geq 3$ can be modelled equivalently as $x_0 - x_1 \leq -3$.
Despite every DBM represents a zone, and every zone can be represented by a DBM, the correspondence is not unique and a same zone can be represented by several DBMs.
In order to reduce the computing cost of a DBM, alternative solutions known in literature are *minimization algorithm for DBMs* [156], *federations* [87] that are in charge to merge efficiently DBMs or *clock difference diagrams* (CDDs) [155].
Zones are also related to regions through the following properties:

- each region is a zone;
- if Z is a zone, then $Z = \bigcup_i r_i$, where each r_i is a region;
- if $W = \bigcup_i Z_i$ is convex and each Z_i is a zone, then W is a zone.

To better understand the concept of zone, an example taken from [75] is shown in figure 1.11. The model describes a simply timed automaton of a periodic task, whose period is 6 time units. Each task instance consists of two sequential actions, having a not deterministic duration in the range $[1, 3]$ and $[2, 3]$ respectively. Locations L_0 and L_1 model action execution, while L_2 is used to model the waiting period of the next task activation. Two clock variables are used: x that is in charge to measure actions duration and y that is used to measure the periodic behaviour. In order to ensure the non-deterministic attendance, invariant conditions coupled with appropriate guards are introduced such as $x \leq 3$ attached to L_0 and $x \geq 1$ on the output edge from L_0 to L_1. Some edges are

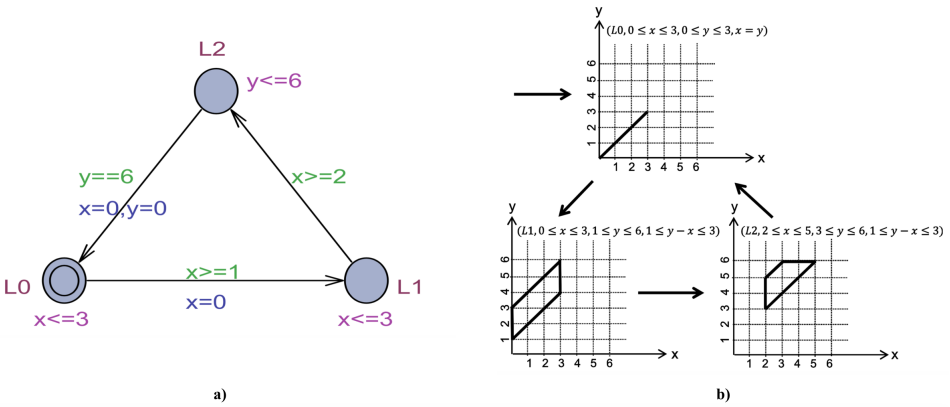


Figure 1.11: a) A simple of periodic timed automaton b) Associated zone state graph

equipped with clocks resetting actions, needed to allow the correct measure of the sojourn time. For example, on the edge from L_2 to L_0 the values of both clocks are set to zero, to indicate the beginning of a new instance of the task and to count the flow of other 6 time units, while on the edge from L_0 and L_1 x, is reset to measure the duration of the second action.
While the zones for the locations L_0 and L_1 are identifiable in an easy way, the clock constraints on L_2 are not obvious and need to be clarified. On L_2 the constraints are $2 \leq x \leq 5$ and $3 \leq y \leq 6$, because in the best case L_2 can be reached after the sum of $1tu$ needed to leave L_0 and $2tu$ to exit from L_1 on y, but after $2tu$ on x due to the x reset from L_0 to L_1, while the upper bounds can be immediately calculated. Considering the difference $y - x$, doing maths, it is deduced that $1 \leq y - x \leq 3$, where $1 = y - x$ and $y - x = 3$, that are respectively the remaining lines used to delimit the zone.

1.3 Some Available Model Checking Environments

Besides UPPAAL, several tools are currently available (and constantly evolving), useful for making model checking. In this section, a few of them are summarized.

1.3.1 Java PathFinder

Java PathFinder (JPF) is a model checker that has been developed as a verification and testing environment for Java multithreaded programs [172]. It is open source and online available and consists of a custom-made Java Virtual Machine (JVM) that is in charge to execute a program (Java bytecode) not only once as a normal VM, but theoretically in all the possible ways, exploring thread interleaving and non-deterministic assignments, for checking property violations like deadlocks or unhandled exceptions. For properties specification JPF uses:

- Java assertions inside the application under analysis (in this case any violation is captured and notified to the user);
- instance of gov.nasa.jpf.Property, gov.nasa.jpf.GenericProperty, or through the implementation of two listener classes gov.nasa.jpf.SearchListener and

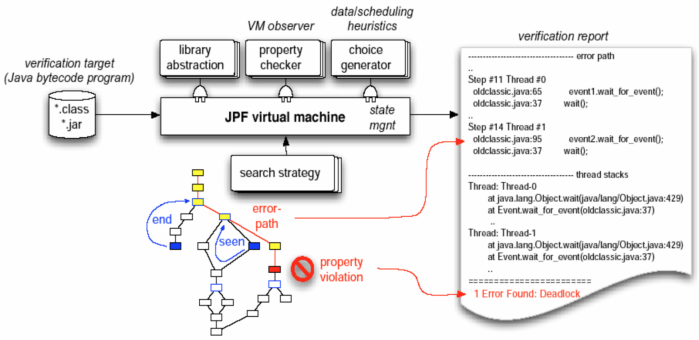


Figure 1.12: Java PathFinder architecture, taken from [172]

gov.nasa.jpf.VMListener.

JPF cannot analyze Java native methods and if the system under test calls such methods, these have to be provided within peer classes or intercepted by listeners.
The two main features of JPF are:

- *backtracking*, which indicates the ability to restore previous execution states, to see if there are unexplored choices left;
- *state matching*, during the execution a match is made between a new reached status and any other previously examined. JPF can then backtrack to the nearest non-explored non-deterministic choice.

Like other model checkers, also JPF is prone to state-explosion problems. Since concurrent actions can be executed in any arbitrary order, all their possible interleavings can lead to a very large state space. In particular, for n threads with m statements each, the number of possible scheduling sequences is equal to:

$$K = \frac{(nm)!}{m!^n} \tag{1.1}$$

JPF model checking can purposely exploit two techniques:

- *state abstraction*, which eliminates details irrelevant to a property, obtaining a simple finite model sufficient to verify the property. It can produce false positives/negatives due to the loss of precision;
- *Partial Order Reduction* (POR) which basically groups all instructions in a thread, that do not have any effects outside the thread, into a single transition.

Other detail can be found in [42], [172] and [207].

1.3.2 SPIN and PROMELA

SPIN is a popular open-source model checker, that can be used for formal verification of multi-threaded applications and for analyzing the logical consistency of concurrent and distributed systems [123].

The tool supports a high-level modelling language called PROMELA (PROcess MEta LAnguage), which allows the creation of processes by using classical data types, control flow instructions, loop structure, atomic statements, complex data structures, and process communication through the use of shared memory or message channels that can be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered).
SPIN has been used to trace logical design errors in distributed systems design, such as operating systems, communications protocols, switching systems, concurrent algorithms, railway signalling protocols, control software for spacecraft, nuclear power plants, etc. highlighting the presence of deadlocks, race conditions, different types of incompleteness, and unwarranted assumptions about the relative speeds of processes. Moreover, that tool directly supports the use of embedded C code as part of model specifications and using the model extractor *modex*, it is also possible to automatically generate a PROMELA model from a concurrent C code.
SPIN enables the use of multi-core computers for model checking, supporting the verification of both safety and liveness properties. It works on-the-fly, avoiding the need to preconstruct a global state graph, thus making it possible the checking of large system models. Property specification can be based on LTL, but other alternative and efficient on-the-fly ways, for safety and liveness verification exist. Correctness properties can also be specified as a system or process invariants (using assertions) and as formal Büchi automata.
Spin can be used in four modalities:

- as a *simulator*, that allows a rapid prototyping of the model under investigation;
- as an *exhaustive verifier*, to demonstrate the validity of user-defined requirements (using partial order reduction to optimize the search);
- as *proof approximation system*, that can validate very large system models, covering the maximum state space available;
- as a *driver for swarm verification*, which can make optimal use of large numbers of computing cores, to detect defects in large models.

All Spin software is written in ISO-standard C, and is portable across all versions of Unix, Linux, cygwin, Plan9, Inferno, Solaris, Mac, and Windows.

Chapter 2

The UPPAAL Symbolic Model Checker

UPPAAL is a popular, online available, continuously evolved, symbolic model checker jointly developed by the Uppsala University and the Aalborg University. It runs on the most common operating environments: Windows, Linux, Mac OS. It is designed for the exhaustive verification of real-time systems modelled as networks of timed automata (TA) [27]. Compared to similar tools, such as HYTECH [117] and KRONOS [266], UPPAAL is characterized by the adoption of compact data structures to represent the state graph of a model, and efficient traversal algorithms.
The main features of UPPAAL which motivated its use in this work are:

- its high expressive and simplicity of modeling, through the use of a user-friendly GUI;
- its faster verification engine;
- its extension with a statistical model checker.

In the past, UPPAAL was successfully exploited for off-line verification of industrial protocols, such as the collision handling in the *Philips Audio Control Protocol (PACP)* [38]. More recently, it was applied, e.g., to the on-line optimization of home automation and traffic control systems [157], such as the *Intelligent Control of Traffic Light (ICTL)* [100], or the CASSTING project, where the tool is used to synthesize an improved controller for a floor heating system of a single family house [154].
The toolkit is composed by three sub-environments:

- an intuitive graphical front-end *editor*, written in Java, used to design a TA model;
- a *simulator*, which allows to animate a model in order to observe graphically its execution;
- a *verifier* or *server*, written in C++, that implements the model checking engine which generates the state-graph of a model and verifies the satisfaction of the property queries of interest. The verifier is able to analyze all

the possible interleavings among the actions of the concurrent component processes.

This chapter briefly reviews the Uppaal symbolic model checker by focusing on the modelling capabilities and the query specification language.

2.1 Modelling language

The Uppaal modelling language extends the classical timed automata formalism with the following features:

- availability of integer and boolean primitive data, together with high-level data structures (arrays and structs);
- instantiation of parametric automata, called *template processes*, with possible local declarations (of integers, boolean, clock etc.);
- global declarations of *integer* (possibly bounded), *boolean* and *arrays* variables (of integers, boolean, clock and channels);
- *two-way synchronization* between processes, that is *rendezvous* as in CSP [177], [122]) based on unicast channels, where *!* indicates an output and *?* an input operation. As in classical TA, synchronization does not carry any data. However, any transmitted data can be simulated using global variables;
- broadcast (multicast) channels where a single sender can synchronize with a group of 0, 1 or multiple receivers; broadcast synchronization is always non-blocking;
- support of C-like syntax with the possibility of introducing *user-defined functions* either globally or locally to a template process.

2.1.1 Normal, urgent and committed locations

In Uppaal the term *state* represents a state of the whole system, obtained through the composition of multiple automata; the term *location* refers to the local state of a single automaton.
Three types of locations are recognized:

- *normal*, in which a TA can stay an arbitrary, possibly infinite, time;
- *urgent* (U) and *committed* (C), where no delay is allowed and from which an automaton has to go out with no time passing.

As shown in Fig. 2.1 a) an urgent location semantically corresponds to a location labelled with the invariant $x \leq 0$. A committed location, depicted in figure 2.1 b), introduces a more restrictive constraint. In a network of TA, the state of the overall system is marked as committed if at least one of its TA is in a committed location. Since delay for this configuration is not allowed, the committed location must be left in the successor state, so only action transitions starting from a committed location are allowed. This means that although urgent and committed locations both guarantee a zero sojourn time, committed locations have priority over the urgent ones.

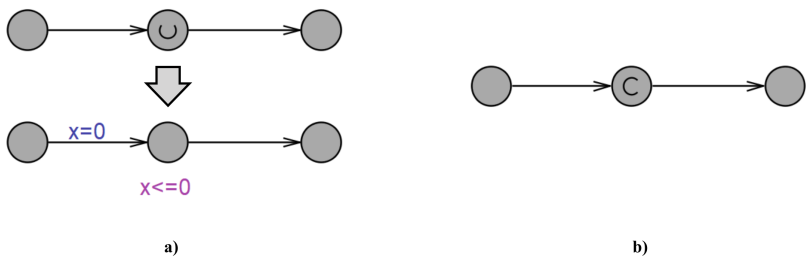


Figure 2.1: a) Urgent semantics b) Committed Location

2.1.2 Guarded commands

A *transition*, (referred as an *edge* in UPPAAL), can be labelled with an *action* which, in general, consists of four optional fields, that are *select*, *guard*, *synchronization* and *update*, as follows:

$$[select][guard][synch - i/o][update]$$

Each edge is enabled if the associated guard is true. If the guard is omitted, it evaluates to true. If an edge has no label, as shown in figure 2.2 a), the transition is said *spontaneous*.
The select component is useful for realizing a non-deterministic assignment, where the value is chosen in an interval of values. In addition, the select can be exploited for choosing a synchronization would multiple ones be ready.
The synchronization part of a command specifies one channel and an input/output operation on it. As indicated in Fig. 2.3, when a synchronization over a unicast channel is ready, with one sender and multiple receivers (or vice versa), one receiver is chosen non-deterministically by UPPAAL. However, during verification, all the possible choices are examined.
The update component is a comma-separated list of assignments, clock reset, or function calls. The operations in an update are executed from left to right.
A guarded command constitutes the fundamental element for modelling concurrent systems in UPPAAL. It represents a basic atomic action. Semantics of UPPAAL establishes that the update of the sender of a synchronization (*ch*!) is executed before that of the receiver(s) (*ch*?). This way the transmission of some arguments can be easily organized. In the case of a broadcast synchronization, the order of execution of the updates of the various engaged receivers, are carried out in the order in which the TA are listed in the *system configuration* statement which specifies the TA to be parallel composed.

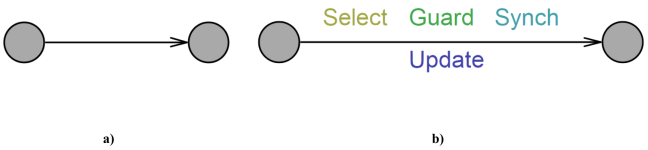


Figure 2.2: Command syntax: a) spontaneous edge b) labelled edge

2.1.3 Progress conditions

Besides the use of urgent and committed locations, the dwell-time in a normal location can be bounded through an invariant, which typically consists of a clock constraint (or conjunction of clock constraints). However, there are circumstances where it cannot be possibly anticipated the amount of staying time in a location, but when a condition states the location should be abandoned, the exiting should be realized without delay. Toward this, the notion of an urgent channel (unicast or broadcast) can be exploited. If a synchronization is ready on an urgent channel, it must be taken immediately. In particular, the use of an urgent and broadcast channel can sometimes be preferable where, for instance, an output operation which is heard by no other process can be used for forcing the exit from a normal location. However, for implementation/efficiency reasons, Uppaal does not allow to check clocks on a guard in combination with a synchronization of an urgent channel. All of this, in turn, is related to the requirement that zones of a state graph be always convex.

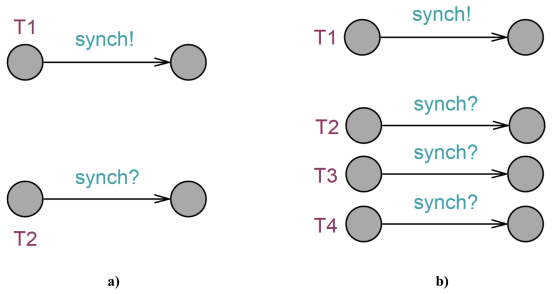


Figure 2.3: Unicast synchonization: a) deterministic b) non-deterministic

2.1.4 Global time and clocks

Uppaal uses a continuous time model, in which the global time value is not directly accessible, but it can be checked through the use of clock variables. Clocks allow to measure relative time amounts, such as the elapsed time from a given global instant when the clock was last reset. The value of a clock varies in dense intervals. It can be reset and compared exclusively using natural constants. After being reset, all the clocks of a model automatically advance with the same rate (first derivative equals to 1).
The example in figure 2.4, taken from [27], shows typical clock behavior. The overall system is the composition of two automata, which model a user and a lamp respectively. The user can press a button to switch on a lamp. The first press puts the lamp in a moderate light. A subsequent press, which occurs within 5 time units from the previous one (two consecutive close presses) causes the lamp to bright. A next press will put off the lamp. From the low location, if 5 or more time units elapse, the lamp gets switched off. The behavior is regulated through setting and checking a clock variable y.

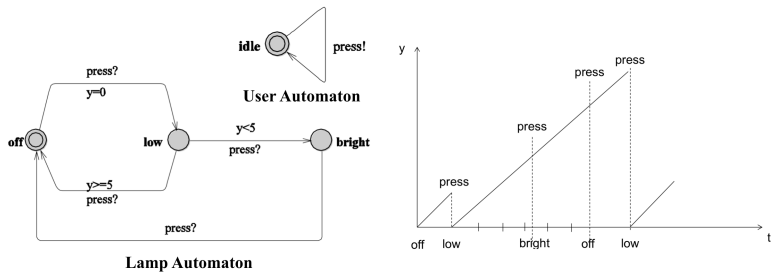


Figure 2.4: A simple lamp example taken from [27]

2.2 Query language

A network of TA can be inspected (i.e., tested) using the *simulator* component, by launching a single execution with a random choice of the next transition to be performed, or, it can be verified against some properties through the *model checker engine*, i.e. the *verifyta* kernel (see Fig. 2.5).
The properties to be verified are to be formally expressed using a subset of the Timed Computation Tree Logic (TCTL) [14]. Each query returns *true* when the property is satisfied, *false* otherwise. Like in TCTL, the query language consists of *path formulae*, that are in charge to infer properties over path or traces of the model, and *state formulae*, that describe what happens in individual states. Uppaal, though, does not allow nesting of path formulae [27]. A *state formulae*

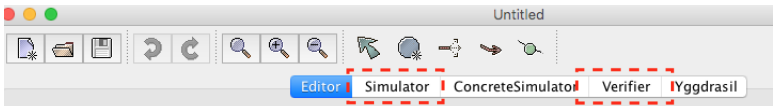


Figure 2.5: Example of how in the Uppaal toolkit GUI is possible to select a sub-environment

is an expression that can be evaluated for a state, without considering the overall behaviour of the model. The syntax of state formulae is a superset of that of guards, which provides for the use of logical connectors (&&, *and*, ||,*or*, !, *not*, *imply*) and it includes:

- logical conditions on data or clock variables;
- testing if an automaton is in a particular location;
- or a logical combination of the conditions above.

A default property, which is not strictly a state formula, is expressed by the keyword *deadlock*, used for checking for the possible presence of deadlocked states.
Path formulae can be classified into three categories:

- *reachability*: which has the task of verifying if a certain state can be achieved;
- *safety*: who is responsible for ensuring that the system does not come into a dangerous state (such as deadlock);

Formula	Meaning
$E <> \varphi$	Possibly φ: a state can be reached in which φ holds
$A[\,]\ \varphi$	Invariantly (in all states) φ holds. Equivalent to: not E<> not φ
$E[\,]\ \varphi$	Potentially always φ (a path exists where φ holds in all reached states)
$A <> \varphi$	Always eventually φ. Equivalent to: not $E[\,]$ not φ
φ –> ψ	φ always leads-to ψ. Equivalent to: $A[\,]$ (φ imply $A <> \psi$)

Table 2.1: Path formulae supported in Uppaal

- *(bounded) liveness*: whose task is to test if the system evolves towards some state (possibly within a given finite time);

that can are always investigated in terms of reachability on the state-graph.
Reachability properties can be used to validate a model, by checking the occurrence of expected or unexpected behaviour. Given a state formula φ, they are in charge to verify if φ can be satisfied by any reachable state (even after an infinite time), i.e., if there exists a path starting from the initial state, such that φ is eventually satisfied along that path. The path formula used to check these kinds of properties is in the form $E \diamond \varphi$, while in Uppaal it can be expressed using the syntax E<> φ.
Safety properties are used to ensure that the system does not have unwanted behaviours. In Uppaal these properties are formulated positively, i.e., in terms of something good is invariantly true. Let φ be a state formula, it is possible to express by using:

- A$\Box\varphi$ that φ should be true in all reachable states;
- E$\Box\varphi$ that should exist a maximal path (that is an infinite path or where the last state has no outgoing transitions) such that φ is always true;

In Uppaal these formulae can be written as $A[\,]\ \varphi$ and $E[\,]\ \varphi$ respectively. *Liveness properties* aim to guarantee that the desired state will be finally reached in the system. In its simple form, liveness is expressed with the path formula A$\diamond\varphi$, meaning that φ is eventually satisfied. The more useful form is the *leads to* property, written as $\varphi \rightsquigarrow \psi$ which is read as whenever φ is satisfied, then inevitably ψ will be satisfied. In Uppaal these properties are written as $A <> \varphi$ and φ - - > ψ. Note that the liveness property cannot be verified from normal locations, where on the exit edges there are at the most normal channels, because the process can stay there for an unbounded time.
Figure 2.6 and Table 2.1 summarizes the operational meaning of the basic queries.

When a property is not satisfied, Uppaal (if requested) can generate a counter-example, called *diagnostic trace*, automatically loaded in the simulator, which allows to see step-by-step the sequence of transitions that lead to the problem. A diagnostic trace can also be created when an existential query gets satisfied.

2.3 Advanced features

In latest versions of Uppaal, new features were introduced which enable the modelling of hybrid systems, that combine discrete and continuous behaviors.

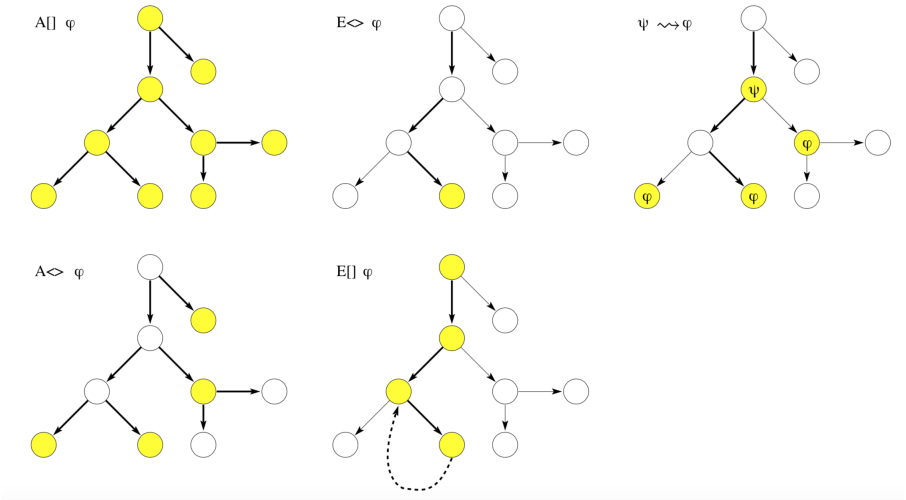


Figure 2.6: Path formulae supported in UPPAAL, taken from [27]

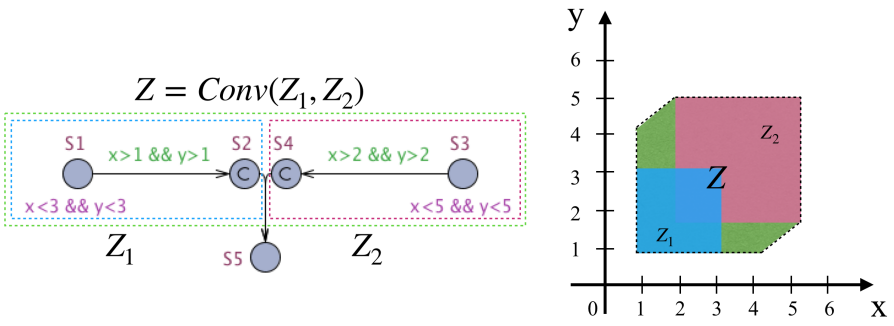


Figure 2.7: Example of over-approximation

Toward these clock variables are allowed to have different rates of advancement, regulated by putting into locations, as invariants, a first derivative of the clock different from the 1 default value. In addition, clocks are allowed to be stopped (stopwatches), thus retaining their values, and subsequently restarted from the value they were last stopped. These features are, for example, of interest when modelling preemptive real-time tasks [68, 88]. A preempted task, caused by the arrival of a greater priority one, has the need to frozen its clock which is measuring the execution time of the task body. All of this can be achieved by entering a location where the invariant states the first derivative of the clock variable is equals to zero. When the scheduler decides that a preempted task can again be released on the cpu, it is sufficient to force the exiting from the frozen clock location.
Unfortunately, the use of stopwatches does not come without problems. On the other hand, hybrid systems in most cases tend to be undecidable. It can be shown that a model with stopwatches generates not convex zones thus complicating the model checking (which sometimes cannot possibly terminate). In these cases, Uppaal resort to using the Over Approximation (OA) technique

[31] which is not always capable of furnishing certain results. Under OA a not convex zone is enlarged so as to cover the original zone and become convex (see the example in Fig. 2.7 taken from [] for the general concept of how a not convex zone can be transformed into a convex one). Obviously, an enlarged zone includes states which do not exist in the original model. Therefore, the issue of some queries can be inconclusive. In particular, properties which are invariably true (safety $A[\,]$ queries) on the OA based model, are certainly true also on the original model. An existential property which is satisfied on the OA based model, can be true on states introduced by the OA which do not belong to the modelled system. As a consequence, the concept of a property which *might* be or not satisfied arises.

Chapter 3

Probabilistic and Statistical Model Checking

Distributed probabilistic/stochastic timed systems represent a grand challenge for modelling and analysis. Symbolic model checkers like Uppaal can only evaluate some logical/temporal properties of these systems. In fact, when faced with a probabilistic and timed model, Uppaal ignores probabilistic aspects and turn probabilistic execution paths into non-deterministic paths. All of this is important but a full characterization of the behavior of these system models requires deriving a probability measure of event occurrences. On the other hand, exhaustive verification can incur into state explosion problems. Nowadays, *Probabilistic* and *Statistic Model Checking* represent two automated formal verification methods, that given a stochastic model $\mathcal{M}$ and a logical property φ, can perform two kind of analysis (possibly approximate) [153]:

- *qualitative*: is the probability for $\mathcal{M}$ to satisfy φ greater than or equal a certain threshold k ($P(\varphi) \geq k$)?
- *quantitative*: what is the probability for $\mathcal{M}$ to satisfy φ ($P_{=?}(\varphi)$)?

Analysis can exploit *numerical* or *statistical* techniques. The *numerical* approach checks the system using *symbolic methods* based on boolean formulas, or *numerical methods* [230]. Although accurate, this approach does not scale well and can be computationally expensive, being centred on the construction of a *probabilistic timed system* based on Markov chains and related timed variants [204], it can also be affected by state-space explosions [164]. The *statistical* approach, instead, although potentially less accurate, does not suffer of state-explosions, because it relies on Monte Carlo simulations. It provides a quantitative measure (p-value) of confidence of its answer using first a *sampling phase*, in which the system is *simulated* for a finite number of runs, e.g., established by Wald's hypotheses [255]. Then statistical inference is applied either according to the *hypothesis testing phase*, used to infer whether the samples provide statistical evidence for the satisfaction or violation of the specification [263], or with an *estimation phase* whose aim is to determine likely values of parameters, based on the assumption that the data are randomly drawn from

a specified type of distribution [9]. Intuitively, since this technique rests on a proportion of the results extracted from the various runs (with the amount of required memory which is linear with the model size), to obtain more accurate results the number of samples needs to be increased, thus implying a longer computation time.
In this chapter, first concepts of Probabilistic Model Checking are discussed, then the operation of Statistical Model Checking (SMC) is reviewed. SMC is chosen in this work for quantitative evaluation of distributed probabilistic and timed systems.

3.1 Probabilistic Models

Probabilistic model checking is an extension of model checking, applied to transition systems augmented with information about the likelihood that each transition will occur [141]. The behavior of a model with a set of states $\mathcal{S}$ is not specified by a *transition relation* on $\mathcal{S}$, but through a *transition function*. In order to carry out their operations, probabilistic model checkers require as input:

- a probabilistic model of the system;
- a temporal logic specification language, used to express qualitative and quantitative properties under investigation.

Table 3.1 summarizes some probabilistic models often used to build a formal description of a system with stochastic behavior, whose concepts are then briefly summarized.

Time	Non Deterministic	Probabilistic Models
Discrete	no	Discrete-time Markov Chains (DTMCs)
	yes	Markov Decision Processes (MDPs) Probabilistic Automata (PAs)
Continuous	no	Continuous-time Markov Chains (CTMCs)
	yes	Probabilistic Timed Automata (PTAs) Priced Probabilistic Timed Automata (PPTAs)

Table 3.1: Types of probabilistic models

3.1.1 Discrete-Time Markov Chains (DTMC)

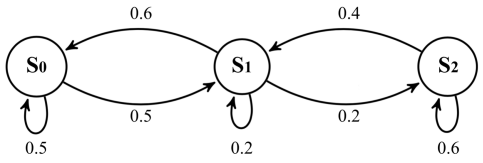
Discrete-Time Markov Chains (DTMCs) are fully probabilistic transition systems, in which the successor state of a process does not depend on the satisfaction of a guard, but it is chosen probabilistically within a set of admissible states. In a DTMC, transitions can occur at discrete time-step $n = 0, 1, 2, \ldots$, so if the system enters a state s at time n, it stays there for one time unit and then moves to s' at time $n + 1$ [204].
DMTCs are well suited for modelling, for example, simple random algorithms or synchronous probabilistic systems where components evolve in lock-step.
Formally, a DTMC is a tuple $\mathcal{M} = (S, s_0, M, L)$ where:

- S is a finite set of states;

- $s_0 \in S$ is the *initial state*;
- $M : S \times S \to [0, 1]$ is a *transition probability function*, that assigns probabilities to successor states, with the requirement: $s \in S$, $\sum_{s' \in S} M(s, s') = 1$;
- $L : S \to 2^{AP}$ is a *labelling function*, which assigns to each state $s \in S$ a set $L(s)$ of atomic propositions $a \in AP$;

A DTMC satisfies the *Markovian property (absence of memory)*: the probabilistic description of the system at time $n + 1$, only depends on the current state at time n, and not on the previous history; this implies that the sojourn time in a state is distributed according to a geometric distribution.
Graphically, a DMTC is represented:

- by an *direct graph*, in which states, labelled with atomic propositions, indicate a specific execution circumstance of the system, and edges, weighted with probability values, represent the eligible transitions with non-zero probabilities;
- by a *transition probability matrix*, in which rows and columns are labelled with states and the element at (i, j) represents the probabilistic weight associated to the transition from the status s_i to s_j.



M	S0	S1	S2
S0	0.5	0.5	0
S1	0.6	0.2	0.2
S2	0	0.4	0.6

Figure 3.1: Example of a Discrete Time Markov Chain and its transition probability matrix

Figure 3.1 shows an example of the two ways of representing the so called *homogeneous DMTC*, in which the probability values associated with transitions are costant and do not depend on the time-step n.
A *run* or *path* π in a DMTC $\mathcal{M}$ is defined as a sequence of states and transitions $s_0 \xrightarrow{p_0} s_1 \xrightarrow{p_1} \ldots \xrightarrow{p_{i-1}} s_i \xrightarrow{p_i} \ldots$ with $i \in \mathbb{N}$, $s_i \in S$, and $p_i \in \mathbb{R}_{(0,1]}$ such that $p_i > 0 \wedge p_i = M(s_i, s_{i+1})$ for all $i \geq 0$. It is indicated with $\pi^{:n} = s_0, s_1, \ldots, s_{n-1}, s_n$ [212]. Due to the Markovian property, transitions between states are selected independently from each other, and the probability of reaching a given state, following a path π, is given by:

$$P_{\mathcal{M}}(\pi) = \prod_{i=0}^{n-1} M(s_i, s_i + 1) \qquad s_i, s_{i+1} \in \pi$$

Considering the DMTC $\mathcal{M}$ in figure 3.1 and the path $\pi = s_0, s_0, s_1, s_0, s_1, s_2$, the probability of π is $P_{\mathcal{M}}(\pi) = 0.5 \cdot 0.5 \cdot 0.6 \cdot 0.5 \cdot 0.2 = 0.015$.
In order to carry out performance analysis on discrete-time and stochastic model, classical Computation Tree Logic (CTL), introduced by [82], has been

enriched to handle and formulate probabilistic queries. The *Probabilistic Computation Tree Logic* (PCTL) proposed by [116], provides the following syntax for query formulation [166]:

$$\phi ::= true \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid P_{\sim p}[\psi]$$
$$\psi ::= X\phi \mid \phi_1 U^{\leq k} \phi_2 \mid \phi_1 \, U \, \phi_2 \mid F\phi \mid G\phi$$

where a is an arbitrary atomic proposition, $p \in [0,1]$ a path probability value, $\sim \in \{<, \leq, \geq, >\}$ one of the possible inequality operators, and $k \in \mathbb{N}$ an arbitrary natural number. The ϕ formula is used to verify properties on a specific state of the model, whereas ψ formula is used to investigate the satisfaction of constraints over a path.
Compared to the basic CTL version, two new expressions are added:

- the formula $P_{\sim p}[\psi]$, that is true in a state $s \in S$ when the probability that ψ holds on paths starting from s exceeds a certain threshold ($\sim p$);
- the bounded until operator $\phi_1 U^{\leq k} \phi_2$, that is true for a path $\pi = s_0, s_1, \ldots, s_k, \ldots$ whenever there is a state s_{n+1} with $n+1 \leq k$ where ϕ_2 holds, and for all states up to including s_n, ϕ_1 holds (see equation 3.1) [9].

$$\begin{array}{cccccc} s_0 & s_1 & \ldots & s_n & s_{n+1} & \ldots \\ \phi_1 & \phi_1 & \ldots & \phi_1 & \phi_2 & \ldots \end{array} \tag{3.1}$$

3.1.2 Markov Decision Processes (MDPs)

Some aspects of a system which may not be modelled probabilistically, include:

- concurrency in the scheduling of parallel components;
- unknown environments;
- underspecification, i.e., unknown model parameters.

To model these aspects, the concept of Labelled Transition System (LTS) where the choice of the next state is nondeterministic, can be combined with that of DTMC where the next state is chosen probabilistically.
A Markov Decision Process (MDP) is a discrete-time state-transition system with both nondeterministic and probabilistic behavior. As in a DTMC, transitions between states occur in discrete time-steps and a discrete set of states represent possible configurations of the modelled system. Formally, it can be described as a tuple $\mathcal{M} = (S, s_o, \mathcal{A}, T, R(s))$ where:

- S is a finite set of states;
- s_o are the initial states;
- $\mathcal{A}$ is a set of *actions* used to *control* the system state. The set of ations that can be applied in $s \in S$ are labelled as $\mathcal{A}(s)$, where $\mathcal{A}(s) \subseteq \mathcal{A}$;
- T is the *transition probability function* $T : S \times \mathcal{A} \times S \to [0,1]$ that describes the probability of reaching state s' after doing an action a in state s: $T(s,a,s') = Pr(s'_{t+1}|s_t, a_t)$. For all states s and actions a, $\sum_{s' \in S} T(s,a,s') = 1$, i.e. T defines a probability distribution over *possible next states.*

- R is a *reward function* that gives a reward (a real number) for:
  - performing an action in a state: $R : S \times \mathcal{A} \to \mathbb{R}$;
  - particular transitions between states: $R : S \times \mathcal{A} \times S \to \mathbb{R}$.

Rewards can be considered as a short-term *utility function*, which associate events to positive or negative values, to offer a measure of the *usefulness of decisions*. In a MDP is still valid the *Markovian property*: current state s holds enough information for making an *optimal decision* (i.e., a decision with maximum reward), and it is not important which states and actions occurred previously [248].
The behavior in a state s is both probabilistic and nondeterministic, in fact, first an available action ($a \in \mathcal{A}(s)$) is selected nondeterministically, then a successor state is chosen according to the probability by $T(s, a)$ [143].

3.1.3 Probabilistic Automata (PAs)

Probabilistic Automata (PA) [215] [209] are state machines useful for modelling systems that at any time-step can evolve according to a probability distribution. PA differs from an ordinary automata in the transition relation, because the successor s' of a state s is selected by:

- choosing non-deterministically one transition among the ones outgoing s;
- selecting the action to be performed and the state that will be reached through a discrete probability distribution;

MDPs are closely related to PA. In fact, by identifying the controls of MDP with the inputs of PA, the two computational models are almost identical. They differ, though, in the way they are used. Generally PAs are adopted to study the input acceptance sequences of a system, under different acceptance conditions. MDPs are employed to identify control policies that maximize or minimize given functionals [91].
Formally, a PA is a tuple of $\mathcal{M} = (S, s_o, \mathcal{A}, T, L)$ where:

- S is a set of states;
- s_0 is the initial state;
- $\mathcal{A}$ is a set of *actions* (known as the *alphabet* of the model);
- T is the *transition probability function* $T : S \times \mathcal{A} \times S \to [0, 1]$, which associates actions with a probability weight to reach a successor state;
- $L : S \to 2^{AP}$ is a *labelling function*, which assigns each state $s \in S$ with a set $L(s)$ of atomic propositions $a \in AP$.

3.1.4 Continuous-Time Markov Chains (CTMCs)

A Continuous-Time Markov Chain (CTMC) is an extension of DTMC, that allows to incorporate real-time information. In contrast to the discrete version, transitions are not triggered by probabilities at fixed times $n = 0, 1, 2, \ldots$, but

can occur at any arbitrary time t, according to a movement rule determined by the q_{ij} rate variables. Since the *Markovian memoryless property* still remains valid, the future state of the model depends only on the current one. Consequently, the sojourn time in a state s_i can be modeled by a sample extracted from an *exponential distribution function*, whose *rate* is equal to the sum of the all q_{ij} values, associated to the outgoing transitions from s_i:

$$F_S(t) = 1 - e^{-(\sum_j q_{ij})t}$$

Formally, a CTMC is a tuple $\mathcal{M} = (S, s_0, Q, L)$ where:

- S is a finite set of states;
- $s_0 \in S$ is the *initial state*;
- $Q : S \times S \to \mathbb{R}_{\geq 0}$ is the *transition rate matrix*, which expresses the instantaneous tendency of the system (q_{ij}) to move from s_i to s_j: $q_{ij} = \lim_{\Delta t \to 0} \frac{P(S(t+\Delta t)=s_j | S(t)=s_i)}{\Delta t}$;
- $L : S \to 2^{AP}$ is a *labelling function* which assigns to each state $s \in S$ a set $L(s)$ of atomic propositions $a \in AP$.

In general, the elements of the Q matrix are time-variant ($Q(t)$), but for *homogeneous CTMCs*, they are supposed to be constant.
In order to analyze a CTMC, the *Continuous Stochastic Logic* (CSL) specification language is used [166]:

$$\phi ::= true \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid P_{\sim p}[\psi] \mid S_{\sim p}[\phi]$$
$$\psi ::= X\phi \mid \phi_1 U^I \phi_2 \mid F\phi \mid G\phi$$

where a is an arbitrary atomic proposition, $p \in [0, 1]$ a path probability value, $\sim \in \{<, \leq, \geq, >\}$ one of the possible inequality operators, and $I \in \mathbb{R}_{\geq 0}$ is an arbitrary interval of $\mathbb{R}_{\geq 0}$. The ϕ formula is used to verify properties on a specific state, whereas ψ formula is used to investigate the satisfaction of constraints over a path.
The new expressions added by CSL are:

- $S_{\sim p}[\phi]$: used to determine the probability that a certain (state) property *finally* holds, in the long term;
- $\phi_1 U^I \phi_2$: used to assure that a certain property ϕ_1 holds until another property ϕ_2 holds, which has to occur within the given time interval I.

3.1.5 Probabilistic Timed Automata (PTAs)

Probabilistic Timed Automata (PTAs) are finite-state machines with features of TA and MDPs. They are useful for modelling systems with probabilistic, nondeterministic and real-time aspects. As in TA, also in PTA time is *dense* ($t \in \mathbb{R}$), and modelled through *clocks*, whose values increase simultaneously and uniformly. *Guards* and *invariants* are assigned respectively to transitions and states for:

- imposing a restriction on when a transition can occur;

- enforcing an upper bound on the instant at which certain probabilistic choices are made, so defining how long the model can stay in a state [144].

The formalism also allows the use of internal finite-ranging *data variables*.
Formally, a PTA is a tuple $\mathcal{M} = (S, s_0, L, \mathcal{X}, inv, Pr, \langle \tau_s \rangle_{s \in S})$, where:

- S is a finite set of states;
- s_0 is the initial state;
- $L : S \to 2^{AP}$ is the *labelling function* that assigns to each node a set of atomic propositions;
- $\mathcal{X}$ is a finite set of *clocks*;
- $inv : S \to Z_{\mathcal{X}}$ is the function that assigns to each node an invariant condition (with $Z_{\mathcal{X}}$ the set of all zones, conjunctions of clock constrains, of $\mathcal{X}$);
- $Pr : S \to P(\mu(S \times 2^{\mathcal{X}}))$ is a function that assigns to each node a (finite, non-empty) set of discrete probability distributions on $S \times 2^{\mathcal{X}}$;
- a family of functions $\langle \tau_s \rangle_{s \in S}$ where, for any $s \in S$, $\tau_s : Pr(s) \to Z_{\mathcal{X}}$ assigns to each $p \in Pr(s)$ an enabling condition.

During a state transition, clocks can be reset (to integer values) and data-variables can be updated.
Discrete transitions are instantaneous and consist of two steps:

- a *nondeterministic choice* between the set of distributions $p \in Pr(s)$ whose corresponding enabling condition $\tau_s(p)$ is satisfied by current values of the clocks;
- supposing that the probability distribution p is chosen, a *probabilistic transition* is performed according to p; that is, for any $s' \in S$ and $X \subseteq \mathcal{X}$, the probability that the system will make a transition to s', and reset all clocks in X, is $p(s', X)$.

Verification of PTAs addresses a wide range of *quantitative properties*, from reliability to performance, like the following ([258], [200]):

- the maximum probability of an airbag failing to deploy within 0.02 seconds;
- the minimum probability that a packet is correctly delivered with 1 second;
- the maximum expected time for the protocol to terminate.

3.2 The PRISM probabilistic model checker

PRISM is a notable general stochastic model checker, mainly developed at the University of Oxford [144], for formal modelling and analysis of systems with a probabilistic behaviour. It incorporates state-of-the art symbolic data structures and algorithms based on BDDs (Binary Decision Diagrams) and MTBDDs

(Multi-Terminal Binary Decision Diagrams). The tool supports both numeric/symbolic techniques and statistical model checking. Prism is based on Java and runs on the most common operating systems like Windows, Linux and Mac OS, with 64-bit variants. For improved verification, Prism relies on such techniques as *quantitative abstraction refinement* [58] and *symmetry reduction* [141]. The PRISM language, although in a textual not graphical form, recognizes stochastic systems of the kind DTMCs, CTMCs, MDPs and PTAs. The tool can simulate systems at the language level (on-the-fly), and hence does not need to store complete representations in memory. The supported property specification language incorporates the temporal logics PCTL, CSL, pLTL and PCTL$*$, as well as extensions for quantitative specifications and costs/rewards.
The *discrete-event simulation engine* enables two different confidence interval methods to carry out approximate/statistical model checking; one is based on Chernoff bounds and the other depends on the sequential probability ratio hypothesys test [59], [142].
In its current version, PRISM does not support high-level data structures (e.g., arrays of integers) and user-defined functions which can contribute to compact modelling in specific application domains.

3.3 Statistical Model Checking

In cases where the application of probabilistic model checking is infeasible, probability evaluation can be approximated by a statistic estimator [199]. Statistical Model Checking (SMC) [9] is a simulation-based approach that aims to do this by adopting Monte-Carlo like simulations. SMC does not build the state graph of a model. Therefore, it can be applied to a larger class of systems than numerical model checking algorithms, including black-box systems and infinite state systems. SMC works on a set of independent stochastic simulation traces, obtained in an easy way by simulating model execution. Such traces are then checked with respect to a logic such as Bounded Linear Temporal Logic (BLTL) [37] and the results are combined with statistical methods such as the hypothesis testing. Moreover, being independent, traces may be generated on different machines, so SMC can efficiently exploit modern parallel architectures.
Recently, SMC developers are engaged in a new challenge: that of allowing the analysis of *rare-events* [152], that are events whose occurrence probability p is very low. To overcome issues such as requiring an excessively large number of samples to observe the target event, new techniques based on the *Importance Sampling* (IS) and the *Importance Splitting* (IP) are being studied and are under investigation [220].

3.3.1 SMC Operation

Statistical model checking operations are summarized in figure 3.2. The first step is generating a set of samples (simulation runs) on which perform the analysis, according to two possible strategies:

- *fixed-size sampling*, in which the sample size is predetermined;
- *incremental or sequential testing*, in which samples are collected and statistical tests performed, until a decision is reached; therefore, the number

of required observations need not be known beforehand.

Since taking a limited number of runs obviously offers a low guarantee about the result, in paragraph 3.3.3 are indicated some strategies for computing the optimal number of samples needed to obtain a given level of reliability.
Once extracted the paths, the verification phase consists in estimating if each of them satisfies a property. In the analysis phase, the estimator is updated with the evaluations coming from the verification phase and it will be possibly returned to the user [199]. The result of an SMC analysis is an estimate $\tilde{\gamma}$ of the quantity γ that expresses the exact measure with which the system satisfies a property, together with a statistical statement about the potential error. A typical guarantee is a probability δ representing the confidence, any $\tilde{\gamma}$ will be within $\pm\epsilon$ of γ, where ϵ indicates the precision. To strengthen the guarantee, i.e. increase δ or decrease ϵ, more samples are needed [47].

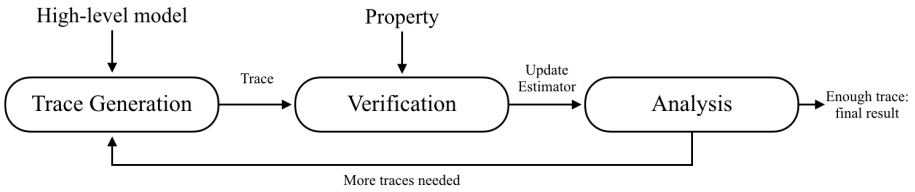


Figure 3.2: Statistical Model Checking operation, taken from [199]

SMC verification algorithms are divided into two different classes, that are:

- *quantitative algorithms* that calculate a probability measure satisfying a property;
- *qualitative algorithms* that decide if the probability of a requirement is above a given threshold, choosing between two possible contrary hypotheses.

3.3.2 Monte Carlo simulations

Classical Monte Carlo technique is a quantitative algorithm that uses N simulation traces (w_i, $i \in \{1, \ldots, N\}$), to calculate a measure:

$$\tilde{\gamma} = \sum_{i=1}^{N} \frac{\mathbf{1}(\omega_i \models \varphi)}{N}$$

that represents an estimation of the probability γ with which the model satisfies a logical formula φ. The operator $\mathbf{1}(\cdot)$ is a function, whose value is 1 if its argument is *true* and 0 otherwise.
Using the Chernoff-Hoeffding bound [203] and setting the number N of simulations to the value:

$$N = \left\lceil \frac{ln(2) - ln(\delta)}{2\epsilon^2} \right\rceil$$

it is guaranteed that the probability of error is $Pr(|\tilde{\gamma} - \gamma| \geq \epsilon) \leq \delta$, where, as noted previously, ϵ indicates the precision and δ the confidence. Parameter δ is related to the number N of simulation, through the equation:

$$\delta = 2e^{-2N\epsilon^2}$$

3.3.3 Hypothesis testing

The main approaches to investigate qualitative properties are based on the *hypothesis testing* [9]. Hypothesis testing uses statistics to determine the probability that a given *hypothesis* is true, where a statistical hypothesis is a statement about the values of the parameters of a probability distribution [178].
The hypothesis testing procedure consists of the following steps:

- formulating two competing and complementary hypotheses:
  - H_0, called *null hypothesis*, that represents what it would be accepted until it is proved otherwise and is the hypothesis that the probability measure p of φ for a run is greater than a certain fixed threshold Θ ($H_0 : p \geq \Theta$);
  - H_1, called *alternative hypothesis*, is the hypothesis that the probability measure p of φ for a run is less than a certain fixed threshold Θ ($H_1 : p < \Theta$);
- setting an acceptable significance value α, that acts as threshold for the refusal or not of the null hypothesis;
- identifying a *test statistic* to use for assessing the truth of the null hypothesis (reject or fail to reject H_0);
- following a choice of the appropriate test for the problem, two equivalent methods for proceeding are:
  - critical test values method: it uses the α value as a threshold to decide the rejection of H_0, by dividing the sample-space into two regions:
    - an *acceptance region*, in which if the test statistic value falls within it, H_0 is accepted;
    - a *critical region* or *reject region*, in which if the test statistic is enclosed in it, H_0 is reject;

    such regions can be an interval, or a union of two intervals, depending on whether the test is unilateral or bilateral.
  - compute the *p-value* $\tilde{p}$, which is the probability that a test statistic at least as significant as the one observed would be obtained by assuming the null hypothesis was true. The smaller the p-value, the stronger is the evidence against the null hypothesis. And then compare the p-value with α: if $\tilde{p} \leq \alpha$, H_0 is rejected and H_1 accepted.

It is important to note that when a null hypothesis is not rejected, it can not be considered as *true*, but only that it could be true but the samples do not provide sufficient evidence for its refusal. Inference, in fact, represents only an indication that the evidence of an hypothesis is supported by the available data.
Two kinds of errors may be committed when testing hypotheses:

- *type I error* when the null hypothesis is rejected but it is true;
- *type II error* when the null hypothesis is not rejected but it is false.

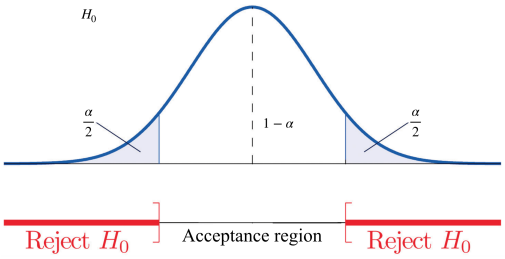


Figure 3.3: Sample-space regions for the decision rule

The probabilities of these two types of errors are:

$$\alpha = P\{type\ I\ error\} = P\{reject\ H_0 \mid H_0\ is\ true\}$$

$$\beta = P\{type\ II\ error\} = P\{fail\ to\ reject\ H_0 \mid H_0\ is\ false\}$$

where $\alpha+\beta \leq 1$ and the test cannot work if $\alpha = \beta$. Figure 3.4 summarizes all the possible outcomes and admissible combinations of hypothesis, from which one can deduce that if the α value will be small, the probability of falsely rejecting the null hypothesis will be smaller.

	Decision	
Truth	*accept* H_0, *reject* H_1	*reject* H_0, *accept* H_1
$p \geq \Theta$: H_0 *true*, H_1 *false*	correct ($> 1-\alpha$)	type I error ($\leq \alpha$)
$p < \Theta$: H_0 *false*, H_1 *true*	type II error ($\leq \beta$)	correct ($> 1-\beta$)

Figure 3.4: Schema of errors in a statistical test

Test statistic

The *test statistic* is a quantity computed from the observed data, that summarizes the relevant sample information for the evaluation of the likelihood of the hypothesis. Different tests can be adopted depending on the hypothesis system and the auxiliary assumptions, but since main problems have an already developed solution, the work becomes that of trying to bring the real problem back to one of these standard situations.
For example, the procedure for testing an hypothesis H_0 about a random variable x, with unknown mean value μ:

$$\begin{cases} H_0 : \mu = \mu_0 \\ H_1 : \mu \neq \mu_0 \end{cases}$$

consists in taking a random sample of n observations on x and compute the statistic test *Z-test* [178]:

$$Z_0 = \frac{\overline{x} - \mu_0}{\frac{\sigma}{\sqrt{n}}}$$

and **reject** H_0 if $|Z_0| > Z_{\frac{\alpha}{2}}$ where $Z_{\frac{\alpha}{2}}$ is the upper $\frac{\alpha}{2}$ percentage point of the standard normal distribution (i.e., the reference distribution for the Z-test). In some situations H_0 could be rejected only if the true mean is larger than μ. Thus, the *one-sided* alternative hypothesis is $H_1 : \mu > \mu_0$, and $H_0 : \mu = \mu_0$ would be **rejected** only if $Z_0 > Z_\alpha$. If rejection is desired only when $\mu < \mu_0$, then the alternative hypothesis is $H_1 : \mu < \mu_0$, and H_0 is **rejected** only if $Z_0 < -Z_\alpha$.
To clarify the analytics details about the functioning of this technique, in the following a practical example is showed (taken from [178]).

Example: *Computer response time*
The response time of a distributed computer system is an important quality characteristic. The system manager wants to know whether the mean response time to a specific type of command exceeds 75 ms. From previous experience, he knows that the standard deviation of response time is 8 ms. The type I error if fixed at $\alpha = 0.05$.

The appropriate hypotheses are:

$$\begin{cases} H_0 : \mu = 75 \\ H_1 : \mu > 75 \end{cases}$$

The command is executed 25 times (n) and the average response time is $\overline{x} = 79.25$ ms. The value of the test statistic is:

$$Z_0 = \frac{\overline{x} - \mu}{\frac{\sigma}{\sqrt{n}}} = \frac{79.25 - 75}{\frac{8}{\sqrt{25}}} = 2.66$$

Since $\alpha = 0.05$ and the test is one-sided, from the tabled values of the normal distribution: $Z_\alpha = Z_{0.05} = 1.645$. Therefore, $H_0 : \mu = 75$ is rejected and it is possible to conclude that time exceeds 75 ms.

If the σ^2 value is unknown an additional assumption is needed to carry out the test. The random variable x is considered distributed according to a normal, and the unknown variance is replaced with s, that is the variance value obtained from the samples:

$$s^2 = \frac{\sum(x_i - \overline{x})^2}{n - 1}$$

In this case the statistic test adopted is:

$$t_0 = \frac{\overline{x} - \mu_0}{\frac{s}{\sqrt{n}}}$$

and the reference distribution is the t-distribution with *n-1* degrees of freedom.

Confidence Interval

The information about values of the process parameters that are in a sample can be expressed in terms of an interval estimate called a *confidence interval*. An interval estimate of a parameter is the interval between two statistics that

includes the true value of the parameter with some probability. For example, to construct an interval estimator of a parameter μ, two statistics L and U must be identified, such that:

$$P\{L \leq \mu \leq U\} = 1 - \alpha$$

and the resulting interval

$$L \leq \mu \leq U$$

is called a $100(1-\alpha)\%$ *confidence interval (CI)* for the unknown mean μ. L and U are the lower and upper *confidence limits*, respectively, and $1-\alpha$ is the *confidence coefficient*. Sometimes the half-interval width $U-\mu$ or $\mu-L$ is called *accuracy* of the confidence interval.
Assuming that x is a normal random variable with unknown mean μ and unknown variance σ^2, and that from a random sample of n observations, the sample mean $\overline{x}$ and the sample variance s^2 are computed. The confidence interval is obtained by:

$$\overline{x} - t_{\frac{\alpha}{2},n-1}\frac{s}{\sqrt{n}} \leq \mu \leq \overline{x} + t_{\frac{\alpha}{2},n-1}$$

Error control in hypothesis testing

Considering a property φ and the query that would be checked, written in a PCTL-like specification language, $P_{\geq\Theta(\psi)}$. Assuming that to verify the query it is necessary to monitor the random variable X in the simulation traces, whose values are: $Pr[X=1] = p$ and $Pr[X=0] = 1-p$. The goal is to obtain results that bound the probability of a Type I error (α) and Type II error (β), respectively.
Let L_p be the probability of accepting H_0 in a statistical test. To provide the desired error bounds, the test has to guarantee that whenever $p < \Theta$, $L_p \leq \beta$ and whenever $p \geq \Theta$, $L_p > 1-\alpha$ [9]. As shown in figure 3.5 a), plotting a theoretical trend of p for varying L_p, it emerges a critical case in which in corrispondence of $p \approx \Theta$, $L_p > 1-\alpha$ and $L_p \leq \beta$ hold at the same time. This

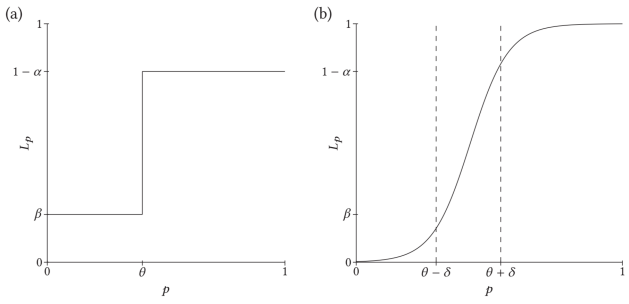


Figure 3.5: Probability L_p of accepting $H_0 : p \geq \Theta$, taken from [9]

is a problem, because to ensuring the fulfillment of both conditions, the test requires:

- sampling more extensively;
- setting $\beta = 1-\alpha$;

but the former constraint may be infeasible, and the latter implies that the error probabilities for Type I and Type II errors become not independently controllable [265].
Another solution to carry out more flexible tests, consists into the introduction of an *indifference region* around the Θ value, whose width depends from a parameter δ. When p belongs to this area, the tests provide no error-related guarantees.
By adding this feature, the new curve for the p-L_p relation is depicted in figure 3.5 b). To take the indifference region into account, the original hypothesis $H_0 : p \geq \Theta$ is changed to $H_0 : p \geq \Theta + \delta$ and similarly the alternative hypothesis passes from $H_1 : p \leq \Theta$ to $H_1 : p \leq \Theta - \delta$. Consequently, both hypotheses H_0 and H_1 are considered *false* when $p \in (\Theta - \delta, \Theta + \delta)$ and the new resulting schema of errors is shown in the following figure [9].

	Decision	
Truth	*accept* H_0, *reject* H_1	*reject* H_0, *accept* H_1
$p \geq \Theta : H_0$ *true*, H_1 *false*	correct ($> 1 - \alpha$)	type I error ($\leq \alpha$)
$\Theta - \delta < p < \Theta + \delta : H_0, H_1$ *false*	indifferent	indifferent
$p < \Theta : H_0$ *false*, H_1 *true*	type II error($\leq \beta$)	correct ($> 1 - \beta$)

Figure 3.6: Errors in a test with α, β and δ parameters, taken from [9]

Sequential Probability Ratio Test

The *Sequential Probability Ratio Test (SPRT)* of Wald [255] is used to evaluate the hypothesis of the form $Pr(\omega \models \varphi) \bowtie p$ where $\bowtie \subset \{\leq, \geq\}$. For choosing between the following H_0 and H_1 hypotheses:

$$\begin{cases} H_0 : Pr(\omega \models \varphi) \geq p^0 \\ H_1 : Pr(\omega \models \varphi) \leq p^1 \end{cases}$$

SPRT defines a Bayesian likelihood measure, called probability *ratio*, as:

$$f_m = \prod_{i=1}^{N} \frac{(p^1)^{\mathbf{1}(\omega_i \models \varphi)}(1-p^1)^{\mathbf{1}(\omega_i \not\models \varphi)}}{(p^0)^{\mathbf{1}(\omega_i \models \varphi)}(1-p^0)^{\mathbf{1}(\omega_i \not\models \varphi)}}$$

where N is the number of simulation traces.
The main idea behind a sequential test is to perform a simulation and to estimate the obtained step-by-step ratio value, until the collected samples are sufficient to end the process decision, according to the rule [114]:

$$f_m = \begin{cases} \geq \frac{1-\beta}{\alpha} & H_1 \text{ is accepted} \\ \leq \frac{\beta}{1-\alpha} & H_0 \text{ is accepted} \end{cases}$$

More details concerning the statistical problems or the $P_{=?}(\varphi)$ and $P_{\sim p}(\varphi)$ estimation are reported in [9] and [199].

3.4 Plasma Lab statistical model checker

Plasma Lab (Platform for Learning and Advanced Statistical Model checking Algorithms) is a compact, efficient and flexible platform for statistical model checking of stochastic models. It may be used as a stand-alone tool with a graphical development environment or invoked from the command line for high-performance applications.
The tool is written in Java for cross-platform compatibility and differs from other SMC due to the presence of an API abstraction of the concepts of *stochastic model simulator*, *property checker (monitoring)* and *SMC algorithm*, that allow it to use external simulators or input languages [165]. The tool architecture is

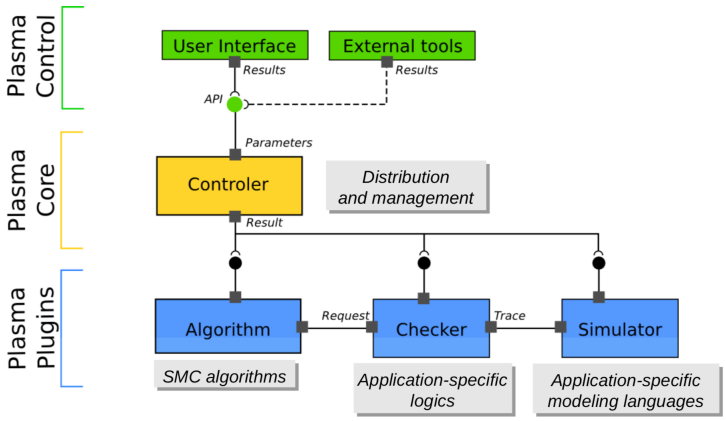


Figure 3.7: Plasma Lab architecture taken from [165]

summarized in Fig 3.7. The core of Plasma Lab is a light-weight *controller* entity that manages the experiments and the distribution mechanism, implementing an API that allows to control the experiments either through a user interface or external tools. The *controller* loads three types of plugins, that are:

- algorithms,
- checkers,
- and simulators,

that are enabled to communicate each other and with the controller.
An SMC algorithm works by collecting samples obtained from the checker component. In turn, the checker asks the simulator to initialize a new trace, with a state on-demand approach: new states are generated only when needed to decide a property. Depending on the query, the checker either returns boolean or numerical values. Finally, the algorithm notifies the progress and sends the results to the user interface, through the controller API. The tool supports Bounded LTL specifications and offers three analysis methods in the form of simple Monte Carlo, Monte Carlo using Chernoff bounds and sequential hypothesis testing.
More information on Plasma are reported in [39], [165].

3.5 VeStA and PVeStA

VeStA is an on-line available SMC tool, written in Java and developed at the University of Illinois [9]. Its simulation engine works on models generated by using two modelling languages:

- PMaude, that is an executable algebraic specification language, which allows to describe models in probabilistic rewriting logic dialect [140];
- a language very close to the PRISM syntax, oriented to for specifying the system as a DTMC/CTMC model.

Properties are checked by exploiting a sequence of inter-related statistical hypothesis testing, and are specified in PCTL, CSL or Quantitative Temporal Expressions (i.e. QuaTEx). In particular, the results of a QuaTEx expression [8] is statistically evaluated by sampling, until the size of $(1-\alpha)100\%$ confidence interval gets bounded by δ, where α and δ are provided as input.
PVeStA is an extension of the VeStA tool, with a parallel implementation, which improves specifically the analysis execution efficiency. PVeStA includes a client-server architecture, whose aim is to distribute and to parallelize the execution of model checking algorithms, where as shown in Fig. 3.8:

- the client component is in charge of implementing the sequential part of the SMC algorithm, spreading the simulation of runs to be performed, among different R_i server components;
- the server component, instead, is in charge of performing the requested number of simulations, by adopting a pseudo-random seed to guarantee statistical independence of the results.

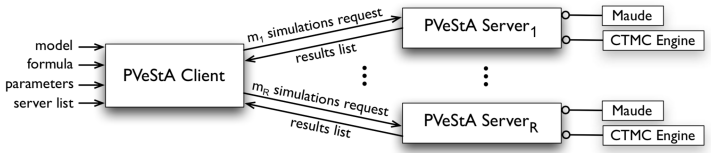


Figure 3.8: PVeStA architecture schema, taken from [13]

More details about these tools can be found in [231] and [13].

Chapter 4

The UPPAAL Statistical Model Checker

In last years the UPPAAL toolbox was extended with a statistical model checker (SMC) [90] to allow probabilistic modelling and verification of *hybrid* and *stochastic* systems. UPPAAL SMC represents systems through a *network of stochastic timed automata* (NSTA) [89] which communicate each another by broadcast synchronizations only. The success of UPPAAL SMC depends on its ability to:

- provide a simple implementation of reactive systems through NSTA;
- model check properties that cannot be expressed in classical temporal logics;
- require no extra modelling or particular specification effort.

In this chapter, the main features of UPPAAL SMC, selected in this work for quantitative evaluation of probabilistic and timed actors and more in general for analyzing multi-agent systems, are summarized, by focusing on the query language used to formulate and check stochastic properties. Although SMC is potentially less accurate with respect to a probabilistic model checker based on numerical methods and stochastic transition system, it can anyway provide, e.g., a confidence interval adequate, from a practical engineering point of view, to evaluate system behavior and event occurrences.

4.1 Network of Stochastic Timed Automata

A *Stochastic Timed Automata (STA)* combines the general structure of a timed automaton with underlying stochastic processes [166]. STA replace non-deterministic choices between multiple enabled transitions with probabilistic choices, as well as non-deterministic choices of time-delays with customizable probability distributions, so that the set of successors of a state is defined through a probability measure.
An STA composition constitutes a *Network of Stochastic Timed Automata (NSTA)*, in which the different components are able to communicate each other by using

broadcast synchronization channels or *shared variables*. The choice to restrict communications to the broadcast type only is due to keep always components in an unblocked state.
Uppaal SMC uses the NSTA formalism and adds the following features:

- built-in uniform or exponential distributions to model the sojourn time in a normal location. General probability distributions, if needed, can also be reproduced explicitly by the modeler (see later in this chapter);
- clocks with an arbitrary integer rate;
- branching edges in automata (see figure 4.1 a)) with probability weights which provide a discrete probability distribution to transitions;
- support for double variables. A double can be used to save the value of a clock at a given time, but also for decorating a model so as to facilitate output collection and extraction.

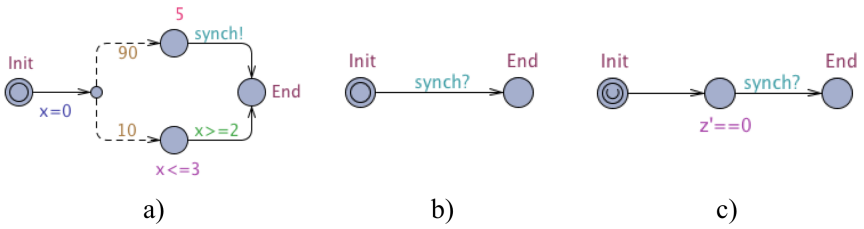


Figure 4.1: Example of NSTA in Uppaal SMC

Dashed edges outgoing from a branching node (see figure 4.1 a), have a probability weight (non-negative integer) and can admit synchronization and update fields. If p_i is the probability weight of an edge, this edge will be selected with probability:

$$\frac{p_i}{\sum_{j \in bpe} p_j}$$

where *bpe* represents all the branch point edges.
For property evaluation of NSTAs, an extended version of the *Metric Interval Temporal Logic (MITL)* [18] is adopted. MITL is defined as follows:

$$\phi ::= a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid O\phi \mid \phi_1 U^x_{\leq d} \phi_2$$

where a is an arbitrary atomic proposition over states of an NSTA, $d \in \mathbb{N}$ is a natural number and x is a clock variable. Besides the usual logical operators [166], MITL offers a *next-state operator* O and an extended bounded-until operator $\phi_1 U^x_{\leq d} \phi_2$ which allows to specify a clock to be considered in the evaluation. A bounded-until is satisfied by a run if ϕ_1 is satisfied on the run until ϕ_2 is satisfied and this happen before the value of the clock x exceeds d [89].

4.2 Query language

Uppaal SMC supports a set of queries for stochastic analysis, evaluated by incremental or sequential testing.

4.2.1 Bound and number of runs

All SMC queries are evaluated according to a simulation horizon, whose width depends on a *bound* variable. *bound* can be expressed by:

- a positive integer ($<= k$);
- specifying a threshold for a specific clock ($x <= k$);
- fixing a discrete number of steps ($\# <= k$).

The number of runs to infer a certain property can be optionally set within the query, through the parameter $\#run$, alongside the bound value: $(bound; \#run)$. When the number of runs is omitted, a default value applies which is 1 for a *simulation* query, and a dynamically adjusted value for *probability* queries where the number of simulations varies adaptively on the basis of the observed outcome.
The symbols [ ] and $< >$ can be used to denote which states should satisfy a property on a path: [ ] indicates all states on the path, while $< >$ means at least one state in the path.

4.2.2 Simulation

For preliminary/functional model assessment, e.g. for debugging purposes, a simulation run of the model can be asked, thus monitoring and having that UPPAAL SMC automatically plots, at query termination, the observed trajectories vs. time of specified expressions of interest:

```
simulate [bound] {E1, ..., Ek}
```

where *bound* is the simulation horizon and $E1, \ldots, Ek$ are the k expressions to be monitored.

4.2.3 Statistical algorithms

Since, in general, it is not decidable whether the probability emerging from one run of the model $\mathcal{M}$ satisfies a given threshold value, i.e. $P_M(\varphi) \geq p$ with $p \in [0, 1]$, UPPAAL SMC tries to approximate the answer by using simulation-based algorithms, by restricting the MITL cost-bounded in the form $P_M(\Diamond_{x \leq C} \phi)$, according to principles of *Weighted Metric Temporal Logic* (WMTL) [166], where x represents a clock variable and $C \in \mathbb{N}$ is a general bound.
Accordingly, the tool is able to perform the following kinds of analysis:

$$\begin{aligned} \textbf{Probability Evaluation}&: P_M(\Diamond_{x \leq C} \phi) \\ \textbf{Hypothesis Testing}&: P_M(\Diamond_{x \leq C} \phi) \geq p, \quad p \in [0, 1] \\ \textbf{Probability Comparison}&: P_M(\Diamond_{x \leq C} \phi_1) > P_M(\Diamond_{x \leq C} \phi_2) \end{aligned}$$

and to infer results by performing:

- Monte-Carlo simulation for answering a quantitative question;
- sequential hypothesis testing to infer qualitative questions.

4.2.4 Probability Estimation

To quantify the probability $P(\psi) = p$ of a path expression being *true* given that the predicate *psi* in probability brackets is *true*, UPPAAL SMC uses the probability estimation algorithm. The result is an estimated probability interval measure $[p-\epsilon, p+\epsilon]$, with a confidence $1-\alpha$. To perform this type of analysis, the query is:

```
Pr[bound] (<> psi)
```

where *bound* is the simulation horizon, *psi* the expression to be monitored and the <> symbol could possibly be replaced by [ ].
It seems that the tool derives a confidence interval by exploiting the Clopper-Pearson method [85] for a given α, performing a number of run N, evaluated according to the Chernoff-Hoeffding bound [203]. Runs are stopped when the confidence interval width becomes less than 2ϵ.

4.2.5 Hypothesis testing

By using the Wald's sequential hypothesis testing [255], UPPAAL SMC allows to estimate if an event occurrence probability exceeds a given threshold. The query is:

```
Pr[<= bound] (<> psi) >= p_0
```

where *bound* is the simulation horizon, p_0 is the comparison probability to test for and the <>, >= symbols can be replaced with [ ], <= respectively.
Such a query is more efficient than that of probability estimation as it is one sided and requires fewer simulations for the same significance level.

4.2.6 Probability comparison

Two probabilities of event occurrence can be compared, without estimating them, through a query like this:

```
Pr[<= bound] (<> psi_1) >= Pr[<= bound] (<> psi_2)
```

where *bound* is the simulation horizon, while psi_1 and psi_2 are two predicates over the state and the <>, >= symbols can be replaced by [ ] and <= respectively.

4.2.7 Value bound determination

To estimate the expected maximum or minimum value of an expression, by running a given number of simulations, the following query can be used:

```
E[<=bound; N] (min:expr)        E[<=bound; N] (max:expr)
```

4.2.8 Support for WMITL

Uppaal SMC also supports the checking of extended Weighted MITL (WMITL) properties.
The probability of satisfying a property ψ is estimated by the query:

```
Pr psi
```

where ψ corresponds to:

$$\begin{aligned}\psi ::= & \ BExpr \\ & \mid (\psi \ \&\& \ \psi) \mid (\psi \ || \ \psi) \\ & \mid (\psi \ U[a,b] \ \psi) \mid (\psi \ R[a,b] \ \psi) \\ & \mid (<> [a,b]\psi) \mid ([\ ][a,b]\psi)\end{aligned}$$

where $a, b \in \mathbb{N}$, $a \leq b$ and BExpr is a boolean expression over clocks, variables and locations.
In WMITL the operators U (until) and R (release) are bounded to arbitrary clocks. Informally:

- $psi_1 \ U^x_{[a,b]} \ psi_2$ is satisfied by a run if ψ_1 is satisfied in the run *until* ψ_2 is satisfied and this should happen before the value of the clock x exceeds b time units, and after a time units are elapsed;
- $\psi_1 \ R^x_{[a,b]} \ \psi_2 = \neg(\neg\psi_1 \ U^x_{[a,b]} \ \neg\psi_2)$ is the *release* operator (dual of U), indicating that ψ_2 must be true until both ψ_1 and ψ_2 are true and they should be true after a time units and before b time units. Alternatively ψ_2 is true from now until b time units have passed.

The following SMC query:

```
Pr(<>[low, upp] ([] [t1,t2] psi))
```

evaluates the probability that psi keeps true in the interval $[t1, t2]$, being true at some instant in the time-interval $[low, upp]$. Quantities low, upp, $t1$, $t2$ are integers.
Two different ways exist in Uppaal SMC for using WMTL. For the MTL fragment of WMTL, an MTL property can be passed directly to the engine [48]. For the full WMTL language, the formula can be converted to an *observer automaton* composed with the system [49]. The observer is guaranteed to reach a specific location if the property is satisfied and another one if it is not satisfied. Therefore, an optimised reachability engine of Uppaal SMC can be used to verify WMTL. However, a problem could arise: sometimes an exact observer cannot be constructed and only an over or under-approximation can be made [212].

4.2.9 Additional queries

Since Uppaal SMC also allows for the dynamic creation of automata [], other constructions are available to check properties for them.
The query

```
forall (i : T) (psi)
```

allows to verify if psi is true for all the dynamically created instances of a T-type template, while i may be used to refer, in psi, to a specific instance of T.
The construct

```
exists (i : T) (psi)
```

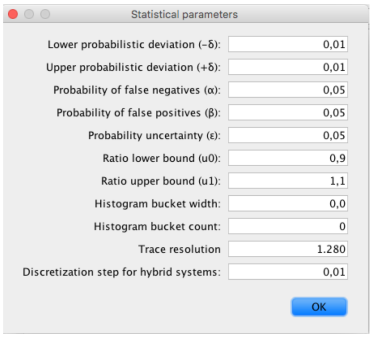
evaluates psi as true if some i-automaton of the T-type satisfies the property.
The expression

```
sum (i : T) (phi)
```

returns an integer equals to the sum of the expression phi evaluated with i ranging over the given type argument.

4.3 SMC options

The Uppaal SMC behavior during stochastic verification, can be configured by setting some global statistic parameters as follows:



Probabilistic Deviation:	lower($-\delta$), upper($+\delta$)
Probability of false results:	negatives(α), positives(β)
Probability Uncertainty:	ϵ
Ratio Bounds:	lower(u_0), upper(u_1)

Figure 4.2: SMC statistical parameters settings

whose meaning refers to the implementation of statistical algorithms discussed in the previous chapter. The *probabilistic deviation* values and *probability of false positives and negatives*, are used for the hypothesis testing, and indicates, respectively, the width of the indifference region and the probabilities that the alternative hypothesis is accepted by mistake. The *probability uncertainty* defines an interval around the determined probability value, in which the real value falls. Finally, the *lower* and *upper ratio bound* influence the probability comparison step, by providing bound values for the ratio of the two probabilities, to deduce a clear result statement [166].

4.4 Dynamic template processes

The standard Uppaal language allows to create *static models*, that are network of timed automata with a fixed size, instantiated when the overall system is bootstrapped. This is in contrast with the fact that multiprocessing systems are able to create threads and processed when needed, in order to let programs take advantage of concurrent execution *David: Quantified2014*.
Uppaal SMC enables *dynamic models*, through the use of *Dynamic Networks of*

Stochastic Hybrid Automata, that are Timed Automata extended with a *spawning* and a *tear-down* primitive. Any automata in the model can *spawn* instances of processes declared as *spawnable*, that operate as classical static templates, with the difference that at any moment they could end their execution and be removed from the model.
A template that will be dynamically spawned is defined the normal way in the Editor window, but must be declared with the keyword *dynamic* in the global declarations:

```
dynamic NameProcess(list parameters)
```

where the list of parameters include only by-value parameters or reference to a broadcast channel.
The spawn operation can be executed by a generator process, thus:

```
spawn NameProcess(parameters)
```

in the update attached to an edge. The spawned template can tear itself down, through an update which invokes the operation:

```
exit()
```

Figure 4.3 summarizes the mechanism of dynamic automata [90].

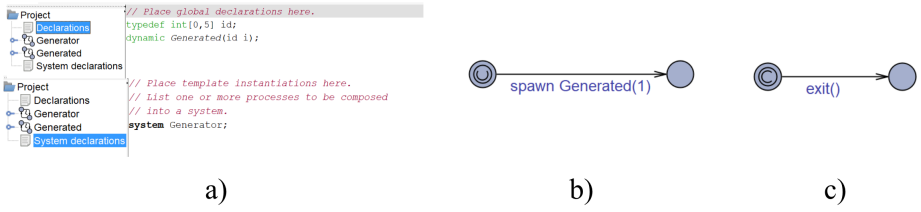


Figure 4.3: Dynamic spawning: a) global and system declaration b) Generator Template c) Generated Template

Although potentially useful, dynamic template processes can introduce an unacceptable performance cost in the evaluation of complex models.

4.5 Custom probability distribution functions

Stochastic behavior can be defined in a Uppaal SMC model by attaching to normal locations the rate of a negative exponential distribution function. Each time the model enters one such a location, a dwell-time is established by sampling the exponential function. However, modelling non-Markovian systems requires custom probability distribution functions to be managed. The following pattern, suggested at Fig. 22 of the Uppaal SMC tutorial, can be used. It is based on a function $f()$ which returns the next sample of a custom distribution function, a stopwatch d (duration), and a clock x used to measure the time elapsing in the $SWait$ (stochastic wait) location. $SWait$ is exited when x reaches the value of d, which purposely is kept frozen in the meanwhile. Although d could be replaced by a double variable, discretization and time resolution aspects make the solution based on the stopwatch d more efficient in the practical case.

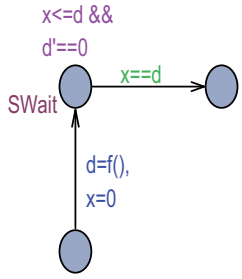


Figure 4.4: Emulating custom probability distribution functions

4.6 Non-deterministic vs. stochastic interpretation

To some extent, an Uppaal SMC model could preliminarily be investigated with the symbolic model checker. For this to be possible, the model has not to depend only on double variables. Indeed, doubles and probabilistic/stochastic aspects of the model are ignored by the symbolic model checker. Probabilistic choices are then replaced by non-deterministic choices. Although this represents a worst case scenario, because a high probable execution path is handled as any other alternative path, performing a non-deterministic analysis can allow to check safety (e.g., absence of deadlocks), (bounded) liveness and reachability properties on the model. In particular, the non-deterministic analysis can reveal something of interest (an event) can occur. The quantitative analysis carried by the statistical model checker then permits to estimate a probability measure for the event to actually occur. Therefore, when both kinds of analysis are possible, they can be exploited in a synergic way.

Part II

Distributed Probabilistic Timed Actors

Chapter 5

Actor-based Development of Distributed Probabilistic Timed Systems

The main goal of this work is to establish and to experiment with a methodology for the development of distributed, probabilistic, timed systems belonging to such application domains as cyber-physical systems, embedded real-time systems, wireless sensor networks, general IoT, multi-agent systems and so forth. The methodology aims at addressing all the phases of the system lifecycle, from modelling, to analysis and property checking (by exhaustive model checking and/or by simulations e.g. through a statistical model checker), down to design, prototyping, implementation and real-time execution.
The chosen approach is based on actors, that are encapsulated software entities which communicate to one another by asynchronous message passing. Many years of experience have witnessed actors have the great potential for modularizing complex and scalable distributed systems. A grand challenge has been that of adding and ensuring, in an effective way, timing to actors, and developing tools assisting modelling and analysis so as to favor a model synthesis into a programming language capable of guaranteeing "faithfulness" of an implementation to its specification. Different efforts are reported in the literature and are summarized in this chapter.

5.1 Untimed actors

Actors were introduced by Hewitt in [118, 119] as active *agents* for Artificial Intelligence, providing a fundamental concept combining *control* and *data*, on top of which to achieve program structures, by relying only on the notion of sending messages to actors. Actors were turned in a concurrent object-oriented language by Agha [10]. Actor computation was formalized in a functional context by Talcott et al. in [11, 174, 242] to serve as a basis for a theoretical understanding of concurrency. Actors were soon recognized as a modelling and implementation framework for general untimed distributed systems based on

asynchronous message passing.
Actors encapsulate an internal data status and expose a message interface. Receiving a message represents the basic action (see Fig. 5.1) by which an actor can change local data variables, create new actors, send messages to known actors (acquaintances), including itself.

In its basic form, the computational model of actors associates one *thread* of

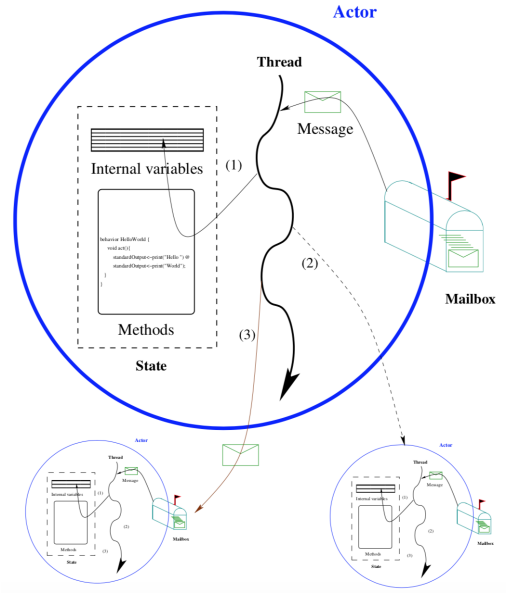


Figure 5.1: An actor responds to a message by (1) updating local variables (2) creating new actors, (3) sending messages to acquaintances (picture taken from [250])

control to each actor which is responsible for message processing. Messages directed to an actor get stored into an input *mailbox* owned by the actor, from where they are extracted, one at a time, and processed. When the mailbox is empty, the control thread pauses by yielding the processor.
Examples of common libraries and languages supporting actor programming include: ActorFoundy [21], Salsa [249, 250], Scala/Akka family [115, 202], Erlang [20].The Asynchronous Agents Library (AAL) is an actor-based framework that was added to Microsoft Visual Studio 2010. An example of a real application which was achieved by actor programming is the Twitter message queuing system [236].
Being based on asynchronous message passing and not on shared data, common problems and pitfalls of multi-threaded programming [159] , e.g., data races, misuses of locks, deadlocks and so forth, are avoided when programming with actors. However, correctness issues raised by actors and requiring suitable analysis tools [236] , are tied to non-deterministic message delivery (multiple messages sent to a same destination actor may be received in different orders), which is a problem similar to the basic thread non-deterministic action-interleaving which makes it hard to analyze and predict the behavior of a concurrent application.
The tool Basset [158] is an efficient and general framework which was developed for testing (by model checking and state space exploration) concurrent actor

programs developed according to different libraries and languages (specifically ActorFoundry and Scala) preliminarily translated into bytecode. Basset was achieved on top of Java PathFinder (JPF) and proved effective in detecting bugs (e.g., undeliverable messages, deadlocks and so on) in actor systems. In a case, it was capable of finding a bug into Scala. To improve space/time of state exploration, Basset could be adapted to reduce dynamically the partial-order (by pruning generated state space) in message order delivery by keeping a Lamport "happens-before" relationship among messages. In addition, states are compared for isomorphism (by exploiting the state comparison mechanism already implemented in JPF) so as to reuse states as they repeat.
The Jacco toolset [267] represents an improvement w.r.t. Basset, and deals with actor programs written in Java. It does not depend on JPF as the back-end model checker, but directly implements the model checking procedure. This way Jacco avoids the fine-grain interleaving at the low-level bytecode instructions, which is unrequired since the high abstraction level of actor programs which do not share data variables (Basset recognizes the macro-step semantics, namely executing atomically a message before proceeding with the next one, but the use of underlying JPF and fine-grain interleaving do not avoid the inefficiency). Jacco develops a new message scheduling approach which can adapt to standard libraries like Akka, and realizes a more efficient state saving mechanism during state exploration which can speedup model checking. The point of message scheduling is pragmatic: actual actor libraries ensure message order of messages by relying on TCP communications. As a consequence, if an actor A sends two consecutive messages m_1 and m_2 to B, B will receive first m_1 then m_2. But Basset using a pessimistic scheduling discipline, considers that m_1 and m_2 can be received in any order (non-determinism). Thus are possible message deliveries which can raise false negative problems, which in practice do not exist. Non-deterministic delivery is instead kept for messages sent by different actors to a same receiver actor. Due to the implemented improvements, Jacco was successfully applied to case studies where Basset proves inefficient or not applicable.

5.2 Timed actors

In our opinion, classical multi-threaded actors are best suited to the development of untimed concurrent/distributed software systems. A first attempt for specifying timing aspects in untimed actors was the concept of RT-Synchronizer [222, 185] [221, 184]. The idea was to separate concerns (see Fig. 5.2) of *functional behavior* (actors) from those of *temporal behavior*, through a modular specification of timing constraints existing in actor interactions. Concretely, an RT-Synchronizer was devised as a declarative composition software agent, which filters (by computational reflection) messages directed to actors. Messages get buffered in the RT-Synchronizer when they do not satisfy a timing constraint (specified by a constraint pattern); otherwise, they are transmitted to the relevant actors for them to be processed. It is the principle of *"safe progress; unsafe block"* in actor interactions. This way functional actors are not aware about timing issues and timing constraints are part of RT-Synchronizers, which in general can reflect on distinct assigned groups of application actors. RT-Synchronizer specifications were intended to be transparently weaved with regulated actors

so as to be observed during the runtime.

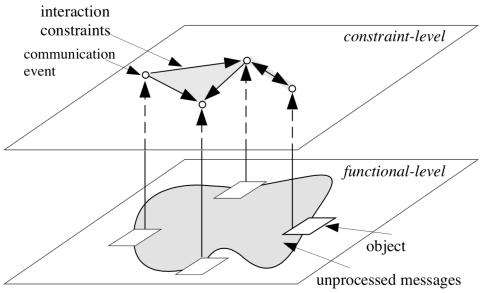


Figure 5.2: Separation of concerns among functional and temporal behavior operated by an RT-Synchronizer, taken from [184]

In [192] it was experimented a framework based on the RT-Synchronizer for the schedulability analysis of distributed real-time actor systems. The framework is based on modelling and analysis by Colored Petri Nets. The key point of the framework is the introduction of a control layer which reflects on timing and regulates message exchanges.
The idea of RT-Synchronizers was at the hearth of the design of a time warp algorithm for high-performance optimistic distributed simulations [32]. Pattern constraints were embedded into the regulating control structure which is transparent to application actors (Logical Processes-LPs). The realization was characterized by a particularly aggressive cancellation technique which is capable of stopping as early as possible an incorrect computation, by exploiting a coarse-grain notion of undoing messages, which proves effective in loosely coupled distributed contexts. The time warp schema was extended in [33] so as to support *temporal uncertainty* in distributed simulation models, whose exploitation can contribute to runtime speedup. The idea is that messages normally have a temporal interval of occurrence times and not just a punctual time. The interval can then be exploited, especially in network communication messages, to reduce the incidence of *stragglers*, i.e., messages whose occurrence time is already past and normally trigger rollbacks and recovery in time warp. The actor-based implementation in [33] rests on Actor Foundry as the underlying middleware for distributed services (e.g., naming). However, for efficiency in the network communications, a *direct network* concept was realized so as to directly exploit TCP sockets.
In an important case, the RT-Synchronizer concept was specialized as a QoS-Synchronizer [223, 252] for specifying and reasoning upon the *quality of service* (QoS) in multimedia applications, that is a weak real-time context. The typical scenario consists in a multimedia session composed by multiple independent (but time related) components (media) like audio and video. The goal of the synchronizer is to keep synchronized the (timed) messages (frames) of audio and video (transported, e.g., by the Real Time Protocol-RTP) so as to control the well-known *lip-synch* problem. When temporarily the synchronization is lost (because some video packets are missing) messages get buffered and the synchronization is restored at the next possible time, so as to guarantee an acceptable fruition of the multimedia session from the end user. A concrete realization based on the QoS-Synchronizer for the lip-synch problem was described

in [107]. In [103] the concept was experimented in the design and prototyping of real/virtual teleconferences.

5.3 Actor Extensions for Real Time Modelling and Analysis

Whereas a concurrent actor-based program focuses mainly on what components are to be present in the program to address its mission, the viewpoint significantly changes in a real-time program where not only *what* actions (functions) are required in the program but mostly *when* they are to be executed is the essence for correctness. In [183] a distributed real-time actor language was proposed which extends classical actors by defining *timing constraints* on *message interactions*. At each message send a couple of information are specified: a *release time* (r) and a *deadline* (d) which constrain the delivery of the message to its receiver. Such times are relatives to the invocation time of the message and express respectively the earliest time and the latest time for message delivery. The semantics of the language was provided by timed graphs, which are similar to timed automata. The goal was to provide semantics independently from the resources which will be used to support the execution of the real-time actor program. In particular, a specification would enable different implementations, as long as it can be quantitatively guaranteed that timing constraints are fulfilled. However, no tool for quantitative timing analysis nor how to ensure an implementation is effectively a refinement of its specification were addressed issues.
The research ideas in [183] are the logical basis upon which the modern and notable actor-based modelling language Probabilistic and Timed Rebeca (PTRebeca) [7, 129], together with supporting analysis tools, were achieved.
The basic untimed and imperative Rebeca (**Re**active O**bjec**ts L**a**nguage) [218, 237] modelling language remains in the tradition of classical threaded actors. An actor is modelled as a reactive class which encapsulates local data variables, including acquaintances. The behavioral part is expressed by a collection of *message server methods* (*msgsrv*), each one being dedicated to the processing of one expected message (see also [236]). Message servers are invoked asynchronously using a syntax similar to the object-oriented method invocation. In [237] a formal operational semantics for Rebeca is defined which provides a transition system for property analysis.
Rebeca is worth mentioning because it was the first attempt to add formal verification to an actor system with asynchronous message passing [236], using a temporal logic (e.g., LTL and CTL) for specifying the properties to check. Rebeca models are translated into the modelling language SMV [57] or Promela [240] and then formal properties checked using an existing model checker like NuSMV [57] and SPIN [240]. To make possible exhaustive model checking, Rebeca models were restricted to having: bounded queues (in the basic language, the inbox message queues of actors are unbounded), bounded data types, static configuration (no creation of rebecs allowed during the runtime). In addition, message parameters can be omitted by assuming, due to bounded data variables, the existence of multiple msgsrv methods each devoted to a particular data configuration. In [237] it was also advocated a different approach to formal verification of Rebeca models: designing a *front-end* tool capable of generating

the transition system corresponding to the operational semantics of a source model and to use it to special case the translation into the modelling language of a back-end model checker.
In [236] it is described RMC, a Rebeca Model Checker, which founds on the *Modere* (**Mod**el checking **e**ngine for **re**beca) tool which exploits the features of Rebeca (e.g., atomic method execution) to provide benefits in time and space for model checking Rebeca models. RMC enables property specification by LTL and CTL formulas. A *rebec manager* component of RMC is worth mentioning. It relies on a preliminary translation of a Rebeca source model in C++ code. During model checking, the rebec manager can be asked to put a rebec into a given state, to execute the rebec and returning the resulting state. In [236] specific techniques to improve state space exploration in Rebeca models are discussed such as partial-order reduction and symmetry.
A contribution brought by the work described in [237] [236] refers to experimenting with the challenging *compositional verification* problem [238], which amounts to decomposing a Rebeca model into components, verifying the components separately and then inferring system properties from those checked on the individual components. In formal terms, if the model is split into two components P and Q whose specifications are respectively φ_P and φ_Q, the goal is trying to support the rule:

$$\begin{aligned} P &\models \varphi_P \\ Q &\models \varphi_Q \\ \varphi_P \wedge \varphi_Q &\Rightarrow \varphi \\ \text{------} & \\ P \parallel Q &\models \varphi \end{aligned}$$

but, as observed in [237], the problem is, for example, that φ_P could not possibly hold in the composition $P \parallel Q$.
Composition verification aims at tackling the state explosion problem by reducing the state space by focusing on model and analysis of components. Modularization of a Rebeca program around encapsulated rebec units (actors), potentially helps to split a system into components. A whole Rebeca model is *closed*: it is constituted by a collection of rebecs interacting to one another by message exchanges. A component is an *open* subset of rebecs which need to interact with an *environment*, that is the remaining external rebecs. The environment requires a proper abstraction to allow verification of single components. In [237] the *composition minimization* technique is adopted, where a component Q is reduced to a version Q' which reproduces the "essential behavior" of Q as viewed by P. Q' is a reduced environment which improves the analysis process.
An interesting issue discussed in [236] concerns *task schedulability* in *timed actors*. In a Rebeca model, tasks are naturally associated with method (message) invocations and can have an execution time and a deadline. Ensuring, in certain applications, an acceptable level of QoS thus requires analyzing a Rebeca model properly abstracted with timing. Instead of concrete algorithms in msgsrv, the corresponding time passage can be modelled and replace implementation code. In addition, a deadline can be attached to messages at the sending time (see also later in this section). Studying schedulability problems also necessitates the definition of a custom scheduling strategy of message reception in rebecs. Whereas in standard untimed Rebeca, messages in a rebec are retrieved from

its inbox queue one at a time and in FIFO order (more precisely, the arrival order of messages in the queue), ensuring a good level of QoS needs a different scheduling discipline which e.g. recognizes priority or timing information. In [236] tasks (namely msgsrv) are modelled as Timed Automata and model checking tools like Uppaal [27] used for schedulability analysis.
The Rebeca modelling language was extended, in different steps, so as to making it more suited to general distributed and timed systems. First probabilistic aspects and timed aspects were separately added to Rebeca, and investigated, giving rise to Probabilistic Rebeca (PRebeca) [251] and Timed Rebeca (TRebeca) [224]. Probabilistic Timed Rebeca (PTRebeca) is the latest version which combines both probabilistic and timed aspects; it is the more powerful and expressive version, more challenging from the point of view of analysis. In the following, also considering its influence on the Theatre actor system which is the main focus of this dissertation, the PTRebeca modelling language will be summarized, together with its supporting analysis tools. PTRebeca favors a model-driven development methodology of distributed timed systems, based on formal modelling and formal verification tools.
PTRebeca maintains the syntax structure of a Rebeca model, which is composed of a given number of classes (rebec definitions) plus a *main* rebec for bootstrapping purposes. Only integer and boolean primitive data types are admitted. Elementary statements within the body of a msgsrv include the assignment $v = e$, the $if - else$ (loop constructs are avoided) and the method invocation (*call*) which has an asynchronous semantics. New instructions specific of PTRebeca are the *non-deterministic assignment*:

$$v = ?(e_1, e_2, \ldots, e_n)$$

which assigns to v the value of an expression e_i, $1 \leq i \leq n$, chosen non-deterministically; and the *probabilistic assignment*:

$$v = ?(p_1 : e_1, p_2 : e_2, \ldots, p_n : e_n)$$

where p_i, $1 \leq i \leq n$, are *probabilistic weights*, $\sum_{i=1}^{n} p_i = 1$. The value of expression e_i is chosen for the assignment to v with probability p_i. Non-deterministic and probabilistic assignments can be exploited for choosing a time value in a *call* or *delay* statement, as explained in the following, thus making the behavior of a PTRebeca model probabilistic *and* timed.

Sending a message to a known rebec (acquaintance) kr (the identity of the *sender* rebec is implicitly transmitted with the message; the noun $self$ identifies the currently executing rebec) is realized by the extended *call* instruction:

$$kr.message\_name([args])[after(a)][deadline(d)]$$

As one can see two possible times can be attached to each message invocation: an $after$ time (whose amount is a) and a *deadline* time (whose value is d). Both are relative times with respect to the instant in time the message was sent. The meaning is that the message cannot be delivered to its destination before $after$ time units are elapsed, and should be delivered within *deadline* time units. Failing to dispatch the message before or at the *deadline*, causes the message to be discarded. The timing information are optional. A missing

$after$ evaluates to 0; a missing *deadline* evaluates to ∞. A further instruction admitted by PTRebeca is the *delay*:

$$delay(d)$$

which blocks the execution of the msgsrv body for d time units (duration). The instruction is useful for expressing the duration of a code segment.
As a consequence of the timing model of PTRebeca, a major departure occurs with respect to the semantic model of Rebeca. Whereas in Rebeca a msgsrv body is always atomic and does not permit suspension, a msgsrv in a PTRebeca model can introduce suspension due to the use of delay statements.
In [129] a *structural operational semantics* (SOS) of a PTRebeca model is given using production rules in the style of Plotkin [210] and [132] (for an example of an SOS description, see later in this thesis), extended with a representation of the discrete probability distribution whose values tag each possible reachable state. The semantic rules capture basic *scheduler* message dispatching, *msgsrv body execution* with detailed semantics of each component statement. A particular rule concerns the *time-progress*, which makes an advancement of the time when no event (either an eligible statement in a msgsrv or a scheduled message in the bag of already sent but not yet dispatched messages) can occur at current time. The semantic description saves the program counter of a rebec msgsrv so as to enable the scheduler to evaluate if the time is arrived for a suspended rebec to continue its execution.
The overall SOS of a PTRebeca model establishes a transition system in the form of a Timed Markov Decision Process (TMDP) [129]. Transitions in a TMDP are realized into two steps: starting from a given state, in the first step a non-deterministic choice (of an action or delay transition) is carried out, followed, in the second step, by a transition to a next state according to a discrete probability distribution.
The (hopefully finite) TMDP of a PTRebeca model is automatically built by the *Afra* tool [218] [129]. The resultant TMDP provides the semantics of a probabilistic timed automaton PTA) with one digital clock (see below). The output file of Afra can then be translated into the terms of the modelling language of a back-end probabilistic model checker like PRISM [144] or, more recently, IMCA (Interactive Markov Chain Analyzer) [112]. For efficiency of the model checking process, the resultant final model does not rest on the parallel composition of multiple processes (one process can be associated to each distinct rebec) but generates one single process (module) whose state transitions are directly those of the TMDP transition system. Being a PTRebeca model discrete, the final translated model makes use of *rewards* and *digital clocks* (a digital clock is an integer variable which gets incremented when, e.g., a certain state transition occurs). Therefore, using in a case the PCTL temporal logic supported by PRISM, properties can be checked related to (*maximum/minimum*) *expected time reachability* and *expected reward reachability*. Actually, the IMCA translation proves more efficient than that of PRISM but, as noted in [129], there are still problems in verifying timing properties with the IMCA.
A TMDP is an example of a timed transition system (TTS) which (implicitly) is based on global time, uniformly viewed by all the rebecs. Global time and arbitrary interleaving at the instruction level of the message server methods, cause in general the state space to become very large thus complicating model checking. In [136] the notion of a Floating Time Transition System (FTTS) is

proposed for Timed Rebeca (TRebeca) (in a TRebeca model all the instructions of PTRebeca are allowed except the non-deterministic and the probabilistic assignments) which exploits intrinsic features of the actor model. Since actors are single threaded encapsulated concurrent units (they share no data), with non-blocking send and receive operations, and whose msgsrvs are not preemptive, there is no need to model arbitrary interleaving among the instructions of the message servers. In addition, each rebec can maintain a local time notion (clock) which is advanced by the timed instructions (i.e., *delay* statements) during a msgsrv execution. Rebec clocks can thus be widely apart each other. Transitions in FTTS are associated to a *complete and atomic* execution of a message server. However, as shown in [136], even under FTTS semantics, the state space can still be large. As a consequence, a bounded FTTS (BFTTS) is proposed which significantly can reduce the state space by *reusing* states. More in particular, an equivalence relation among states is defined which is based on a *time shift* operation. Two states are equivalent if they contain the same data variable values and if *all* the timing information of the timed messages in the bag of one state differ of a same time shift t (a natural number) from the identical messages in the bag of the other state. As a consequence, one state s' which is time shifted of t with respect to a state s, can be reduced to s, thus shortening the state space. The theory behind BFTTS owns to Lamport's logical time. When a timed message $tmsg$ is sent from a rebec A to a rebec B, the $after$ and *deadline* (relative) times of $tmsg$ are "*absolutized*" according to the clock of A. Let these values are ar (arrival time) and dl (deadline). Messages in the bag of B are said to be *enabled* if they have the same and smallest ar time. In the case multiple messages have the same ar, one of them is chosen non-deterministically, is extracted from the bag and atomically executed. Before execution, the clock value of B is updated thus: $nowB = max(nowB, tmsg.ar)$. Recall also that the execution of the message server can augment the $nowB$ according to the duration of a *delay* statement. It can be shown that BFTTS is bisimilar to FTTS. In addition, although in the presence of not aligned rebec clocks, the model event ordering is preserved.
Under BFTTS a TRebeca model can be verified for *deadlock freedom* and *message schedulability*. The absence of deadlocks amounts to checking that no state in the state graph exists which has no outgoing transitions. Schedulability analysis means assessing, for each rebec r, that in no case for an enabled message $tmsg$, it happens that $tmsg.dl > now_r$ (deadline miss).
Although BFTTS enables timed schedulability analysis and deadlock freedom, the use of unsynchronized clocks does not permit to check more general timing properties, for example, determining the worst case of an end-to-end delay (EED) between a stimulus and the corresponding response provided by the model. For these cases, the use of global time can be recommended.

5.4 The Theatre infrastructure

5.4.1 Basic Concepts

Theatre (see, e.g., [192, 106, 63, 65, 64]) is an adaptive software framework and a methodology whose aim is supporting the development of distributed probabilistic timed actor systems. It is actually implemented in Java. Other languages

are possible. THEATRE represents a variant of the classical Agha actors model [10], specifically tailored to *predictable* time-sensitive applications. THEATRE actors are light-weight, thread-less encapsulated software entities, which do not have a local mailbox for incoming messages. Rather, THEATRE actors are truly reactive: they are programmed to provide reactions to each received message. An actor is at rest until a message arrives. Message processing is atomic and cannot be suspended nor preempted.
A fundamental design aspect of THEATRE concerns the *separation* of *concerns* between the *application logic* (*business*) and a *reflective control layer* which, transparently, intercepts the exchanged messages and schedules (stores) and dispatch them according to a customizable *control structure*. The control layer is responsible for time management (real-time or simulated time) and of the enforcement of the timing constraints. It logically follows the RT-Synchronizer concept [222].
A distributed THEATRE system consists of a federation (see Fig. 5.3) of computing nodes (theatres). Each theatre hosts a *subsystem of local actors*, a *control machine* and a *transport layer*. The transport layer allows messages to be communicated from a theatre to another through a network. A universal global naming for theatres and actors is assumed. The name of an actor (string) is unique and system-wide. THEATRE is based on *global time*. The control machines of the various component theatres of a system, coordinate each other, through the mediation of a Time Server, in order to keep synchronized the local time notion of each theatre with global time.
Two levels of concurrency exist in a Theatre system: the *intra-theatre cooperative concurrency*, which depends on message interleaving as ensured by the local control machine, and the *inter-theatre concurrency* which can be truly parallelism (e.g. by mapping theatres onto JVMs which execute on separate cores). It is worth noting that control machine operation obeys to the *macro-step semantics* [134]: at each moment, only one message can be under processing in a theatre. At message processing termination, the control machine proceeds with selecting and dispatching the next message and so forth.
The light-weight control-based design of Theatre actors can be related to the Ptolemy modelling and analysis framework and toolbox [43]. Ptolemy supports different models of computation in actor-based applications. In a case, Ptolemy exposes a not thread-based notion of actors where the exchange of messages is asynchronous and where a *Director* (a concept similar to the control machine of Theatre) can supervise the actual delivery of messages to actors. The Ptolemy toolbox is characterized by its extensive set of tools for the analysis and synthesis of a modelled system.

5.4.2 Programming in-the-small concepts

THEATRE is characterized by its *open customizable design*: it can be adapted not only in the reflective control structure, but also in the timing model and behavioral programming style of actors. As a common design guideline, many previous THEATRE applications used actors modelled as *finite state machines* (FSM). In particular, the behavior of an actor was captured into an *handler(message)* method which receives the message dispatched by the control machine, and decides the reaction to perform (e.g., establishing the next state in the FSM) on the basic of the current status and the identity of arrived message (event). In

[104] the actor behavior was modelled as a *statecharts* for real-time systems. The control machine directly founds on the RT-Synchronizer and its declarative programming style. Timing constraints affecting message interactions are derived by temporal activity diagrams (TAD) which capture causal effect relationships on message interactions and associated timing constraints. Validation of timing constraints relies on simulation. Statechart-based actors were also used in [66] for modelling and performance evaluation of complex manufacturing systems.

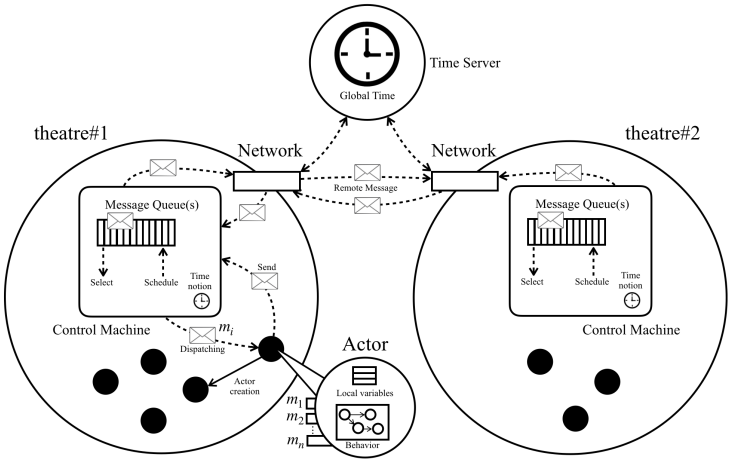


Figure 5.3: A distributed Theatre system

The programming model of actors was enhanced in [66, 73] by the concepts of *actions*, i.e. independent programming units. The handler(message) methods can spawn actions, which have a *duration*, an *input-parameter-list* and an *output-parameter-list*. Actions have no visibility to the internal data of the spawning actor. As a consequence, (a) no interference can occur from action execution on the actor data status, (b) actions can be naturally executed in parallel, would sufficient resources be available. Actions and control machines can be differently reified when moving a model along the system development lifecycle (see also below).

5.4.3 Programming in-the-large concepts

The Theatre actor framework was experimented with different transport layers and middleware, which provide services for inter-theatre message communication and actor migration. In [106] the socket API of Java were used to prototype Theatre configuration in general distributed Internet-based applications. In [33] ActorFoundry was used as a middleware for distributed simulations using the TUTW optimistic time warp mechanism. In [64, 65] the distribution services of the High-Level Architecture (HLA) were used as a basis for Theatre-based distributed simulations. The APIs of HLA were exploited for time coordination among the various theatres/federates. Also HLA supported an original development of Theatre actors for distributing Repast models [61] for high-performance simulations, using a conservative time synchronization technique. In [62] Terracotta [246] which provides the illusion of a shared heap memory

(Net Attached heap) among distributed programs, was used for supporting actor interactions over a network. In [67] an experience porting Theatre over the peer-to-peer architecture of the Globus Toolkit 4 is described. A special design and implementation in Java of the Theatre transport layer was described in [71] which refers to multi-core clusters. The realization proved very effective for high-performance distributed simulations. Recently, the JADE agent-based framework [29] and its FIPA [102] adherence (e.g., the Agent Communication Language ACL) were used as a middleware for prototyping Theatre with actions [66, 73]. The realization enabled distributed simulations over JADE which has no primitive support for discrete-event simulation.

5.4.4 Simulation applications

Previous applications of Theatre were mainly directed to modelling and analysis of complex systems through simulation and particularly distributed simulation. Some significant experiences are reported in [64, 71, 61, 69]. Theatre actors were also used successfully in modelling and analysis of large multi-agent systems [65]. In [74] Theatre was exploited for modelling and performance prediction of an original variant of the Minority Game, the Dynamic Sociality Minority Game (DSMG), where a player can interact with a group of partners whose composition and identity can vary dynamically.

5.4.5 Development methodology and model-continuity

A key factor of Theatre rests in its *control-based* character. A family of recurring control machines was prototyped which includes standalone and distributed versions of the control forms either based on real-time or simulated time. Such a family was significantly extended with new members as part of this thesis work. The possibility of changing the control machine of theatres (as a plug-in component) is at the heart of what in the literature is often referred to as *model continuity* (see, e.g., [124, 260]). Model continuity can be related to the *faithfulness* requirement which Marjan Sirjani (which develops with her team the Rebeca formal modelling language) advocated in [235]. Guaranteeing that an implementation is faithful to its analyzed model is a well-known difficult issue, especially when the modelling language (e.g., a Petri net) is far (semantically) from the implementation language. Model continuity means transitioning a *same* model from the early analysis phase down to design, prototyping, implementation and final real-time execution phases. Theatre *favors* model continuity by enabling a development lifecycle which transforms in a natural way the initial analyzed model. Assuming that the modeler has created the "right" model for her/his system (this is obviously a fundamental aspect [163]), in terms of Theatre the model is an abstraction where functions and timing behavior are taken into account. Timing aspects replace concrete code and network communication/propagation issues. But even with this abstraction, a *closed* Theatre model (i.e., one that includes modelling of the influencing environment) is articulated in terms of actors and message interactions among actors. Upon this abstract model, property analysis is responsible for assessing functional and temporal behavior. The analysis phase of a Theatre model can be based on simulation or, better (when it is possible) by symbolic exhaustive model check-

ing. An analyzed model can then be transformed for preliminary execution. A preliminary control machine uses the real-time but message processing is still abstract as in simulation. The goal of a preliminary execution is to check the overhead of message exchanges (during analysis, message scheduling, dispatching and processing are instantaneous) and how they affect the timing behavior. Preliminary execution can be conducted on a standalone or distributed setting. In the latter case some tool for keeping real-time clocks aligned has to be used. In [73, 79] some experience was carried out using, for clock alignment of Windows machines, the Dimension 4 software [97]. Therefore, in preliminary execution, the model is that of analysis, only the control machine was changed. Moving a model toward final implementation means introducing concrete code to implement message reactions. The control machine still is based on real-time. Of course, code implementation can introduce errors (although the code segment in a message reaction normally is very simple, for example it does not use loop constructs) but in Theatre the model structure of actors and message exchanges remains unchanged.
It is our opinion that THEATRE *can* help producing the final implementation of a system in a "faithful" way to the analysis phase, by transitioning, without distortions, the *same* model along the system lifecycle.

5.4.6 Implementation status

THEATRE concepts were first prototyped in Java in [106]. Such implementation was completely redesigned and reworked in this thesis for supporting specifically a new version of Theatre customized according to the timing model and programming style of PTRebeca [129]. The new implementation (described in [79]) is characterized by:

- xml scripting and network socket parsing and automated configuration of a THEATRE system;
- minimal use of threads. One thread serves the operation of the control machine. Other threads are associated with input/output operation of socket connections;
- adoption of lock-free data structures in a theatre for transferring external incoming/outgoing messages to/from the control machine without blocking;
- support of actor migration through a customized serialization mechanism;
- extended library of control machines which fully supports model continuity. The control machines cover both analysis (by simulation), prototyping (by preliminary execution on the target architecture) and final implementation (with real-time execution).

5.4.7 Contributions of this dissertation

Starting from previous work on THEATRE carried out in the Software Engineering Laboratory, this thesis first has continued experimentation with THEATRE with actions on top of JADE, particularly for developing a cyber-physical system

devoted to power management in a smart micro-grid [78], as well as prototyping distributed measurement systems for real-time control of sensor networks [92, 53]. This experience suggested the need to improve predictability of Theatre distributed software systems by replacing the JADE middleware with a more efficient and native Java implementation of Theatre (also considering the very preliminary work described in [106]. In addition, the attractive and lucid on-going work about formal modelling and verification of distributed actor systems based on the Rebeca language [218] and more exactly on its probabilistic and timed extension PTRebeca [129], has inspired a customization of Theatre centred on the timing model and programming style of PTRebeca which structures an actor class as a collection of message servers (msgsrv) each dedicated to processing the reaction to a distinct expected message. The new version of Theatre was then formally described as an abstract modelling language for distributed real-time systems and its formal semantics (transition system) specified through a structural operational semantics (SOS) definition [196]. The SOS semantics was systematically used to design an original reduction of the extended Theatre on top of Uppaal model checkers, thus enabling formal verification of Theatre models. All of this improved the analysis phase of a Theatre system model which previously was based only on simulation, by non-deterministic exhaustive model checking and/or statistical model checking. Another contribution is tied to an original implementation in Java [79] of the new version of Theatre, together with the realization of a new library of control machines plus the envGateway component [78] which interfaces a Theatre system with peripheral devices (e.g., sensors and actuators) of e.g. a cyber-physical system. Theatre was successfully experimented in the development of several applications ranging from:

- modelling and formal analysis of
  - emerging properties in general multi-agent systems [189];
  - the time synchronization problem in complex wireless sensor networks [197];
  - web-based and network-based protocols [193, 187];
- concrete prototyping systems in the areas of
  - structural health monitoring [147];
  - management of body sensor networks for health monitoring [92]

Furthermore, during the work of this thesis, some other formal language was also considered, and some tool achieved, for modelling and analysis of timed systems, like stochastic Time Petri nets [76, 191, 190].

Part III

Theatre in Action

Chapter 6

Model Continuity in Cyber-Physical Systems: a control-centered methodology based on agents [1]

A Cyber-Physical System (CPS) is given by the integration of cyber and physical components, usually with feedback loops, where physical processes affect computations and vice versa. Design and implementation of complex CPSs is a multidisciplinary and demanding task. Challenges arise especially for the exploitation of heterogeneous and different models during the various phases of the system life cycle. This paper proposes an agent-based and control-centric methodology which is well suited for the development of complex CPSs. The approach is novel and supports *model continuity* which enables the use of a unique model along all the development stages of a system ranging from analysis, by simulation, down to real-time implementation and execution. In the paper, basic concepts of the methodology are provided together with implementation details. Effectivenesses of the approach is demonstrated through a case study concerning a prototyped CPS devoted to the optimization of power consumption in a smart micro-grid automation system.

6.1 Introduction

Cyber-physical systems (CPSs) [160, 137, 161, 228] integrate a physical system with a computational part through a network infrastructure. Their exploitation is advocated in various domains including avionics, automotive, traffic management, health care system, mobile communications, medical technology, manufacturing, smart grid, procurement and logistics, industry and building automation, plant construction and engineering [45]. A correct design for CPSs is of

[1] The material in this chapter is related to publications [78, 76, 53, 52, 92]

great importance as they are often applied in safety or business-critical contexts [135].

CPS development challenges arise from the necessity of adopting powerful software engineering methods for the cyber part, capable of ensuring modularity and evolution of a software architecture, while at the same time guaranteeing an effective control of the runtime platform and communication network for the fulfillment of the physical plant real-time constraints. Design difficulties [130, 135, 98] are related, for instance, to the needs of conjoining continuous dynamics of the physical components with the discrete-time model of the cyber components. In addition, the use of open and public networks requires the handling of security concerns [50] arising from the real-time operation of a CPS.

Architectural means for CPS modelling are described, for instance, in [243], where the use of agents [259] and their interactions (events) to one another and with the external controlled environment are the basic concepts. The adoption of crosscutting agent coordination policies at both the local and the global/system level emerged as a fundamental way to control the achievement of system goals. Multi-agent systems have demonstrated their advantages as an open and flexible software technology capable of unifying control aspects in smart grid applications [225]. As an example, agents were used to handle the power management problem in a smart home automation system [225]. Holonic agents, instead, are used in [254] as basic architectural building blocks for the development of manufacturing automation systems.

In this work an original agent-based control framework [72, 73] is advocated for CPSs, which rests on mechanisms for managing control and coordination aspects of agents as in [243]. Managing control aspects means that the approach makes it possible to use, in a transparent way, different message scheduling and dispatching policies according to a chosen time notion (real or virtual) so as to fulfill specific application requirements. The control framework acts as an *operating software* solution that integrates both flexibility of an agent-based design [225] with time-sensitive control structures which coordinate agents' evolution. A unique feature of the adopted framework, not supported by other existing agent-based approaches for CPSs, is *model continuity* [124, 260], which consists in the possibility of transitioning unaltered an agent model throughout the entire development life cycle, from analysis, down to design, implementation and real-time execution. The approach provides also a concurrency model which favours predictability and determinacy by avoiding common pitfalls of multi-threaded programming [159] (see section 6.3.2).

With respect to other approaches supporting model continuity [124, 260], the proposed framework distinguishes by its abstraction mechanisms which enhances separation-of-concerns during the development of CPSs. In particular, the following abstraction entities can be exploited: (i) agents to structure the *business logic* of the application to realize, (ii) *boundary elements* to interface the application with the external physical environment, (iii) the environment Gateway (*envGateway*) taking into account aspects related to modelling, analysis and implementation of the physical part of a CPS and more in general of the external environment in which an application runs, and (iv) customisable time-sensitive *control structures* suited to scheduling and dispatching system events and message exchanges. Model continuity depends on different concretizations of the boundary elements, the envGateway and the control-specific components. The envGateway requires to be re-interpreted when moving from the analysis to

the implementation phase. It offers a transparent yet uniform way for dealing with communication protocols and hardware equipments needed for sensing and acting upon a controlled environment. During system analysis, besides the modelling of single sensors and actuators, the envGateway takes also into account the causal-effect relations tied to operations carried out on the environment. As an example, turning on a lamp through a relay implicitly affects the value read by a luminosity sensor, placed near the lamp itself.

During the simulation phase, the envGateway can also interface software components like ordinary differential equations (ODEs), modelling continuous-time behavior of a system plant. From this point of view, the proposed approach can integrate continuos models within an overall discrete-event based framework. As an example, such techniques as quantization [34, 138], experimented, e.g., in the DEVS community [268], can be used.

The above mentioned agents and control framework is tailored to CPSs and the focus will be on proposing a methodology which addresses all the development stages of a system.

6.2 Related Work

CPS engineering challenges include the use of integrated models, facing issues related to interoperability, reconciliation of Newtonian time of the physical part with the discrete time of the cyber part [98], privacy protection, security, non-functional requirements, timing constraints, humans-system cooperation and so forth [45].

Service-oriented architectures (SOA) and multi-agent systems (MAS) are two important software technologies which have proved their effectiveness in general ICT systems and whose exploitation for CPS is deemed promising to sustain a revolution in industry automation and smart factories [145, 121, 227, 93, 168, 256].

A service-based approach for developing CPS is proposed in [145] which exploits service-oriented architecture concepts and/or cloud concepts to realize service-based CPS. The approach deals with some design challenges of CPSs such as dynamic composition, dynamic adaptation, and high confidence CPS management, hardware heterogeneity. Three tiers were defined: an *Environmental Tier* for dealing with the target physical environment, a *Control Tier* for making decisions for networked physical devices, and a *Service Tier* for managing reusable services. The final goal is that of allowing the handling of complex and resource-consuming processes even on downsized mobile Internet devices which are usually involved in a CPS.

Another service-based approach is discussed in [121] where the WebMed middleware is proposed. The goal is promoting the use of the service metaphor for the development of CPS applications. By exploiting the service-oriented computing, WebMed fosters the realization of loosely coupled CPS infrastructures that expose the functionality of physical devices as Web services. Exposed functionalities can be easily integrated with other existing software components. The middleware consists of: (i) a *WebMed device adapter*, aiming at hiding heterogeneity related to the use of specific hardware, data structures and communication protocols; (ii) a *Web service enabler*, which provides a mechanism for the data and functionalities of a physical device to become accessible as a

Web service; (iii) a *service repository*; (iv) an *engine*, which is the core element providing a runtime environment for all Web services and operations in the middleware; and (v) an *application development* tool providing high-level management of interaction and composition of Web service components in the middleware. The latter serves as user interfaces for developers, and as front-end in order to invoke a developed Web service.

In [168] MAS and SOA are identified as strategic technologies for CPS development and industry automation [254]. Agents contribution mainly derives from being decentralized, autonomous and modular entities, encapsulating data and "intelligence", and interacting to one another (for sociality and holonic aspects [254]) for the fulfillment of goals which could not be reached by each agent operating in isolation (property emergence at the society/population level). Other relevant agent features include robustness, flexibility, learning and adaptation, and self re-configurability. The usage of MAS, though, can be critical from the timeliness point of view. Therefore in [168, 254] the notion of an "industrial agent" is envisioned where an agent is paired with a Programmable Logic Controller (PLC) for low-level control and responsiveness, while ensuring, at the higher level, intelligence and adaptation. Services are purposely combined with agents in [168], by abstracting and exposing agent functionalities and low-level control through services, to favor in-the-large interoperability, modularity and composability. In [254] holonic agents interact and coordinate each other by FIPA [3] inspired mechanisms. Various kinds of simulators, already existing or especially developed for specific needs, are used to validate a control solution.

A methodological approach based on MAS is proposed in [93] for the analysis and prototyping of CPS. The analysis phase is directed to MAS simulation and CPS validation. However, that paper mainly focuses on the system requirements elicitation, e.g., sensor measurements and effector actions, using a specialization of SysML profile, and the assignment of requirements to behaviors of organizations which finally map on agents. The identification of organizations is helped by a problem ontology which describes all the concepts involved in a CPS and their relationships. The proposed methodology appears at a preliminary stage and has to demonstrate its effectiveness in the design of real systems.

Another agent-based approach aiming at developing CPS is proposed in [227]. A goal of the approach is that of trying to assess system behavior. Both qualitative and quantitative system property evaluation is considered. The quantitative evaluation, which is based on the exploitation of the INGENIAS methodology [109], is carried out by using a multi-agent model that supports event-driven behaviors. In the paper, the multi-agent approach is considered as the proper one to model a CPS with dependability features. This is due to the flexibility provided by agents as autonomous and intelligent components in decisions support actions. Raw data-streams, collected by various devices like sensors, video cameras, mobile phones, and measuring devices, are transmitted to the cyber components which, by using hardware and software facilities as well as communication connections, can provide several main functions as learning and adapting for intelligent control, self-maintenance, and self-organization.

An agent-based framework for smart factory is proposed in [256]. The framework consists of four layers, namely *physical resource layer*, *industrial network layer*, *cloud layer*, and *supervisory control terminal layer*. The physical resources are implemented as smart things which communicate each other through the industrial network. The integrated information system exists in the cloud which

collects massive data from the physical resource layer and interacts with people through supervisory control terminals. All of this, actually forms a CPS where physical objects and informational entities are deeply integrated. Furthermore, a negotiation mechanism for agents to cooperate each other is proposed, and four complementary strategies are designed to prevent deadlocks by improving the agents decision making and the coordinators behaviour. Properties of the proposed framework are assessed through simulation. A simulation program has been developed by using the Microsoft VS integrated development environment (IDE).

In [135, 130] it is argued that the design of CPSs strongly requires both an integrated view and co-designing of the physical and the computational part of a whole system. Authors of [135] suggest a model-based view to cope with design aspects of the hardware and software components, and their interactions. The goal is to derive relationships among the design, analysis and implementation models so as to ensure, for example, that analysis results are reflected in the executable system. What is envisioned is an incremental, simulation-based development approach, where initially the whole system is simulated and, subsequently, simulated parts are replaced by real ones.

To summarize, CPS development is currently trying to exploit powerful methodologies capable of systematically addressing all the CPS design issues. Anyway, the research about models, approaches, middleware architectures or platforms for realising CPS applications is still in its infancy [145, 121, 135].

This paper claims that *model continuity* [73, 124, 260] and MAS technology are fundamental tools for CPS development. Model continuity naturally addresses the CPS requirement of *model integration* [135], that is the need of ensuring coherence between the properties assessed during model analysis with the properties exhibited by a system during its execution. However, no experimented methodology based on the concepts of model continuity is actually exploited for CPSs. The contribution of this paper is to propose a novel approach based on MAS and model continuity and to demonstrate its practical usefulness through the realization of a real CPS case study.

The proposed approach mainly focusses on methodological issues embedded within a discrete-event framework. The approach, though, can integrate continuous time (CT) components and co-simulation activities as permitted, e.g., by well-known frameworks and toolboxes supporting CPS modelling and analysis like DEVS [268] and PTolemy II [98]. Both DEVS and Ptolemy enable the construction of hierarchical complex models. In addition, Ptolemy too rests on actors as the basic building blocks. The model of computation of a non atomic actor model can be defined so as to work with either synchronous or asynchronous interactions. Code generators are finally in charge of transforming an analyzed model into a final implementation. DEVS builds on a discrete-event world vision and has an efficient and modular simulation structure, which is open to interact with CT components. The DEVS community has experimented with such techniques as quantization [34, 138] for integrating CT components with discrete-event operation. With respect to CT components, which can be required during system analysis, the approach developed in this paper is able to exploit the same concepts and techniques of DEVS.

6.3 From Modelling to Implementation of a CPS

The life cycle of a CPS can be viewed as composed of different *transition phases*. First a model of the system is built. The model is then used to analyse its functional and temporal behaviour both in a simulation context and in a real execution environment. Thereafter, an analysed system can be put into operation.

In this section, a methodology is proposed which is based on the agent and control framework introduced in [72, 73] and here specialized for its use with CPSs. The approach relies on *pure-software components*, which *remain unchanged* during the transition from model analysis to system implementation, and on *hybrid components* which require to be concretized for actual system implementation. The methodology furnishes also entities suitable to model *external resources* needed by the system, and to capture and abstract the interactions between the system and the *external environment* the system operates in. In the following, the methodology is discussed together with its related entities. Current version of the control framework is prototyped on top of the JADE (Java Agent DEvelopment framework) open source FIPA compliant project [29], which provides a distributed architecture supporting agent naming, creation, execution, message passing, behaviour and mobility.

6.3.1 The proposed methodology

The development of a CPS follows four main phases, namely *modelling*, *analysis* (e.g., by distributed simulation), *preliminary execution* and *real execution*, which are described below.

The modelling phase

A model is built in terms of the following basic abstractions: *actors* (or agents), *messages*, *actions*, *processing units* and the *environmental gateway* (*envGateway*).

Actors and *messages* are pure-software components which capture the business logic of a model. Actors are thread-less agents whose behavior is patterned by a finite state machine, and whose communication model depends on asynchronous message passing (see Figure 6.1). Message processing is atomic and consumes a negligible time. A *time-stamp* can be attached to a message to specify when it has to be consigned to its recipient. If the time-stamp is not specified the message has to be delivered at the current time.

Actions are self-contained computational entities which are submitted for execution by actors. Following its submission, an action can run to completion or it can be suspended/resumed or aborted [72]. Actions are hybrid-components modelling time-consuming tasks which require external entities not owned by actors (e.g., a document to be printed requires a printer to print it). Actions are executed on top of *processing units* which also are hybrid components. An action is ready to be executed as soon as it has been submitted. However, the available processing units actually determine *if* and *when* a submitted action is actually executed.

The *envGateway* is a hybrid component devoted to (i) modelling the external environment the actor-based application runs in, (ii) abstracting the inter-

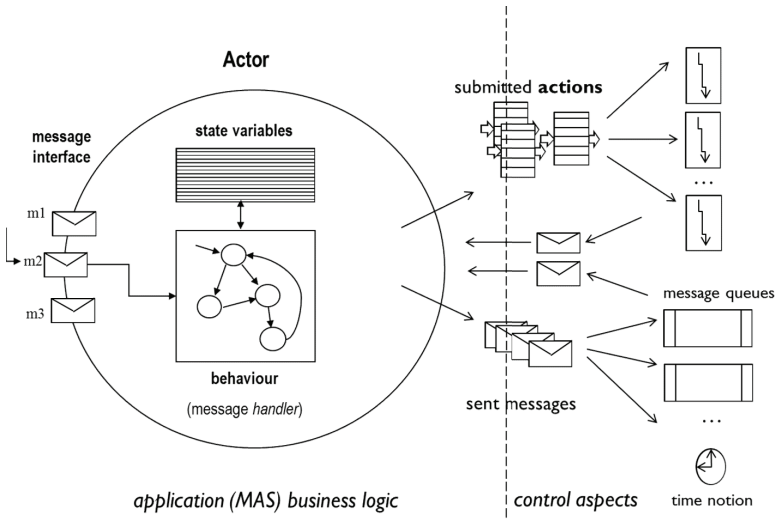


Figure 6.1: Actor structure and cross-cutting control aspects

actions of an actor with its external environment, e.g., for sensing or actuation purposes. The *envGateway* plays the role of abstracting the external devices as well as hiding the used communication protocols. For example, if a temperature sensor is handled by an Arduino device [2], only the *envGateway* is aware of the presence of Arduino and of the specific protocol adopted for interacting with the temperature sensor. From the application viewpoint, the only relevant thing is requiring a read operation from the sensor. In addition, the external environment needs to be modelled, e.g., with the help of continuous time components implementing ODEs, each time a carried out operation has a side effect on the environment itself. For instance, let's consider an application which monitors the temperature in a room and, when the temperature goes below a certain threshold, activates a heating system. Within the real system, the activation of the heating system increases the temperature of the room and, as a consequence, the increased temperature is automatically observed by the sensor. During simulation, instead, the cause/effect relation existing between the heating system and the temperature read by the sensor requires to be explicitly considered. Such aspects are dealt with through the *envGateway* modelling.

The analysis phase

Properties and behaviour of a CPS actor model can be checked by simulation. The model can be simulated in a sequential context or it can be partitioned so as to be handled by distributed simulation, which is actually supported by the proposed framework. Model partitioning is achieved by allocating actors onto different computational nodes (containers or Logical Processes, LPs). Distributed simulation [105] can be required in the case a large/complex model has to be analysed, or in the case the model refers to a system which is intrinsically distributed.

The same model can be simulated in different conditions by simply configuring a different *simulation context*. Setting a particular simulation context

corresponds to defining the *number* and the *behaviour* of the processing units as well as the *policy* adopted for scheduling and executing submitted actions. As an example, actions can be processed in A first-in-first-out order or in a priority-driven way, in which low-priority actions are suspended and subsequently resumed when no more high priority actions exist. The *action schedulers* are the entities which are responsible for managing action scheduling issues. Properties and behaviour of the same model vary as a different simulation context is considered. For example, if a call-centre is modelled where the calls-to-serve are expressed through actions and the receptionists by processing units, the study of how the number of served customers (i.e., the throughput of the model) changes as further receptionists get available, can be carried out by simply changing the number of the exploitable processing units, without any modification to the model. In this phase, all the hybrid-components are configured by their simulated counterpart.

During analysis, a *simulated* (i.e., virtual) *time notion* is used and, in addition, in the case of a distributed simulation, the evolution of the entire actor model has to be time-coherent among all the distributed simulators. Ensuring the right time notion and coordination among simulators is the responsibility of a *control machine*. A control machine is also responsible for coordinating and actualizing message delivery to recipient actors. Message scheduling, dispatching and processing do not increase the simulation time. The simulation time augments only when a timed-message is processed or an action gets executed.

The preliminary execution phase

Preliminary execution is an intermediate stage between the simulation phase and the real execution of a system. A notable difference between this phase and the previous one concerns the used time notion which is no longer a simulated time but the *real time* (the wall-clock time) of the system. Therefore, all the time needed for processing messages, and for sending information through a network, are implicitly taken into account. In fact, the execution of the business logic of the system, e.g., the execution of a control or an optimization algorithm, can be expensive in terms of computational and time resources, and the real time required by the computation and communication ultimately depends on the chosen hardware infrastructure. Such real execution time is taken into account during the preliminary execution phase. As a consequence, this phase can be exploited to assess if the time constraints and system performance, previously checked in the simulation, continue to be satisfied during real-time execution on top of the final exploitable hardware infrastructure. More in particular, the *behavioural drift* existing between simulation and real-time execution, i.e., the deviation between the actual processing of a timed message and its due time, can be quantitatively assessed during this phase.

Analogously to the previous phase, an *execution context* requires to be configured: the processing units and the action schedulers used in simulation must be replaced by the corresponding entities, able to deal with real-time and physical computational resources (e.g., processing units can be mapped on Java threads). Actions are implemented as pure resource-consuming tasks having a time duration and being capable of keeping busy a processing unit. The *envGateway* and all the pure software components remain exactly those used in the analysis phase.

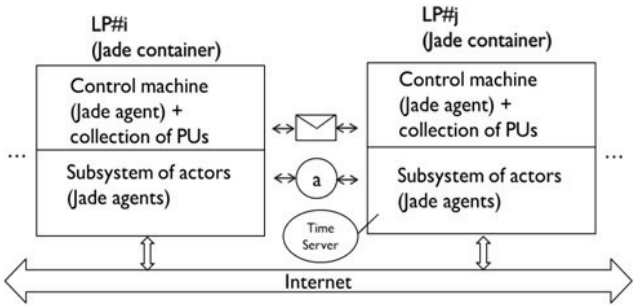


Figure 6.2: A JADE based distributed actor system

A *real-time aware control machine* is now required, both in a sequential or distributed execution scenario.

The real execution phase

In this phase, the system is put into real execution onto the target physical architecture. All the hybrid-components of the model are replaced by their real counterpart. The control machine and the execution context coincide with those used in the previous phase. With respect to the preliminary execution, only the actions and the *envGateway* must be modified. Actions are reified so as to interact with physical devices and carry out real computational tasks, whereas the *envGateway* abstracts the used physical devices together with the communication protocol and physical infrastructure. Obviously, the pure-software components remain unchanged also in this phase.

6.3.2 Control machines and time management

A subsystem of actors (Logical Process or LP) is allocated for the execution on a computing node. All the actors of a same subsystem are governed by a *local control machine*, which transparently buffers exchanged messages into one or more message queues and ultimately consigns messages, one at a time, to recipient actors, according to a proper *control structure*, e.g., based on a specific time notion (simulated or real-time). Message processing is the unit of message dispatching (macro-step semantics). All of this determines a *cooperative* (i.e., not pre-emptive)*concurrency schema* for the local actors of an LP, ensured by message interleaving, which favors time predictability [72, 73].

Multiple actor subsystems (LPs) are federated to constitute a distributed system, using the services of a transport layer and communication protocol. Fig. 6.2 shows a distributed actor system as prototyped by using JADE. Both actors and messages can be dynamically transferred from an LP (JADE container) to another one. Migrating actors can be a need to ensure that an actor is located close, e.g., to a controlled device, or it can respond to dynamic load-balancing issues. A *Time Server* is responsible of maintaining a global time notion across the entire system. In a distributed simulation setting [73], all the control machines interact with the time server in order to negotiate time advancements so as to evolve all together in a coherent way. The exploitable APIs and the library of the available control machines are detailed in [72, 73].

6.3.3 Actions and processing units

By design, an action is a black box with a list of *input parameters* and a list of *output parameters*. Actions have no visibility to the internal data variables of the submitter actor and they do not share any data. Therefore, no mutual exclusion mechanism is needed and no interference problem can derive from the action parallel execution schema and message processing. An *action completion message* can be generated by an action to inform its submitter actor about action termination.

Actions are hybrid-components which have different concretizations during the life cycle of a CPS. *Simulated actions* usually do not carry out any computation except that used to produce output parameters. They have a time duration, specified by an input parameter, which is an estimation of the time duration of the associated modelled task. *Real* or *effective* actions hide a concrete algorithm implementing a computational task. The execution of a real action increases the real time. Pseudo-real actions, used during a preliminary execution, advances the real time but have no concrete computation to perform.

In the case an action interacts with physical devices, the interactions are simulated in both the simulated and the *pseudo-real* actions. On the contrary, they are concretely implemented when a real action is used. A useful feature of actions, which is exploitable during the development of CPS, refers to the capability of returning partially computed results at some selected time points. The *return* primitive is made available for these purposes (see also the sequence diagram in Fig. 6.3). The *return* statement naturally can serve for implementing a periodic behavior within an action. In this case the time points are equally spaced within a time window assigned to the action. At action completion, an operation result message is issued.

An action scheduler administers the local processing units and stores actions which find no available processing unit in pending action queues. A processing unit is a hybrid-component: it can be a physical core or it can be realized by a Java thread, or it is a fake object in the case of simulated actions. A detailed description of the supported kinds of actions, related schedulers and processing units can be found in [72, 73].

6.3.4 envGateway and environment control

During the analysis and the preliminary execution phases, the *envGateway* is exploited to abstract the environment within which the system operates, in the sense of mirroring the effects of the actuations upon the environment itself. In the following, a description of the implemented *envGateway* for the real-execution phase, is provided (see Fig. 6.4). During real-execution, the read/write operations are typically requested by submitted actions which, for generality, can be executed on dedicated Java threads. In a case, one action can be interested to get multiple sensor data, each one being related to a given time point within an assigned time window. The *envGateway* maintains a collection of data variables, which correspond to sensor/actuator devices. An In/Out layer in the *envGateway* is in charge of controlling the communication links with the physical devices and to update the data variables. In particular, the In/Out layer is composed of input/output Java threads, which interface the communication channels with a number of I/O hardware components, e.g., Arduino

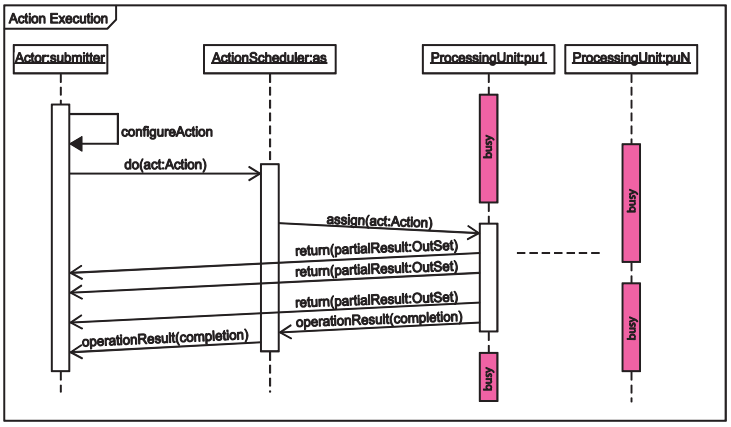


Figure 6.3: Message interplay during an action execution with multiple returns

[2] or similar equipments. Sensors/actuators are physically linked to the I/O hardware. To simplify configuration and operation, each I/O hardware can be specialized to handling a disjoint subset of sensors or actuators.

The communication channels between the *envGateway* and the I/O hardware components, can either be based on the serial connection or on a wireless connection. A suitable protocol requires to be established for the exchange of information between the *envGateway* and the I/O hardware devices. The protocol specifies the input/output operation, the involved physical device and (possibly) accompanying data (in an output command).

The *envGateway* was implemented as a monitor, which manages the action threads and the input/output threads, thus guaranteeing interference-free access to the I/O device data variables. More precisely, separate concurrent hash maps are used for handling the input (sensor) data variables and the output (actuators) commands and data. A design issue of the *envGateway* is concerned with the adoption of an anticipation schema as described in the following. The I/O hardware components are supposed to be programmed so as to repeatedly reading the sensors and providing the data to the *envGateway*. At any instant in time, the values of the data variables represent the most recent data values. Such values are then acquired by actions according to their own timing. For generality concerns, an action which needs some sensor data can provide a filter object at the request time. The filter exposes a *guard method* (i.e., a boolean function) which must be satisfied by the values of involved data variables for them to be actually returned.

Finally, it is worth noting, that correct behavior of a real execution of a CPS system, can require that the reaction to sensed data generated by a controller actor in the cyber part be provided within the sampling period of sensors of the physical part.

6.3.5 Specializing the *envGateway* to work with Arduino

The experiments described in this paper were accomplished by using the serial connection managed by the RXTX.jar Java library. The defined protocol

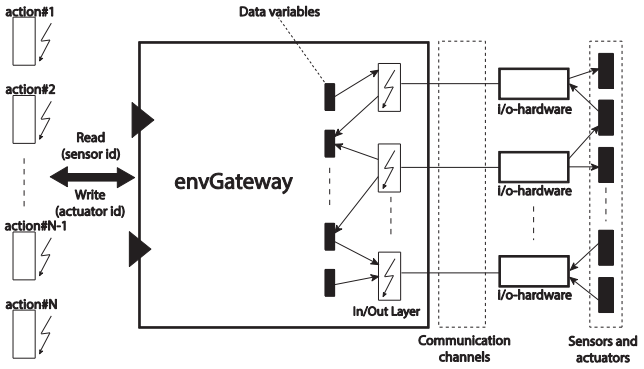


Figure 6.4: Organization of an *envGateway* component

between the *envGateway* and the I/O hardware components clarifies the input/output operation, the involved physical device and (possibly) accompanying data (for an output command). The *envGateway* was concretely interfaced with some Arduino [2] devices. Arduino configuration is carried out in the `setup()` function, where the details of pin connections with physical devices are defined. `setup()` is executed only once following a reset of the device. After that, Arduino enters its main `loop()`. The `loop()` instructions can be directed to reading from sensors and to put the data, after some A/D conversions, onto the communication channels towards the *envGateway*. At the end of the loop, a delay statement is executed before starting the next loop iteration. During its loop operation, Arduino can also receive and process interrupt signals. For example, a *serialEvent* interrupt which is raised whenever new data arrive through the serial communication link (RX), can be heard and managed only at the end of each loop iteration. The mechanism can be exploited to modify dynamically the amount of the loop delay. An Arduino dedicated to controlling only actuators, has an empty loop and all its output operations are delegated to the serial interrupt handling mechanism. Each interrupt signal is expected to be accompanied by all the command information needed for completing the output operation.

During analysis by simulation, the In/Out and I/O hardware layers can be transparently replaced by software agents which provide, in simulated time, pregenerated input data to the *envGateway* or simply consume output commands. Moreover, an *EnvAgent* can be introduced which through a mathematical model, fuzzy logic etc., is able to reproduce the necessary changes in the environmental variables monitored by the *envGateway*, implied by an actuation.

6.4 A case study using power management

A problem of electric power management [6, 131, 60], whose context can be a domestic home or an industrial plant, is considered. Motivation behind the problem stems from the need to exploit to the greatest extent the power generated, e.g., by a local photovoltaic panel, thus ensuring that the power loads in the context are dynamically activated/deactivated (i.e., scheduled) so as to optimally fit, at any instant in time, to the available generated power (reference

or threshold power signal). Indeed, it is not economically viable to sell the surplus of the local produced energy to the external electric provider as it is not properly paid.

The input for the case study is constituted by a threshold signal representing the generated power, and by a certain number of user power loads. Each load is characterized by a dynamic temporal behavior such as the start time and the duration (or computation cost) in the case of one-shot load, or the start time, the duration and a period in the case of a periodic load. Every load is tagged with a *utility* measure. The scheduler gives priority to loads having a greater utility. To avoid starvation, the utility is aged (i.e., increased) in the case a load gets not selected by the scheduler. To capture the quality of scheduler decisions, a *fitting* measure, inversely proportional to the offset between the overall consumed power and the reference threshold signal, is also considered. The scheduler decision takes place as soon as a variation is sensed either in the threshold reference signal and/or in the power loads (e.g., a load notifies it would execute, or it informs it just finished its execution).

The problem of CPS systems like the chosen case study, is to keep aligned the Newtonian time of the physical part with the discrete time of the cyber part. Depending on the physical dynamics of the controlled system, communication and computational delays can make a control reaction inconsistent (and possibly useless) with the actual state of the physical plant. The development of the case study was aimed at both assessing the timing problem and demonstrating the application of the proposed methodology with model continuity. The impact of the computational/communication overhead was checked by designing a scheduler agent which can search for an optimal solution (i.e., optimal load configuration), if there are any, through a full exploration of the solution space which can be computational demanding, or it can exploit a greedy heuristic which looks for a suboptimal, approximate solution, generated in a small amount of time.

For the purpose of the case study, an optimal solution is looked for by an iterative backtracking technique. An approximate solution, instead, is greedy searched by examining the candidate set of active loads, preliminarily ranked by decreasing utility and for the same utility by increasing power.

From a practical point of view, since the backtracking technique can be applied by specifying the number of required solutions, the heuristic solution can be generated as the first-found solution by backtracking, which operates on the candidate set of loads ranked by decreasing utility and then by increasing power. The backtracking process prunes, as early as possible, those partial solutions which can be predicted they cannot become a full solution. Among the acceptable solutions, the optimal one is selected as the one which maximizes the overall utility of loads, and, among the solutions having the same maximal utility, the one which optimizes the fitting measure is preferred.

In the following, the development of the case study is detailed according to the various transition phases enabled by the approach. The provided description highlights the achieved benefits which stem from the exploitation of the same model during the development, to the capability of validating the correctness of the scheduling algorithms and predicting their overhead when executed on a real platform. Finally, it is shown how the implemented system is achievable by concretizing only the hybrid components as described in Section 6.3. The methodology is demonstrated without considering distribution aspects.

6.4.1 Modelling the system

The developed multi-agent model for the power control system consists of load agents (instances of the *LoadAgent* class), one *ThresholdAgent*, one *SchedulerAgent* and the *envGateway* for interacting with the external environment.

The *ThresholdAgent* handles the samples of the generated power signal. The *ThresholdAgent* helps separating the functionalities of the *SchedulerAgent* from those of the *envGateway*. It is the *ThresholdAgent* which reads, through a periodic action, the samples of the reference power signal and transmits them to the *SchedulerAgent*.

Each load agent manages a single physical power load. A *publish/subscribe* design pattern is adopted among the scheduler and the load agents. At its arrival, a *LoadAgent* first registers itself at the *SchedulerAgent* by an *Announcement* message (see also Fig. 6.5) which carries the identification data about the load. Subsequently, the *LoadAgent* communicates with the *SchedulerAgent* each time a variation in the temporal behaviour of the load occurs. Similarly, the *SchedulerAgent* sends commands to load agents for activating or deactivating the corresponding power load. When a load decides to abandon the candidate set of loads, it detaches from the *SchedulerAgent* through a Detach message. A *LoadAgent* is configured with the *id* of the controlled actuation device, the temporal parameters which regulate the load behaviour, and the utility value of the correlated load. The agent also holds the active/inactive status of the corresponding physical load. Each time a load agent receives a command from the scheduler agent, it submits a one-shot action which implements the scheduler command through an interaction with the *envGateway*. An activation command sent to an already active load agent is simply ignored.

A load agent sends to itself a time-stamped message to step along the load power consumption curve, according to the basic time unit (e.g., 1s). Processing such a message causes an interaction with the *SchedulerAgent*, for communicating current load power level, activation/deactivation status and its actual utility value.

By design, the scheduler algorithm of the *SchedulerAgent* was directly coded in the `handler()` method of the agent. To avoid taking so long for the `handler()` to complete its calculations about an optimal solution, the `handler()` of the scheduler is kept busy only for the (minimal) time required for finding the next solution. A *Next* message is sent by the scheduler to itself for starting the computation of the next solution and so forth. In this way the scheduler remains capable of quickly sensing and processing variation events, which can require starting from scratch the scheduling process, just after one further solution was found.

6.4.2 Data configuration

The multi-agent system model was experimented using the configuration data reported in Table 6.1 and Table 6.2. Table 6.1 specifies the adopted threshold (or available) power signal. The time-span of the available signal is 480 time units (*t.u.*) where a time unit corresponds to 1s of Newtonian time of the physical plant. Table 6.2 details the assumed power loads. Only the load #3 is periodic, and, following its termination, it becomes ready again to be scheduled after 12 *t.u.*.

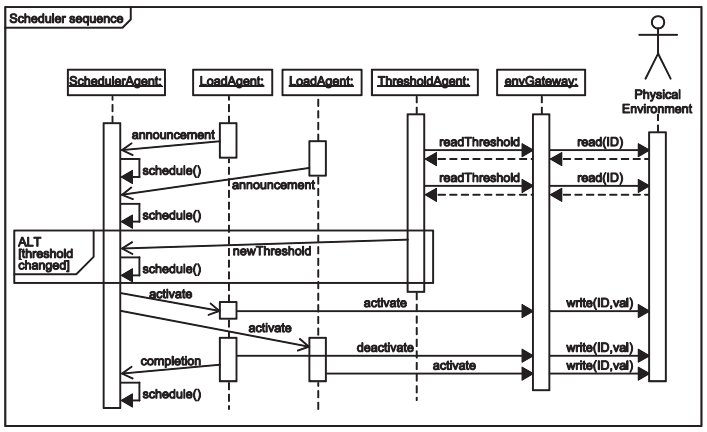


Figure 6.5: Main interactions among model agents

Table 6.1: Reference threshold power signal

Available Power (W)	Time Duration (t.u.)
750	10
900	15
1300	20
2400	60
3300	205
1800	5
3100	100
2100	25
1800	10
1100	10
600	20

6.4.3 Analysis phase

A first concern was studying in simulation the multi-agent system model. The goal was checking both functional and non functional (temporal) properties, particularly the behaviour of the scheduling algorithm. It is worth noting that, in simulation, message processing consumes 0 time. As a consequence, the scheduler algorithm, being implemented in the `handler()` method of the *SchedulerAgent*, either searching an optimal or suboptimal solution, is always virtually completed in 0 time. Time advancement is mainly related to the duration of actions, and then to stepping through the power signal samples.

The model was executed on a standalone machine with the *Simulation* control structure [73]. *Simulated actions* are used and their execution immediately schedules the *completion message* or the message of the next *return*, which is time-stamped with their due time. The samples of the available power signal are pre-loaded in the *envGateway*. Output commands are simply logged.

When an active load is interrupted because the *SchedulerAgent* chooses a different load, its completion message is invalidated and the remaining time to completion is stored by the *LoadAgent* so as to be exploited at its next activation.

Fig. 6.6 portrays the scheduling effects on the total consumed power by loads with respect to the available threshold power signal, when an optimal or

Table 6.2: Power loads parameters

Load ID	Utility	Power Request (W)	Time duration	Available at	Periodic
1	3	500	300	8	No
2	2	250	380	18	No
3	4	500	400	12	Yes
4	5	750	350	0	No
5	4	250	330	0	No
6	3	250	350	0	No
7	2	250	320	0	No
8	1	250	410	0	No

a suboptimal solution is adopted.

For the assumed loads (Table 6.2), the power consumption curves in Fig. 6.6 are very similar. However, since the scheduler tends to select different loads, differences will ultimately emerge between the two curves. At time 437 the periodic load is reactivated and the consumed power suddenly raises under optimal scheduling. Under suboptimal scheduling, instead, the same load reactivates at time 457. This is due to the fact that the optimal scheduler is capable of ensuring a greater power consumption in the time interval from 332 to 410 t.u..

Fig. 6.7 depicts the observed fitting, i.e., the deviation between the available power and the total consumed power, vs. time, in the two cases optimal/suboptimal scheduling.

As one can see from Fig. 6.7, the two algorithms tend to behave differently in the long time when, definitely, it *seems* that the optimal algorithm is outperformed by the suboptimal algorithm. In reality, since the optimal algorithm is capable of activating simultaneously more loads (although with a same total consumption power level) which in the considered case are almost one-shot, in the long time the optimal scheduler has fewer loads to manage and then it exhibits a greater deviation of the consumed power from the available power.

The above observations are confirmed by Fig. 6.8 which shows the total observed utility of the scheduled loads vs. time, in the two scenarios. Initial and final differences are respectively due to the ageing process of the load utility and to the fact that definitely the optimal algorithm handles fewer loads.

6.4.4 Preliminary execution

Under preliminary execution, the model was run using the *RealTime* control machine (with the time unit set to 1s) along with simulated actions and the *FirstComeFirstServedAS* action scheduler [73]. No change was introduced in the agent model. The goal was to check the effects of message processing, which now is no longer negligible, on the scheduling process. Message processing overhead obviously depends also on the performance of the hosting computer machine. Fig. 6.9 depicts the total consumed power vs. time when the optimal scheduling is used, both in the simulation and the preliminary execution scenarios. The experiments refer to the use of a Macbook Pro Intel Core i5, 2.9GHz, 16GB, OS X El Capitan. A delay clearly emerges in the scheduler reaction due to the algorithm overhead. Such a delay disappears in the case the suboptimal algorithm is adopted (picture not reported for brevity).

In order to highlight the usefulness of the preliminary execution phase, the experiment was repeated also on a less performing (older) Macbook, Intel Core

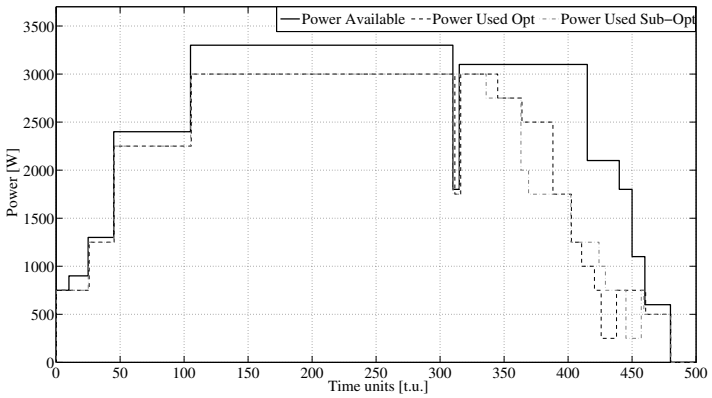


Figure 6.6: Power available and total consumed power vs. time - optimal/suboptimal scheduling

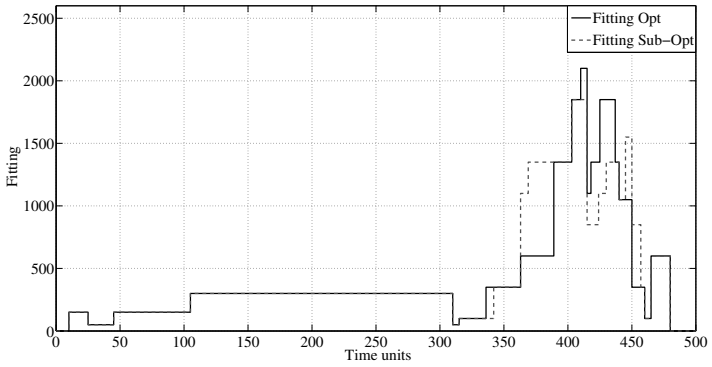


Figure 6.7: Observed fitting vs. time

2 Duo, 2GHz, 2GB, OS X Mavericks. In this case the delay of the optimal algorithm gets increased (see Fig. 6.10) as expected. A closer examination of the generated logs reveals that at time 315 a reaction is required but, whereas the simulation is capable of producing an optimal schedule by instantaneously (although ideally) responding to the variation event, the new Macbook employs 10 time units for finding an optimal solution in preliminary execution, and the old Macbook requires more time and it happens that a new variation event is sensed during the scheduler operation which forces it to restart its computation thus missing one reaction.

The worst case of observed *drift*, i.e., the time deviation with which a message is processed with respect to its due time, was found to be about 101ms on the less performing Macbook and about 48ms on the high performing Macbook. The worst case occurs during the initialization/bootstrap of the model. Definitely, the drift tends to be a few ms.

The documented experimental results show that the optimal algorithm, despite computational and communication delays, is capable of managing loads by

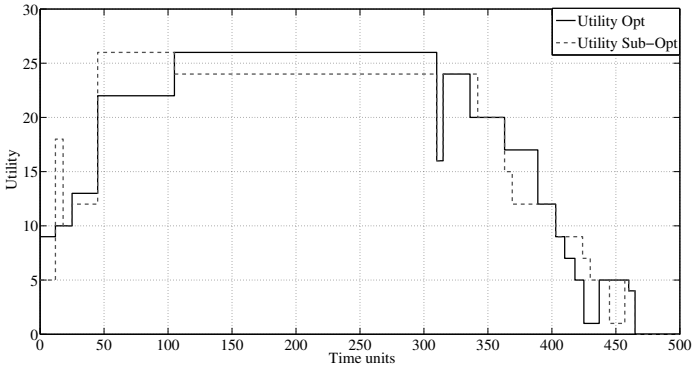


Figure 6.8: Total utility vs. time

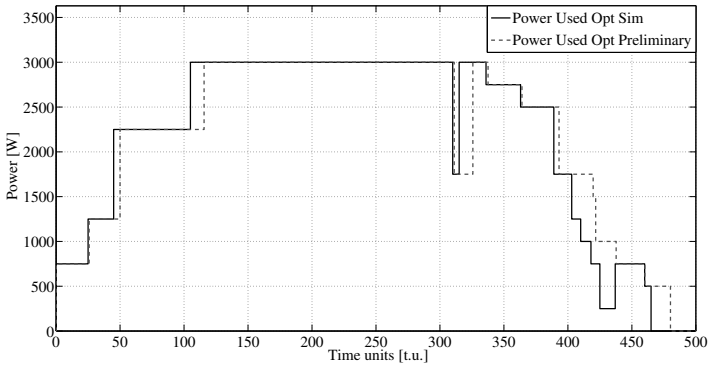


Figure 6.9: Used power vs. time - optimal scheduling, simulation/preliminary execution

generating control actions consistent with system dynamics. However, the sub-optimal algorithm is preferable because it favours correct temporal dynamics. Its use, in fact, guarantees that the reaction to a variation event is computed and actuated during the same sampling period or, in the worst case, until the next one would a particular disalignment between the physical clock (of Arduino) and the cyber clock occur. All these properties were confirmed by considering both the overhead of the scheduler algorithm and the bookkeeping of message scheduling and dispatching, also in the case a low performing computer is adopted.

6.4.5 Prototype implementation and real execution

The CPS power management case study was assembled in the context of an academic electronic measurement laboratory, where loads are realized by lamps of a basic power of 250W. Single or groups of lamps whose power is a multiple of 250W were realized so as to be controlled by few relays. This explains the data assumed in Table 6.2. Fig. 6.11 provides an overview of the overall CPS, in which a group of physical loads (lamps) are controlled by corresponding load

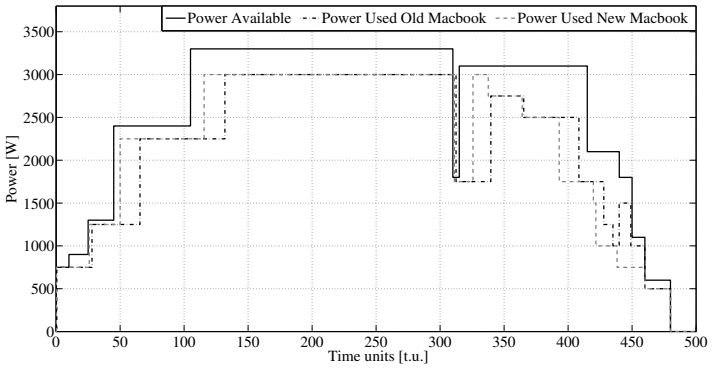


Figure 6.10: Power available and total consumed power vs. time - optimal scheduling, preliminary execution, the two Macbooks

agents, which in turn receive control commands from the *SchedulerAgent* which is in charge of monitoring the reference input power signal and to adapt the power loads accordingly. The threshold power signal was generated by LabView software and loaded in the memory of an Arbitrary Waveform Generator AWG2021, configured to generate the signal with a frequency of 10Hz. Two Arduino Uno [2] were used as I/O hardware components, with serial communication channels. The first Arduino is devoted to reading the threshold power signal samples. The second one serves to effecting the commands to relays which activate/deactivate the power loads. The prototyped model implementation was executed using the *RealTime* control machine, effective actions and the *envGateway* which provides interactions with real input and output physical devices, controlled by the two Arduino. Except for an initialization time (due to setting up the Arduino, opening the serial communication channels etc.), which establishes an initial offset of about 10s, the behaviour of the available power and used power is that observed during the preliminary execution. As a final remark, although the case study was driven by pre-configured signals for the generated power and the consuming loads, the agent-based solution is open and flexible to work with, e.g., more dynamic load configurations. This is due to the reactive character of the scheduler which is capable of intervening at each occurrence of a variation event. In the following, a description of the used measurement bench, the load subsystem, and the adopted communication protocol is provided.

Measurement bench

The assembled measurement bench is shown in Fig. 6.12. It is composed of an Arbitrary Waveform Generator Sony/Tektronix AWG2021, an Arduino One board, a Digital Oscilloscope Tektronix TDS220, and a personal computer. The personal computer interfaces the AWG2021 by a General Purpose Interface Bus (GPIB) and runs a proper program developed in the NI LabView environment. The output channel of the AWG2021 feeds the analog channel 0 of the Arduino board to provide the emulated power signal. Moreover, the marker output signal of the AWG2021 feeds the digital I/O pin 7 of the Arduino One board in order

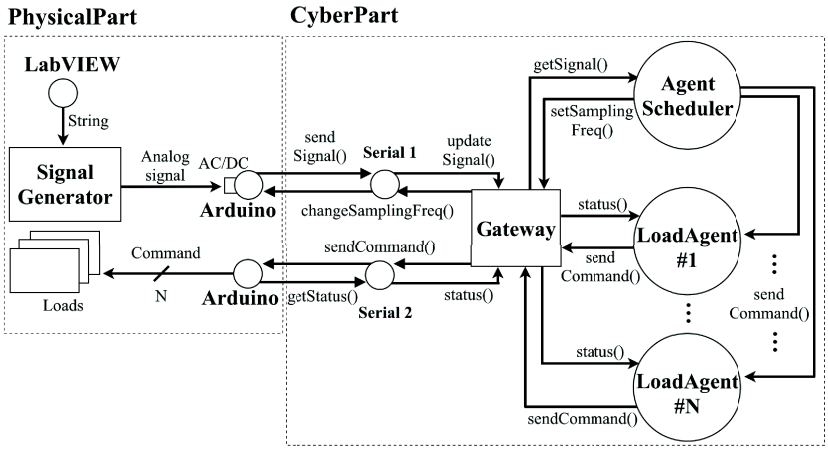


Figure 6.11: An overview of the realized CPS

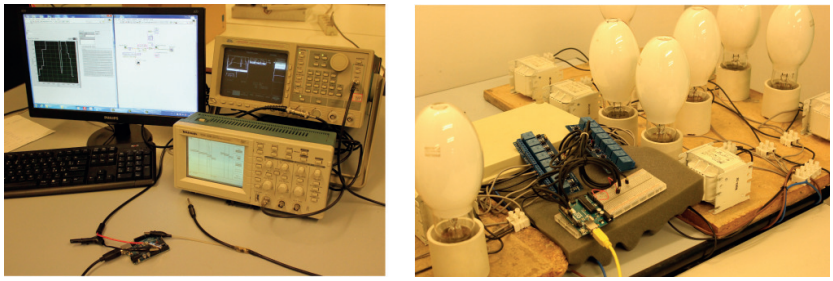


Figure 6.12: Measurement bench and controlled loads (lamps)

to synchronize the two devices. The TDS220 is connected in parallel to the output channel of the AWG2021 in order to visualize the trend of the output signal and then the correct shape of this signal.

Load subsystem

The load subsystem consists of one Arduino One board, 2 relay boards with 8 high voltage channels characterized by rating of 10A at 250 and 125 V AC and 10A at 30 and 28 V DC, managed by means of 8 digital pins adapted to work with the Arduino output operating voltage, and 12 Standard High Pressure Mercury lamps Philips HPL-N 250 W and 12 ballasts. Each ballast is connected to a lamp in order to regulate the current to the lamps and provides sufficient voltage to start the lamp. The loads are obtained by connection of a pre-established number of lamps. The digital pins of the Arduino board feed the digital pins of the relay modules, so permitting to turn on and off the loads.

Communication protocol

The cyber and the physical subsystems interact with each other through the exchange of character strings. Strings can express commands to be executed by the loads, or can capture information that Arduino achieve from sensors,

ultimately destined to agents. The following clarifies the adopted format: *sensorId/actuatorId # message*, where the content of the token *message* can vary. Information about sampling the available power signal is transmitted from Arduino as: *watt # powerLevel* where *powerLevel* is a double number. To change the sampling period the following command can be sent by agents: *arduinoId #samplingTimeInMillis*. To act on a relay it is necessary to send to Arduino the load *id* and the *type* of the command to be executed. The format is: *loadId #command powerLevel* in which the type of the command can be:
⇒ **A**, to activate. It allows to (re)connect a load with a given power level;
⇒ **C**, to change load power. It permits to change the power level of an already connected load;
⇒ **D**, to deactivate. It asks to disconnect a load (in this case no power level is furnished).

Chapter 7

Qualitative and Quantitative Model Checking of Distributed Probabilistic Timed Actors [1]

This chapter describes an evolution of the Theatre actor infrastructure whose design aims to favoring the development of predictable time-dependent applications. Adopted actors are thread-less and their execution 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). This chapter describes the Theatre architecture and introduces a real-time case study which is used as a running example. 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.

7.1 Introduction

The described work is concerned with a model-driven methodology for the development of distributed real-time systems such as cyber-physical systems [160, 78].

[1] The material in this chapter is related to publications [196, 194, 193, 188]

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 [10] which is a well-known formal framework [11] 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 [159]. 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 [241], or discrete-event simulation which requires high-performance execution [105]. 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 [234] along with its probabilistic and timed extension (PTRebeca) [129]. Rebeca represents an interpretation of the classical actors model [10], formally defined through a structural operational semantics [129]. 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 [144] or the IMCA (Interactive Markov Chain Analyzer) [112], or targeting the model to Erlang with the timed McErlang tool [224] used for model checking or, to avoid state explosion problems, by statistical model checking activities [127]. 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 [73] 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 ad-

dressed: a same model can be transitioned without distortions (*model continuity* [78]) 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 [74], to the real-time control of power management in a smart micro grid [78], to the support of modelling and analysis of general complex multi-agent systems [189, 188]. A library of prototyped control forms can be found in [73, 79], 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 [73] 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 [129] 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 [17] of the UPPAAL popular toolbox [27, 90] 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 [9, 90]. Current paper significantly extends the preliminary experience described in [194] where only statistical model checking activities were enabled. A major difference from [194] consists in the replacement of *dynamic message templates* [90] 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. First 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 summarizes the implementation status of THEATRE and illustrates some methodological guidelines.

7.2 THEATRE concepts

7.2.1 Architectural view

A THEATRE system (see also Fig. 7.1) 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 a

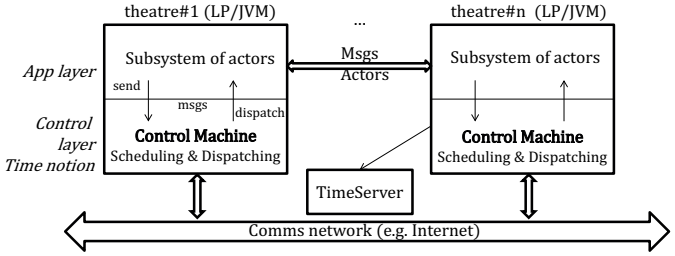


Figure 7.1: A THEATRE system

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 [134]. 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 chapter 8). However, other solutions were experimented as well. For example, in [73, 78] the JADE agent infrastructure [29] was used as a middleware providing naming, messaging and network services; in [62] the Terracotta services were exploited, in [67] the Globus middleware was used etc.
A THEATRE system (see Fig. 7.1) can admit a *time server* (allocated to a given

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 7.2: Theatre abstract modelling language

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 [10] by allocating one actor per theatre.

7.2.2 Abstract modelling language

In the following, the "in-the-small" modelling aspects of Theatre actors are tailored according to the PTRebeca modelling style [129]. In Fig. 7.2 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 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 chapter 8) which exposes all the fundamental services: send, now(), etc. Actor classes rely only of the default language constructor implicitly used at each actor creation.

7.2.3 A modelling example

Fig. 7.3 depicts a THEATRE model of a dependable real-time toxic gas sensing system (TGSS), adapted from [129]. 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. 7.3) which affect actor operation. Fig. 7.4 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. 7.3), 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. 7.3 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 danger
    bool meetDangerousLevel=false
    msgsrv init(Main m, Scientist
        sc=s; m.done();
    }//init
    msgsrv changeGasLevel(){
        if(gasLevel==2)
            gasLevel=?(0.98:2,0.0
    ;
        if(gasLevel>2 &&
           !meetDangerousLevel){
            sc.die()
            after(SCIENTIST_DEADL
    ;
            meetDangerousLevel=tr
        }
        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, Environm
    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.checkScientistAc
      after(SCI_ACK_DEADI
    abortSent=true;
    }
   }
   self.checkSensors() af
   CONTROLLER_CHECK_DELAY
  }
}//checkSensors
msgsrv ack(){ sciAlive=tr
msgsrv checkScientistAck(
    if(!sciAlive)
    re.go() after(NET_DEI
}//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 7.3: A Theatre model for the toxic gas sensing system, adapted from [129]

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 controller an ack() message. The controller checks the arrival of the scientist ack by sending to itself a checkSci-

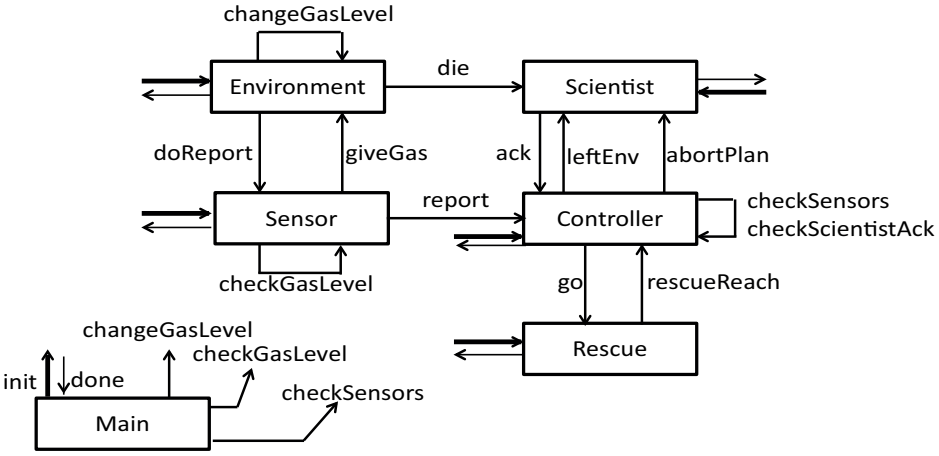


Figure 7.4: Message exchanges in the toxic gas sensing system model

entistAck() 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 [129], 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. 7.3). Further details about the semantics of Theatre are given in the next

section.

7.3 An operational semantics of THEATRE

As in [129], a structural operational semantics of THEATRE in the style of the transition rules of Plotkin [210] and Kahn [132] 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: $C[pu \triangleright free]$, $C[pu \triangleright occupied]$. 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. 7.2 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 [90] 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 [129]. 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 [90, 27]. Anyway, the provided semantic rules could be extended to specify the probability distribution of transitions using an approach similar to that shown in [129].
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.

7.3.1 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. 7.5) 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. 7.6).

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'

$(msg - dispatch)$

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

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 7.5: Message dispatch rule

$(delay - expiration)$

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

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

Figure 7.6: Delay expiration rule

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.

As a consequence of the $delay - expiration$ rule in Fig. 7.6, the configuration C will be changed to $C' = C[pu(Ai) \triangleright free]$ 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 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. 7.7 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

$(time - progress)$

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

Figure 7.7: Time progress rule

$(send)$

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

$(delay)$

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

$(assignment)$

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

$(non\ deterministic - assignment)$

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

$(probabilistic - assignment)$

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

$(create)$

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

$(move)$

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

Figure 7.8: Statement rules - 1st part

$(cond_1)$

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

$(cond_2)$

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

$(sequence)$

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

$(msgsrv - end)$

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

Figure 7.9: Statement rules - 2nd part

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. 7.8 and Fig. 7.9, each corresponding to a distinct basic action admitted by the THEATRE modelling language (see Fig. 7.2).
The *probabilistic − assignment* rule in Fig. 7.8 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. 7.8) 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. 7.5), and it is freed at the message server end (see

the *msgsrv − end* rule in Fig. 7.9) provided no delay operation was raised during the message server. All of this ensures the macro-step semantics of messages within a same theatre.

7.4 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 [27, 90] using timed automata (TA) [17]. 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. 7.12). For example, a probabilistic assignment is reproduced by a branch point whose dashed exiting edges (Fig. 7.12) 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 [90]. Dynamic automata were experimented, e.g., in [194]. 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 7.2.1).
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 7.2.3.

7.4.1 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;
```

Table 7.1: 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

7.4.2 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.

```
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 →free, false → 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.

7.4.3 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.

7.4.4 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 [27] 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.

7.4.5 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.

7.4.6 The Message automaton

Fig. 7.10 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 7.3), 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 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

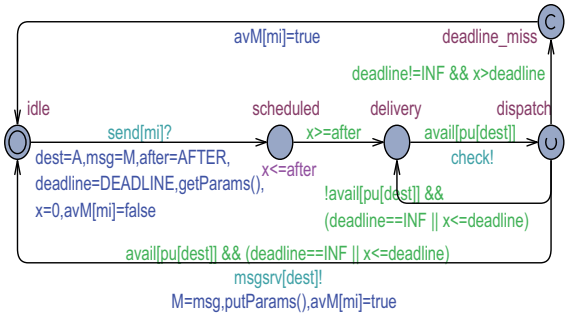


Figure 7.10: The Message timed automaton

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. 7.10 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. 7.10 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. 7.10, 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. 7.10).
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.

7.4.7 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. 7.11 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

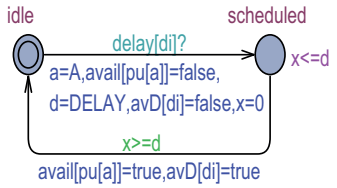


Figure 7.11: The Delay automaton

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.

7.4.8 An actor automaton

The model of an actor (see, for example, the Environment automaton of the toxic gas sensing system in Fig. 7.12 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. 7.12, 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.

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

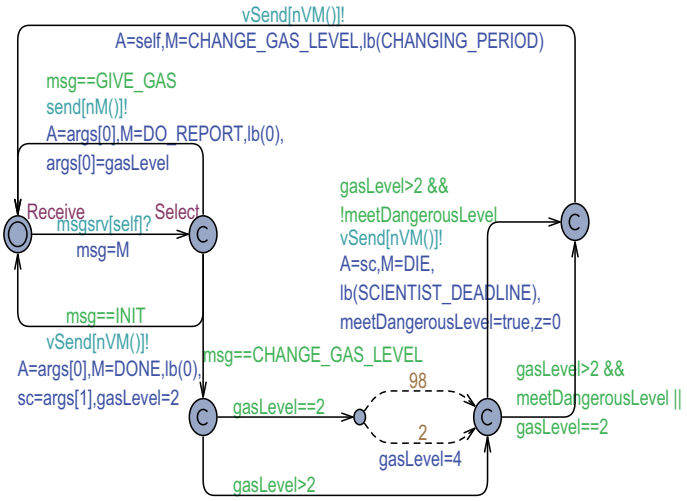


Figure 7.12: The Environment actor automaton

number of messages which are sent within it. Indeed, only one message send (channel synchronization) can be specified per edge.

7.4.9 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 7.3) 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 7.3) 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 7.2.1), 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*

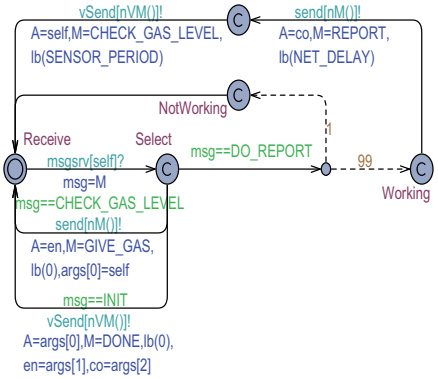


Figure 7.13: The Sensor actor automaton

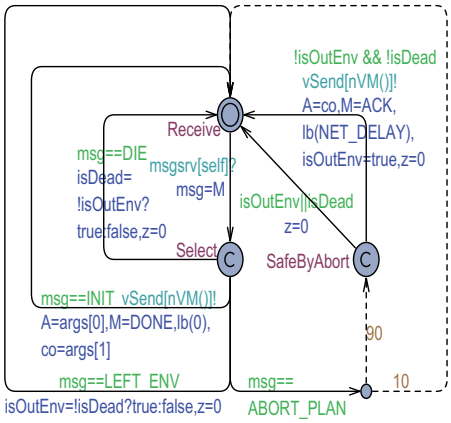


Figure 7.14: The Scientist actor automaton

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 7.3). 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.

7.4.10 Translated UPPAAL model of the toxic gas sensing system

Fig. 7.13 to Fig. 7.17 show all the remaining Uppaal actor timed automata for the case study (the Environment automaton is shown in Fig. 7.12). Each actor model is parameterized with only one parameter of its corresponding sub_range type (see section 7.4.2).
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 controller. Care was taken in the model in Fig 7.3 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,
```

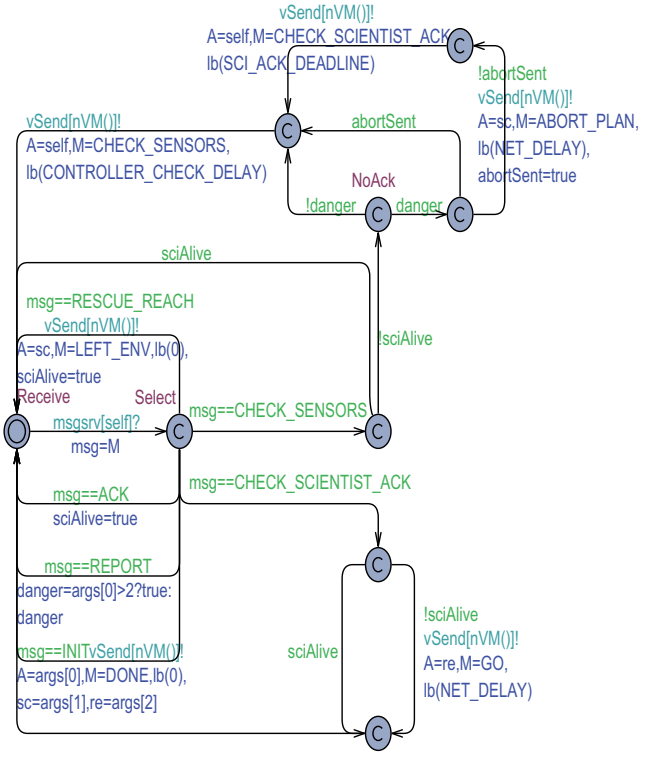


Figure 7.15: The Controller actor automaton

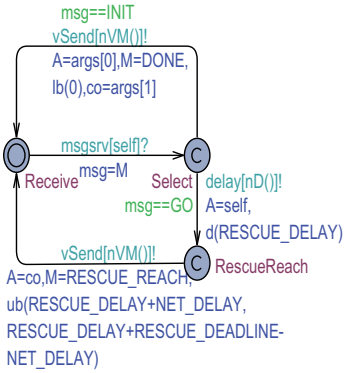


Figure 7.16: The Rescue actor automaton

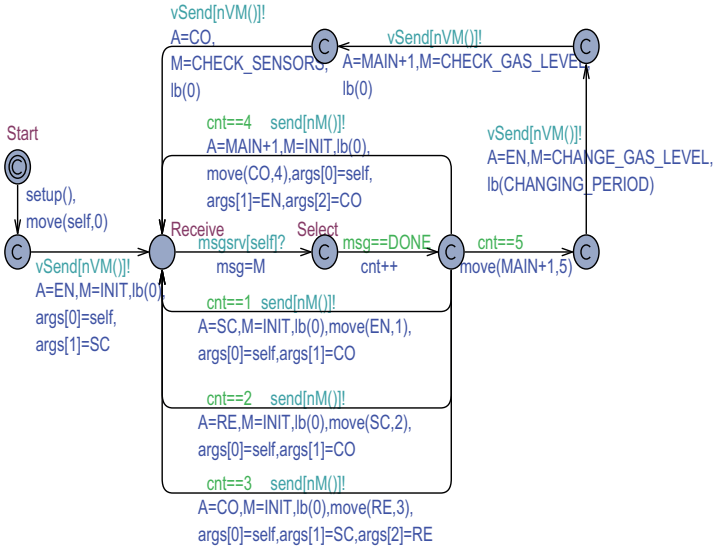


Figure 7.17: The Main actor automaton

```
Message,VoidMessage,Delay;
```

Of each template automaton a number of instances is automatically created as determined by the corresponding sub-range type (see section 7.4.2). 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.

7.5 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 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) [27] which does not admit formula nesting. The statistical model checker of UPPAAL uses, instead, an extended version of the Metric Interval Temporal Logic (MITL) [18]. All the experiments were carried out on a Linux machine, Intel Xeon $CPUE5-1603@2.80GHz$, 32GB, using UPPAAL 4.1.19 64bit.

7.5.1 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 7.4.3) 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. 7.10). Both queries terminate by saying they are not satisfied.
Due to the asynchronous message exchanges among actors, a reduced THEATRE

Table 7.2: 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

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 7.2.3) 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. 7.3. 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 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. 7.18 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.

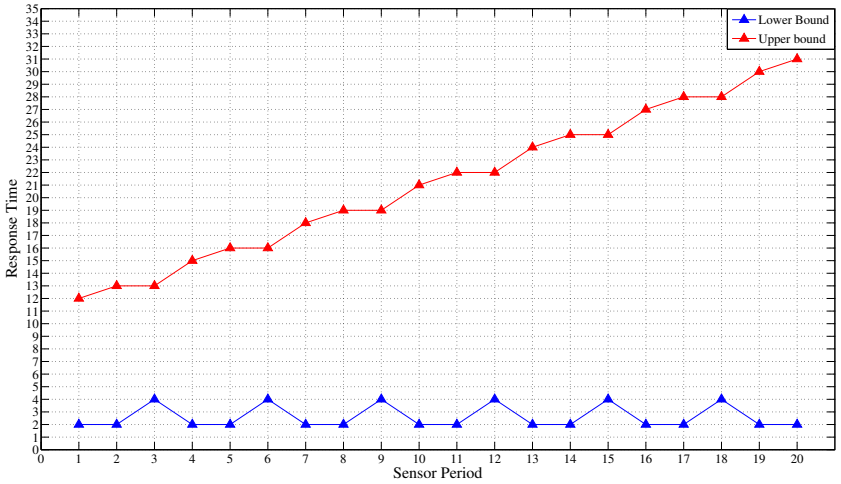


Figure 7.18: Response time windows when the scientist *can* perceive an ABORT_PLAN

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. 7.19 depicts the observed time windows when the scientist, optimistically, is always capable of perceiving an ABORT_PLAN (in Fig. 7.14 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:

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

With respect to Fig. 7.18, the lower bounds of the response time in Fig. 7.19 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.

7.5.2 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*",

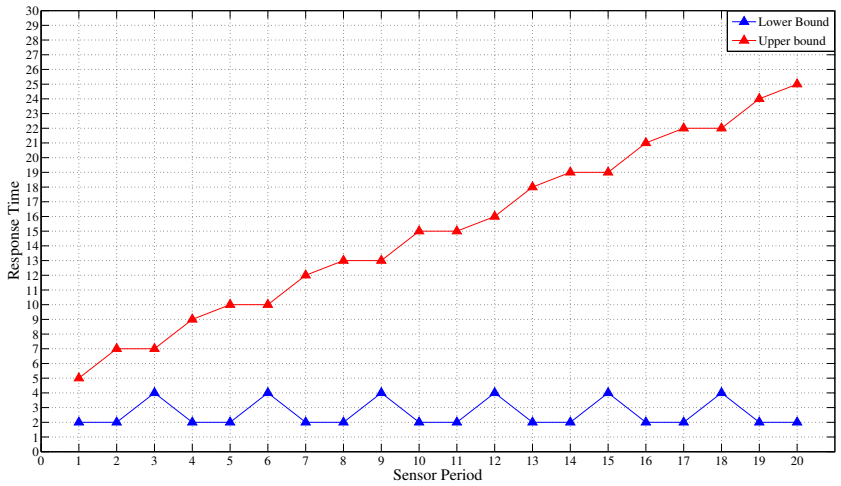


Figure 7.19: Response time windows when the scientist *always* perceives an ABORT_PLAN

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 [9], 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. 7.20 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. 7.20 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:

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

As one can see from Fig. 7.20, 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

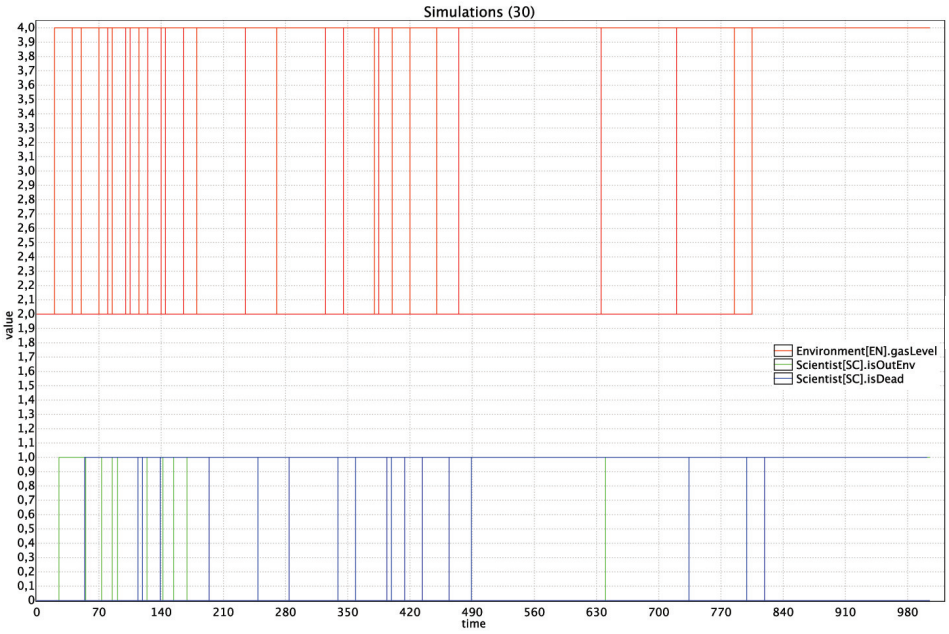


Figure 7.20: Traces of 30 simulations monitoring the gasLevel, isOutEnv and isDead variables

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. 7.21 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. 7.21 depicts the bounds of the confidence intervals proposed after launching the query:

$$Pr[<= 5000](<> (Scientist(SC).isDead))$$

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 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. 7.18) 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:

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

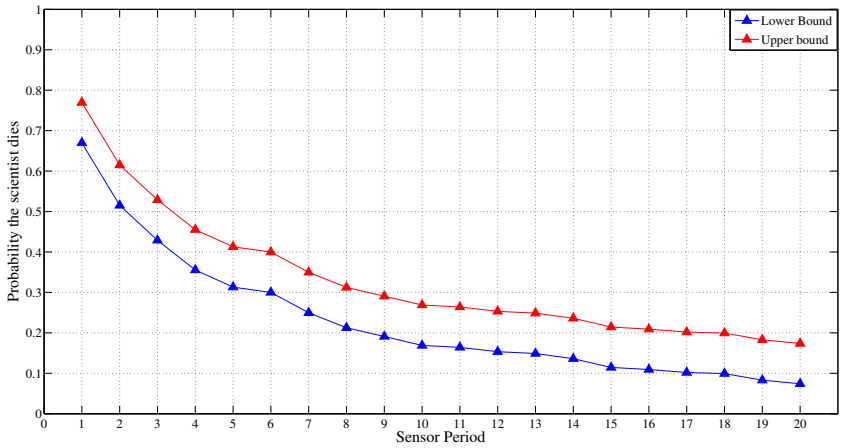


Figure 7.21: Probability of the scientist to die vs. sensor period, when only one sensor is used

proposes, after 29 runs, a confidence interval of [0.901855,1], thus confirming the event has a great occurrence probability.
The results in Fig. 7.21 are also a consequence of the transformation of a non-deterministic behavior into a true probabilistic one guided by the adopted 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. 7.14), 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 [18]. 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. 7.21) 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]. 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 be-

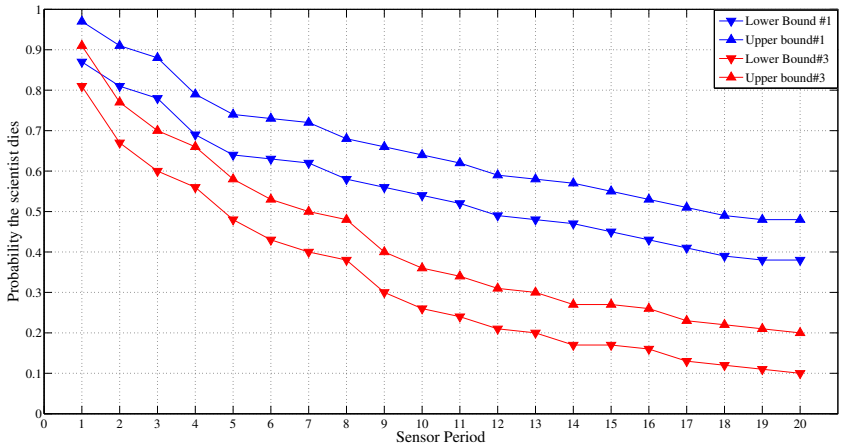


Figure 7.22: Probability of the scientist to die vs. sensor period, when one or three sensors are used and the probability for a sensor to be working is 95%, and not be working is 5%

havior 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. 7.13) 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. 7.22 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. 7.22, the probability is greater than that shown in Fig. 7.21 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 emerges from Fig. 7.22, that the probability value significantly decreases when 3 sensors are used.
Finally, in Fig. 7.23 and Fig. 7.24 the probability of the scientist to die is shown when the model is that of Fig. 7.3, 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. 7.23 refers to the case the SCIENTIST_DEADLINE is 10 and in Fig. 7.24 it is 12. The following query was used for Fig. 7.23. Each point is the result of 738 runs. The until interval is turned to [0,12] for Fig. 7.24.

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

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 7.18 and 7.19 confirm there is a value for the sensor period where

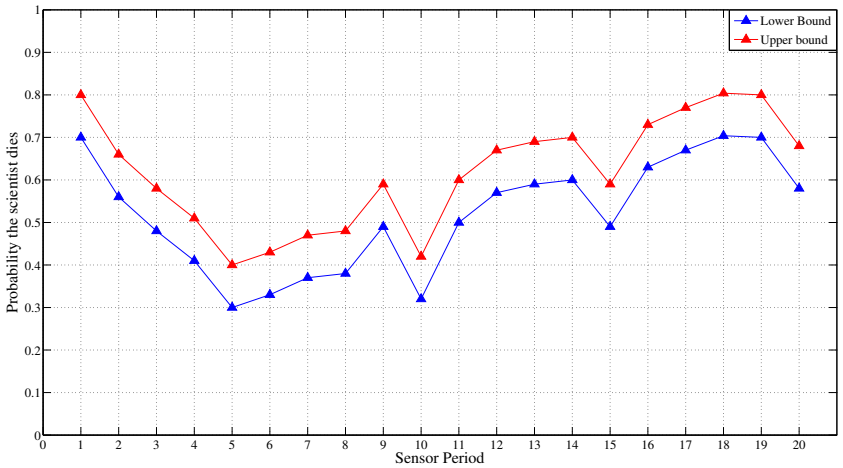


Figure 7.23: Probability of the scientist to die vs. sensor period, when the SCIENTIST_DEADLINE is 10

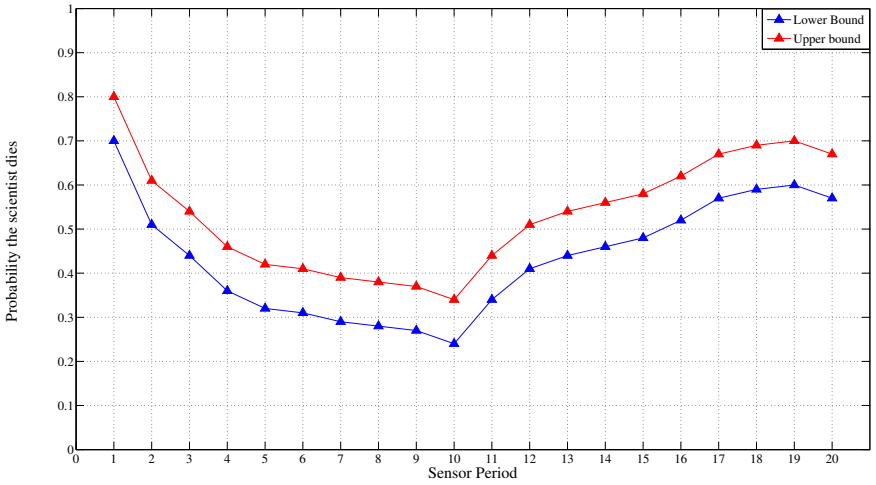


Figure 7.24: Probability of the scientist to die vs. sensor period, when the SCIENTIST_DEADLINE is 12

the probability gets to a minimum. In the case of Fig. 7.23 this occurs at abscissa 5, whereas it shifts to 10 in Fig. 7.24, due to the greater value of the SCIENTIST_DEADLINE. In reality, in fig. 7.23 there are more local minima. This is due to a parameters alignments configuration in the proposed 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. 7.24 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.

7.5.3 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. 7.3) 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 7.4.2), (possibly) adapting the required number of Message instances and, finally, adjusting the move() operations in the main automaton (see Fig. 7.17).
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.

Chapter 8

Seamless Development in Java of Distributed Real-Time Systems using Actors [1]

8.1 Introduction

This chapter describes a full implementation in Java of THEATRE, extended with PTRebeca timing model and programming style. Java is used in all the development phases of the system lifecycle. This makes it possible to model and analyze a system and then seamlessly implement it for real-time execution. The solution is self-contained and no third-party middleware is required or used. This represents a major extension of previous work exploited in chapter 6. Pre-existing THEATRE was based on the distribution services provided by such middleware as JADE [78, 29]. From [78] it is retained the runtime infrastructure for controlling the input/output devices (sensors and actuators) of a cyber-physical system. Moreover, with this new THEATRE scheme the execution can be easily distributed across several computers rather than being limited to just one, as in [78].

8.2 An overview of THEATRE

A Theatre distributed system (see Fig. 7.1) consists of a federation of computing nodes (theatres or logical processes LPs). Each theatre is organized around four functional layers: the *application layer* (AL), the *control layer* (CL), the *transport layer* (TL) and the *time server layer* (TS). AL is made up of a collection of local actors assigned to the theatre for execution. CL is a fundamental layer which keeps a reflective link with AL so as to (transparently) regulate the evolution of local actors, e.g., according to a given time notion (real or simulated). CL mainly is devoted to message scheduling and dispatching. TL opens the theatre to remote interactions with peers reached through a communication

[1] The material in this chapter is related to publications [79, 80]

network. TL is normally based on a reliable transport layer like TCP sockets. The TS layer, finally, is responsible to guaranteeing a common global time notion in a system.
The source of flexibility for customizations stems from the possibility of specializing the programming style in AL, and the services implemented in CL and TS which can be tailored to work with simulated time or real-time or a combination of the two.
Time predictability is the result of having local *actors* which are *reactive* in character: they are at rest until a message arrives. No mailbox and no built-in thread exists within an actor. Messages sent are ultimately captured, buffered, managed and dispatched by the *control machine* component of CL. Message processing is atomic: it cannot be preempted nor suspended. The control machine repeats a basic loop: at each iteration, one pending message, if there are any, is selected according to a given strategy, and delivered to its recipient actor thus triggering a reaction. At the end of message processing, the control machine loop is re-entered and proceeds with the next message and so forth.
Within a theatre a *cooperative* (not preemptive) *concurrency* schema exists, ensured by interleaving of message dispatches. Actors belonging to *distinct theatres* can instead be executed in *true parallelism*.

8.2.1 Basic Java Framework

The THEATRE vision is enforced by some basic classes implemented in Java. A fundamental class is the Actor abstract class which exposes the basic services to actors. A programmer defined actor class derives directly or indirectly from *Actor*. *ControlMachine* is the base and abstract class which has to be extended for achieving a custom control structure. Actor and ControlMachine are synergic to each other: the services provided by Actor are ultimately handled by a specific control machine class. Behind the scene classes capture the structure of local and remote transmitted messages among actors. A *universal naming* schema is adopted for *actors* which relies on *unique* system-wide *strings*. Unique identifiers are also used for theatres.
The Actor class is currently shaped according to the PTRebeca [224, 127] programming style and timing model. Message reactions in an actor class are programmed as void methods which can have parameters, annotated as message servers (msgsrv):

```
@Msgsrv public void messageName( [params] ){ msgsrv body }
```

Basic services of Actor are the non-blocking *send*, the time service *now()* which returns the current time, the *delay(d)* operation expressing the duration of a code segment within a message server, and a *move(theatreName)* operation which migrates an actor to a given theatre. Three versions of the *send* operation are provided:

```
void send( "messageName", Object... args );
void send( after, "messageName", Object... args );
void send( after, deadline, "messageName", Object...args);
```

A message send can be untimed (first version) or it can be equipped by one or two (relative) time values: *after* and *deadline*. It is intended that the message cannot be dispatched before *after* time units are elapsed from the sending time,

but it should be dispatched within its deadline. Beyond the deadline, a message can no longer be dispatched and it is lost. When not specified, *after* defaults to 0 and *deadline* to ∞. For generality reasons, time values are assumed to be specified as double quantities. Java reflection is used to link a message send with an underlying message object along with a message server method invocation. Normal Java methods, not annotated by $@Msgsrv$, cannot be asynchronously invoked by a send operation. They, though, can be useful to structure the body of a msgsrv.
The *delay(d)* operation is mainly useful during modelling and analysis when a message server is obviously not concrete and an estimation of (or a part of) its duration can be anticipated. Also the duration d is a relative amount of time.
The following control classes were developed. Specific control forms were designed to sustain the advocated methodological phases of a Theatre system. Some simulation control machines, though, are useful for general discrete-event stochastic, high-performance simulations. Control machines come normally in pairs: a standalone version referring to one single theatre, and a distributed version which exploits a partitioning of the actors over a collection of theatres. In this case, a time server component (see Fig. 7.1) should be used.
Concurrent, DConcurrent - Control machines supporting the execution of general untimed actor applications.
Simulation, DSimulation - Control machines specialized to simulation and performance prediction of discrete-event stochastic systems. *DSimulation* should be used in the case of large and scalable models, and depends on a conservative synchronization algorithm [105] for the advancement of the global time.
AbstractSimulator - A control machine dedicated to the simulation on a standalone workstation of a Theatre distributed real-time model. It serves for early assessment of functional and timing behavior of a system.
Preliminary, DPreliminary - Control machines which use real time on an abstract model. They are useful to check the overhead introduced by message scheduling and dispatching which have a zero duration during analysis.
RealTime, DRealTime - Control machines based on real time. They regulate the evolution of a concrete THEATRE model, partitioned to run over one theatre or multiple theatres, where effective algorithms are adopted in the message servers.
Other classes of the Java framework (see also later in this paper) are specialized to interfacing theatres to network services and to the management of global time by a time server through the exchange of special control messages.

8.2.2 Development Phases And Control Machines

Modelling and Analysis

A development starts by constructing an *abstract model* of a system for analysis purposes. Abstractions refer to message server methods in actors, that initially, necessarily, can only be characterized by timing information and not by detailed code, and to the distribution issues. By default, message scheduling and dispatching are assumed, during analysis, to consume no time. Moreover (see also below in this section) message servers, despite the use of delay operations, are considered atomic and instantaneous. Multiple theatres are abstracted by corresponding processing units (PUs). A PU serves a group of assigned (moved)

actors. A PU can be occupied (e.g., during the execution of a delay operation issued in a message server of an assigned actor) or free. A message can be dispatched to an actor, provided the corresponding PU is free. An abstract model can be simulated for property checking by the *AbstractSimulator* control machine. A subtle semantic point concerns the interpretation of the delay operations during analysis. Since adopted actors are not thread-based, a delay operation has an asynchronous meaning: it asks occupying the corresponding PU for a time amount equal to the delay. When the delay is expired, the PU comes back to the free state. As a consequence, during modelling and analysis, the after/deadline times of any message send which textually follows a delay operation, must include the delay amount (see Fig. 8.4). The following points recapitulate the behavior of the *AbstractSimulator* engine.

- At each message send the *after* and *deadline* relative times get absolutized: after+now(), deadline+now().
- Similarly, the duration time d of a delay is absolutized to an expiration time: d+now().
- When multiple events (message dispatches and/or delay expirations) can occur at the current time, one of them is chosen non-deterministically and executed (a message server is activated or the status of a PU is turned to be free).
- If no event can occur at the current time, the simulation clock is first advanced to the minimum occurring time of next events.
- The execution continues until a termination condition holds (e.g., a maximum number of time units are elapsed).

Preliminary execution

It is important to note that an abstract model represents an almost *complete* Theatre model where actors and their interactions are fully detailed. After modelling, the next development step aims at correcting some assumptions of the abstract model regarding the temporal overhead of message exchanges which in a real execution cannot be ignored and which can have an impact upon the timing constraints of the system. The so-called *preliminary development* phase is assisted by control machines (*Preliminary/DPreliminary*) which still act on an abstract model but use instead of the real-time not the simulated time. In a sense, both simulated and real time are employed. Event occurrences follow the same rules explained for the *AbstractSimulator*. The difference is that now when no event is eligible to occur at the current time, the control loop is void iterated only to allow the real-time to advance. The *Preliminary* control form executes the model on a single machine. It is useful to observe the deviations in time of message delivery with respect to their due times. During preliminary execution, processing units (PUs) are turned to be fake thread objects upon which the delay operations are executed as *sleep* operations. The *DPreliminary* control machine executes a model physically split among concrete theatres (JVM instances). It permits to monitor the overhead introduced by network communications. As for a real-time distributed execution, the clocks of the various theatres/cores can

be required to be kept aligned through some algorithm/tool (for an example see [73]).

Real-time execution

The last development phase is based on a concrete version of the Theatre model. In particular, message servers are detailed, delay operations are replaced by a concrete code, and a real-time control machine is used. Message dispatches occur as in the preliminary execution. Depending on the partitioning requirements, a concrete model can be run on a single theatre (see the *RealTime* control machine), or it can be configured to run in a distributed context (the *DRealTime* control machine is used). Distributed execution necessitates of real-time clock alignment of the involved machines on a regular basis. The use of Dimension4 [97] was successfully experimented in [73].
It is important to stress that a Theatre model, consisting of actors and message interactions, flows almost unchanged from a development phase to the next one. What changes are the exploited control machine and the adopted time notion. All of this tends to reduce the semantic gap existing from an implementation and its modelling counterpart which in different approaches can be far each other.

Message ordering and non-determinism

It has been said that control machines, see e.g. the simulated ones, depend on a non-deterministic selection when multiple events can occur at the same time. Non-determinism can be a natural hypothesis (which complies with the general Actors model [10]) when, instead of simulation, one would consider a model checker for property analysis. In addition, non-determinism can influence the modelling activity because one cannot rely on a particular order of message dispatches, e.g., respectful of causal/effect relationships.
However, both within a same theatre and also when actors which are belonging to physically distinct theatres interact through a $FIFO$ transport layer, it is natural to assume that if an actor A sends to an actor B a sequence of concurrent messages $< m1, m2, \cdots >$, B should receive the messages in the sending order. Non-determinism could apply, instead, to different senders which transmit messages to a common destination. In the actual implementation of the control machines provisions were taken to ensure that, at the same time, messages are received in the sending order. Each message is time-stamped with a Lamport logical clock (LC) [151] which counts message generations within a same theatre. LCs are adjusted when messages cross the theatre boundaries. At the same time, the message with the lowest LC is chosen. Obviously, this choice is perfectly compliant with non-determinism which would imply any selection to be correct among concurrent events.

8.3 A modelling example

In the following, the dependable real-time modelling example of the Toxic Gas System (TGS) shown in the Chapter 7 and adapted from [129], is directly programmed, checked and prototyped in Java. There is a research lab wherein a

scientist is working. In the lab environment a toxic gas level can accumulate to the point to require the scientist to immediately abandon the lab to avoid a fatal consequence.
The model consists of six actors: Environment, Sensor, Scientist, Rescue, Controller and the Main, whose behavioral specification is described in the Section 7.2.3. Java actor modelling is shown in the Fig. from 8.2 to Fig. 8.6). Global constants acting as *scenario parameters* are depicted in Fig. 8.2.

The Environment (Fig. 8.2) has a changing period after which a normal gas

```
public final class ScenarioParameters{
    public static final int SCIENTIST_DEADLINE=10,
        SCI_ACK_DEADLINE=3, RESCUE_DEADLINE=5,
        NET_DELAY=1, CONTROLLER_CHECK_DELAY=3,
        SENSOR_PERIOD=2, RESCUE_DELAY=2,
        CHANGING_PERIOD=5,
        NR_SENSORS=1;
}//ScenarioParameters
```

Figure 8.1: Class of scenario parameters

level (value 2) can be changed, with 0.98 probability, to a dangerous level (value 4). As soon as a dangerous level occurs, the Environment sends a die() message to the scientist whose *after* time reflects the maximum time within which the scientist should be saved (see the constant $SCIENTIST\_DEADLINE$ in Fig. 8.1).

```
public class Environment extends Actor{
    private Scientist sc; //acquaintance
    //state vars
    private int gasLevel;
    private boolean meetDangerousLevel=false;
    @Msgsrv public void init( Scientist sc ){
        this.sc=sc; gasLevel=2; //2=safe, 4=dangerous
    }//init
    @Msgsrv public void changeGasLevel(){
        if( gasLevel==2 ) gasLevel=Math.random()<0.98?2:4;
        if( gasLevel>2 && !meetDangerousLevel ){
            sc.send( SCIENTIST_DEADLINE, "die" );
            meetDangerousLevel=true;
        }
        this.send( CHANGING_PERIOD, "changeGasLevel" );
    }//changeGasLevel
    @Msgsrv public void giveGas( Actor sender ){
        sender.send( "doReport", gasLevel );
    }//giveGas
}//Environment
```

Figure 8.2: Environment actor class

The Sensor (Fig. 8.3a) periodically (see the $SENSOR\_PERIOD$ in Fig. 8.1) asks the gas level to the Environment and then transmits it to the Controller for a decision. The sensor, though, can possibly be not working (with probability 0.01) in which case it cannot transmit the gas level to the Controller. When a dangerous gas level is received by the Controller (see Fig. 8.5), it first

```
public class Sensor extends Actor{
    //acquaintances
    private Controller co;
    private Environment en;
    //state vars
    private boolean working;
    //message servers
    @Msgsrv public void init( Controller co, Environment en ){
        this.co=co; this.en=en;
    }//init
    @Msgsrv public void checkGasLevel(){
        en.send( "giveGas", this );
    }//checkGasLevel
    @Msgsrv public void doReport( Integer value ){
        working=Math.random()<0.01?false:true;
        if( working ){
            co.send( NET_DELAY, "report", value );
            this.send( SENSOR_PERIOD, "checkGasLevel" );
        }
    }//doReport
}//Sensor
```

(a) *Sensor actor class*

```
public class Scientist extends Actor{
    private Controller co; //acquaintance
    private boolean isDead, isOutEnv; //state vars
    @Msgsrv public void init( Controller co ){
        this.co=co;
    }//init
    @Msgsrv public void die(){
        if ( !isOutEnv ) isDead = true ;
        else isDead = false ;
    }//die
    @Msgsrv public void abortPlan(){
        if( Math.random()<0.90 ){
            if ( !isOutEnv && !isDead ){
                isOutEnv = true ;
                co.send( NET_DELAY, "ack" );
            }
        }
    }//abortPlan
    @Msgsrv public void leftEnv(){ isOutEnv=true; }
}//Scientist
```

(b) *Scientist actor class*

Figure 8.3: Sensor and Scientist classes

alerts the Scientist (Fig. 8.3b) to exit the lab environment through an abortPlan() message, a reply of which the Controller expects within a maximum allowed deadline ($SCI\_ACK\_DEADLINE$ constant in Fig. 8.1). Would the ack not arrive in time, the Controller proceeds by asking the rescue team to go physically to the lab and try to save the scientist. The Rescue (Fig. 8.4) informs the Controller when it reaches the scientist (see the rescueReach() message). For the example to be more realistic, the Rescue can experiment some delay (due to encountering some obstacles) when going to the lab, and the Scientist can (with 0.90 probability) or not perceive an abortPlan() from the Controller. The

```
public class Rescue extends Actor{
    private Controller co; //acquaintance
    @Msgsrv public void init( Controller co ){
        this.co=co;
    }//init
    @Msgsrv public void go(){
        delay( RESCUE_DELAY ); //unpredictable obstacle
        co.send( RESCUE_DELAY+NET_DELAY,
            RESCUE_DELAY+
            RESCUE_DEADLINE-NET_DELAY,
            "rescueReach" );
    }//go
}//Rescue
```

Figure 8.4: Rescue actor class

problem is to characterize quantitatively the model so as to predict the expected probability, under different operating conditions, with which the scientist would die or (hopefully) is saved in time.

In the Controller class (Fig. 8.5), the sciAlive instance variable reflects some status information received from the scientist. Its true value following an ack() message would testify the scientist was saved by the previously sent abortMessage(). However, its true value following the reception of a rescueReach() sent by the rescue when it reaches the scientist, only potentially expresses the fact that the scientist was rescued. In any case, the values of isDead and isOutEnv variables of the Scientist (see Fig. 8.3b) reflect the resultant scientist status.

The scenario parameters class in Fig. 8.1, is statically imported by each actor class so that constant values are immediately accessible. The UML sequence diagram in Fig. 8.7, summarizes some message exchanges among the actors of the TGS model in a situation where, following a dangerous gas level, the scientist does not perceive the *abortPlan*() and does not sent the *ack*() message (dashed line) to the controller. The scientist, though, gets finally saved thanks to the intervention of the rescue team.

The *Main* (Fig. 8.6) is modelled as a normal actor which can receive messages from the other application actors. It contains the main() method which configures the actor model and launches its execution. First actor instances are created. As one can see in the Fig. from 8.2 to Fig. 8.5, actor classes rely only on the default constructor. For the initialization, each actor expects an init(...) (or similar) message which carries any required initialization data. After the initialization, the model execution is started by sending a changeGasLevel() message to the environment, a checkGasLevel() to the sensor(s), and a checkSensors() to the controller.

```
public class Controller extends Actor{
    //acquaintances
    private Scientist sc;
    private Rescue re;
    //state vars
    private boolean danger, abortSent, sciAlive; //false by default
    @Msgsrv public void init( Scientist sc, Rescue re ){
        this.sc=sc; this.re=re;
    }//init
    @Msgsrv public void report( Integer value ){
        if( value>2 ) danger=true;
    }//report
    @Msgsrv public void rescueReach(){
        sciAlive = true ;
        sc.send( "leftEnv" );
    }//rescueReach
    @Msgsrv public void checkSensors(){
        if ( !sciAlive ){
            if( danger ){
                if ( !abortSent ){
                    sc.send( NET_DELAY, "abortPlan" );
                    this.send(   SCI_ACK_DEADLINE,
                                 "checkScientistAck" );
                    abortSent = true ;
                }
            }
            this.send(   CONTROLLER_CHECK_DELAY,
                         "checkSensors" );
        }
    }//checkSensors
    @Msgsrv public void ack(){
        sciAlive=true;
    }//ack
    @Msgsrv public void checkScientistAck(){
        if( !sciAlive )
            re.send( NET_DELAY, "go" );
    }//checkScientistAck
}//Controller
```

Figure 8.5: Controller actor class

```
public class Main extends Actor{
    @Msgsrv public static void main( String[] args ){
        Main m=new Main();
        Environment en=new Environment();
        Sensor se1=new Sensor(); Scientist sc=new Scientist();
        Rescue re=new Rescue(); Controller co=new Controller();
        ControlMachine cm=new AbstractSimulator(6,tEnd);
        //partitioning and configuration
        m.move(0); en.move(1); se1.move(2); sc.move(3);
        co.move(4); re.move(5); //maximal parallelism
        //initialization
        en.send( "init", sc ); se1.send( "init", co, en );
        sc.send( "init", co ); re.send( "init", co );
        co.send( "init", sc, re );
        //execution
        en.send( CHANGING_PERIOD, "changeGasLevel" );
        se1.send( "checkGasLevel" ); co.send( "checkSensors" );
        cm.controller();
    }//main
}//Builder
```

Figure 8.6: Main actor class

In Fig. 8.6 an instance of the *AbstractSimulator* control machine is created. *AbstractSimulator* receives the number of used processing units and the time limit for the simulation experiment. The control loop of *AbstractSimulator* is actually started by invoking on it the *controller()* method.
By default, all the created actors are assigned to one processing unit (PU) associated to the $Main$. The use, though, of $move()$ operations before any send operation, permits to establish a different configuration and partitioning of the actors to multiple PUs. In Fig. 8.6, the *maximal parallelism* configuration is defined where each actor is allocated onto a distinct PU. However, instead of 6 PUs, 3 PUs could also be used, by grouping together the Main, Environment, Sensor and Scientist on one PU, whereas the Controller and the Rescue are moved to two distinct PUs. All of this emerges from the message exchanges among actors, which is some cases have NET_DELAY as their after time, that is the assumed network communication delay.
The $main()$ method can easily be adapted for executing a batch of simulation runs. At each simulation, the model can be re-created and model information (e.g., the number of times the scientist was found dead in the simulation runs, or the number of times the scientist was saved by an $abortPlan()$ message (then without the recourse to the rescue team) or by a $leftEnv()$ message sent by the Controller following the intervention of the rescue) can be stored into a few additional *decoration* variables which can be put, e.g., in the ScenarioParameters class.

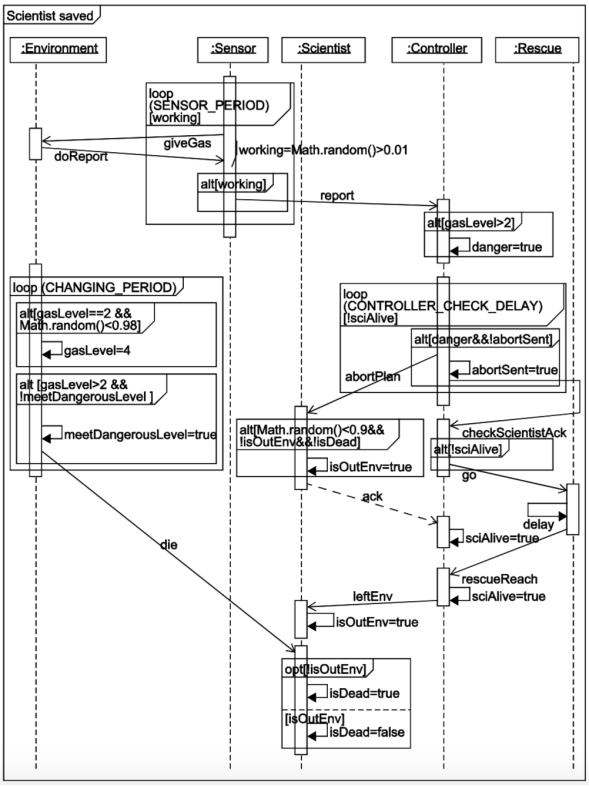


Figure 8.7: A pattern of message exchanges in the toxic gas system

8.4 Analysis of the toxic gas system

The abstract model of TGS was extensively analyzed in simulation using the *AbstractSimulator* control machine, e.g., under the maximal parallelism assumption.

8.4.1 Qualitative Experiments

30 simulation runs were preliminarily carried out using only one sensor with $SENSOR\_PERIOD = 8$, $SCIENTIST\_DEADLINE = 10$ and the other parameters set as in Fig. 8.1. Such parameter values were chosen by taking into account uncertainty and timed behavior of the system. Each simulation experiment lasts 1000 time units.
Fig. 8.8 shows the observed times the scientist dies after a dangerous toxic gas level, or it gets saved. The grey height of the columns reflects the instant in time the dangerous gas level occurs into each experiment. The top of each column then registers the corresponding scientist outcome.
The results can mirror the general cases where the controller is not informed of a dangerous level because the sensor was not working, and cases when the controller gets informed late about the danger and it is not possible to save the scientist within the remaining portion of the $SCIENTIST\_DEADLINE$.
Choosing 8 as the sensor period reduces the risk of the sensor to become not working (a sensor with a lesser sampling frequency has a higher probability to be working at each period). On the other hand, the controller acquires late a dangerous stimulus when the gas level change occurs immediately after (1 time unit after) the last reading of the sensor. As a consequence, the sensor will perceive the dangerous gas level at the next period and then informs the controller which starts the reaction. As one can see from Fig. 8.8, 1000 time units are

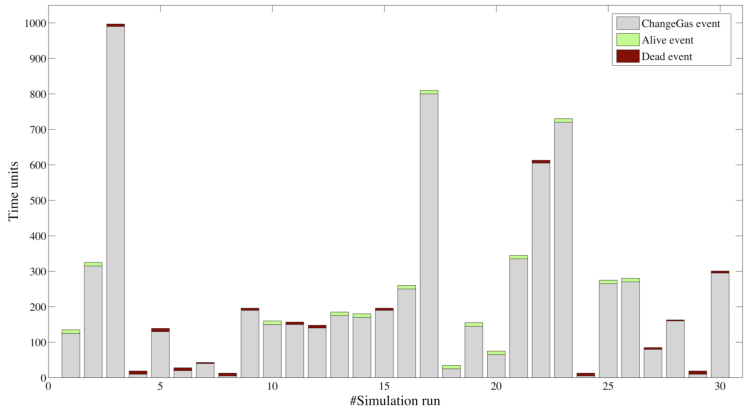


Figure 8.8: Trace of 30 simulation runs observing the scientist rescue

sufficient to trigger the occurrence of a toxic gas level. In addition, it clearly emerges that the scientist *can* die.
The next step was then to deepen quantitatively the timing behavior of the TGS model under various operating conditions. Critical parameters for the TGS model are the $SCIENTIST\_DEADLINE$, the $SENSOR\_PERIOD$, and the probability for a sensor to be working The value of the $SCIENTIST\_DEADLINE$

can be dictated from the physical system, that is the lethal characteristics of the toxic gas. In the following, the performance curves refer to an experimental frame where the time limit for each simulation is tEnd=5000 time units. Each execution consists of a batch of 5000 simulation runs and the scientist die probability is inferred by Monte Carlo-like techniques.

8.4.2 First Scenario: 2 Sensors and Scientist Deadline set to 10

The first scenario uses 2 instances of the Sensor actor (Fig. 8.3a), and estimates the probability of the scientist to die or to be saved, when the sensor period is varied from 1 to 20 and the $SCIENTIST\_DEADLINE$ is 10.
Fig. 8.9 shows the observed probabilities. Both the probability to die (red curve) and the complementary probability to be saved, distinguished in the two components of first level intervention of the controller through an abortPlan() message (blue curve), or through a leftEnv() message which follows to a rescueReach() message (green curve) sent by the Rescue to the controller, are depicted.
From Fig. 8.9 it emerges that when the scientist is saved it mainly occurs without the intervention of the rescue team (the green probability curve remains always very low).

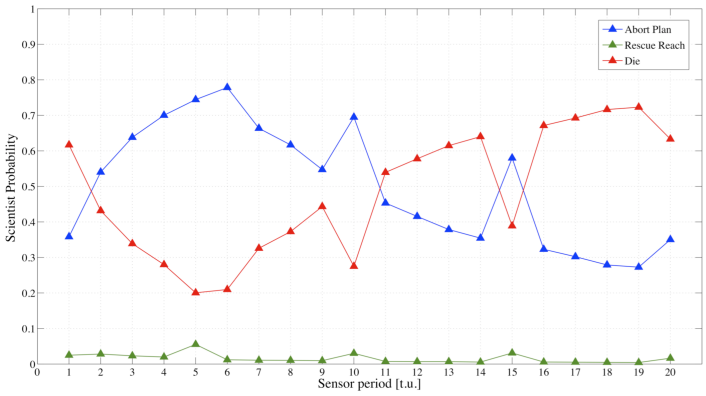


Figure 8.9: Scientist probabilities in the first scenario

An interesting and not obvious behavior in Fig. 8.9 concerns the evolution of the die probability of the scientist (red curve) or, dually, its complement of the saving probability (see the blue curve) of the scientist. The die probability is high when the sensor period is low. This is due to the fact that despite the use of two sensors, a high frequency of reading raises the probability for sensors to be not working. As the sensor period increases, the die probability diminishes by reaching a minimum at the abscissa 5. After the minimum, the die probability tends to augment as the sensor period continues to be increased. For higher sensor periods, the die probability is again high because of the problem of late informing the controller about a dangerous gas level, although a great sensor period reduces the risk for the sensors to become not working.

In reality, in Fig. 8.9 different local minima exist for the die probability. They correspond to points where the sensor period is a multiple of the environment $CHANGING\_PERIOD$ which is 5, that is at abscissas 10, 15 and 20. In these points, it becomes possible for the sensor(s) to read a dangerous gas level as it occurs in the environment and to promptly transmit it to the controller. Of course, as the sensor period increases from 10, to 15, to 20, there is the problem of having less time to respond with respect to the $SCIENTIST\_DEADLINE$.

8.4.3 Second Scenario: 2 Sensors and Scientist Deadline set to 13

The same experiments used for deriving Fig. 8.9 were repeated when the $SCIENTIST\_DEADLINE$ is raised from 10 to 13. The results are shown in Fig. 8.10.

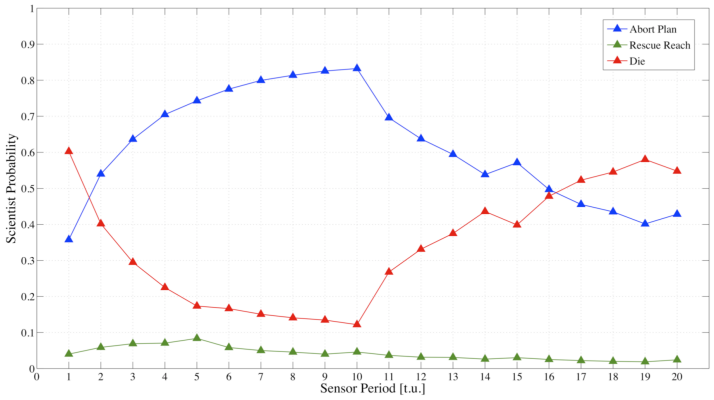


Figure 8.10: Scientist probabilities in the second scenario

Fig. 8.10 conserves the same trend observed in Fig. 8.9 but now the die probability values are smaller because there is more time for the controller to respond to the environmental critical stimulus. The minimum of the die probability moves from 5 to 10 as the sensor period. For the sensor period 5 the behavior is not significantly different form near values of the sensor period, always due to the greater value of the $SCIENTIST\_DEADLINE$. As the die probability diminishes, the saving probability by first (abortPlan()) or second (rescueReach()) intervention obviously increases.

8.4.4 Third Scenario: 3 Sensors vs. 1 Sensor, 95% Working

In order to study the effects of using multiple sensors, the TGS model was investigated by using 3 sensors vs. 1 sensor but lowering from 99% (see Fig. 8.3a) to 95% the probability for a sensor to continue to be working, and increasing from 1% to 5% the probability of becoming not working. The used $SCIENTIST\_DEADLINE$ is 16. Fig. 8.11 shows the collected results.

Fig. 8.11 confirms the expectation that 3 sensors allows the TGS model to behave better than the case where 1 sensor is used, even when the probability of

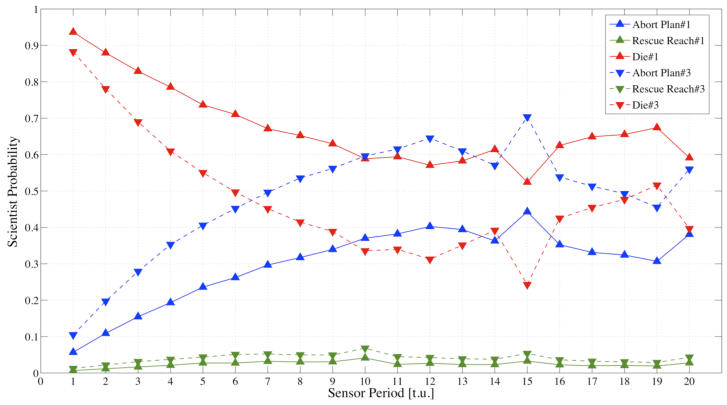


Figure 8.11: Scientist probabilities when 3 vs. 1 sensors are used, 95% working

becoming not working is raised from 1% to 5%. In fact, although the not working probability is higher, the probability that all the 3 sensors are simultaneously out-of-order is low.
As one can see from Fig. 8.11, the probability of saving the scientist through the rescue team is not significantly different when passing from 1 to 3 sensors. The minimum for the die probability now occurs when the sensor period 15 is used.

8.4.5 Fourth Scenario: Scientist Die Probability vs. Number of Sensors

Since it is of utmost importance rescuing the scientist when a dangerous gas level occurs, also considering the very low cost, today, of sensor devices, in this scenario the $SCIENTIST\_DEADLINE$ was kept to 15, the $SENSOR\_PERIOD$ was set to 4 with the working probability being 99% as in Fig. 8.3a, and the number of sensors was varied from 1 to 20 and then 30. The gathered scientist die probability results are portrayed in Fig. 8.12.
It emerges from Fig. 8.12, that when the number of sensors is increased, the die probability for the scientist sharply decreases. With 30 sensors, the observed die probability is less than 1% (about 0.005).
As a final remark, all the documented experimental results achieved with the maximum parallelism, were also confirmed with a fewer number of PUs.
For example, when only one sensor is involved, three PUs were configured as in the following:

```
m.move(0); en.move(0); se1.move(0);
sc.move(0); re.move(1); co.move(2);
```

8.5 Preliminary Execution of the TGS Model

After modelling and analysis, the proposed methodology continues by checking, in real time, the effects of the inevitable overheads introduced by message

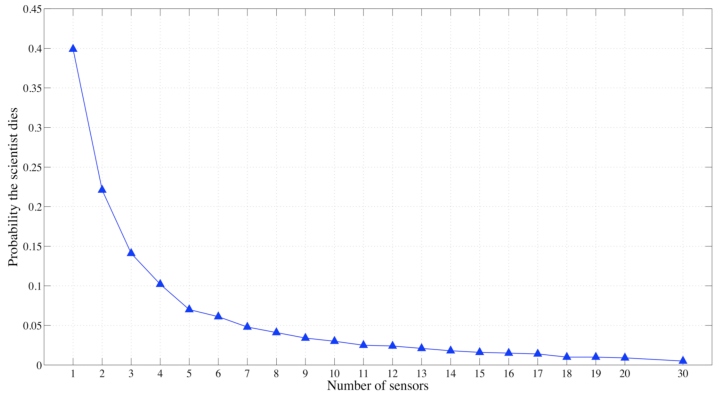


Figure 8.12: Scientist die probability vs. number of sensors

scheduling, dispatching, network communications and message server executions, on the model timing constraints.
The TGS model was evaluated using the *DPreliminary* control machine on a concrete distributed context constituted by three theatres configured (see the next section for the configuration details) for the execution on two computing nodes (High Sierra OS, MacBookPro Retina 2.9GHz 16GB and MacBookAir Retina 2.4GHz 8GB) in the presence of a wireless communication network sharing Internet traffic, nominally operating at 5Gb per second. The Main, environment, sensor and scientist are put on one theatre. The controller and the rescue are each allocated on a distinct remaining theatre. The two involved physical machines are initially time aligned and the execution of each experiment started at an absolute date/time.
The time behavior of the TGS model was first observed in real time execution by setting the NET_DELAY scenario parameter (see Fig. 8.1) to 0, so as to affect the model by real communication delays. A number of runs were carried to monitor specifically the scientist die probability. Each real experiment lasts on the average 15 minutes. Only one sensor was used, with a $SENSOR\_PERIOD$ of 4 and a $SCIENTIST\_DEADLINE$ of 15. After 30 runs of real execution (1 time unit=1 sec) it emerged a die probability of 0.41, confirming the previously achieved value shown in Fig. 8.12. After all, despite the involvement of real communications, the timing of the model is exactly that considered during the analysis phase in simulation.
In order to better qualify/quantify the effects of the message overheads on the overall system timing, some experiments with *DPreliminary* were also performed using two different configurations. The goal was capturing the *time deviations*, that is the drifts with which messages are finally dispatched in real-time with respect to their *due* time (absolutized after times). Time deviations include the network delays, message server body executions, bookkeeping times of the control machine and message interleaving within a same theatre.
In Fig. 8.13 the monitored deviations are reported when (first case) exactly the abstract model is executed over the chosen physical architecture. As one can see from Fig. 8.13, the great majority of messages were found to be "unaffected" by time deviations (it should be noted that on common operating systems, 1ms is the time resolution, so lower times are virtually 0). Only in one case the

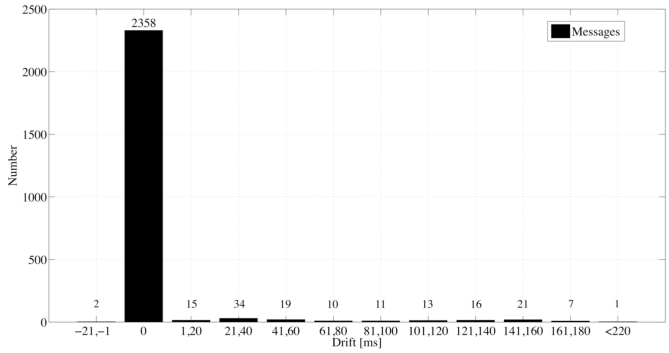


Figure 8.13: Time deviations - First case

worst case measured drift was about 212ms. In a few cases the drift was found to be virtually negative. These cases correspond to clock misalignment among the computing machines. From the experimental log it emerged that the great drifts are related to network messages received by the less performing machine. In the second scenario the TGS model was run on the same actor/theatre configuration as in the first scenario, but now the parameter NET_DELAY (see Fig. 8.1) was set to 0 (thus converting network messages to instantaneous messages). The goal was to observe the impact of real communication delays on the model time behavior. Fig. 8.14 depicts the monitored drifts in the second case. In Fig 8.14, the number of messages with a 0 drift is lesser than that in

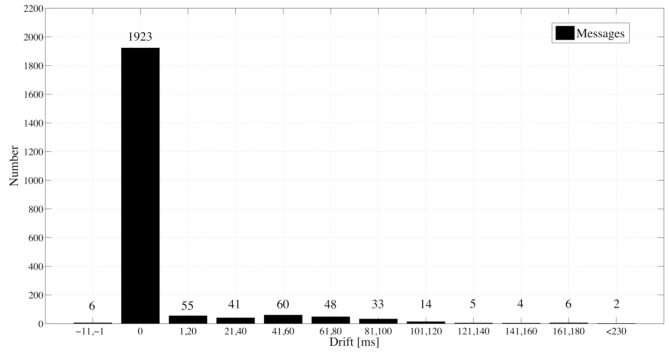


Figure 8.14: Time deviations - Second case

Fig. 8.13, because messages with the original $NET\_DELAY = 1 = 1000ms$ naturally hide the execution drift within their absolute dispatch time, whereas when they are converted into instantaneous messages the drifts are inevitable shown because of the immediate dispatch.
In the third case (see Fig. 8.15) the environment actor is moved to the same theatre of the controller and the net delays mirror the real communication delays. In this configuration the number of messages sent across the network increases. As a consequence, the number of messages with a 0 drift remains almost the same as in Fig. 8.14, but the intermediate values of time deviations are greater than those observed in Fig. 8.14. The documented results in the figures from 15 to 17 confirm that the observed drifts on a real physical architecture remain

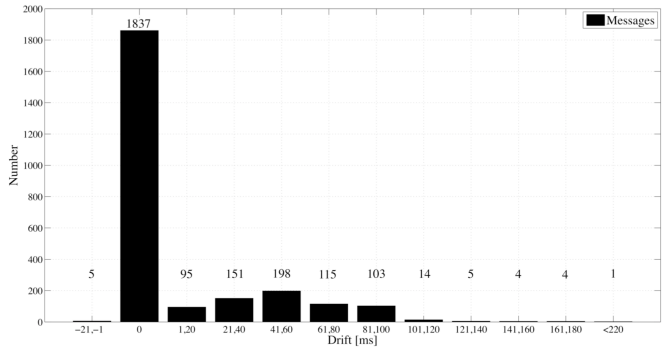


Figure 8.15: Time deviations - Third case

less than 1/4 of the time unit assumed in the TGS model. In other terms, the time deviations do not impair the system timing constraints.

8.6 Real time execution

For the purposes of the chosen case study, the preliminary execution is indicative of what happens when a Theatre model is put into real-execution. Transforming the TGS model for final implementation requires concretization of message servers and the use, e.g., of the *DRealTime* control machine with a time server in charge of keeping regularly aligned the clocks of the physical machines hosting the theatre nodes (an experience is reported in [73]). However, besides any change to the model structure (the Environment actor could be omitted by having those sensors directly and periodically read from the physical environment) an important concern is interfacing the cyber software part of a system with the sensors/actuators devices of the controlled physical plant, and keeping aligned (a challenging issue) the Newtonian time of the physical part with the discrete time of the cyber part.
In [73] the concept of an *EnvGateway* was proposed to properly abstract the cyber part from the details of contacting and carrying read/write operations and associated protocols on the input/output devices of the physical part.
In particular, the operations of interfacing and managing peripheral devices (such as sending signals to actuators, triggering the sensing operations or making available the sensed data) are entrusted into IOHardware components such as Arduino, Raspberry Pi and other similar boards. Using wired or wireless communication protocols, these boards allow the bidirectional information transfer, also providing the routing and transformation of the high-level commands coming from the *EnvGateway* to a specific device, into signals compatible with its functioning.
The *EnvGateway* works according to an anticipative schema. It asynchronously receives sensors data (as programmed, e.g., in the periodic loop of Arduino [151]) and store them into corresponding variables (precisely, in a ConcurrentHashMap) of the *EnvGateway*. Such values are subsequently retrieved by actors of the cyber part according to the application timing model. It is guaranteed that application actors always get the more recent read values and not stale values.

As a final remark, it is worth noting that being the chosen case study a modified version of a similar example described in [129], it was not possible to directly compare the experimental results reported in this paper with those of [129]. Moreover, whereas the approach proposed in this paper addresses all the development phases and includes preliminary and real-time execution of a system, the work described in [129] seems more related to only modelling and analysis aspects without an explicit indication to the design and implementation phases. For validation purposes, the modelling case study was reduced into Uppaal (see [196] for details of the reduction process) and the analysis results re-achieved using the Statistical Model Checker of Uppaal [90].

8.7 Theatre Implementation Status

Theatre is currently implemented in standard Java as a framework (see Fig. 8.16), that is a collection of classes plus an *event flow* which means that a programmer-defined class directly or indirectly inheriting from the Actor base class automatically and implicitly participates to the event flow. The event flow is realized in a control machine (a class specializing the base class *ControlMachine*) and consists in the activation of message servers (message dispatches) in user-defined actors through reflection (method activation service). Message

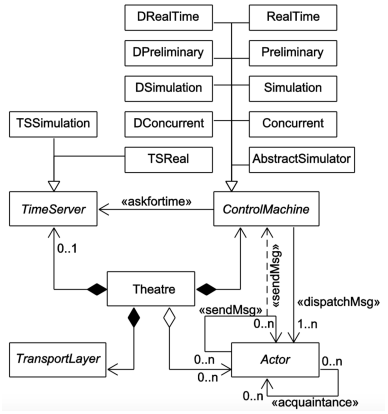


Figure 8.16: A Theatre UML Class Diagram

sends are specified to occur among acquaintance actors but in reality message objects, reflectively built behind the scene of a theatre through the services of the Actor class, get captured and managed (scheduled) by the control machine chosen for regulating the local actors. Control machines act as plug-in components: they are transparently installed in the various theatres. The Actor class loaded in the JVM of a given theatre, knows the adopted control machine. The design logic of the provided control machines tailored to the PTRebeca timing model [224, 127], was anticipated in section 8.2. In the following some in-the-large aspects, that are related to theatre configuration, bootstrap execution, and (possibly) dynamic re-location of actors, are summarized.

8.7.1 Configuration and Bootstrap of a Distributed System

A distributed Theatre system is configured through an XML file like the one reported in Fig. 8.17. The file contains, among other, information about the list of the theatres composing the system. For each theatre the following data are specified: (i) its unique name, (ii) its IP address and port number; (iii) the type of the required control machine; (iv) a boolean value indicating whether the theatre should provide the time server (at most one time server can exist in a system), and (v) the class name (a string value) of the Main (see Fig. 8.6) which will be executed by the *master* theatre in order to startup the application (one single master is admitted in a system).
Beside the above information, the XML file also contains the address of the class server hosting the Java classes of the application to execute, and a date information indicating when the configured application has to startup. This date information can be exploited, for synchronization purposes, by control machines that use a real time notion (i.e., *Preliminary*, *DPreliminary*, *Realtime* and the *DRealtime* control machines, see Fig. 8.16).
The class server is a network repository from which Java classes can be retrieved at runtime.
The THEATRE configuration file is feed to each theatre so as to permit system bootstrap. System bootstrap consists of the following phases: (a) each theatre begins by properly establishing socket connections with all the peers listed in the configuration file. Managing such connections is a responsibility of the *TrasportLayers* components (see Fig. 8.16); (b) all the control machines are created and, in the case a virtual time notion is used (see *DSimulation* in Fig. 8.16), the time server is instantiated too; (c) the master theatre executes the Main which creates/configures/migrates applicative actors so as to startup the application. The time server can also be required in a distributed real-time application (see *DPreliminary* and *DRealTime* control machines in Fig. 8.16) to supervise clock alignment of the various theatres.

In particular, during the socket network setting up phase, theatres will play, following directions of the master theatre, the server or client socket role for establishing the socket connections. Moreover, separate input/output threads are associated to each socket connection (see Fig. 8.18). Concurrency control problems among the input/output threads and the control machine thread are avoided by the adoption of lock-free data structures. Incoming messages get stored into one input buffer from where they are extracted at each iteration of the control loop of the control machine. External Output messages are stored into output buffers, one per socket connection, with the final transmission being responsibility of the socket thread.
A protocol based on special/control messages (see Fig. 8.18) is defined for communications among control machines and the time server in order to ensure correct advancement of the global time. For example, the protocol adopted by *DSimulation* rests on a conservative synchronization algorithm [105], which exploits, for each theatre, the counter values of sent/received messages/actors for detecting possibly ongoing transmissions. All the actors, either created or migrated to a given theatre, are automatically added to the so called *Local Actor Table* (LAT), that is the local registry for actors. System bootstrap completes as the Main ends its execution. Fig. 8.18 summarizes a running distributed The-

```
<?xml version="1.0"?>
<theatreList>
   <startDate>
      <minute>08</minute>
      <second>15</second>
      <millis>00</millis>
   </startDate>
   <classServer>
      <ip>192.168.1.27</ip>
      <port>9091</port>
   </classServer>
   <theatre>
      <name>Theatre1</name>
      <master>it.unical.GasAppBuilder</master>
      <timeServer>false</timeServer>
      <ip>192.168.1.7</ip>
      <CMType>DPreliminary</CMType>
   </theatre>
   <theatre>
      <name>Theatre2</name>
      <timeServer>false</timeServer>
      <ip>192.168.1.4</ip>
      <port>8889</port>
      <CMType>DPreliminary</CMType>
   </theatre>
   <theatre>
      <name>Theatre3</name>
      <timeServer>true</timeServer>
      <ip>192.168.1.7</ip>
      <port>8989</port>
      <CMType>DPreliminary</CMType>
   </theatre>
</theatreList>
```

Figure 8.17: Example of a theatre-config.xml file

atre system consisting of N theatres. The figure highlights connections among theatres, messages in transit across the network, and the actors in the system. More in particular, black circles denote actors executing in a given theatre, white circles correspond to actors that have leaved an originating theatre, and gray circles show actors that are in the phase of migrating toward a destination theatre. More details about actor migration are provided in the next subsection.

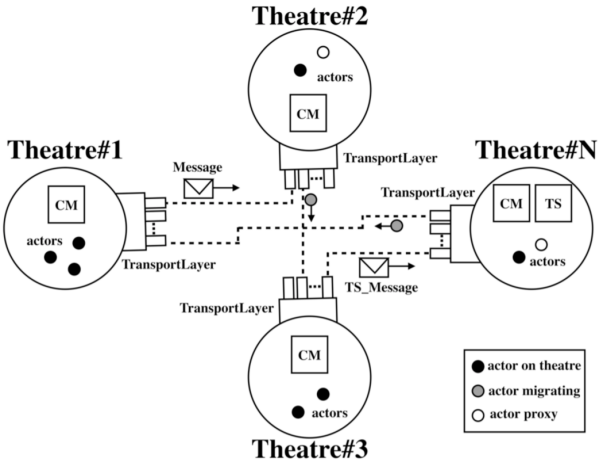


Figure 8.18: Multi-theatre configuration with socket channels

8.7.2 Actor Migration

During its life cycle, an actor can move from one theatre to another. For example, during initial configuration an actor can be moved to a particular theatre where some physical devices are attached so as to be locally and directly controlled. In addition, more in general, dynamic re-configuration can be required for load-balancing issues.
When an actor leaves a theatre it is not definitively removed from the theatre. Rather, the actor gets automatically turned into a *proxy* version of itself. A *proxy actor* is a place holder of the original one and acts as a message *forwarder*. It keeps information about the last known destination theatre toward which the actor moved when leaving the current one. When a message is directed to a proxy actor, the message is transparently forwarded, through the network, to the remote theatre indicated in the proxy. Since potentially an actor can move multiple times, different proxies can exist, at a given time, of the same actor on different theatres, and multiple hops can be required to deliver the message to the migrated actor. During its movements, if an actor comes back to a theatre where a proxy version of itself exists, the corresponding proxy is upgraded to the normal version, and its state transparently replaced with the one carried by the incoming actor.
To specify operations that an actor should perform just before leaving a theatre and immediately after it reached the destination one, the Actor abstract class makes it possible to override the *beforeMove()* and *afterMove()* methods, which are automatically invoked by the THEATRE runtime system.
Actor mobility rests on a *weak migration* semantics [26]. Since an actor does not have an internal thread, its *move()* operation reduces to Java serialization/deserialization of its state variables.
In order to avoid a *move()* operation to interfere with current message processing, a migration request is implemented by scheduling an instantaneous *Move* control message, that is subsequently dispatched by the control machine.

Chapter 9

Case Studies[1]

This chapter reports some case studies about modelling and verification of Theatre systems. The first two examples are based on the use of Uppaal SMC only. In particular, the first one models messages through dynamic template processes. The third example applies Theatre to formal modelling and reasoning on knowledge and commitments in general multi-agent systems.

9.1 Asynchronous Leader Election

The problem is considered of electing a leader in a ring of N not distinguishable processor nodes. A solution is challenging when nodes behave asynchronously [126]. The following illustrates an untimed actor model adapted from [251], which elects a leader probabilistically, i.e., the leader can be elected with a certain probability. As a scenario parameter, the number N of the nodes composing the ring is established by an declaration (see Fig. 9.1).
Each node knows only its right neighbor in the ring, communicated at the initialization time, and can send messages only to it. Two basic messages are *elect* and *leaderElect*. During the trying process of leader election, the *elect* message is exchanged between neighbor nodes. As soon as a leader emerges, the *leaderElect* message circulates instead in the ring, to terminate the algorithm.
A node can be active or not active. Initially, all the nodes are active. Node behavior starts by flipping a coin ($false$ is head, $true$ is tail) and sending its result to the right partner through an *elect* message. If a node achieves a head and its predecessor got a tail, the node ceases to be active. Otherwise, the flipping process is repeated. An inactive node simply forwards received messages to its right neighbor.
Fundamental to the model operation is keeping up to date the counter of active nodes in the ring. Initially, each node sets its *nActive* local variable to N. Subsequently, on receiving an *elect* message, which carries the flipping result and the active count of which the predecessor is aware, a node which finds the value of its *nActive* is 1, becomes the leader and starts circulating the *leaderElected* message. Otherwise, the node (possibly) updates its *nActive* from the prede-

[1]The material in this chapter is related to publications [193, 197, 196, 186]

cessor value or it decrements its counter if it becomes not active due to the flipping process. In the latter case the new value of $nActive$ is transmitted to the right partner. The model is completed by a $Main$ actor which creates N $Node$ instances and initializes them according to the ring topology. Other details should be self-explanatory from Fig. 9.1.

```
env int N=3; //demo

actor Node{
  Node rNeighbor; //acquaintance
  boolean leader, active, x;
  int nActive;

  msgsrv init( Node rN ){
    rNeighbor=rN;
    nActive=N; active=true;
    leader=false; self.flip();
  }
  msgsrv flip(){
    x=?(false,true);
    rNeighbor.elect( x,nActive );
  }//flip
  msgsrv leaderElected(){
    if( !leader ){
      active=false;
      rNeighbor.leaderElected(); //propag.
    }
  }//leaderElected
msgsrv elect( boolean i, int n ){
  if( active ){
    if( nActive==1 ){
      leader=true; rNeighbor.leaderElected();
    }
    else{
      if( nActive>n ){ nActive=n; }
      if( !x && i ){ active=false; nActive--;
        rNeighbor.elect( x, nActive );
      }
      else{ self.flip(); }
    }
  }
  else{ rNeighbor.elect(i,n); }//forward msg
 }//elect
}//Node
actor Main{
  main(){
    Node n0=Node(), n1=Node(), n2=Node();
    n0.init( n1 ); n1.init( n2 ); n2.init( n0 );
  }//main
}//Main
```

Figure 9.1: A THEATRE asynchronous leader election model [251]

Fig. 9.2 shows the $Node$ actor template, developed by using UPPAAL SMC, parameterized by $const\ aid\ self$. Since the analysis depends on the evalua-

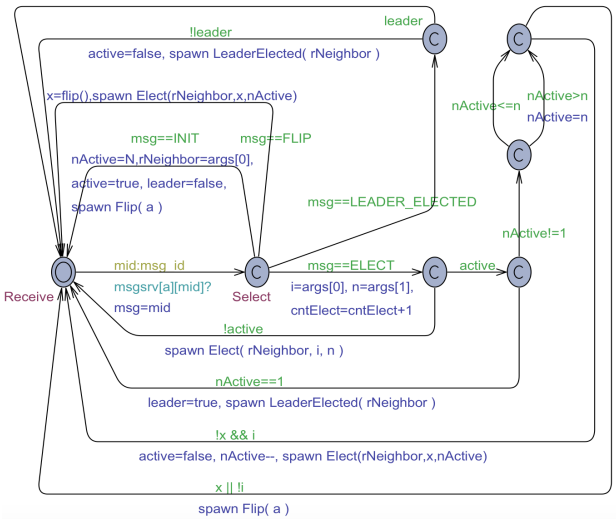


Figure 9.2: $Node$ actor template for the Asynchronous Leader Election model

tion of only stochastic queries, exchanged messages are modelled as *dynamic automata* scheduled by a *spawn* command, whose global declarations is illustrated in Fig. 9.3. It should be noted that a reaction could be fully realized in a local function (see the $FLIP$ message and the $flip()$ function in Fig. 9.4 which

```
dynamic Init( const aid a, const aid rN );
dynamic Flip( const aid a );
dynamic Elect( const aid a, const bool i, const int n );
dynamic LeaderElected( const aid a );
dynamic InitManager();
```

Figure 9.3: Basic global declarations of the Asynchronous Leader Election model

realizes a non-deterministic assignment) of the automaton. However, since the tool forbids a *spawn* command to be put in a function, it can be necessary to split the reaction in a sequence of committed locations, as for the $ELECT$ and $LEADER\_ELECT$ messages.

```
bool flip(){
  if( random(1)<=0.5 ) return true;
  return false;
}//flip
```

Figure 9.4: The $flip()$ function used in Fig. 9.2

9.1.1 Experimental results

The model was analyzed under maximal parallelism hypothesis. Being untimed, the model was checked by a given number (#) of steps (or transitions). The following query was used to check that the leader, if elected, is at most one:

```
Pr[#<=50000](<> (sum(i:aid)Node(i).leader)>1)
```

This query, in particular, asks to quantify the probability of the event that multiple nodes have their *leader* field which becomes true (a safety property). Uppaal SMC, through 36 runs, responds that such probability has a confidence interval (CI) of $[0, 0.0973938]$ with a confidence degree of 95%. Therefore, the event has a very low occurrence probability. After that, the probability of electing a leader was estimated with the query:

```
Pr[#<=50000](<> (sum(i:aid)Node(i).leader)>0)
```

Such a query was launched for N varying from 2 to 6 and the corresponding lower and upper bounds of the emerged CI 95% are as shown in Fig. 9.5. As expected, for $N = 2$ the leader election is almost certain. The probability diminishes as N increases. The worst case number of required simulation runs was 400 and relates to $N = 4$.

9.2 A Time Synchronization Algorithm

Due to technological advances, miniaturization, low power and a favorable function/cost ratio, more and more wireless sensor networks (WSNs) [239, 229] get employed for monitoring purposes of a given environment/territory. Tasks assigned to a WSN include: object tracking, temperature and climate control, data fusion and more in general measurement and sensing applications. However, for proper operation, the sensor nodes of a WSN require to be *time synchronized*,

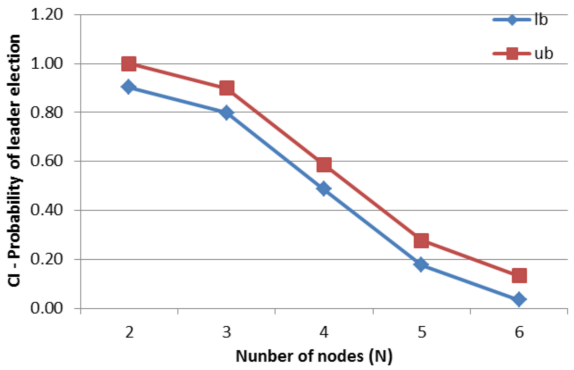


Figure 9.5: Probability of leader election vs. the number N of nodes

that is a global common time notion to be enforced.
Each node is typically equipped with a given energy (battery), radio support for communications, and hardware clock which is characterized by its frequency and phase. The use of sensor nodes always aims at prolonging to the largest extent the network lifetime, e.g., by node sleep periods and wakeups. The various hardware clocks are unsynchronized each other and can experiment the known phenomena of frequency drifts and/or differences in their offsets.
The clock synchronization problem [239, 216] refers to building and maintaining a *software clock* synchronization, that is a *reference* logical global clock which can guide sensor operation. The reference time can help to identify causal relationships between events of the physical world, identify and remove redundant data, synchronize sleep periods and so forth. More in general, it is commonly accepted that for it to be useful, sensor data should be accompanied by sensor position and timestamp.
Different algorithms are described in the literature for clock synchronization [239, 216]which can be roughly classified as hierarchical or fully distributed. In a hierarchical solution, the nodes of a WSN are tree-structured and child-parent relationships are exploited to compensate clock drifts (skew) and offsets. However, hierarchical solutions suffer from node failures. In the case a parent node becomes unreachable, a subnet can be excluded from the clock synchronization until a new organization of the WSN is re-established.
Fully distributed solutions tend to be more robust with respect to topology changes, that is node exiting and entering at the runtime the WSN. In addition, distributed solutions are normally based on asynchronous messages for carrying the clock adjustment operations. Only when a message is received from a neighbour, a given node updates its logical (software) clock.
Available solutions to clock synchronization can be characterized by their synchronization accuracy, robustness to message losses and so forth.
In this work, a simulation approach is advocated for deep analysis of an algorithm for clock synchronization in WSNs. The chosen algorithm [229, 150] is fully distributed and has a natural multi-agent semantic interpretation. Clock synchronization represents an emerging property of the WSN. An adaptation of the selected algorithm is proposed which tries to conserve energy during time re-synchronizations.

Synchronization algorithms are generally based on a *virtual clock* concept. In fact, although each node owns an internal clock, determined by the number of the crystal oscillations, and increased periodically as:

$$clk_{hw}(t_k) = \lfloor f t_k \rfloor + o \tag{9.1}$$

where f is the oscillation frequency and o is the offset due to the uncertainty of f, *hardware clocks* synchronization it is not feasible because the pair (f, o) is non-tunable, is time varying and is different from node to node.
The Average TimeSynch (ATS) algorithm, proposed in [229, 150], bases its synchronization procedure on a *software clock* tuning, using the linear function:

$$clk_{sw}(t_k) = a(t_k) clk_{hw}(t_k) + b(t_k) \tag{9.2}$$

where $(a(t_k), b(t_k))$ are *corrector parameters*, adapted following a synchronization message received from another node of the network. It has been proved that the ATS converges within a certain finite time t, whose amount depends on the network topology used. The ATS algorithm is totally asynchronous, fully

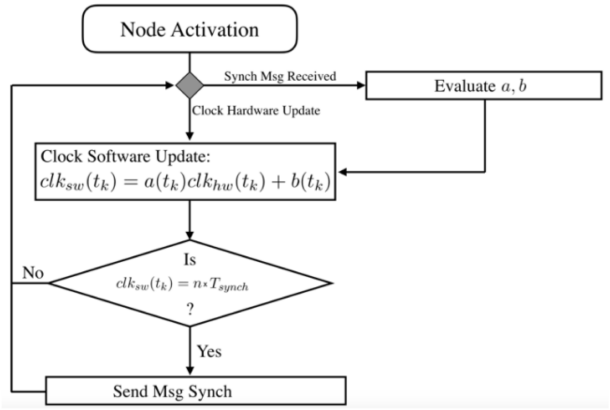


Figure 9.6: Block scheme of the ATS algorithm performed by each node

distributed and based on the consensus procedures [229, 150]. No centralized node is required to manage the operations, because each node adapts its own clk_{sw} value using information coming from its neighbors, without other mediations. Moreover, ATS is resilient to packet loss, node failure, replacement or relocation and requires minimal memory and computational resources.
Fig. 9.6 summarizes the synchronization procedure under the hypothesis that:

- each node has a unique identifier (id);
- each node is equipped with an omnidirectional radio antenna, that supports broadcast message sending;
- each message communicated through the network is supposed to be instantaneous: no transmission and propagation delays are considered;
- the period T_{synch}, used by each node to trigger the synchronization message exchange, is fixed.

The ATS algorithm works by updating the *software clock* values, by applying into each node the equation 9.2 every time one of two possible events occurs:

- the hardware clock value is increased;
- a synchronization message is received and processed.

When the *software clock* reaches a value that is a multiple of the synchronization period, i.e.:

$$clk_{sw}(t_k) = nT_{synch} \qquad n \in \mathbb{N} \tag{9.3}$$

the node reacts by broadcasting a synchronization message with data the neighbors require to correct their $(a(t_k), b(t_k))$ parameters. The initial values of these parameters are (1,0). The data transmitted by a *sender node* (s) are: id_s, $a_s(t)$, $b_s(t)$, $clk_{hw_s}(t)$. When a *receiver* node (r) receives a synchronization message from a *sender* (s), it updates its corrector parameters. The a parameter is modified as follows:

$$a_r(t_k^+) = \rho a_r(t_k) + (1-\rho)\frac{clk_{hw_s}(t_k) - clk_{hw_s}(t_{k-1})}{clk_{hw_r}(t_k) - clk_{hw_r}(t_{k-1})} a_s(t_k) \tag{9.4}$$

The contribution of the neighbors' node clocks to the overall alignment is due to the ratio part of 9.2, between the *hardware clock skew* of the sender and the *hardware clock skew* of the receiver, evaluated at the instant of reception of the current t_k and previous t_{k-1} synchronization message. The ρ constant, whose value belongs to the range $[0, 1]$, is in charge of modulating the weight of the correction influence due to the values of the sender node data.
As implied by 9.2, every node can store the parameters coming from each distinct neighbor in a row of a local matrix.
The b paramater value is updated according to the equation:

$$b_r(t_k^+) = b_r(t_k) + (1-\rho)(clk_{sw_s}(t_k) - clk_{sw_r}(t_k)) - (a_r(t_k^+) - a_r(t_k)))clk_{hw_r}(t_k) \tag{9.5}$$

In this case the contribution of a neighbor' sender node clock to the correction, is specified by the *software clock skew* $clk_{sw_s}(t_k) - clk_{sw_r}(t_k)$ between sender and receiver. To reduce the amount of the exchanged data, the $clk_{sw_s}(t_k)$ is evaluated by the receiver using the 9.2.
After the achievement of the first WSN clock alignment, an adaptation of the ATS algorithm is introduced for energy saving, that dynamically evolves the T_{synch} parameter so as to reduce the amount of exchanged messages.

9.2.1 Modelling the Time-Synch algorithm using THEATRE

The ATS algorithm was modelled as shown in Fig. 9.7, which refers to a single WSN node, under the hypothesis of *maximal parallelism* (one actor node per theatre/PU). The model is flexible enough to work with different WSN topologies, by specifying (into a matrix) the neighbors for each node. The full mesh topology can be defined by connecting each node to the remaining $N-1$ nodes, if N if the network size.
The WSN model is bootstrapped by the Main actor (see Fig. 9.8) which initially moves each node onto its PU and sends them the $INIT\_HW$ message which simulates the physical activation of the nodes. Each node responds to $INIT\_HW$ by self-sending an $OFFSET$ message which really activates the

node after a random time within $2000ms$. Function $nM()$ reserves and returns the id of the first free message instance in the message pool.
As one can see in Fig. 9.7, the design of the Node automaton (whose only parameter is its id) is organized into three logical sub-parts associated respectively with the update of the *hardware clock, software clock* and *adaptation* mechanism. These sub-parts are mainly triggered into execution through self-sent messages. The hardware clock increases its value according to the HW_PERIOD which is set to $1ms$. The updating operations of the software clock are triggered either when a hardware clock update occurs (see the notify message HW_UPDATE which follows an HW_PERIOD dispatch) or a $SYNCH\_UPDATE$ message is received from a neighbor, which adjusts the corrector parameters a and b. The periodicity of software clock synchronization operations is $TSynch = 1000ms$. $TSynch$ defines the after attribute of the $SYNCH\_PERIOD$ message. Processing a $SYNCH\_PERIOD$ causes a broadcast send of $SYNCH\_UPDATE$ to all the neighbor nodes.
A subtle point is concerned with the interpretation of the next scheduled $TSynch$, which obviously is related to the (under evolving) software clock value owned by the node. Care was used into the model in Fig. 9.7 to dynamically adjust the after attribute of the current $SYNCH\_PERIOD$ message so as to mirror the time distance existing between the new value of the software clock and the current TSynch's after. In the case the value of the new software clock is beyond the currently scheduled TSynch, current synchronization phase gets lost. Purposely, though, to avoid scheduling of a new message, the existing $SYNCH\_PERIOD$ is simply time redefined.
Modelling the radio antenna in Fig. 9.7 implies that multiple synchronization

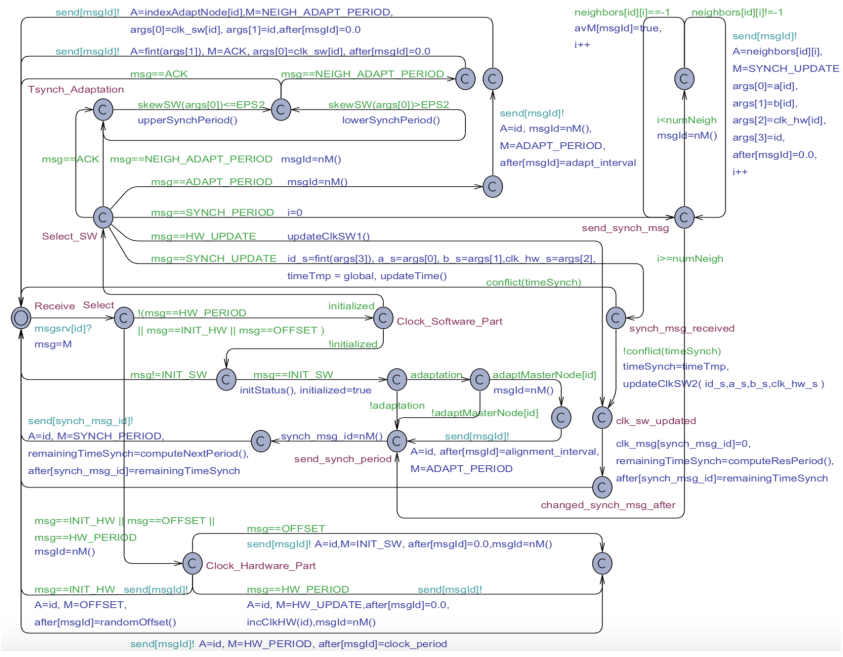


Figure 9.7: The actor *Node* automaton

messages arriving to a node at the same time (and thus conflicting each other)

get discarded (packet loss) except for the first one. Toward this, a *global* clock is introduced which provides the system absolute time.
It is worth noting that all clock comparisons (double values) in the UPPAAL SMC model of Fig. 9.7 are actually constrained, for precision granularity, to only two fractional digits.
The model in Fig. 9.7 admits two *operating modes* controlled by the *adaptation* boolean variable. The first (*synch*) mode, implied by the preceding discussion, aims at estimating the global time required for clock alignment.
Unfortunately, clock synchronization is an emerging property of the WSN model and it is unknown to each node. Subsequently, when the $alignment\_interval$ was estimated, a switch occurs, for energy saving, to the *adaptation* mode which reduces the burden of exchanged radio synchronization messages.
For the adaptation mode, the WSN is partitioned in pairs of master-slave nodes. In addition, an $ADAPT\_PERIOD$ message is initially scheduled to occur at the time of (supposed known or set to a very high value otherwise) $alignment\_interval$ value. After the arrival of the first $ADAPT\_PERIOD$ message, it gets re-scheduled according to the $adapt\_interval$ time, e.g., set to $5000ms$. The adaptation idea stems from the fact that after the time synch, the $TSynch$ value can reasonably be relaxed, e.g., changing it from $1000ms$ to $10000ms$. Moreover, since node clocks inevitably tend again to misalign, the proposed mechanism captures the *symptom* of misalignment in the pairs of nodes. Then misaligned pairs, which are identified when the relevant software clocks differ of about $0.8ms$, react locally by resetting $TSynch$ to $1000ms$ thus starting re-alignment, whose achievement causes $TSynch$ to come back to $10000ms$ and so forth.

A master node sends a $NEIGH\_ADAPT\_PERIOD$ message to its slave,

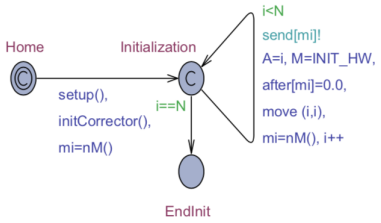


Figure 9.8: The $Main$ automaton

passing the value of its software clock. The slave evaluates the difference between master and slave software clocks and possibly adjusts its $TSynch$.
In any case the slave node replies to the master with an ACK message carrying the slave software clock, which will allow the master to possibly modify its own $TSynch$.
It is ensured that (a) pairs which do not sense a misalignment, do not start a local re-synchronization and then do not send network messages; (b) all the network is eventually re-synchronized as a consequence of local re-alignments.

9.2.2 Experimental results

A WSN of 16 nodes with full mesh topology was used for the experimental work. The model in Fig. 9.7 was decorated with auxiliary variables and functions to

help collecting and depicting information from the simulation runs.
Fig. 9.10a shows the basic behavior of the time-synch algorithm which effectively converges, after some global time, to software clocks alignment. The experiment was conducted for $3x10^4ms$ using the query:

```
simulate [<=30000] {clk_sw[0],clk_sw[1],...,clk_sw[15]}
```

It emerged that after $1.6x10^4ms$ the synchronization accuracy is about $1ms$. In a similar way, queries were prepared for revealing other aspects of the time-

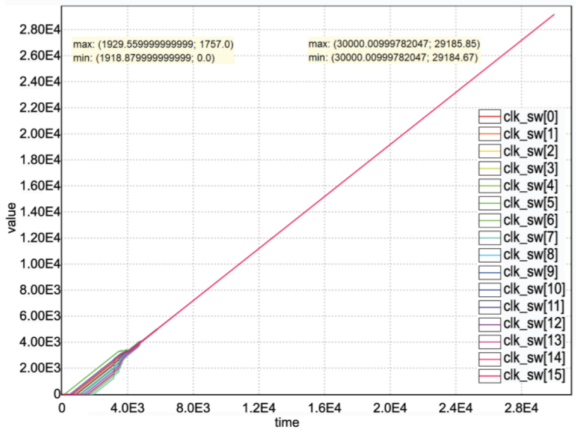
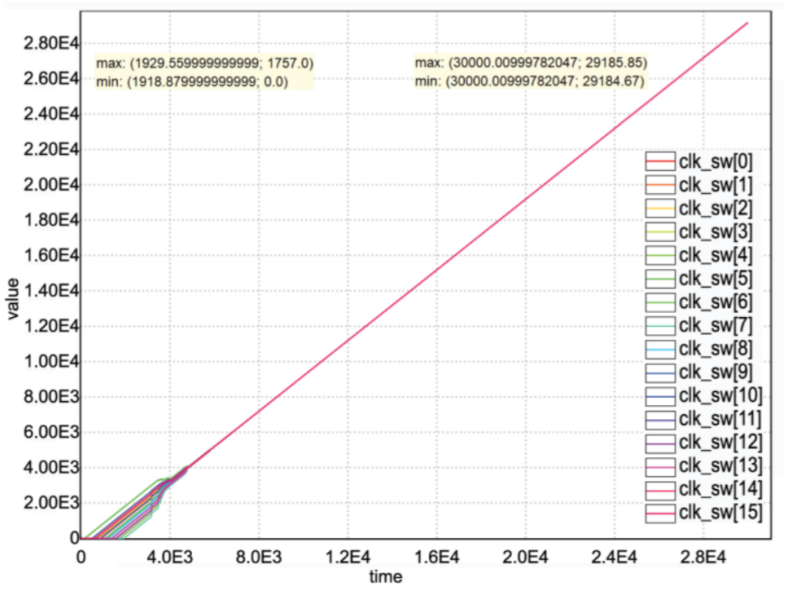


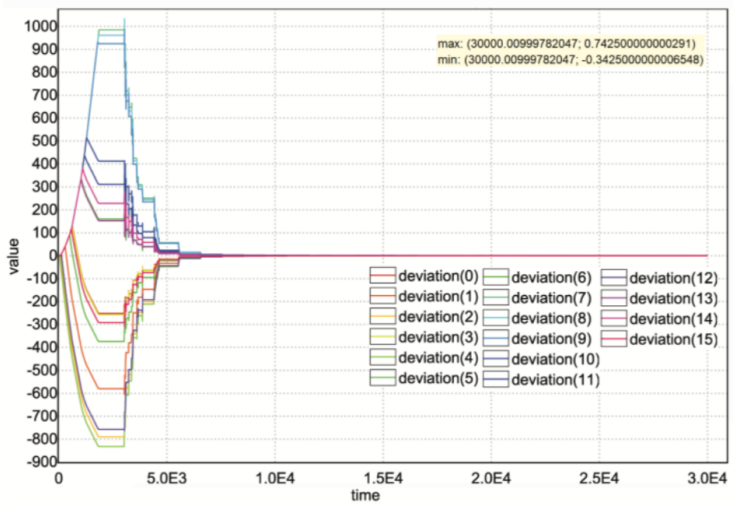
Figure 9.9: Software clocks alignment vs. time

synch model. Fig. 9.10b gives more light in the synchronization process, by depicting a fitness value vs. time. Fitness is computed, at each time instant, as the deviation between the average value of software clocks and each particular software clock. Fig. 9.11a portrays software clocks skew, i.e., the difference, on a time span of $6x10^4ms$ (this time span was chosen to allow comparison with the adaptation mechanism shown later in this section), between the maximum and minimum values of the software clocks.

Of course, the time-synch algorithm requires an amount of exchanged messages among the actor nodes, which directly impacts on the energy consumption and network lifetime. Fig. 9.11b reports the measured number of network messages vs. time. It is to be recalled that in the initial time interval $[0, \sim 2000]ms$ the network gets activated and that the first synchronization phase occurs after next $1000ms$. As stated also in Fig. 9.11a, before $9000ms$ the software clocks are highly misaligned, therefore conflicts among network messages sent at a "same" time are actually not existing.
The above described behavior is also confirmed in Fig. 9.12a by the measured number of cumulative packet losses vs. time. Packet losses reveal themselves as soon as the synchronization starts to be attained which occurs at $10000ms$ considering the $TSynch$ which must elapse since $9000ms$.
It is useful to note that the size of the Theatre cloud of messages sent but not yet delivered registers a maximum value of 78 messages at about $4x10^4ms$, with an average of about 32 pending messages which correspond to the scheduled HW_PERIOD and $SYNCH\_PERIOD$ messages (see Fig. 9.7).
Subsequent experiments were carried out to monitor the adaptive behavior which is triggered following the achievement of the time synchronization, i.e.,

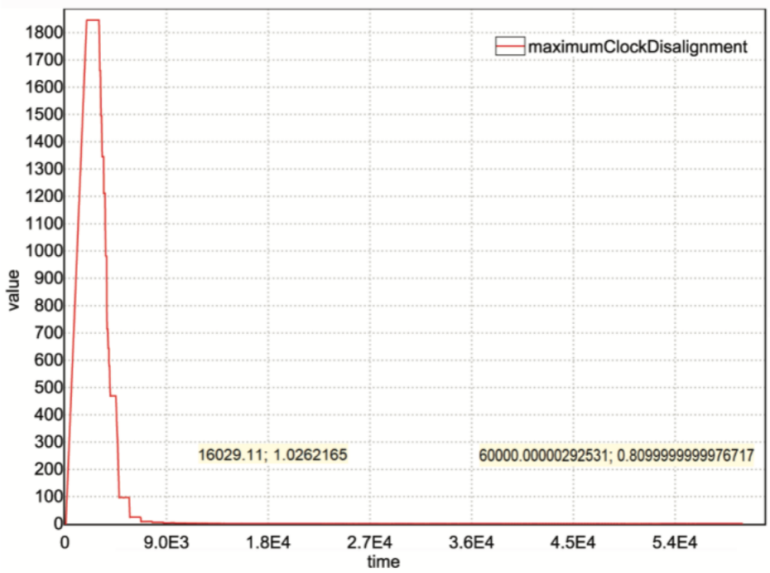


(a) *Software clocks alignment vs. time*

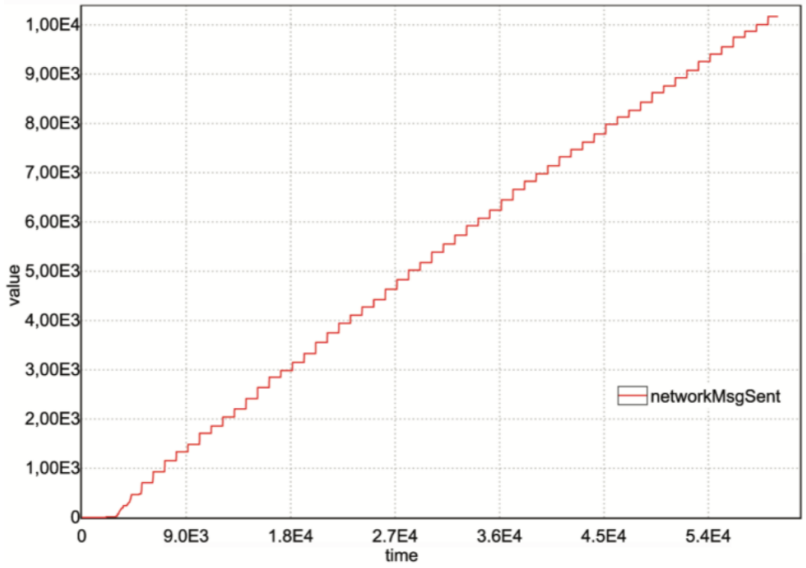


(b) *Software clocks fitness vs. time*

Figure 9.10: Software clocks simulations



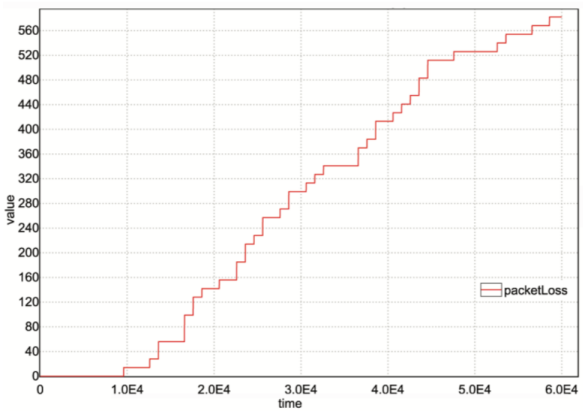
(a) *Software clocks skew vs. time*



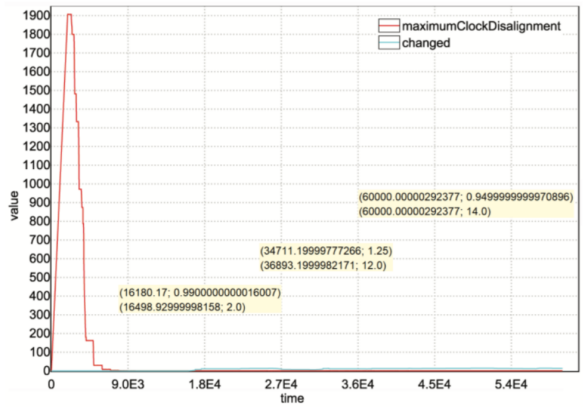
(b) *Number of exchanged network messages vs. time*

Figure 9.11: Simulation results

after $1.6x10^4ms$ for the considered WSN topology and size. Therefore, experiments were extended to $6x10^4ms$ in order to observe clock misalignments which occur *after* the first synchronization phase was reached.
First it was quantified the probability of the clock alignment to be maintained



(a) *Packet loss vs. time*



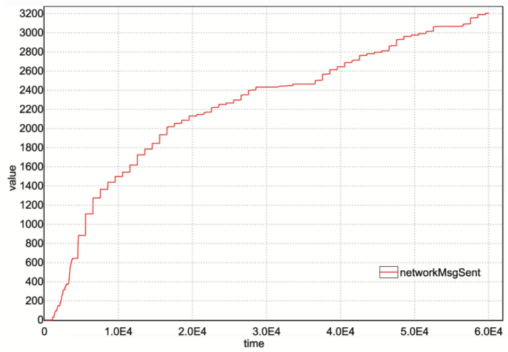
(b) *Software clocks skew vs. time with synch-period adaptation*

through the query:

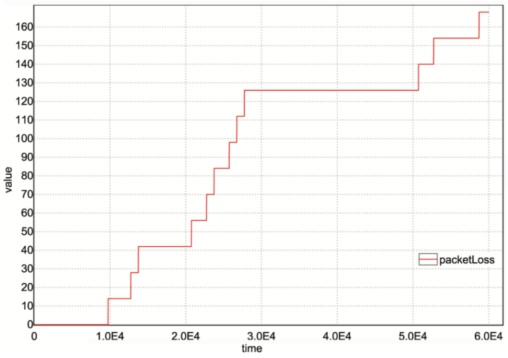
```
Pr[<=60000] ([]( global<=16000 || !existsPairMisaligned()))
```

which checks if after $16000ms$ never exists a master-slave pair of nodes whose clock misalignment exceeds $1.3ms$. UPPAAL SMC proposes, after 29 runs, a confidence interval of $[0, 0.0981446]$ with a confidence degree 95%, thus stating the event has a very low probability of occurrence, i.e., effectively clocks tend to misalign again. This, in turn, motivates the need to run the adaption mechanism.
Fig. 9.12b is analogous of Fig. 9.11a about observed software clocks skew when the adaptation mechanism is active. The curve called *changed* shows the number of nodes that switch their $TSynch$ from $1000ms$ to $10000ms$. As one can see, the number of changed nodes definitely is greater than zero (e.g., 12 at about $3.6x10^4ms$) to testify that only a few nodes register a clock misalignment

which is promptly reacted.
Fig. 9.12a reports the number of exchanged network messages when the adaptation mechanism is enabled. It is confirmed the same number shown in Fig. 9.11b up to $1.6x10^4ms$. Such number sharply decreases as an effect of having achieved the synchronization state. The fewer exchanged messages (at $6x10^4ms$ it is 1/3 of those observed in Fig. 9.11b) positively affect the node energy saving. The benefits of the adaptation mechanism can also be watched in Fig. 9.12b



(a) *Number of network messages vs. time with synch-period adaptation*



(b) *Packet loss vs. time with synch-period adaptation*

where a fewer number of packet loss is reported. At $6x10^4ms$, the number of packet loss reduced from about 580 (see Fig. 9.12a) to about 170.
For completeness, the peak size of the cloud of pending messages within THEATRE was found to be 67 with an average value of 40. The increased value of the average is due to the schedule of $ADAPT\_PERIOD$ messages required by the adaption mechanism.
The model in Fig. 9.7 was also checked with other topologies of the WSN with 16 nodes, including a graph-based (see Fig. 9.12) and a star-based one.
For the graph-based topology, it emerged that the global time required for the first clock alignment is about $105ms$ with an accuracy of about $1.26ms$. For the star-based topology, $7x10^4ms$ are required for the first clock alignment with an accuracy of $1.39ms$.
Experiments were carried out on a Win10 Pro workstation with Intel Core i7-7700 CPU@3.60 GHz, 16 GB Ram, using the development version Uppaal-

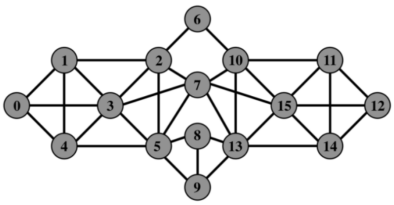


Figure 9.12: Graph-based WSN topology

4.1.20.beta10.

9.3 Actor-based NetBill Protocol

This example focuses on modelling and analysis of knowledge and commitments in MAS which naturally arise in business web-based protocols and applications. Commitments represent the willingness for agents to do something (e.g., paying for an accepted offer of a good over the Internet). Commitments have a status which can transition from creation, to fulfillment, to discharge, to cancellation. Knowledge denotes the epistemic relation of agents being aware about the status of a commitment.
Reasoning on knowledge and commitments in MAS is typically carried out, in the literature, through the development of a special temporal logic language like CTLKC+ [12] whose models can be verified through a reduction to an existing model checker. In the work of [12], the use of the NuSMV model checker is demonstrated with the overall approach which proves to be efficient and scalable.
In the following, the use of Theatre actors is shown for modelling and analysis of multi-agent systems. Instead of shared variables and dedicated communication channels as in [12], commitments are naturally tied to actor interactions (message exchanges). Knowledge relationships can be checked during analysis. The approach is illustrated by modelling and analysis of the NetBill protocol [233, 12]. The example confirms that more simple, not-nested queries can be used for checking properties of a MAS.
The NetBill protocol is played by a customer agent (cus) and a merchant agent (mer). The message conversation relates to buying and selling an encrypted software good over the Internet.
Two commitments are handled in the protocol: $C_{cus \to mer} Pay$ and $C_{mer \to cus} Deliver$. The Pay commitment is raised by the customer toward the merchant to testify the customer intention, after having accepted an offer from the merchant, to proceed with the payment of the good. The $Deliver$ commitment is played by the merchant toward the customer, to express its willingness to deliver to it the required (and paid) good. However, uncertainty in the agent behaviors can change the course of expected actions following the initial intention to commit, as witnessed in the Figures 9.15 and 9.16. The NetBill model is composed of three types of actors: $Customer$, $Merchant$ and $Main$. Different configurations (i.e., number of pairs $< cus, mer >$) can be established by adapting the behavior of the $Main$ actor (Fig. 9.16). $Main$ creates and initializes actors by sending them an $init()$ message. Each actor informs the $Main$ with a $done()$

```
env int SI=0, S0=1, S1=2, …, S9=10; //state ids of cus and mer
env int PAY=0, DELIVER=1; //commitment ids
env int N=1; //number of pairs <cus,mer>
env int PU=N+1; //demo: one PU per pair, +1 for main
```

Figure 9.13: Model constants

message when its initialization is completed. As an example, in Fig. 9.16 each pair $< cus, mer >$ is allocated to a distinct PU.
$Customer$ and $Merchant$ are modelled as finite state machines (see Fig. 9.14 and 9.15). After being started, the $Customer$ sends a $request()$ to the $Merchant$ and awaits for a $quote()$ (in state S1). After that, the customer can (probabilistically) reject or accept the offer by sending a corresponding message to the $Merchant$. Being not thread-based, proactive behavior is achieved by self-sending a message like $next()$. In the case the customer accepts the offer, it

```
actor Customer{
      Merchant mer;
      int cs=SI; //initial status

      msgsrv init(Main m, Merchant me){
            if( cs==SI){
            mer=me; m.done(); cs=S0; }
      }//init

      msgsrv start(){
            if( cs==S0 ){
            reset(); mer.request(); cs=S1; }
      }//start

      msgsrv quote(){
            if( cs==S1 ){
            self.next(); cs=S2; }
      }//quote

      msgsrv next(){
            if( cs==S2 ){
            //uncertainty through probabilistic behavior
                  if( ?(0.8:true, 0.2:false) ){
                  mer.accept();
                  //message ordering through timing
                  self.commit() after(1);
                  co(PAY,self); cs=S3;
                  }
                  else{ mer.reject();
                        self.next() after(1); cs=S4; }
            }
            else
            if( cs==S4 || cs==S8 || cs==S9 ){
               self.start(); cs=S0; }
      }//next

      msgsrv commit(){
            if( cs==S3 ){
               if( ?(0.95:true, 0.05:false) ){
                  mer.Payment(); co(DELIVER,self);
                  fu(PAY,self); cs=S5;
               }
               else{
                  self.next(); mer.notPayment() after(1);
                  fail(PAY,self); cs=s4;
               }
            }
      }//commit

      msgsrv notDelivery(){
            if( cs==s5 ){
               fail(DELIVER,self); cs=S6; }
            }//notDelivery

      msgsrv delivery(){
            if( cs==S5 ){
               fu(DELIVER,self); cs=s7; }
            }//delivery

      msgsrv refund(){
            if( cs==S6 ){
               self.next(); cs=S8; }
            }//refund

      msgsrv receipt(){
            if( cs==S7 ){
               self.next(); cs=S9; }
      }//receipt
      }//Customer
```

Figure 9.14: The Customer model

moves towards commitment (payment) by self-sending a $commit()$ message. Following a $commit()$, though, the customer can proceed, again probabilistically, with the payment or it can choose not to make the payment by informing the merchant about its decision. After a payment, another source of uncertainty exists: it can follow a $deliver()$ or a $notDeliver()$ message from the merchant. In the case of a $delivery()$ a $receipt()$ message must follow (in state S7). Instead, after a $notDelivery()$ message the customer awaits (in state S6) a $refund()$ from the merchant.
A dual behavior is observed by the $Merchant$ and can be easily traced in Fig. 9.15. A common issue of the two actors is concerned with *message order*. As a consequence of the assumption of non-deterministic delivery of concurrent messages, to ensure correct behavior of the model some messages are scheduled to occur after 1 time unit. As an example, in the $Customer$ model in Fig. 9.14, after sending an $accept()$ message to the Merchant, the $commit()$ message is

self-sent with 1 time unit as the after time. All of this guarantees that in no case the $Merchant$ can receive a $notPayment()$ before of an $accept()$. Similar provisions were taken in the Merchant model in Fig. 9.15. In [12] a social

```
actor Merchant{
      Customer cus;
      int cs=SI;

      msgsrv init( Main m, Customer c ){
            if( cs==SI ){ cus=c; m.done(); cs=S0; }
      }//init

      msgsrv request(){
            if( cs==S0 ){ self.next(); reset(); cs=S1; }
      }//request

      msgsrv next(){
            if( cs==S1 ){ cus.quote(); cs=S2; }
            else if( cs==S5 ){ cus.refund(); cs=S0; }
            else if( cs==S6 ){ cus.receipt(); cs=S0; }
      }//next

      msgsrv reject(){
            if( cs==S2 ){ cs=S0; }
      }//reject

      msgsrv accept(){
            if( cs==S2 ){ co(PAY,self); cs=S3; }
      }//accept

      msgsrv payment(){
            if( cs==S3 ){
                  self.commit(); fu(PAY,self);
                  co(DELIVER,self); cs=S4;
            }
      }//payment

      msgsrv notPayment(){
            if( cs==S3 ){ fail(PAY,self); cs=S0;}
      }//notPayment

      msgsrv commit(){
            if( cs==S4 ){
                  if( ?(0.9:true, 0.1:false) ){
                        cus.delivery(); self.next() after(1);
                        fu(DELIVERY,self); cs=S6;
                  }
                  else{
                        cus.notDelivery();
                        self.next() after(1);
                        fail(DELIVER,self); cs=S5;
                  }
            }
      }//commit
}//Merchant
```

Figure 9.15: The Merchant model

commitment implies a communication between the customer and the merchant in order to reflect the status of the commitment in shared variables of the two agents. Such a communication can be naturally expressed by an explicit message exchange in the proposed actor modelling which does not admit data sharing. However, the $accept()$ message itself serves the purpose of transmitting to the merchant the customer intention to (possibly) proceed with the payment, then it plays also the role of communicating that the PAY commitment was issued. Similarly, when transmitting a $payment()$ message to the merchant, the PAY commitment is automatically fulfilled in the customer behavior.
The status of a commitment cm is supposed to be updated by the functions $co(cm, req)$, $fu(cm, req)$ and $fail(cm, req)$ which respectively set the status of cm, as viewed by the requestor actor req, to $QillingToDo$, $Fulfilled$, $Violated$. Knowledge of commitments will be checked during model analysis.

```
actor Main{ //demo: one pu per pair <cus,mer>
      Customer cus;
      Merchant mer;
      int cnt=0, p=0;

      msgsrv pair(){
            cus=Customer(); mer=Merchant();
            move(cus,p); move(mer,p);
            cus.init(self,mer);
      }//pair

      msgsrv done(){
            cnt++;
            if( cnt==1 ){
                  mer.init(self,cus);
            }
      else
            if( cnt==2 ){
                  cnt=0; p++;
                  cus.start() after(1);
                  if(p<N) self.pair();
            }
      }//done

      msgsrv main(){
            move(self,0);
            self.pair();
      }//main
}//Main
```

Figure 9.16: A Main actor

9.3.1 Modelling the NetBill protocol into Uppaal SMC

A set of constants and sub-range types are introduced, which give dimension to each entity category (actors, messages, delays, PUs, etc.) and provide unique

ids to each category instance. In addition to the other parameters introduced in

```
const int N=1; //number of <cus,mer> pairs
typedef int[0,N-1] cus_id; //customer ids
typedef int[N,2*N-1] mer_id; //merchant ids
typedef int[2*N,2*N] main_id;
typedef int[0,2*N] aid; //whole agent ids
const int NPU=N+1; //e.g. one PU per pair <cus,mer>
typedef int[0,NPU-1] pu_id; //pu ids
const int INF=-1;
//PU declarations
bool avail[pu_id]; //is a pu available?
pu_id pu[aid]; //actors to pus mapping
//Message names
const int Init=0,Next=1, Start=2, Quote=3,
Delivery=4, Receipt=5, NotDelivery=6, Refund=7,
Request=8, Reject=9, Accept=10, Payment=11,
NotPayment=12, Commit=13, Done=14, Pair=15;
const int MSG=16;
typedef int[0,MSG-1] msg_id;
//pool dimensioning
const int MI=1;//number of Messages
typedef int[0,MI-1] mid; //Message istance ids
const int VMI=2*N; //number of VoidMessages
typedef int[0,VMI-1] vmid; //VoidMessage instance ids
//pool declarations
bool avM[mid]; //pool of Messages
bool avVM[vmid]; //pool of VoidMessages
//buffer for message arguments
const int MAX_ARGS=2;
int args[MAX_ARGS];
```

Figure 9.17: Global declarations of the transformed NetBill model

section 7.4, useful for modeling the message exchanges, Fig. 9.18 depicts other global declarations, including the possible states of *Customer* and *Merchant* and the commitment and knowledge declarations.

Figures 9.19 and 9.20 depict respectively the TA for the *Customer* and

```
//customer/merchant states
const int SI=0,S0=1,S1=2,S2=3,S3=4,
S4=5,S5=6,S6=7,S7=8,S8=9,S9=10;
//commitment declarations
const int PAY=0, DELIVER=1; //commitments
typedef int[PAY,DELIVER] cid; //commitment ids
const int Invalid=0, WillingToDo=1, Fulfilled=2;
typedef int[Invalid,Fulfilled] status; //comm states
status k[cid][aid]; //agent knowledge of comm. states
```

Figure 9.18: Other global declarations

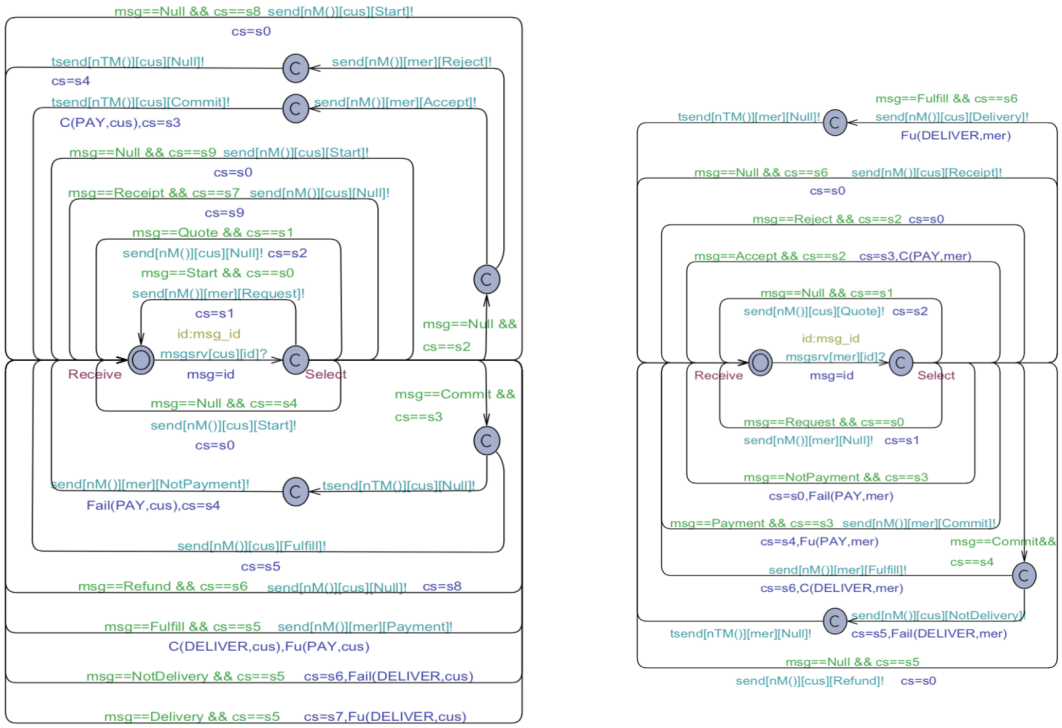
Merchant, and the *Main* actor of the NetBill model. The template processes admit only one parameter $self$, whose type is respectively cus_id, mer_id and $main\_id$.

9.3.2 Experimental results

The following analysis claims that it is not always necessary to write complex and unreadable specifications. Simple, not nested queries in the TCTL subset supported by UPPAAL are sufficient for property checking of MAS models like the NetBill protocol. In addition, THEATRE modelling also opens to the use of the Statistical Model Checker of UPPAAL which permits quantitative evaluation, that is estimating probability measures for event occurrence, in probabilistic models not considered in [12].

Non-deterministic analysis

The UPPAAL reduced NetBill protocol was first exhaustively verified by the symbolic model checker, which builds the (hopefully finite) model state graph. This qualitative analysis is very important because it allows to check general properties like the absence of deadlocks, various kinds of liveness issues etc. The NetBill model with one single pair $< cus, mer >$ $(N = 1)$ was verified through the following queries. The status of a commitment c, as known to an actor a, can be checked by the epistemic (Knowledge) function: $K(c, a)$.



a) The Customer automaton

b) The Merchant automaton

Figure 9.19: Actor automaton

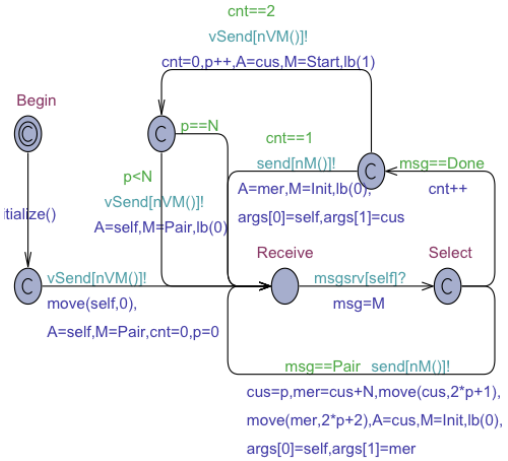


Figure 9.20: A *Main* actor automaton

It is worth noting (see Fig. 9.17) that customer with id 0 interacts with merchant id N and so forth.

1) $A[]!deadlock$ (satisfied)

The model has no deadlocked state.

2) $Customer(0).cs! = S0 -- > Customer(0).cs == S0$ (satisfied)

$Merchant(N).cs! = S0 -- > Merchant(N).cs == S0$ (satisfied)

After being started, actors regularly come back to their home state.

3) $A[]Customer(0).cs == S0 \&\& Merchant(N).cs == S0 \ \ imply \ \ now <= 2$

(satisfied; but it is not satisfied with now<=1)

Customer and Merchant come back to their home state after at most 2 time units. This in turn anticipates the duration of a simulation experiment.

4) $Merchant(N).cs == S6 -- > K(DELIVER, N) == Fulfilled$

$\&\& K(DELIVER, 0) == Fulfilled$

$(satisfied)$ (satisfied)

When the merchant is in state S6, it necessarily follows that both actors know the $DELIVER$ commitment is fulfilled.

5) $K(DELIVER, 0) == Fulfilled -- > K(DELIVER, N) == Fulfilled$ (satisfied)

If the Customer knows $DELIVER$ is fulfilled, it necessarily happens the same for the $Merchant$.

6) $E <> K(DELIVER, 0) == Fulfilled$ (satisfied)

It *can* happen that the customer knows the $DELIVER$ commitment is fulfilled.

7) $K(PAY, 0) == Fulfilled -- > K(DELIVER, 0) == Fulfilled$ (not satisfied)

Even after payment, the customer is not guaranteed that the $DELIVER$ commitment gets fulfilled.
The above queries confirmed their results when, e.g., each actor is assigned to a distinct processing unit (PU)(maximal parallelism), when all the actors are allocated to the same PU, and when each pair of $< cus, mer >$ is assigned to a distinct and the $Main$ runs on a separate PU.
To give an idea of the model checking performance, Table 9.1 collects the wall-clock time and the RAM memory peak observed when launching the $A[]!deadlock$ query on some number N of pairs $< cus, mer >$ when only one PU is used. With $N = 3$ there is state explosion. Obviously, even one single pair $< cus, mer >$ enables the protocol to be analyzed.
Experiments refer to a Linux machine, Intel Xeon CPU E5-1603@2.80GHz, 32 GB, using UPPAAL 4.1.19 64 bit which allows to exploit as much RAM is available.

Table 9.1: Observed CIs with adaptation errors, $\epsilon = 0.01$

N	Wall-clock time (sec)	RAM Memory Peak (MB)
1	≈ 0	7.6
2	4.55	72

Quantitative analysis

From the non-deterministic analysis it emerged that it *can happen* that the customer eventually knows the $DELIVER$ commitment is fulfilled (query 6)). In addition (see query 7)), even after the payment, the product is not guaranteed to be received. This is a consequence of the model *uncertainty* expressed by probabilistic behavior. As a consequence, it is interesting to estimate the probability with which events can occur.
In the following, the use of UPPAAL default statistical options is assumed, which means, e.g., the uncertainty error of a confidence interval is $\epsilon = 0.05$, 95% of confidence degree.
The query:

$$8) \qquad simulate[<= 100]Customer(0).cs$$

makes one simulation of the model lasting 100 time units and asks to monitor the changes with time of the $Customer$ current state variable cs. The generated Fig. 9.21 confirms functional behavior already observed during the exhaustive model checking. The customer, after its operation, regularly returns to its home state $S0$ (= 1). From Fig. 9.21 one can see that sometimes the customer takes the $S9$ state (= 10) to which corresponds the fulfillment of the $DELIVER$ commitment (see also Fig. 9.14).
The qualitative behavior of the $DELIVER$ commitment as observed from the

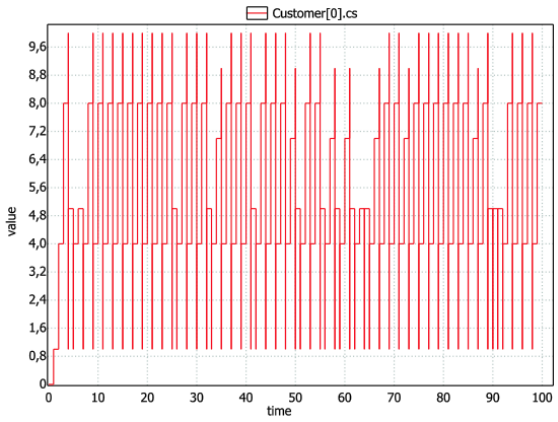


Figure 9.21: Evolution of the Customer state following query 8)

$Customer$ viewpoint is specifically collected and shown in Fig. 9.22 generated by the query:

$$9) \qquad simulate[<= 50]K(DELIVER, 0)$$

Fig. 9.22 testifies there are cases when the $DELIVER$ commitment can effectively be fulfilled ($Fulfilled = 2$) for the $Customer$.

The following query asks to estimate, through a certain number of simulation experiments (each one lasting in 2 time units), the probability for the $DELIVER$ commitment to be fulfilled as perceived by the $Customer$:

$$10) \qquad Pr[<= 3](<> K(DELIVER, 0) == Fulfilled)$$

UPPAAL SMC, after 339 runs, proposes a confidence interval (CI) of $[0.650287, 0.750272]$ with 95% confidence. The MITL query:

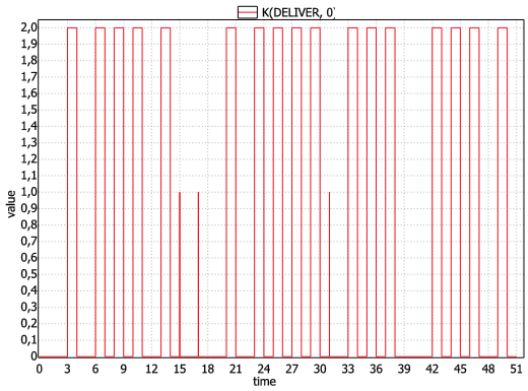


Figure 9.22: Monitored $K(DELIVER, 0)$ values with query 9)

$$11) \qquad Pr(<> [0,3](K(PAY, 0) == FulfilledU[0,1]K(DELIVER, 0) == Fulfilled))$$

estimates the probability of the event: "*assuming that at an instant in* $[0, 3]$ *the* PAY *commitment is fulfilled for the customer, what is the probability that within* 1 *time unit the* $DELIVER$ *commitment gets fulfilled for the customer?*". Using 738 runs UPPAAL SMC proposes a CI of $[0.647832, 0.747832]$ with 95% confidence.

The results of queries 10) and 11) were further validated by decorating the

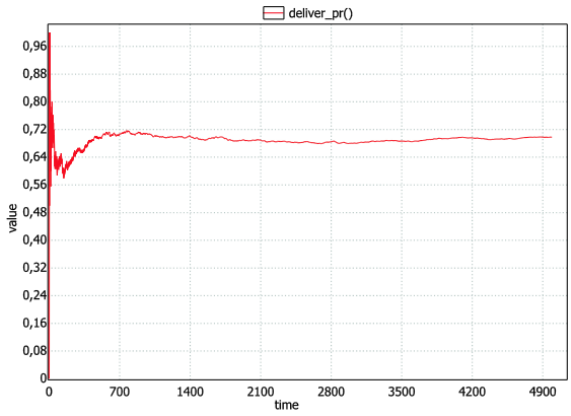


Figure 9.23: Direct estimation of $K(DELIVER, 0)$ fulfillment probability

NetBill protocol model with two double variables nr and sr which respectively

counts the total number of iterations of the customer and the number of successful iterations when the customer receives the *Delivery* message, switches to state $S7$ and fulfills the $DELIVER$ commitment. The function $deliver\_pr()$ returns initially 0.0 then it returns the instantaneous value of the ratio sr/nr which gives an estimation of the fulfillment probability of the $DELIVER$ commitment. Fig. 9.23 was generated by UPPAAL SMC through the query:

$$12) \qquad simulate[<= 5000]deliver\_pr()$$

which plots the values of $deliver\_pr()$ vs. time, which are in good agree with the results of queries 10) and 11).

Chapter 10

Modelling and Analysis of Multi-Agent Systems Using UPPAAL SMC [1]

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

10.1 Introduction

Multi-agent systems (MAS) [259]are widely recognized as an important paradigm for modelling and analysis through simulation (M&S) of complex and adaptive systems [201, 74]. Power and flexibility of MAS derive from their ability of modelling both the individual behavior of agents and their social interactions, i.e., the exchanges of information, and then the possibility of observing the emergence of properties at the society/population level. Agents are characterized by their basic abilities [259] of autonomy, sociality, pro-activity, "intelligence" for

[1] The material in this chapter is related to publications [189]

deliberative behavior, learning and adaptation mechanisms, and so forth.
In this work a minimal yet efficient actor computation model [10, 73] is adopted for supporting MAS, which addresses complex models. A key feature of this framework is a light-weight notion of actors which (i) are thread-less agents, (ii) hide an internal data status, and (iii) communicate each other by asynchronous message passing. Message exchanges are ultimately regulated by a customizable *control structure* which can reason on time (simulated or real-time). The actor framework can be effectively hosted by popular languages like Java.
An original contribution is a structural translation of a MAS actor model into the terms of Uppaal SMC [90]. Challenging is an effective support of asynchronous timed message passing. Main motivations underlying the proposed work are the following (i) using a formal modelling tool based on Timed Automata [17] to capture, in a natural way, actor behaviors and message exchanges; (ii) expressing in the associated temporal logic language, model specific properties to be checked on the MAS model by simulations; (iii) exploiting statistical model checking techniques [264, 153], that is automatizing multiple executions of a MAS model, estimating the required number of simulation runs, and using statistical properties (Monte Carlo-like simulations and sequential hypothesis testing) to infer system properties from the observables of the various runs.
Uppaal SMC was chosen among other competitive tools like PRISM [120], Plasma LAB [4] etc., because it is a popular and efficient toolbox based on a stochastic extension of Timed Automata [17, 90], it supports graphical, intuitive modelling and offers high-level data structures and functions which improve the modelling of complex systems. As for PRISM (see e.g. [128, 129]), Uppaal SMC can also be exploited for modelling and quantitative assessment of timing constraints in probabilistic real-time systems [194].
To the best of authors' knowledge, this is a first attempt to support general actor-based MAS models and their quantitative evaluation using Uppaal SMC, with the approach which can be concretely used by modelling and simulation practitioners and engineers.
The approach proposed in this paper is practically demonstrated through a case study concerned with a complex and adaptive model based on the Iterated Prisoner's Dilemma (IPD) game [23]. The model is challenging and aims at studying the emergence of cooperation among competitive agents in the presence of different social interaction networks. SMC results confirm previous indications of Axelrod et al. work, that is that cooperation is possible among players when their interaction links are preserved during the game execution. In addition, the accomplished detailed probabilistic analysis permits to observe some new properties not previously revealed.

10.2 Modelling the Iterated Prisoner's Dilemma

The following considers, as a case study, a modelling of the Iterated Prisoner's Dilemma which is a variant of the basic Prisoner's Dilemma (PD) game [24, 259]. PD is a binary game in which two players have to decide independently and without any form of communication, between two alternative choices: to defect (D , e.g., 0) or to cooperate (C , e.g., 1). The decision implies that each player gets a payoff as follows: $(D, D) \rightarrow (P, P)$, $(D, C) \rightarrow (T, S)$, $(C, D) \rightarrow (S, T)$, $(C, C) \rightarrow (R, R)$, where P means punishment for mutual defect, T temptation

to defect, S sucker's payoff, and R reward for mutual cooperation. Classically $T > R > P > S$ and $R > (S + T)/2$. Common adopted values are $S = 0, P = 1, R = 3, T = 5$.
Under the uncertainty of partner decision, players acting rationally direct themselves to defect in order to optimize their payoff, with (D, D) being the Nash equilibrium of the game. Indeed, players spontaneously are driven by selfish behavior due to the suspect about the opponent decision. In this situation, it would be extremely risky to decide C. In fact, if the partner choses D, the first player would achieve a 0 payoff and the partner the maximum reward of 5.
But if the one shot game admits only the outcome of (D, D), things are not determinate in the case the game is long iterated, with the number of iterations being unknown to players. The Axelrod book [24] triggered much interest in the social science toward studying conditions under which cooperation can emerge. In the basic Iterated Prisoner's Dilemma (IPD) (see Axelrod tournaments [24, 23] a certain number of players N, each equipped with a suitable strategy, repeatedly plays in turn with each of the other $N - 1$ partners and the payoff is accumulated so as to detect some dominant strategy. Each player has the memory of what the opponent did in the previous move. The winner of the first tournament was the strategy Tit-for-Tat (TFT) proposed by Anatol Rapaport. TFT cooperates on the first move (i.e., it is a nice strategy) and then mimics the opponent decision taken in the previous move. Also in the second Axelrod tournament, with more competing strategies, TFT emerged as the winner strategy, but in addition the experiment revealed that "altruistic" strategies instead of "selfish" and "greedy" behavior, in the long time can do better toward cooperation, particularly if strategies can evolve and adapt, thus learning from the experience, during the iterated game.
In [86, 23] the IPD was studied from a different perspective, to investigate the role of a social interaction network upon player behavior. In particular, the goal was to check the influence of link persistence (also said context preservation) on the emergence of cooperation, in the presence of learning and evolution of the strategies. The study confirms cooperation is possible under link persistence.

10.2.1 Case study description

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

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

At the beginning of each simulation run, each player is assigned a strategy (y, p, q) of three probability values in $[0, 1]$, where y is the probability of choosing C at the first period, p is the probability of choosing C when the partner's last

move was C, and q is the probability of choosing C when the partner's last move was D. The space of strategies includes the binary strategies $ALL - C$ ($y = p = q = 1$), TFT ($y = p = 1, q = 0$), anti TFT ($aTFT : y = p = 0, q = 1$) and $ALL - D$ ($y = p = q = 0$). The model initially configures the population of shuffled agents by an even distribution of strategies where $y = p$ and p and q can assume the sixteen probability values in the vector $[1/32, 3/32, \ldots, 31/32]$. At each period, each player plays 4 times the PD game separately with each of its neighbors, and the payoff is accumulated (and finally normalized) and the last move recorded, move by move, for both the player and its neighbors.
At the end of each period, following the PD moves, each player A adapts its behavior by copying (imitation) the strategy of the best performing neighbor (say it B), would the payoff of B be strictly greater than the period payoff of A. In addition, since the adaptation process can realistically be affected by errors (a comparison error can occur during the selection of the best performing neighbor, and a copying error can introduce a noise during the copying process) the following hypothesis are made. At each adaptation time, there exists a 10% chance that the comparison between A and B payoffs is wrongly performed and the best payoff misunderstood. Moreover, even in the case the strategy of A was not replaced with that of B, there is 10% chance that each "gene" of A strategy, i.e., the parameters y, p, q, be affected by a Gaussian noise with mean 0 and standard deviation 0.4.
The main goal of the case study is to monitor the *fitness* of the model vs. time, using a number T of 2500 steps or periods. First the average payoff per period is determined by adding all the period payoff of players and dividing the total period payoff by the population size N. Then the fitness is extracted by accumulating, at each time t, all the population average payoffs up to t, and dividing this sum by t. Other observables are the average values at each time of the probabilities p and q, averaged over all the population, so as to monitor the trend of strategy adaptation. Of course, a fitness value definitely moving toward 1 mirrors the emergence of defection, whereas its tendency to 3 (actually to a value greater than 2) testifies cooperation. The above described observables must be checked in all the possible model configurations.

10.2.2 An IPD actor-based model

The model (see Fig. 10.1) consists of $N + 1$ actors: one $Manager$ actor who directs the games, and N players who actually do the game actions.
A minimal Main program is responsible for creating the actors and sending an

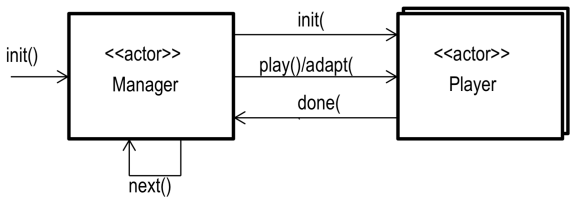


Figure 10.1: IPD actor model

$init()$ message to initialize the $Manager$. The $Manager$ continues by initializing all the players by sending to each of them an $init()$ message. After initialization,

the *Manager* enters its main cycle of operations. Each cycle begins by the *Manager* which sends to itself a *next*() message (see also Fig. 10.2) which lasts one time units, thus realizing the basic step or period. On receiving *next*() the *Manager* sends to all the players a *play*() message. Each player reacts by executing the 4 moves with each of its neighbors and accumulates the period playoff. After that, a *done*() message is sent to the *Manager*. When all the *done* messages are received, the *Manager* broadcasts an *adapt*() message to all the players so as to trigger the learning-and-adaptation phase. Finally, the players send a *done*() message to the *Manager*. When all the N done messages are received, the *Manager* starts the next cycle and so forth. It should be noted that the initialization messages are sent with a 0 delay. Since the *next*() message is received after one time unit is elapsed, it is guaranteed the model has finished the initialization phase before the first cycle begins. This explains why *done* messages are avoided during the IPD initialization.
When the interaction network is persistent during the whole game, each player establishes its neighborhood at the initialization time. In the case of a temporary network, links are re-defined just before starting the next cycle of operations.

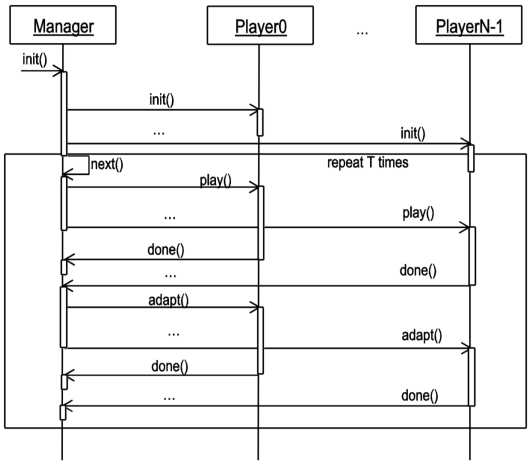


Figure 10.2: Message exchanges during IPD operation

10.3 A structural translation from actors to Uppaal SMC

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

in a non-deterministic way. Details of the translation will be provided in the following using the IPD model as an example.

10.3.1 Actor and message naming

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

```
const int N= 256; //player population
const int dim=16; //sqrt(N)
typedef int[0,N-1] pid; //player ids subrange type
typedef int[N,N] mid; // manager ids subrange type
typedef int[0,N] aid; // agent ids subrange type
```

Using a *const pid a* parameter for the *Player* automaton (see Fig. 10.5b), and a *const mid m* parameter for the *Manager* automaton (see Fig. 10.5a) guarantees N instances will be created of the *Player* template automaton, with the ids ranging from 0 to $N-1$, and only one instance for the *Manager* template automaton whose *id* is N. Messages of the IPD model can be classified as follows:

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

Then the following sub-range type can be introduced:

```
const int MSG= 5;
typedef int[0,MSG-1] msg_id;
```

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

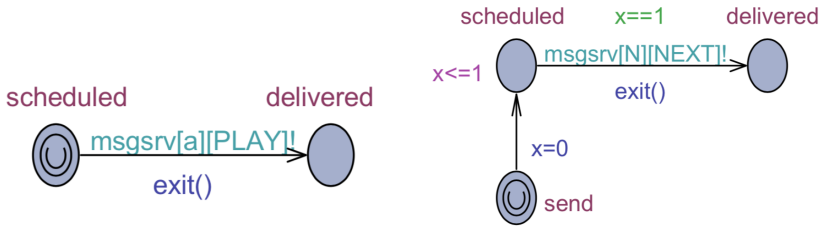
```
broadcast chan msgsrv[aid][msg_id];
```

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

10.3.2 Dynamic messages

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

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



(a) *Immediate message* (b) *Timed message*

Figure 10.3: Message automata

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

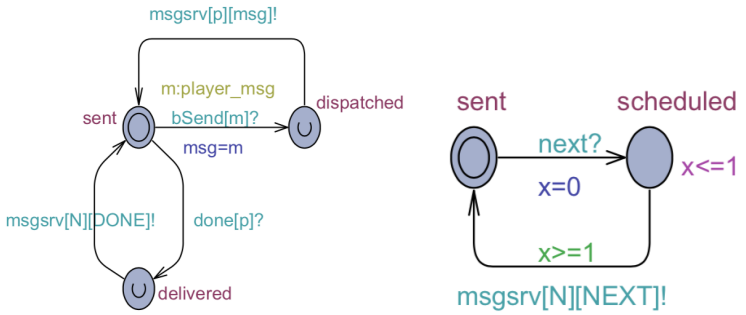
```
spawn tName(args);
```

Similarly, it can be terminated by an *exit*() expression in the update of a command in the *tName* template process.
Two typical examples of message dynamic templates, respectively for an instantaneous (or immediate) message and a timed message, are shown in the Fig. 10.3a and 10.3b.

Message automata, here tailored to the IPD model, start in an urgent location and admit two key locations: *scheduled* and *delivered*. The *scheduled* can be time-sensitive **(possibly stochastic)**. In Fig. 10.3b the message cannot be delivered before 1 time unit is elapsed from the send time. In Fig. 10.3a the scheduled message must be immediately delivered. Delivering is achieved by sending a synchronization over the *msgsrv* channel corresponding to the destination actor and the involved message id. Dynamic instantiation of a message automaton occurs in an actor automaton and it is supposed to transmit as parameter the identity of the recipient actor (see the *a* parameter in Fig. 10.3a).
Although dynamic messages are a flexible and elegant mechanism, a penalty in the execution performance can practically prohibit its exploitation, especially in large MAS models like the IPD model. In preliminary work [195], the IPD model was completely prototyped using only dynamic message instantiations. The consequence was that a single simulation of 2500 time steps required about 6 hours of wall-clock time (WCT) to complete.
In this paper a more efficient solution is adopted which is based on statically pre-allocated automata for all the messages involved in the IPD model. Message automata are dynamically activated by specific synchronization channels. When a message is eventually dispatched, the corresponding automaton resets its behavior so as to be subsequently re-used and so forth. After a careful examination of the IPD model, two static automata were designed as depicted in Fig. 10.4a and 10.4b, along with the following global declarations:

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

The parameterless *StepMessage* template automaton exists in one instance only. It is activated by a synchronization over the scalar *next* channel. The



(a) *Immediate Message automaton* (b) *Timed StepMessage automaton*

Figure 10.4: Message automata

Message automaton, instead, is designed so as to preallocate N instances of immediate messages. Towards this it is sufficient to parameterize the *Message* template with the only parameter *const pid p*. The N instances of *Message* serve all the purposes in the IPD model, and are continually re-used. The *Manager* actor (see Fig. 10.5a) activates all the N instances of *Message* through a broadcast send (see the *bSend*[] channel array) which specifies the particular message to be sent to players among: $INIT$, $PLAY$ and $ADAPT$. On the other hand, following a $PLAY$ or $ADAPT$ message, a player sends back to *Manager* (which has the id N) a *done* reply using the particular *done*[·] channel indexed by the player id p. This way, the synchronization is heard *only* by the certainly free *Message* instance corresponding to p.
To give an idea of the practical impact of the new message design, a simulation of 2500 time steps now lasts in than 17 minutes of wall-clock time.

10.3.3 Actor automata

Figures 10.5a and 10.5b portray respectively the automaton of the *Manager* and that of the *Player*. An actor automaton receives a message server invocation from a normal location, and then processes it through (in general) a cascade of committed locations which ends in a normal location too.
A *Main* automaton is avoided by having that initially the *Manager* starts its execution by initializing itself and then by broadcasting (through a *bSend* channel) an $INIT$ message to all the players. Then a *next* synchronization is sent which causes the *StepMessage* (see Fig. 10.4b) to be activated. On receiving a $NEXT$ message, the *Manager* executes its basic cycle of operations as stated in Fig. 10.2. As one can see from Fig. 10.5a, in the normal locations $WaitDone1$ and $WaitDone2$, the N done messages from the players are awaited and counted.
The finite state machine of the *Player* automaton in Fig. 10.5b, first waits for an $INIT$ message, then it expects a $PLAY$ and then an $ADAPT$ message. The response actions are purposely confined in the functions *init_player*(), *do_play*() and *do_adapt*(). After a $PLAY$ or $ADAPT$, a $DONE$ message is sent to the *Manager*.
Both automata in Fig. 10.5a and Fig. 10.5b are finite state machines. In

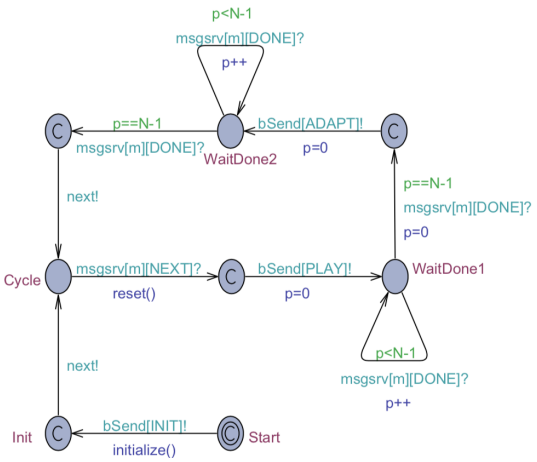
reality, only the *Manager* needs a finite state machine behavior. Since the *Manager* asks the player reactions in the right order (first for $INIT$, then for $PLAY$ and then for $ADAPT$ repeatedly) the *Player* automaton can more easily be modelled according to the "pure reactive" style of agents shown in Fig. 10.5c. This version of the actor illustrates the "input determinism" principle required by UPPAAL SMC: from a location only one edge can exit with the next received message. As a general rule, a pure reactive actor model can be organized according to a couple of locations such as *Receive* and *Select*. To the edge outgoing *Receive* is attached a command with a non-deterministic selection of the received message id in an input *msgsrv* synchronization. Then in the committed *Select* location, the particular message id is checked and a cascade of committed locations is entered, which realizes the corresponding reaction. At the reaction end, the *Receive* location is re-entered.

10.3.4 Other global declaration

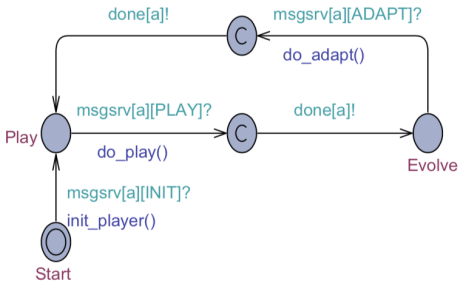
The following are some other global declarations useful for the operation of the IPD model.

```
const int north=0, east=1, south=2, west=3; //links
typedef int[north,west] link;
pid ngh[pid][link]; //agent neighbors
const int D=0, C=1;
typedef int[D, C] decision;
const int MOVES = 4; //nr of moves per pair of players per time step
const int NONE=-1;
int[NONE, C] last[pid][link];
double y[pid], p[pid], q[pid]; //agent strategies
int[NONE, 20] payoff[pid][link];
double period_payoff[pid];
double totpayoff=0.0; //total accumulated payoff per time step
const int[0,5] PAYOFF[decision][decision]=
{{1,5},//{DD,DC}
{0,3}};//{CD,CC}
clock now; // current simulated time
```

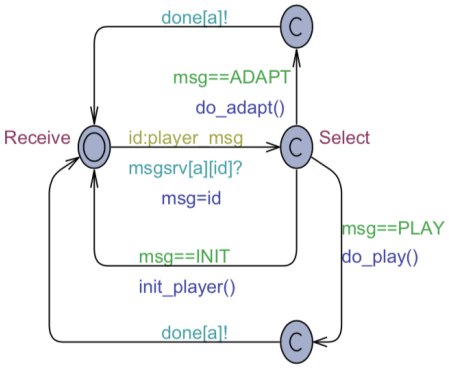
Player links are conventionally named according to the Manhattan neighborhood. However, only for the persistent toroidal grid (PTG) the four players stored in the neighbor matrix $ngh[pid][link]$ effectively mirror the partners of a given player at *north*, *east*, *south* and *west* positions in the grid. In the case of a random network topology (PRN-Persistent Random Network, and TRN-Temporary Random Network), the four links are in reality established randomly, but avoiding duplicate neighbors and having the player as a neighbor of itself. The array $last[pid][link]$ stores the last move taken by a player with respect to a certain link.
At the initialization time (see the *initialize*() function of *Manager* in Fig. 10.5a) *last* is set to undefined (the $NONE$ value is used). The matrix $payoff[pid][link]$ stores the payoff of a player following the four moves with each partner at the four links. The $period\_payoff[pid]$ holds the average payoff of a given player along its four links. The scalar $totpayoff$ accumulates the payoff averaged over the entire population. The constant $PAYOFF$ matrix furnishes the earned



(a) *The Manager actor automaton*



(b) *The Player actor automaton*



(c) *A pure reactive Manager automaton*

Figure 10.5: Message automata

payoff of each partner of a pair of competing players.
Variables like $totpayoff$ and $period\_payoff$ are examples of "decoration variables", useful for extracting interesting properties of the model. At the start of each $Manager$ cycle (see the $reset()$ function in Fig. 10.5a), the payoff values of each player along its four links are set to $NONE$. All of this is a key to handle "reciprocity", i.e., the fact that if A is playing against B, both A and B have to store their achieved result. This in turn avoids to repeat unnecessary moves of B vs. A. The $reset()$ function also updates the $totpayoff$ variable by adding to it the average payoff computed over the entire population. At each time step, the ratio between the $totpayoff$ with current time now (except for the case when now is 0.0) furnishes the instantaneous *fitness* value of the IPD model (see Fig. 10.6).

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

Figure 10.6: Fitness value at current time step

10.4 Experimental work

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

10.4.1 Debugging queries

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

```
Pr[<=5](<> Manager(N).WaitDone1 && Manager(N).p==N-1)
```

Uppaal SMC responds with runs and with a probability estimation having a confidence interval (CI) of $[0.902606, 1]$ with 95% of confidence. Therefore, the event is almost certain. The same query was issued with reference to $WaitDone2$ and Uppaal SMC responds with the same probability estimation. For demonstration purposes, the following query was also used to confirm the prediction about $WaitDone2$:

```
Pr[<>[1,1] (Manager(N).WaitDone2 U[0,0] Manager(N).p==N-1))
```

This query asks about the probability that being at time 1 the state predicate $Manager(N).WaitDone2$ true, following this, and in 0 time (as required by the until time interval $[0, 0]$), it would occur that the p counter of the $Manager$

reaches the value $N-1$. UPPAAL SMC responds, after 738 runs, with a CI of $[0.95, 1]$ with 95% of confidence.
The following query was used to check that effectively, after a $PLAY$ message, a player will receive, at the same time, an $ADAPT$ message from the $Manager$. As an example, the query was directed to the process instance $Player(0)$.

```
Pr[<>[1,1] (Player(0).msg==PLAY U[0,0] Player(0).msg==ADAPT))
```

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

10.4.2 Transient behavior

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

```
simulate[<=50]{avg_payoff()}
```

to show the value of the average payoff in the first 50 steps. Results for the PTG model in the presence of adaptation errors are shown in Fig. 10.7. In reality, the basic behavior holds with or without adaptation errors and also for PRN and TRN topologies. Fig. 10.7 confirms the indications in [86] at pages 24 and 42. The average payoff starts at 2.25 then sharply decreases, after which it will tend to a final possible regime. The initial value is due to an equivalent average strategy (y, p, q) of $(.5, .5, .5)$ being randomly initially distributed. The sharp decline is due to the presence of akin $ALL-C$ strategies which play with akin $ALL-D$ strategies. As a consequence, $ALL-D$ tends to dominate, but as $ALL-D$ plays with other $ALL-D$ it causes a sudden decrease in the payoff.

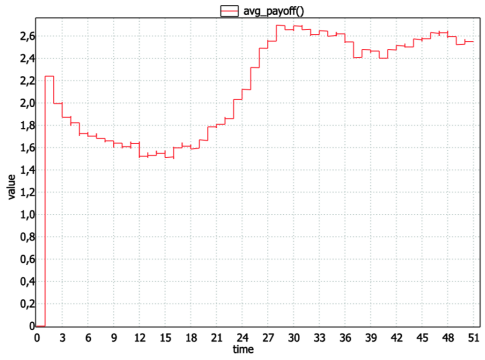


Figure 10.7: Average payoff in the first 50 steps - with adaptation errors

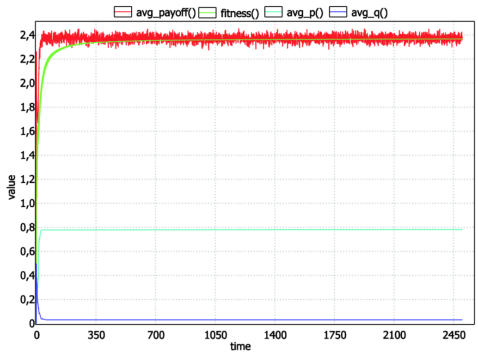
The steady-state analysis was directed to studying the possible emergence of a cooperation regime in two cases: (a) *scenario-1* - each game per period consists of the 4 moves and the last move is recorded and influences the next move in the same game. Of course, strategy adaptations are propagated from a period to the next one; (b) *scenario-2* - the last move of a game affects the first move in the next period. In the scenario- 1 the array *last* (see section 10.3.4) is reset at each period. In the scenario-2 the array *last* is never reset.

10.4.3 Emergence of cooperation in the scenario-1

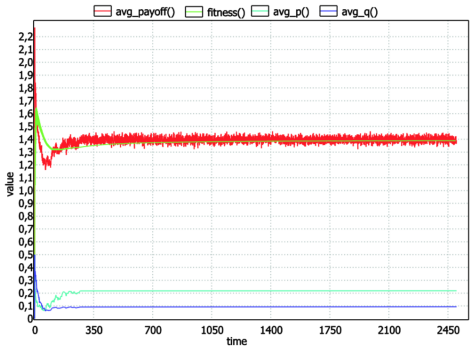
The six models PTG, PRN and TRN with and without adaptation errors, were repeatedly studied using the query:

```
simulate[<=2500]{avg_payoff(), fitness(), avg_p(), avg_q()}
```

where the average payoff at the population level, the fitness and the average values of the agent probabilities p and q are monitored. Figures from 10.8a to 10.9a refer to the case no adaptation errors are introduced. Figures from 10.9b to 10.9d depict collected results when adaptation errors plus noise are possible at each period.
As expected, the evolution of IPD without errors tends to have small fluctuations and to stabilize soon depending on the initial random distribution of strategies to agents and ultimately on "who talks with whom". As a consequence, both the attainment of a cooperative regime (see Fig. 10.8a) or of a defective regime (see Fig. 10.8b) is possible for a persistent interaction network. For a transitory network like TRN, instead, the defective regime was always observed.



(a) *PTG fitness() vs. time, no errors*



(b) *PRN fitness() vs. time, no errors*

Figure 10.8: Experimental results for the scenario 1

In the case adaptation errors are admitted, both persistent interaction networks PTG (Fig. 10.9b) and (Fig. 10.9c) testify the attainment of a cooperative regime with a steady state $fitness()$ value of about 2.5. The TRN network, though, always confirm the achievement of a defective regime. As shown in Fig.

Table 10.1: Observed CIs with adaptation errors, $\epsilon = 0.01$

Model)	Confidence Interval (95%)	Number of runs
PTG	[0.980044,1]	183
PRN	[0.980044,1]	183
TRN	[0, 0.019956]	183

from 10.9b to 10.9d, the adaptation errors imply a greater fluctuation in the payoff values, caused by the dynamic creation of (y, p, q) new strategies in the agent population.

The attainment of a cooperative regime in the presence of adaptation errors, was further checked by using the query:

```
Pr[<=2500]([] (now<2000 || fitness() >= 2.3))
```

which estimates the probability that after 2000 time steps a strictly cooperative $fitness()$ value would emerge. The threshold of 2.3 was chosen according to [23] page 343, as a minimal expectation following the value of 2.25 of an initial random population.
For both PTG and PRN networks, after 36 runs, it emerged a confidence interval of $[0.902606, 1]$ with of confidence 95%, indicating a high occurrence probability. For the TRN network, instead, a CI of $[0, 0.0973938]$ 95% was proposed, thus witnessing the event has a very low probability of occurrence.
Changing the probability uncertainty from $\epsilon = 0.05$ (default) to $\epsilon = 0.01$, implies Uppaal SMC uses more runs and proposes the confidence intervals shown in Table 10.1, thus even better mirroring the event of achievement of a cooperative regime is almost guaranteed for persistent networks. The CI for TRN testifies the attainment of a defective regime.
In the case of absence of adaptation errors, the query

```
Pr[<=2500]([] (now<2000 || fitness() >= 2.3))
```

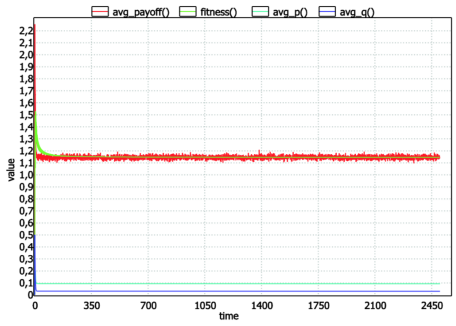
proposes for PTG, after 376 runs, a CI of $[0.300046, 0.399964]$ 95%, and for PRN, after 383 runs, a CI of $[0.339927, 0.43987]$ 95%. All of this indicates that in the case of no errors, cooperation *is* possible although with a smaller probability of occurrence.

It is worth noting that the above presented detailed probability estimations are not part of previous work, e.g., in [23] .

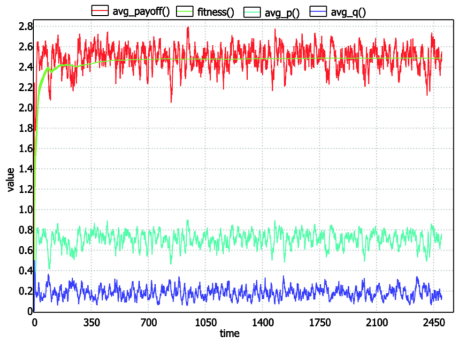
10.4.4 Emergence of cooperation in the scenario-2

The same experiments discussed in the previous subsection were repeated in the case of scenario-2, where the last move of the game of a player in a given period, is exploited and affects the first move of the game of the player in the next period. Such kind of exploration was not covered in [23] . However, these experiments too are capable of drawing some light about the emergence of cooperation in a different context of application of IPD. For simplicity, in the Figures from 10.10a to 10.10c only the $fitness()$ behavior, with the adaptation errors allowed, is reported.

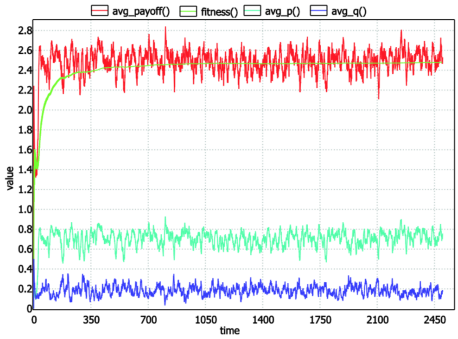
As one can see from Fig. 10.10a and Fig. 10.10b also in the scenario-2 a cooperation regime can ultimately be reached for the PTG and PRN networks, although now with a more modest cooperation level (about). The TRN network



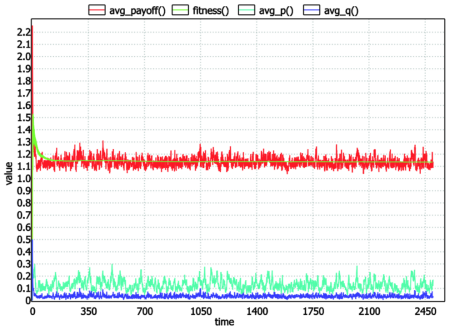
(a) *TRN fitness() vs. time, no errors*



(b) *PTG fitness() vs. time, no errors*

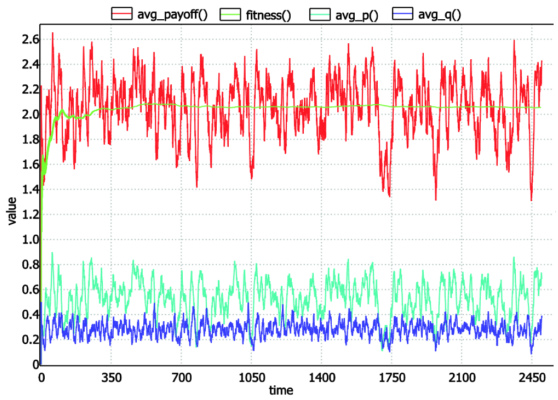


(c) *PRN fitness() vs. time, no errors*

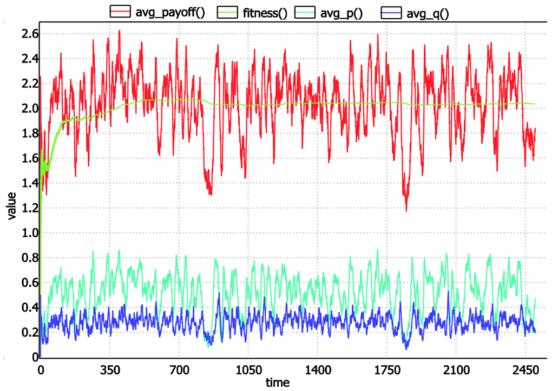


(d) *TRN fitness() vs. time, no errors*

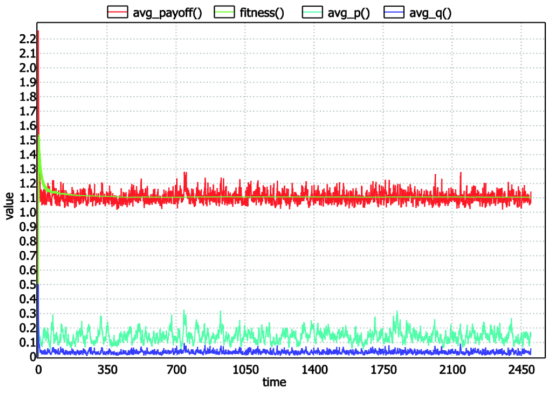
Figure 10.9: Experimental results for the scenario 1



(a) *PTG fitness() vs. time, no errors*



(b) *PRN fitness() vs. time, no errors*



(c) *TRN fitness() vs. time, no errors*

Figure 10.10: Experimental results for the scenario 2

Table 10.2: Observed CIs in the presence of adaptation errors

Model)	**Confidence Interval** (95%)	**Number of runs**
PTG	[0.902606,1]	36
PRN	[0.87865,0.978536]	118
TRN	[0, 0.0973938]	36

Table 10.3: Observed CIs without adaptation errors

Model)	**Confidence Interval** (95%)	**Number of runs**
PTG	[0.18312,0.282947]	291
PRN	[0.222444,0.322213]	322
TRN	[0, 0.0973938]	36

continues to exhibit only the defective regime (Fig. 10.10c). In particular, the query:

```
Pr[<=2500]([] (now<1800 || fitness() >= 2.0))
```

proposes the confidence intervals reported in the Table 10.2.

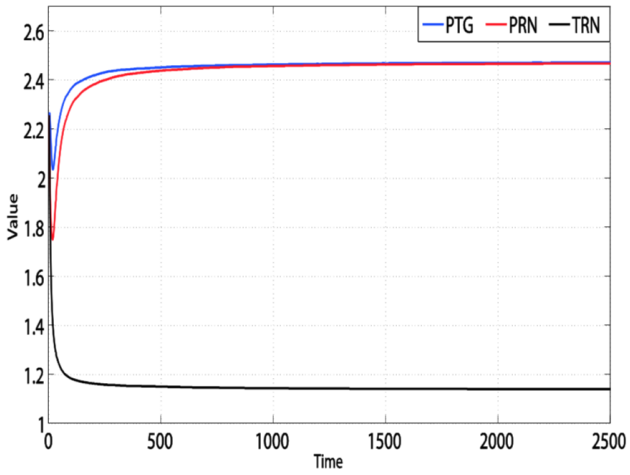
The same query, launched on the models without adaptation errors, suggested the confidence intervals shown in Table 10.2.

Table 10.2 confirms also in the scenario-2 a cooperation regime can be finally attained, when adaptation errors are admitted. Without errors (see Table 10.3) a cooperation regime cannot be excluded, but has a lower occurrence probability. Such probabilities are also smaller than the same probabilities observed in the scenario-1 under the same setting.

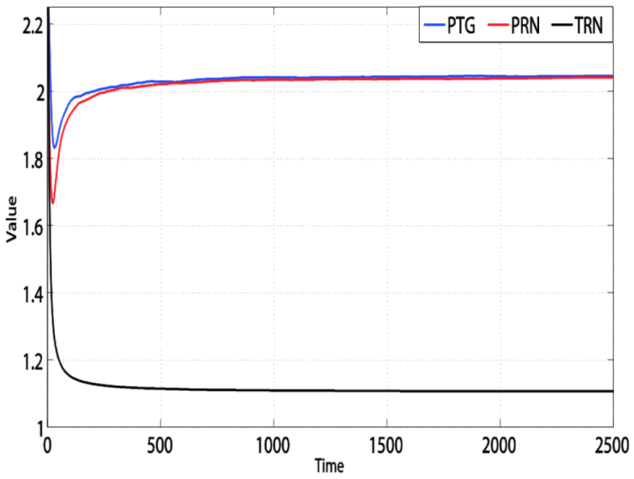
10.4.5 Model validation

The Uppaal SMC models were validated mostly by referring to previous Axelrod work. In particular, experimental results concerning the scenario-1 (no memory between consecutive periods of the last move of players) are in good agreement with the results documented in [86][23] . Further derived results in this paper about scenario-2 (there is the memory of the last move of players between consecutive periods) represent a more challenging application context which anyway follows the shape of the results monitored in the scenario-1 although with a smaller level of the achievable cooperation.

The Uppaal SMC models were also implemented in Java as another source of validation. Figure 10.11a and 10.11b show the observed $fitness()$ from the Java models respectively in the scenario-1 and in the scenario-2, with adaptation errors included. The curves are the average of 100 runs.
The developed IPD models and the obtained experimental results confirm the intuition that link persistence, i.e., playing with the same partners throughout the game, is the key for cooperation because it favors players trustiness. All of this has an obvious interpretation nowadays when one considers people interactions through a social network.
Experiments were carried out on a Linux machine, Intel Xeon CPU E5-1603@2.80GHz, 32GB, using *Uppaal* 4.1.19 64bit.



(a) *fitness*() *vs. time, scenario-1*



(b) *fitness*() *vs. time, scenario-2*

Figure 10.11: Observed *fitness*()

Chapter 11

Conclusions

The work described in this thesis focusses on the development and application of the THEATRE actor system [196]. THEATRE is a mature control-based software engineering framework, which favors the construction of predictable distributed probabilistic and timed systems based on thread-less actors and asynchronous message passing. THEATRE makes it possible to structure a system as a federation of theatres (computing nodes or Logical Processes-LPs). Each theatre hosts a collection of local actors whose evolution is regulated by a customizable control form which manages a time notion and reflectively controls the message exchanges (scheduling and dispatching) among actors. Starting from a preliminary prototype of THEATRE which was achieved on top of JADE [78], the goal was implementing a more efficient version directly upon the built-in mechanisms of Java. In particular, an effective use of network sockets was realized which improves distributed execution and enables migration of actors from a theatre to another. In the course of this thesis, THEATRE was adapted to work with the programming style and timing model of Probabilistic and Timed Rebeca (PTRebeca) [129], that is a well-established formal modelling language suited to specifying and checking properties of complex probabilistic and timed actor systems. A library of control forms was developed for the extended version of THEATRE, which supports both standalone and distributed versions of concurrent, simulation and real-time applications. Original contributions of this thesis are as follows.

- Providing formal semantics to THEATRE models, using a structural operational semantic approach [196, 129, 210].
- Defining a reduction of THEATRE onto UPPAAL model checkers, which permits qualitative, non-deterministic analysis by exhaustive model checking, and quantitative evaluation of model properties by statistical model checking.
- Establishing a development methodology centered on the concept of *model continuity*, that is transitioning, without distortions, a same model from early analysis down to design and implementation, thus contributing to the *faithfulness* of a final system synthesis to the analyzed model.

- Experimenting with the use of THEATRE in several case studies thus demonstrating its effectiveness for formal modelling, verification, prototyping and implementation of time-dependent systems.

The described research work can be continued in the following directions.

- Improving formal verification of THEATRE models by:
  - Optimizing the UPPAAL reduction of THEATRE by, e.g., (a) avoiding explicit message arguments transmission and emulating it by replicated message servers in actor definition; (b) exploiting symmetry reduction techniques [99].
  - Implementing a tool similar to Afra for Rebeca [1] (or possibly directly adapting Afra to Theatre) which starting from the transition system of a THEATRE model formal operational semantics, can automate the generation of a single automaton process (or module) which can be specialized either for UPPAAL or for probabilistic model checkers such as PRISM or IMCA [129]. By avoiding the parallel composition of multiple components, the approach could ensure better model scalability.
  - Representing a THEATRE model into the terms of some more abstract modelling language like stochastic Time Petri Nets, which can facilitate formal modelling and verification. Preliminary experience is reported in [190, 191].
- Continuing experiments with the use of THEATRE for studying and implementing time-synchronization algorithms in large wireless sensor networks [197], with a focus on energy saving. Clock alignment in these scenarios can be essential for proper interpretation of large collection of data.
- Specializing the use of THEATRE for cyber-physical systems design and implementation with the possibility of including, during analysis, continuous time components. Preliminary steps towards the development of an hybrid version of THEATRE, which separates in a clear way the aspects of a CPS model concerning the dynamical laws of the external environment (expressed by continuous modes), from their interaction with the discrete-time, discrete-event actor-based controlling software, are described in [198]. The experience treated in [78] based on the introduction of a specialized component (*envGateway*) for interfacing the system environment, appears as an interesting starting point for this endeavor.
- Exploiting THEATRE actors for embedded real-time systems, IoT-based applications and structural health monitoring and control systems [147].

List of publications

International Conferences:

1. F. Cicirelli, L. Nigro, P.F. Sciammarella. "Agent-based Model Continuity of Stochastic Time Petri nets", in Proc. of 30th ECMS 2016, Regensburg, Germany, May 31th - June 3rd, 2016.

2. F. Cicirelli, L. Nigro, P.F. Sciammarella. "Model Checking Mutual Exclusion Algorithms Using UPPAAL", Advances in Intelligent Systems and Computing - ISSN 2194-5357, pp. 203-215, Springer, 2016.

3. F. Cicirelli, L. Nigro, P.F. Sciammarella."Agents+Control: A Methodology for CPSs", in Proc. of IEEE/ACM 20th Int. Symp. on Distributed Simulation and Real Time Application, 21-23 September, pp. 45-52, 2016.

4. D. L. Carní, D. Grimaldi, F. Lamonaca, P. F. Sciammarella, V. Spagnuolo. "Setting-up of PPG scaling factors for SpO2% evaluation by smartphone", in Proc. of IEEE International Symposium on Medical Measurements and Applications (MeMeA 2016), Benevento, Italy, May 15-18, 2016.

5. D. L. Carní, D. Grimaldi, F. Lamonaca, A. Nastro, P. F. Sciammarella, M. Vasile. "Measurement technique for the healty and carious teeth based on thermal analysis", in Proc. of IEEE International Symposium on Medical Measurements and Applications (MeMeA 2016), Benevento, Italy, May 15-18, 2016.

6. D.L. Carní, F. Cicirelli, D. Grimaldi, L. Nigro, P. F. Sciammarella. "Exploiting Model Continuity in Agent-based Cyber-Physical Systems", Advances in Intelligent Systems and Computing, ISSN 2194-5357, Springer, 2017.

7. D.L. Carní, F. Cicirelli, D. Grimaldi, L. Nigro, P. F. Sciammarella. "Agent-based Software Architecture for Distributed Measurement Systems and Cyber-Physical Systems Design", in Proc. of IEEE Int. Instrumentation and Measurement Technology Conference (I2MTC 2017), Torino, May 22-25, 2017.

8. L. Nigro, P. F. Sciammarella. "Statistical Model Checking of Multi-Agent Systems", in Proc. of 31st European Conference on Modelling and Simulation (ECMS 2017), Budapest, May 23-26, 2017.

9. L. Nigro, P. F. Sciammarella. "Modelling and Analysis of Distributed Asynchronous Actor Systems using Theatre", Advances in Intelligent Systems and Computing, Springer, 2017.

10. L. Nigro, P. F. Sciammarella. "Statistical Model Checking of Distributed Real-Time Actor Systems", In Proceedings of 21st IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'17), Rome, October 18-20, 2017.

11. D. L. Carní, L. De Vito, F. Lamonaca, G. Mazzilli, M. Riccio, P.F. Sciammarella. "An IoT-enabled Multi-Sensor Multi-user System for Human Motion Measurements", in Proc. of IEEE International Symposium on Medical Measurements and Applications (MeMeA 2017), Rochester, USA, May 7-10, 2017.

12. D. L. Carní, D. Grimaldi, F. Lamonaca, L. Martirano, P. F. Sciammarella. "Towards a unified approach for Distributed Measurement System Technologies", in Proc. Of IEEE International Conference on Environment and Electrical Engineering (EEEIC 2017), Milan, Italy, June 6-9, 2017.

13. D.L. Carní, D. Grimaldi, F. Lamonaca, P. F. Sciammarella. "Mobile Object to Speed Up the Synchronization of IoT Network", in Proc. Of IEEE International Workshop on Measurement and Networking (M&N 2017), 2017.

14. C. Nigro, L. Nigro, P. F. Sciammarella. "Model checking knowledge and committments in multi-agent systems using actors and Uppaal", 32nd European Conf. on Modelling and Simulation (ECMS 2018), May 22-25, Wilhelmshaven, Germany.

15. F. Cicirelli, L. Nigro, P. F. Sciammarella. "Seamless Development in Java of Distributed Real-Time Systems using Actors", Int. Symposium Simulation and Process Modelling (ISSPM 2018), July 21-22, Shenyang, Liaoning, China.

16. L. Nigro, P. F. Sciammarella. "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. D. L. Carní, F. Lamonaca, C. Scuro, P. F. Sciammarella, R. Olivito. "Synchronization of IoT Layers for Structural Health Monitoring", 2018 Workshop on Metrology for Industry 4.0 and IoT. IEEE, Brescia, Italy, April 16-18, 2018.

18. D. L. Carní, F. Lamonaca, C. Scuro, P. F. Sciammarella, R. Olivito. "Internet of Things for Structural Health Monitoring", 2018 Workshop on Metrology for Industry 4.0 and IoT. IEEE, Brescia, Italy, April 16-18, 2018.

19. C. Nigro, L. Nigro, P. F. Sciammarella. "Formal modelling and analysis of probabilistic real-time systems", Int. Congress on Information and Communication Technology (ICICT 2019); Best paper award; London (UK), 25-26 February, Springer, Advances in Intelligent Systems and Computing, ISBN Number - 2194-53572018, 2019.

20. C. Nigro, L. Nigro, P. F. Sciammarella. "Modelling and Analysis of Partially Stochastic Time Petri Nets using UPPAAL Model Checkers", In Proc. of Computing Conference (CC 2019) London (UK), 16-17 July, Springer, Advances in Intelligent Systems and Computing, 2019.

21. L. Nigro, P. F. Sciammarella. "Statistical Model Checking of Cyber-Physical Systems using Hybrid Theatre", In Proc. of Inteligent Systems Conference (IntelliSys) 2019 London (UK), 5-6 September, Springer, Advances in Intelligent Systems and Computing, 2019.

22. L. Nigro, P. F. Sciammarella. "Verification of a Smart Power Control Systems using Hybrid Actors", IEEE WorldS4, 30-31 July, London, 2019.

Journals:

1. C. Nigro, L. Nigro, P. F. Sciammarella. "Modelling and Analysis of Multi-Agent Systems Using Uppaal SMC", Int. J. of Simulation and Process Modelling, Vol. 13, No. 1, pp. 73-87, 2018.

2. F. Cicirelli, L. Nigro, P. F. Sciammarella. "Model continuity in Cyber-Physical Systems: A control centered methodology based on Agents", Simulation Modelling Practice and Theory, Vol. 83, pp. 93-107, 2018, doi 10.1016/j.simpat.2017.12.008.

3. L. Nigro, P. F. Sciammarella. "Qualitative and quantitative model checking of distributed probabilistic timed actors", Simulation Modelling Practice and Theory, doi 10.1016/j.simpat.2018.07.011, vol. 87 (September 2018), pp. 343-368.

4. F. Cicirelli, L. Nigro, P. F. Sciammarella. "Seamless Development in Java of Distributed Real-Time Systems using Actors", Int. J. of Simulation and Process Modeling (IJSPM), in press, 2019.

5. D. L. Carní, D. Grimaldi, F. Lamonaca, L. Nigro, P. F. Sciammarella. "From Distributed Measurement Systems To Cyber-Physical Systems: A Design Approach", International Journal of Computing, [S.l.], pp. 66-73, June 2017. ISSN 2312-5381.

6. D. L. Carní, F. Lamonaca, C. Scuro, P. F. Sciammarella, R. Olivito. "Internet of Things for Structural Health Monitoring", IEEE Instrumentation and Measurement Magazine, In press, 2019.

7. D. L. Carní, D. Grimaldi, F. Lamonaca, R. Olivito, C. Scuro, P. F. Sciammarella. "A layered IoT-based architecture for distributed Structural Health Monitoring System", Acta Imeko, to appear, 2019.

8. C. Nigro, L. Nigro, P. F. Sciammarella. "Formal reasoning on knowledge and commitments in multi-agent systems using theatre", Journal of Simulation, submitted.

Bibliography

[1] Afra: a system verifier. http://ece.ut.ac.ir/FML/afra.html.

[2] Arduino, web-site. https://www.arduino.cc.

[3] Foundation for intelligent physical agents, online. http://www.fipa.org.

[4] Plasma lab, web-site. https://project.inria.fr/plasma-lab/.

[5] Wireshark, web-site. http://www.wireshark.org.

[6] Shadi Abras, Stephane Ploix, Sylvie Pesty, and Mireille Jacomino. *A Multi-agent Home Automation System for Power Management*, pages 59–68. Springer Berlin Heidelberg, 2008.

[7] Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. Modelling and simulation of asynchronous real-time systems using timed rebeca. *arXiv preprint arXiv:1108.0228*, 2011.

[8] Gul Agha, Carl Gunter, Michael Greenwald, Sanjeev Khanna, Jose Meseguer, Koushik Sen, and Prasanna Thati. Formal modeling and analysis of dos using probabilistic rewrite theories. In *Workshop on Foundations of Computer Security (FCS'05)*, volume 20, 2005.

[9] Gul Agha and Karl Palmskog. A survey of statistical model checking. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 28(1):6, 2018.

[10] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[11] Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[12] Faisal Al-Saqqar, Jamal Bentahar, Khalid Sultan, Wei Wan, and Ehsan Khosrowshahi Asl. Model checking temporal knowledge and commitments in multi-agent systems using reduction. *Simulation Modelling Practice and Theory*, 51:45–68, 2015.

[13] Musab AlTurki and José Meseguer. Pvesta: A parallel statistical model checking and quantitative analysis tool. In *International Conference on Algebra and Coalgebra in Computer Science*, pages 386–392. Springer, 2011.

[14] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE, 1990.

[15] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and computation*, 104(1):2–34, 1993.

[16] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *International Colloquium on Automata, Languages, and Programming*, pages 322–335. Springer, 1990.

[17] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[18] Rajeev Alur, Tomás Feder, and Thomas A Henzinger. The benefits of relaxing punctuality. *Journal of the ACM (JACM)*, 43(1):116–146, 1996.

[19] A Anastasopoulos, D Kourousis, S Botten, and G Wang. Acoustic emission monitoring for detecting structural defects in vessels and offshore structures. *Ships and Offshore Structures*, 4(4):363–372, 2009.

[20] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang. 1993.

[21] Mark Astley. The actor foundry: A java-based actor programming environment. *University of Illinois at Urbana-Champaign: Open Systems Laboratory*, 1998.

[22] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[23] Robert Axelrod, Gerald R Ford, Rick L Riolo, and Michael D Cohen. Beyond geography: Cooperation with persistent links in the absence of clustered neighborhoods. *Personality and social psychology review*, 6(4):341–346, 2002.

[24] Robert Axelrod and William Donald Hamilton. The evolution of cooperation. *science*, 211(4489):1390–1396, 1981.

[25] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[26] Joachim Baumann, Fritz Hohl, Kurt Rothermel, and Markus Straßer. Mole–concepts of a mobile agent system. *world wide web*, 1(3):123–137, 1998.

[27] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[28] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[29] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing multi-agent systems with JADE*, volume 7. John Wiley & Sons, 2007.

[30] Richard Bellman. *Dynamic programming*. Courier Corporation, 2013.

[31] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Advanced Course on Petri Nets*, pages 87–124. Springer, 2003.

[32] Roberto Beraldi and Libero Nigro. Distributed simulation of timed petri nets. a modular approach using actors and time warp. *IEEE Concurrency*, 7(4):52–62, 1999.

[33] Roberto Beraldi, Libero Nigro, and Antonino Orlando. Temporal uncertainty time warp: an implementation based on java and actorfoundry. *Simulation*, 79(10):581–597, 2003.

[34] Herbert Praehofer Bernard P. Zeigler, Hessam Sarjoughian. Theory of quantized systems: Devs simulation of perceiving agents. *Cybernetics and Systems*, 31(6):611–647, 2000.

[35] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE transactions on software engineering*, 17(3):259–273, 1991.

[36] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE transactions on software engineering*, 17(3):259–273, 1991.

[37] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded ltl model checking. *arXiv preprint cs/0611029*, 2006.

[38] Doeko JB Bosscher, Indra Polak, and Frits W Vaandrager. Verification of an audio control protocol. In *Theories and experiences for real-time system development*, pages 147–176. World Scientific, 1994.

[39] Benoit Boyer, Kevin Corre, Axel Legay, and Sean Sedwards. Plasma-lab: A flexible, distributable statistical model checking library. In *International Conference on Quantitative Evaluation of Systems*, pages 160–164. Springer, 2013.

[40] Robert S Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine intelligence*. Citeseer, 1985.

[41] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 298–302. Springer, 1998.

[42] Guillaume Brat, Klaus Havelund, SeungJoon Park, and Willem Visser. Java pathfinder-second generation of a java model checker. In *In Proceedings of the Workshop on Advances in Verification*. Citeseer, 2000.

[43] Christopher Brooks, Edward A Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, Haiyang Zheng, Shuvra S Bhattacharyya, Elaine Cheong, II Davis, Mudit Goel, et al. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical report, 2008.

[44] James MW Brownjohn. Structural health monitoring of civil infrastructure. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 365(1851):589–622, 2007.

[45] Manfred Broy, María Victoria Cengarle, and Eva Geisberger. *Cyber-Physical Systems: Imminent Challenges*, pages 1–28. Springer Berlin Heidelberg, 2012.

[46] Giacomo Bucci, Laura Carnevali, Lorenzo Ridi, and Enrico Vicario. Oris: a tool for modeling, verification and evaluation of real-time systems. *International journal on software tools for technology transfer*, 12(5):391–403, 2010.

[47] Carlos E. Budde, Pedro R. D'Argenio, Arnd Hartmanns, and Sean Sedwards. A statistical model checker for nondeterminism and rare events. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 340–358. Springer International Publishing, 2018.

[48] Peter Bulychev, Alexandre David, Kim G Larsen, Axel Legay, Guangyuan Li, and Danny Bøgsted Poulsen. Rewrite-based statistical model checking of wmtl. In *International Conference on Runtime Verification*, pages 260–275. Springer, 2012.

[49] Peter Bulychev, Alexandre David, Kim Guldstrand Larsen, Axel Legay, Guangyuan Li, Danny Bøgsted Poulsen, and Amelie Stainer. Monitor-based statistical model checking for weighted metric temporal logic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 168–182. Springer, 2012.

[50] Alvaro Cardenas, Saurabh Amin, Bruno Sinopoli, Annarita Giani, Adrian Perrig, and Shankar Sastry. Challenges for securing cyber physical systems. In *Workshop on Future Directions in Cyber-physical Systems Security*, July 2009.

[51] Laura Carnevali, Leonardo Grassi, and Enrico Vicario. State-density functions over dbm domains in the analysis of non-markovian models. *IEEE Transactions on Software Engineering*, 35(2):178–194, 2009.

[52] D. L. Carní, F. Cicirelli, D. Grimaldi, L. Nigro, and P. F. Sciammarella. Exploiting model continuity in agent-based cyber-physical systems. In *Advances in Intelligent Systems and Computing*. Springer, 2017.

[53] D L Carní, D Grimaldi, L Nigro, PF Sciammarella, and F Cicirelli. Agent-based software architecture for distributed measurement systems and cyber-physical systems design. In *Instrumentation and Measurement Technology Conference (I2MTC), 2017 IEEE International*, pages 1–6. IEEE, 2017.

[54] DL Carnì, D Grimaldi, G Guglielmelli, and F Lamonaca. Synchronization of measurement instruments co-operating into the w-dms. In *Instrumentation and Measurement Technology Conference Proceedings, 2007. IMTC 2007. IEEE*, pages 1–6. IEEE, 2007.

[55] Domenico Luca Carnì, Carmelo Scuro, Francesco Lamonaca, Renato Sante Olivito, and Domenico Grimaldi. Damage analysis of concrete structures by means of acoustic emissions technique. *Composites Part B: Engineering*, 115:79–86, 2017.

[56] Luigi Carullo, Angelo Furfaro, Libero Nigro, and Francesco Pupo. Modelling and simulation of complex systems using tpn designer. *Simulation Modelling Practice and Theory*, 11(7-8):503–532, 2003.

[57] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. Nusmv 2.4 user manual. *CMU and ITC-irst*, 2005.

[58] Pavol Cerny, Thomas A Henzinger, and Arjun Radhakrishna. Quantitative abstraction refinement. In *ACM SIGPLAN Notices*, volume 48, pages 115–128. ACM, 2013.

[59] Milan Češka, Petr Pilař, Nicola Paoletti, Luboš Brim, and Marta Kwiatkowska. Prism-psy: precise gpu-accelerated parameter synthesis for stochastic systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–384. Springer, 2016.

[60] F. Cicirelli, D. Grimaldi, A. Furfaro, L. Nigro, and F. Pupo. Madams: a software architecture for the management of networked measurement services. *Computer Standards & Interfaces*, 28(4):396–411, 2006.

[61] Franco Cicirelli, Angelo Furfaro, Andrea Giordano, and Libero Nigro. Hla actor repast: An approach to distributing repast models for high-performance simulations. *Simulation Modelling Practice and Theory*, 19(1):283–300, 2011.

[62] Franco Cicirelli, Angelo Furfaro, Andrea Giordano, and Libero Nigro. Performance of a multi-agent system over a multi-core cluster managed by terracotta. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 125–133. Society for Computer Simulation International, 2011.

[63] Franco Cicirelli, Angelo Furfaro, and Libero Nigro. Exploiting agents for modelling and simulation of coverage control protocols in large sensor networks. *Journal of Systems and Software*, 80(11):1817–1832, 2007.

[64] Franco Cicirelli, Angelo Furfaro, and Libero Nigro. Actor-based simulation of pdevs systems over hla. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 229–236. IEEE, 2008.

[65] Franco Cicirelli, Angelo Furfaro, and Libero Nigro. An agent infrastructure over hla for distributed simulation of reconfigurable systems and its application to uav coordination. *Simulation*, 85(1):17–32, 2009.

[66] Franco Cicirelli, Angelo Furfaro, and Libero Nigro. Modelling and simulation of complex manufacturing systems using statechart-based actors. *Simulation Modelling Practice and Theory*, 19(2):685–703, 2011.

[67] Franco Cicirelli, Angelo Furfaro, Libero Nigro, and Francesco Pupo. Agents over the grid: An experience using the globus toolkit 4. In *ECMS*, pages 78–85, 2012.

[68] Franco Cicirelli, Angelo Furfaro, Libero Nigro, and Francesco Pupo. Development of a schedulability analysis framework based on ptpn and uppaal with stopwatches. In *Proceedings of the 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*, pages 57–64. IEEE Computer Society, 2012.

[69] Franco Cicirelli, Andrea Giordano, and Libero Nigro. Efficient environment management for distributed simulation of large-scale situated multi-agent systems. *Concurrency and Computation: Practice and Experience*, 27(3):610–632, 2015.

[70] Franco Cicirelli, Christian Nigro, and Libero Nigro. Qualitative and quantitative evaluation of stochastic time petri nets. In *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*, pages 763–772. IEEE, 2015.

[71] Franco Cicirelli and Libero Nigro. An agent framework for high performance simulations over multi-core clusters. In *Asian Simulation Conference*, pages 49–60. Springer, 2013.

[72] Franco Cicirelli and Libero Nigro. Control aspects in multiagent systems. In *Intelligent Agents in Data-intensive Computing*, pages 27–50. Springer, 2016.

[73] Franco Cicirelli and Libero Nigro. Control centric framework for model continuity in time-dependent multi-agent systems. *Concurrency and Computation: Practice and Experience*, 28(12):3333–3356, 2016.

[74] Franco Cicirelli and Libero Nigro. Exploiting social capabilities in the minority game. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 27(1):6, 2016.

[75] Franco Cicirelli and Libero Nigro. Modelling and verification of mutual exclusion algorithms. In *Proceedings of the 20th International Symposium on Distributed Simulation and Real-Time Applications*, pages 136–144. IEEE Press, 2016.

[76] Franco Cicirelli, Libero Nigro, and Paolo F Sciammarella. Agent-based model continuity of stochastic time petri nets. In *ECMS*, pages 18–24, 2016.

[77] Franco Cicirelli, Libero Nigro, and Paolo F Sciammarella. Model checking mutual exclusion algorithms using u ppaal. In *Software Engineering Perspectives and Application in Intelligent Systems*, pages 203–215. Springer, 2016.

[78] Franco Cicirelli, Libero Nigro, and Paolo F Sciammarella. Model continuity in cyber-physical systems: A control-centered methodology based on agents. *Simulation Modelling Practice and Theory*, 83:93–107, 2018.

[79] Franco Cicirelli, Libero Nigro, and Paolo Francesco Sciammarella. Seamless development in java of distributed real-time systems using actors. In *Int. Symposium Simulation and Process Modelling (ISSPM 2018)*, 2018.

[80] Franco Cicirelli, Libero Nigro, and Paolo Francesco Sciammarella. Seamless development in java of distributed real-time systems using actors. *International Journal of Simulation and Process Modelling*, 13(1):73–87, 2018.

[81] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[82] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[83] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[84] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick P Bloem. *Handbook of model checking*. Springer, 2016.

[85] Charles J Clopper and Egon S Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934.

[86] Michael D Cohen, Rick L Riolo, Robert Axelrod, et al. The emergence of social organization in the prisoner's dilemma: How context-preservation and other factors promote cooperation. Technical report, 1999.

[87] Alexandre David. Merging dbms efficiently. In *17th Nordic Workshop on Programming Theory*, pages 54–56. DIKU, University of Copenhagen, 2005.

[88] Alexandre David, Jacob Illum, Kim G Larsen, and Arne Skou. Model-based framework for schedulability analysis using uppaal 4.1. In *Model-based design for embedded systems*, pages 117–144. CRC Press, 2009.

[89] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 2018.

[90] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikuvcionis, and Danny Bogsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[91] Luca De Alfaro. *Formal verification of probabilistic systems*. Number 1601. Citeseer, 1997.

[92] Luca De Vito, Francesco Lamonaca, Gianluca Mazzilli, Maria Riccio, Domenico Luca Carnì, and Paolo F Sciammarella. An iot-enabled multi-sensor multi-user system for human motion measurements. In *Medical Measurements and Applications (MeMeA), 2017 IEEE International Symposium on*, pages 210–215. IEEE, 2017.

[93] Samuel Deniaud, Philippe Descamps, Vincent Hilaire, Olivier Lamotte, and Sebastian Rodriguez. An analysis and prototyping approach for cyber-physical systems. *Procedia Computer Science*, 56:520–525, 2015.

[94] Edsger W Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[95] Edsger W Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer, 2001.

[96] David L Dill. Timing assumptions and verification of finite-state concurrent systems. In *International Conference on Computer Aided Verification*, pages 197–212. Springer, 1989.

[97] Dimension4. Project website. http://www.thinkman.com/dimension4/.

[98] Johan Eker, Jorn W Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. Taming heterogeneity: the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[99] E Allen Emerson, Somesh Jha, and Doron Peled. Combining partial order and symmetry reductions. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 19–34. Springer, 1997.

[100] Andreas Berre Eriksen, Chao Huang, Jan Kildebogaard, Harry Lahrmann, Kim G Larsen, Marco Muniz, and Jakob Haahr Taankvist. Uppaal stratego for intelligent traffic lights. In *12th ITS European Congress*, 2017.

[101] Richard D Finlayson, Mark Friesel, Mark Carlos, P Cole, and JC Lenain. Health monitoring of aerospace structures with acoustic emission and acousto-ultrasonics. *Insight-Wigston then Northampton-*, 43(3):155–158, 2001.

[102] ACL Fipa and ACL FIPA. Message structure specification. 2002.

[103] Giancarlo Fortino and Libero Nigro. Qos centred java and actor based framework for real/virtual teleconferences. *Proc. of SCS EuroMedia*, 98:4–6, 1998.

[104] Giancarlo Fortino, Libero Nigro, Francesco Pupo, and D Spezzano. Super actors for real time. In *Object-Oriented Real-Time Dependable Systems, 2001. Proceedings. Sixth International Workshop on*, pages 142–149. IEEE, 2001.

[105] Richard M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, New York, NY, USA, 1st edition, 1999.

[106] Angelo Furfaro, Libero Nigro, and Francesco Pupo. Actorserver: a java middleware for programming distributed applications over the internet. In *of the Third International Network Conference (INC 2002)*, pages 433–40, 2002.

[107] Angelo Furfaro, Libero Nigro, and Francesco Pupo. Multimedia synchronization based on aspect oriented programming. *Microprocessors and Microsystems*, 28(2):47–56, 2004.

[108] Eli Gafni and Michael Mitzenmacher. Analysis of timing-based mutual exclusion with random times. *SIAM Journal on Computing*, 31(3):816–837, 2001.

[109] Jorge J. Gomez-Sanz. *Ten Years of the INGENIAS Methodology*, pages 193–209. Springer Berlin Heidelberg, 2014.

[110] C Grosse, H Reinhardt, and Torsten Dahm. Localization and classification of fracture types in concrete with quantitative acoustic emission measurement techniques. *NDT & E International*, 30(4):223–230, 1997.

[111] Christian U Grosse and Markus Krüger. Wireless acoustic emission sensor networks for structural health monitoring in civil engineering. In *Proc. European Conf. on Non-Destructive Testing (ECNDT), DGZfP BB-103-CD*. Citeseer, 2006.

[112] Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R Neuhausser. Quantitative timed analysis of interactive markov chains. In *NASA Formal Methods Symposium*, pages 8–23. Springer, 2012.

[113] Beno Gutenberg and Charles Francis Richter. Magnitude and energy of earthquakes. *Science*, 83(2147):183–185, 1936.

[114] Benjamin M Gyori and Daniel Paulin. Hypothesis testing for markov chain monte carlo. *Statistics and Computing*, 26(6):1281–1292, 2016.

[115] Philipp Haller and Martin Odersky. Actors that unify threads and events. In *International Conference on Coordination Languages and Models*, pages 171–190. Springer, 2007.

[116] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.

[117] Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

[118] Carl Hewitt. Description and theoretical analysis (using schemata) of planner: A language for proving theorems and manipulating models in a robot. Technical report, Massachussetts inst of tech Cambridge artificial intelligence lab, 1972.

[119] Carl Hewitt, Peter Bishop, and Richard Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

[120] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: A tool for automatic verification of probabilistic systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444. Springer, 2006.

[121] Dat Dac Hoang, Hye-Young Paik, and Chae-Kyu Kim. Service-oriented middleware architectures for cyber-physical systems. *International Journal of Computer Science and Network Security*, 12(1):79–87, 2012.

[122] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[123] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[124] Xiaolin Hu and Bernard P. Zeigler. Model continuity to support software development for distributed robotic systems: A team formation example. *J. Intell. Robotics Syst.*, 39(1):71–87, 2004.

[125] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[126] Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.

[127] Ali Jafari, Ehsan Khamespanah, Haukur Kristinsson, Marjan Sirjani, and Brynjar Magnusson. Statistical model checking of timed rebeca models. *Computer Languages, Systems & Structures*, 45:53–79, 2016.

[128] Ali Jafari, Ehsan Khamespanah, Marjan Sirjani, and Holger Hermanns. Performance analysis of distributed and asynchronous systems using probabilistic timed actors. *Electronic Communications of the EASST*, 70, 2014.

[129] Ali Jafari, Ehsan Khamespanah, Marjan Sirjani, Holger Hermanns, and Matteo Cimini. Ptrebeca: Modeling and analysis of distributed and asynchronous systems. *Science of Computer Programming*, 128:22–50, 2016.

[130] J. C. Jensen, D. H. Chang, and E. A. Lee. A model-based design methodology for cyber-physical systems. In *2011 7th International Wireless Communications and Mobile Computing Conference*, pages 1666–1671, July 2011.

[131] Hussein Joumaa, Stephane Ploix, Shadi Abras, and Gregory De Oliveira. A mas integrated into home automation system, for the resolution of power management problem in smart homes. *Energy Procedia*, 6:786–794, 2011.

[132] Gilles Kahn. Natural semantics. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer, 1987.

[133] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software Second Edition*. Dreamtech Press, 2000.

[134] Rajesh K. Karmani and Gul Agha. *Actors*, pages 1–11. Springer US, Boston, MA, 2011.

[135] Gabor Karsai and Janos Sztipanovits. Model-integrated development of cyber-physical systems. In *Proceedings of the 6th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, pages 46–54. Springer-Verlag, 2008.

[136] Ehsan Khamespanah, Marjan Sirjani, Mahesh Viswanathan, and Ramtin Khosravi. Floating time transition system: more efficient analysis of timed actors. In *International Workshop on Formal Aspects of Component Software*, pages 237–255. Springer, 2015.

[137] K. D. Kim and P. R. Kumar. Cyber physical systems: A perspective at the centennial. *Proceedings of the IEEE*, 100:1287–1308, May 2012.

[138] E. Kofman and S. Junco. Quantized state systems. a devs approach for continuous system simulation. *Transactions of SCS*, 18(3):123–132, 2001.

[139] Saul A Kripke. A completeness theorem in modal logic. *The journal of symbolic logic*, 24(1):1–14, 1959.

[140] Nirman Kumar, Koushik Sen, José Meseguer, and Gul Agha. A rewriting based model for probabilistic distributed object systems. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 32–46. Springer, 2003.

[141] Marta Kwiatkowska, Gethin Norman, and David Parker. Symmetry reduction for probabilistic model checking. In *International Conference on Computer Aided Verification*, pages 234–248. Springer, 2006.

[142] Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 220–270. Springer, 2007.

[143] Marta Kwiatkowska, Gethin Norman, and David Parker. A framework for verification of software with time and probabilities. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 25–45. Springer, 2010.

[144] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification*, pages 585–591. Springer, 2011.

[145] H. J. La and S. D. Kim. A service-based approach to designing cyber physical systems. In *9th International Conference on Computer and Information Science*, pages 895–900, 2010.

[146] F Lamonaca, D Grimaldi, R Morello, and A Nastro. Sub-μs synchronization accuracy in distributed measurement system by pda and pc triggers realignement. In *Instrumentation and Measurement Technology Conference (I2MTC), 2013 IEEE International*, pages 801–806. IEEE, 2013.

[147] F Lamonaca, PF Sciammarella, C Scuro, DL Carnì, and RS Olivito. Synchronization of iot layers for structural health monitoring. In *2018 Workshop on Metrology for Industry 4.0 and IoT*, pages 89–94. IEEE, 2018.

[148] Francesco Lamonaca, Domenico Luca Carnì, Maria Riccio, Domenico Grimaldi, and Gregorio Andria. Preserving synchronization accuracy from the plug-in of nonsynchronized nodes in a wireless sensor network. *IEEE Transactions on Instrumentation and Measurement*, 66(5):1058–1066, 2017.

[149] Francesco Lamonaca, Antonio Carrozzini, Domenico Grimaldi, and Renato S Olivito. Improved monitoring of acoustic emissions in concrete structures by multi-triggering and adaptive acquisition time interval. *Measurement*, 59:227–236, 2015.

[150] Francesco Lamonaca, Andrea Gasparri, Emanuele Garone, and Domenico Grimaldi. Clock synchronization in wireless sensor network with selective convergence rate for event driven measurement applications. *IEEE Transactions on Instrumentation and Measurement*, 63(9):2279–2287, 2014.

[151] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[152] Kim G Larsen and Axel Legay. Statistical model checking past, present, and future. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 135–142. Springer, 2014.

[153] Kim G Larsen and Axel Legay. On the power of statistical model checking. In *International Symposium on Leveraging Applications of Formal Methods*, pages 843–862. Springer, 2016.

[154] Kim G Larsen, Marius Mikučionis, Marco Muniz, Jiří Srba, and Jakob Haahr Taankvist. Online and compositional learning of controllers with application to floor heating. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 244–259. Springer, 2016.

[155] Kim G Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock difference diagrams. *Nord. J. Comput.*, 6(3):271–298, 1999.

[156] Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 14–24. IEEE, 1997.

[157] Kim Guldstrand Larsen, Florian Lorber, and Brian Nielsen. 20 years of real real time model validation. In *International Symposium on Formal Methods*, pages 22–36. Springer, 2018.

[158] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of java-based actor programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 468–479. IEEE Computer Society, 2009.

[159] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[160] Edward A Lee. Cyber physical systems: Design challenges. In *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[161] Edward A. Lee. The past, present and future of cyber-physical systems: A focus on models. *Sensors*, 15(3):4837–4869, 2015.

[162] Edward A Lee and Marjan Sirjani. What good are models? In *International Conference on Formal Aspects of Component Software*, pages 3–31. Springer, 2018.

[163] Edward A Lee and Marjan Sirjani. What good are models? In *International Conference on Formal Aspects of Component Software*, pages 3–31. Springer, 2018.

[164] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *International conference on runtime verification*, pages 122–135. Springer, 2010.

[165] Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. Plasma lab: a modular statistical model checking platform. In *International Symposium on Leveraging Applications of Formal Methods*, pages 77–93. Springer, 2016.

[166] Sascha Lehmann, Schupp Sibylle, and Ma M Sc Xintao. Online model checking with uppaal smc. 2016.

[167] James J Leifer and Robin Milner. Deriving bisimulation congruences for reactive systems. In *International Conference on Concurrency Theory*, pages 243–258. Springer, 2000.

[168] Paulo Leitao, Armando Walter Colombo, and Stamatis Karnouskos. Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges. *Computers in Industry*, 81:11 – 25, 2016. Emerging ICT concepts for smart, safe and sustainable industrial systems.

[169] Lu Luo. Software testing techniques: technology maturation and research strategy. *Class report for*, 2001.

[170] Jerome P Lynch and Kenneth J Loh. A summary review of wireless sensors and sensor networks for structural health monitoring. *Shock and Vibration Digest*, 38(2):91–130, 2006.

[171] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer Science & Business Media, 2012.

[172] Masoud Mansouri-Samani, Peter C Mehlitz, Corina S Pasareanu, John J Penix, Guillaume P Brat, Lawrence Z Markosian, Owen O'Malley, Thomas T Pressburger, and Willem C Visser. Program model checking–a practitioner's guide. 2008.

[173] Marco Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, Inc., 1994.

[174] Ian A Mason and Carolyn L Talcott. Actor languages their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 220(2):409–467, 1999.

[175] Philip Merlin and David Farber. Recoverability of communication protocols–implications of a theoretical study. *IEEE transactions on Communications*, 24(9):1036–1043, 1976.

[176] EF Miller. Introduction to software testing technology. *Tutorial: Software Testing & Validation Techniques, Second Edition, IEEE Catalog No. EHO*, 1981.

[177] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[178] Douglas C. Montgomery. *Introduction to statistical quality control*. Wiley, 2009.

[179] R Morello, C De Capua, and A Meduri. Remote monitoring of building structural integrity by a smart wireless sensor network. In *Instrumentation and Measurement Technology Conference (I2MTC), 2010 IEEE*, pages 1150–1154. IEEE, 2010.

[180] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[181] Nicolas Navet and Stephan Merz. *Modeling and verification of real-time Systems: formalisms and software tools*. John Wiley & Sons, 2013.

[182] Xavier Nicollin and Joseph Sifakis. The algebra of timed processes, atp: Theory and application. *Information and Computation*, 114(1):131, 1994.

[183] Brian Nielsen and Gul Agha. Semantics for an actor-based real-time language. In *Parallel and Distributed Real-Time Systems, 1996. Proceedings of the 4th International Workshop on*, pages 223–228. IEEE, 1996.

[184] Brian Nielsen and Gul Agha. Towards reusable real-time objects. *Annals of Software Engineering*, 7(1-4):257–282, 1999.

[185] Brian Nielsen, Shangping Ren, and Gul Agha. Specification of real-time interaction constraints. In *Object-Oriented Real-time Distributed Computing, 1998.(ISORC 98) Proceedings. 1998 First International Symposium on*, pages 206–214. IEEE, 1998.

[186] Christian Nigro, Libero Nigro, and Paolo F Sciammarella. Formal reasoning on knowledge and commitments in multi-agent systems using theatre. *Submitted*.

[187] Christian Nigro, Libero Nigro, and Paolo F Sciammarella. Model checking knowledge and commitments in multi-agent systems using actors and uppaal. In *ECMS*, pages 136–142, 2018.

[188] Christian Nigro, Libero Nigro, and Paolo 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*, pages 136–142, May 2018.

[189] Christian Nigro, Libero Nigro, and Paolo F Sciammarella. Modelling and analysis of multi-agent systems using uppaal smc. *International Journal of Simulation and Process Modelling*, 13(1):73–87, 2018.

[190] Christian Nigro, Libero Nigro, and Paolo F Sciammarella. Formal modelling and analysis of probabilistic real-time systems. In *Int. Congress on Information and Communication Technology (ICICT2019)*, 2019.

[191] Christian Nigro, Libero Nigro, and Paolo F Sciammarella. Modelling and analysis of partially stochastic time petri nets using UPPAAL model checkers. In *Int. Congress on Information and Communication Technology (ICICT2019)*, 2019.

[192] Libero Nigro and Francesco Pupo. Schedulability analysis of real time actor systems using coloured petri nets. In *Concurrent object-oriented programming and petri nets*, pages 493–513. Springer, 2001.

[193] Libero Nigro and Paolo F Sciammarella. Modelling and analysis of distributed asynchronous actor systems using theatre. In *Proceedings of the Computational Methods in Systems and Software*, pages 150–162. Springer, 2017.

[194] Libero Nigro and Paolo 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*, pages 188–195, 2017.

[195] Libero Nigro and Paolo F Sciammarella. Statistical model checking of multi-agent systems. In *Proceedings of 31 st European Conference on Modelling and Simulation (ECMS'17)*, pages 11–17, 2017.

[196] Libero Nigro and Paolo F Sciammarella. Qualitative and quantitative model checking of distributed probabilistic timed actors. *Simulation Modelling Practice and Theory*, 87:343–368, 2018.

[197] Libero Nigro and Paolo F Sciammarella. Time synchronization in wireless sensor networks: A modelling and analysis experience using theatre. In *Distributed Simulation and Real Time Applications (DS-RT), 2018 IEEE/ACM 22th International Symposium on*. IEEE, 2018.

[198] Libero Nigro and Paolo F Sciammarella. Statistical model checking of cyber-physical systems using hybrid theatre. In *Proceedings of Inteligent Systems Conference (IntelliSys) 2019*, 2019.

[199] Vincent Nimal. *Statistical approaches for probabilistic model checking.* PhD thesis, University of Oxford, 2010.

[200] Gethin Norman, David Parker, and Jeremy Sproston. Model checking for probabilistic timed automata. *Formal Methods in System Design*, 43(2):164–190, 2013.

[201] Michael J North and Charles M Macal. *Managing business complexity: discovering strategic solutions with agent-based modeling and simulation.* Oxford University Press, 2007.

[202] Martin Odersky. The scala programming language. *http://www. scala-lang. org/*, 2003.

[203] Masashi Okamoto. Some inequalities relating to the partial sum of binomial probabilities. *Annals of the institute of Statistical Mathematics*, 10(1):29–35, 1959.

[204] HA Oldenkamp. Probabilistic model checking: A comparison of tools. Master's thesis, University of Twente, 2007.

[205] Jason Maximino C Ongpeng, Andres Winston C Oreta, and Sohichi Hirose. Monitoring damage using acoustic emission source location and computational geometry in reinforced concrete beams. *Applied Sciences*, 8(2):189, 2018.

[206] Marco Paolieri, Andras Horvath, and Enrico Vicario. Probabilistic model checking of regenerative concurrent systems. *IEEE Transactions on Software Engineering*, 42(2):153–169, 2016.

[207] Pavel Parizek, Frantisek Plasil, and Jan Kofron. Model checking of software components: Combining java pathfinder and behavior protocol model checker. In *Software Engineering Workshop, 2006. SEW'06. 30th Annual IEEE/NASA*, pages 133–141. IEEE, 2006.

[208] David Park. Concurrency and automata on infinite sequences. In *Theoretical computer science*, pages 167–183. Springer, 1981.

[209] Azaria Paz. Some aspects of probabilistic automata. *Information and Control*, 9(1):26–60, 1966.

[210] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[211] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[212] Danny Bogsted Poulsen. *Statistical Model Checking of Rich Models and Properties.* Phd dissertation, Aalborg University, 2015.

[213] Anuj Puri. Dynamical properties of timed automata. *Discrete Event Dynamic Systems*, 10(1-2):87–113, 2000.

[214] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.

[215] Michael O Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963.

[216] Prakash Ranganathan and Kendall Nygard. Time synchronization in wireless sensor networks: a survey. *International journal of ubicomp*, 1(2):92–102, 2010.

[217] MVMS Rao and KJ Prasanna Lakshmi. Analysis of b-value and improved b-value of acoustic emissions accompanying rock fracture. *Current Science*, pages 1577–1582, 2005.

[218] Rebeca. Project website. http://rebeca-lang.org/.

[219] George M Reed and A William Roscoe. A timed model for communicating sequential processes. In *International Colloquium on Automata, Languages, and Programming*, pages 314–323. Springer, 1986.

[220] Daniël Reijsbergen, Pieter-Tjerk de Boer, and Werner Scheinhardt. Hypothesis testing for rare-event simulation: limitations and possibilities. In *International Symposium on Leveraging Applications of Formal Methods*, pages 16–26. Springer, 2016.

[221] Shangping Ren and Gul A Agha. Rtsynchronizer: language support for real-time specifications in distributed systems. *ACM Sigplan Notices*, 30(11):50–59, 1995.

[222] Shangping Ren, Gul A Agha, and Masahiko Saito. A modular approach for programming distributed real-time systems. *Urbana*, 51:61801, 1996.

[223] Shangping Ren, Nalini Venkatasubramanian, and Gul Agha. Formalizing multimedia qos constraints using actors. In *Formal Methods for Open Object-based Distributed Systems*, pages 139–153. Springer, 1997.

[224] Arni Hermann Reynisson, Marjan Sirjani, Luca Aceto, Matteo Cimini, Ali Jafari, Anna Ingólfsdóttir, and Steinar Hugi Sigurdarson. Modelling and simulation of asynchronous real-time systems using timed rebeca. *Science of Computer Programming*, 89:41–68, 2014.

[225] Gregor Rohbogner, Ulf Hahnel, Pascal Benoit, and Simon Fey. Multi-agent systems' asset for smart grid applications. *Comput. Sci. Inf. Syst.*, 10(4):1799–1822, 2013.

[226] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(4):15, 2009.

[227] T. Sanislav and L. Miclea. An agent-oriented approach for cyber-physical system with dependability features. In *Proceedings of 2012 IEEE International Conference on Automation, Quality and Testing, Robotics*, pages 356–361, May 2012.

[228] Teodora Sanislav and Liviu Miclea. Cyber-physical systems-concept, challenges and research areas. *Journal of Control Engineering and Applied Informatics*, 14(2):28–33, 2012.

[229] Luca Schenato and Federico Fiorentin. Average timesynch: A consensus-based protocol for clock synchronization in wireless sensor networks. *Automatica*, 47(9):1878–1886, 2011.

[230] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *International Conference on Computer Aided Verification*, pages 202–215. Springer, 2004.

[231] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pages 251–252. IEEE, 2005.

[232] Giulio Siracusano, Francesco Lamonaca, Riccardo Tomasello, Francesca Garescì, Aurelio La Corte, Domenico Luca Carnì, Mario Carpentieri, Domenico Grimaldi, and Giovanni Finocchio. A framework for the damage evaluation of acoustic emission signals through hilbert–huang transform. *Mechanical Systems and Signal Processing*, 75:109–122, 2016.

[233] Marvin A Sirbu. Credits and debits on the internet. *IEEE spectrum*, 34(2):23–29, 1997.

[234] Marjan 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*, pages 102–126, 2006.

[235] Marjan 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 Birtday*, 2017.

[236] Marjan Sirjani and Mohammad Mahdi Jaghoori. Ten years of analyzing actors: Rebeca experience. In *Formal Modeling: Actors, Open Systems, Biological Systems*, pages 20–56. Springer, 2011.

[237] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S De Boer. Modeling and verification of reactive systems using rebeca. *Fundamenta Informaticae*, 63(4):385–410, 2004.

[238] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S de Boer. Model checking, automated abstraction, and compositional verification of rebeca models. *J. UCS*, 11(6):1054–1082, 2005.

[239] Fikret Sivrikaya and Bülent Yener. Time synchronization in sensor networks: a survey. *IEEE network*, 18(4):45–50, 2004.

[240] SPIN. Project website. http://spinroot.com/spin/whatispin.html, 2018.

[241] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2:247–254, 1990.

[242] Carolyn Talcott. Actor theories in rewriting logic. *Theoretical Computer Science*, 285(2):441–485, 2002.

[243] Carolyn Talcott. Cyber-physical systems and events. In *Software-Intensive Systems and New Computing Paradigms*, pages 101–115. Springer, 2008.

[244] Konglong Tang, Yong Wang, Hao Liu, Yanxiu Sheng, Xi Wang, and Zhiqiang Wei. Design and implementation of push notification system based on the mqtt protocol. In *International Conference on Information Science and Computer Applications (ISCA 2013)*, pages 116–119, 2013.

[245] RC Tennyson, AA Mufti, S Rizkalla, G Tadros, and B Benmokrane. Structural health monitoring of innovative bridges in canada with fiber optic sensors. *Smart materials and Structures*, 10(3):560, 2001.

[246] Inc Terracotta et al. *The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability: Cluster the JVM for Spring, Hibernate and POJO Scalability*. Apress, 2008.

[247] Robert J van Glabbeek. Bisimulation. In *Encyclopedia of parallel computing*, pages 136–139. Springer, 2011.

[248] Martijn Van Otterlo. Markov decision processes: Concepts and algorithms. *Course on Learning and Reasoning*, 2009.

[249] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

[250] Carlos A Varela. *Programming Distributed Computing Systems: A Foundational Approach*. MIT Press, 2013.

[251] Mahsa Varshosaz and Ramtin Khosravi. Modeling and verification of probabilistic actor systems using prebeca. In *International Conference on Formal Engineering Methods*, pages 135–150. Springer, 2012.

[252] Nalini Venkatasubramanian, Carolyn Talcott, and Gul A Agha. A formal model for reasoning about adaptive qos-enabled middleware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(1):86–147, 2004.

[253] Enrico Vicario, Luigi Sassoli, and Laura Carnevali. Using stochastic state classes in quantitative evaluation of dense-time reactive systems. *IEEE Transactions on Software Engineering*, 35(5):703–719, 2009.

[254] P. Vrba, P. Tichy, V. Mar, K. H. Hall, R. J. Staron, F. P. Maturana, and P. Kadera. Rockwell automation's holonic and multiagent control systems compendium. *Trans. Sys. Man Cyber Part C*, 41(1):14–30, 2011.

[255] Abraham Wald. Sequential tests of statistical hypotheses. *The annals of mathematical statistics*, 16(2):117–186, 1945.

[256] Shiyong Wang, Jiafu Wan, Daqiang Zhang, Di Li, and Chunhua Zhang. Towards smart factory for industry 4.0: a self-organized multi-agent system with big data based feedback and coordination. *Computer Networks*, 101:158 – 168, 2016. Industrial Technologies and Applications for the Internet of Things.

[257] GT Webb, PJ Vardanega, and CR Middleton. Categories of shm deployments: technologies and capabilities. *Journal of Bridge Engineering*, 20(11):04014118, 2014.

[258] Simon Wimmer and Johannes Holzl. Mdp+ ta= pta: Probabilistic timed automata, formalized (short paper). In *International Conference on Interactive Theorem Proving*, pages 597–603. Springer, 2018.

[259] M. Wooldridge. *An introduction to multi-agent systems*. John Wiley & Sons, second edition, 2009.

[260] X.Hu and B.P. Zeigler. A simulation-based virtual environment to study cooperative robotic system. *Integrated Computer-Aided Engineering*, 12(4):353–367, 2005.

[261] Cheer-Sun D Yang, Amie L Souter, and Lori L Pollock. All du path coverage for parallel programs. In *ACM SIGSOFT Software Engineering Notes*, pages 153–162. ACM, 1998.

[262] Wang Yi. Ccs+ time= an interleaving model for real time systems. In *International Colloquium on Automata, Languages, and Programming*, pages 217–228. Springer, 1991.

[263] Haakan LS Younes and Reid G Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *International Conference on Computer Aided Verification*, pages 223–235. Springer, 2002.

[264] Håkan LS Younes, Marta Kwiatkowska, Gethin Norman, and David Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer*, 8(3):216–228, 2006.

[265] Håkan LS Younes and Reid G Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Information and Computation*, 204(9):1368, 2006.

[266] Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.

[267] Arvin Zakeriyan, Ehsan Khamespanah, Marjan Sirjani, and Ramtin Khosravi. Jacco: More efficient model checking toolset for java actor programs. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 37–44. ACM, 2015.

[268] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation (second ed.)*. Academic Press, New York, 2000.

Part IV

Appendixes

The following appendixes report some further work carried during the PhD studies. In particular, the use of the Stochastic Time Petri Nets formalism, together with an its support in UPPAAL, is exploited as a more abstract modelling language for checking THEATRE actor systems. In addition, as a more specific application domain for THEATRE, a practical example of structural health monitoring and analysis is shown.

Appendix A

Formal Modelling and Analysis of Probabilistic Real-Time Systems [1]

This appendix considers formal modelling and analysis of distributed timed and stochastic real-time systems. The approach is based on Stochastic Time Petri Nets (sTPN) which offer a readable yet powerful modelling language. sTPN are supported by special case tools which can ensure accuracy in the results by numerical methods and the enumeration of stochastic state classes. These techniques, though, can suffer of state explosion problems when facing large models. In this work, a reduction of sTPN onto the popular UPPAAL model checkers is developed which permits both exhaustive non-deterministic analysis, which ignores stochastic aspects and it is useful for functional and temporal assessment of system behavior, and quantitative analysis through statistical model checking, useful for estimating by automated simulation runs probability measures of event occurrence. The appendix provides the formal definition of sTPN and its embedding into UPPAAL. Two case studies are proposed as running examples throughout the paper to demonstrate the practical applicability of the approach.

A.1 Introduction

Many software systems built today are concurrent/distributed in character and have timed and probabilistic/stochastic aspects. For proper operation of these systems, both functional and non-functional (e.g., reliability, timing constraints) correctness has to be checked early in a development. Building a system with a timing violation, in fact, can have severe consequences in the practical case. Therefore, the use of formal tools both for modelling and property analysis is strongly recommended [129, 196].
In this paper, the Stochastic Time Petri Nets (sTPN) formalism [253, 70] is adopted for abstracting the behavior of a timed and probabilistic system. The modelling language is supported as a special case by the Oris toolbox [46] which admits generally distributed timers for transitions (activities), that is not necessarily Markovian, and can exploit *numerical methods* and the *enumeration of*

[1] The material in this chapter is related to publications [190, 191]

stochastic state classes (state graph or transition system) [253, 51] for quantitative analysis. The approach, although accurate in the estimation of system properties, can suffer of *state explosion problems* when facing complex realistic systems.
The work described in this paper claims that a more practical yet general solution for supporting sTPN is possible by using the popular and efficient UPPAAL model checkers, in particular the symbolic model checker [27], for non-deterministic analysis and/or the statistical model checker (SMC) [90, 9], for quantitative evaluation of probability measures of event occurrences. UPPAAL SMC does not build the model state graph but rather depends on simulation runs which are automated according to the desired level of accuracy in the results. Therefore, the memory consumption is linear with the model size, and thus large systems can be modelled and analyzed. Although potentially less accurate than a method which uses numerical techniques, the SMC approach is anyway capable of generating results which are of value from the engineering practical point of view.

A.2 The formalism of Stochastic Time Petri Nets

A.2.1 Basic Concepts

As in classical Petri nets [180], an [253, 51, 196] is composed of a set of places (circles in Fig. A.2), a set of transitions (bars in Fig. A.2) and arcs (arrows in Fig. A.2) connecting places to transitions or transitions to places only. A place is an *input* or *output* place of a transition, depending on if an arc exists which goes from the place to the transition (input arc), or vice versa (output arc). All the net objects have attributes. Places can have *tokens* (small black dots in Fig. A.2), arcs have *weights* (natural numbers, by default 1) to condition transition enabling on the basis of the tokens into the input places, and transitions have *temporal* and *probabilistic/stochastic* information which constrain their firing. A transition is *enabled* if sufficient tokens exist in its input places, as required by the input arc weights. When a transition is enabled, it can fire. At the fire time, a number of tokens are withdrawn from the input places according to the input arc weights, and a number of tokens are deposited into the output places, always in a measure stated by the output arc weights. The firing of a transition is an *atomic* event and can influence the enabling status of the other transitions in the net model.

A.2.2 Syntax

An $sTPN$ is a tuple $(P, T, B, F, I_{nh}, m_0, EFT, LFT, \pi, F)$ where P is a set of places; T a set of transitions, with $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$; B is the *backward function*: $B : P \times T \rightarrow N^+$ which associates an input arc $(p, t) \in B$ with its natural (not zero) weight (default is 1); F is the *forward function*: $F : T \times P \rightarrow N^+$ which associates to an output arc $(t, p) \in F$ its natural (not zero) weight (default 1); I_{nh} is the set of inhibitor arcs (input arcs ending with a black dot in Fig. A.2): , which have an implicit weight of 0. m_0 is the initial marking of the $sTPN$ model, which assigns a number of tokens (also 0) to each

place: $m_0 : P \to N$. EFT and LFT are respectively the *earliest firing time* and the *latest firing time* of a transition, as in the basic Time Petri nets [175, 36]: $EFT : T \to Q^+$, where Q^+ is the set of positive rational numbers including 0, $LFT : T \to Q^+ \cup \{\infty\}$, with $EFT \leq LFT$. The set of transitions consists of two disjoint subsets: $T = T_i \cup T_s$, $T_i \cap T_s = \emptyset$, where T_i is the set of *immediate transitions* (black bars in Fig. A.2), is the set of *stochastic transitions* (white bars in Fig. A.2). Immediate transitions are implicitly associated with the times $EFT = LFT = 0$. In addition, π is a function which associates to each immediate transition a *probabilistic weight*: $\pi : T_i \to [0, 1]$, where $[0, 1]$ is the dense interval of real numbers between 0 and 1. F is a function which associates to each stochastic transition a *probability distribution function (pdf)*, which is constrained in the timing interval $[EFT, LFT]$ which acts as the *support* for the pdf: $F : T_s \to pdf$. By default, the pdf of a stochastic transition is the uniform distribution function defined on the support $[EFT, LFT]$ of the transition. The pdf can be an exponential distribution function (EXP) defined by its rate parameter λ, or it can be a generally distributed non-Markovian pdf. The $sTPN$ formalism adopted differs from the definitions in [253, 51] because arcs can have an arbitrary weight. Moreover, our $sTPN$ language clearly distinguishes the immediate from the timed/stochastic transitions. Only to immediate transitions a probabilistic weight can be attached, whereas in [253, 51] each transition can have its weight.

A.2.3 Semantics

Enabling

A transition t is enabled in a marking m, denoted by: $m[t >$, iff:

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

Firing

An enabled transition can fire. When $t \in T$ fires, it modifies the current marking m into a new marking m' as follows:

$$\tilde{m}(p) = m(p) - B(p, t) \text{ (withdraw sub-phase)}$$

$$m'(p) = \tilde{m}(p) + F(p, t) \text{ (deposit sub-phase)}$$

where $\tilde{m}$ is the *intermediate marking* determined by the withdrawal sub-phase. The two sub-phases (withdraw and deposit) are executed atomically. They are explicitly indicated because the firing of t can change the enabling status (from not enabled to enabled or vice versa) of other transitions in the model, due to the sharing of some input places (conflict situations), both just after the withdraw or after the deposit sub-phase (also considering the existence of inhibitor arcs). A transition t' is said *persistent* to the firing of t iff: $m[t' > \wedge \tilde{m}[t' >$. Transition t' is said *newly enabled*, since the firing of t, if: $m'[t' >$. $sTPN$ assumes that transitions are regulated by *single server* firing semantics. In other terms, each transition will fire its enablings one at a time, sequentially.
Immediate transitions always are fired before stochastic transitions. Let C_i^m be

the candidate set of immediate transitions enabled in marking $m : C_i^m = \{t_i \in T_i | m[t_i >\}$. Transition t_i is chosen for firing with probability:

$$\frac{\pi(t_i)}{\sum_{t_j \in C_i^m} \pi(t_j)}$$

Fireability of timed/stochastic transitions

The firing process of transitions in a *sTPN* model is now described in more details. Each transition, except for the immediate transitions, has a built-in *timer* which is reset at its enabling and automatically advances toward the firing time. An *sTPN* model rests on *global time* and on the fact that *all* the timers increase at the same rate.
Under *non-deterministic semantics*, as in classic Time Petri Nets [175, 36], probabilistic weights and *pdf*s are ignored, and a transition is said *fireable* as long as the timer value is *within* the $[EFT, LFT]$ interval of the transition. Therefore, the transition cannot fire when $timer < EFT$, but it should fire when the timer has reached the EFT and it is less than or equal to the LFT (*last time point*). It is worth noting that, due to conflicts, an enabled transition can lose its enabling at any instant of the timer and even at the last time LFT. It is not possible for a continually enabled transition, to fire beyond the LFT (*strong firing model*). In the case multiple transitions are fireable, one of them is chosen non-deterministically and fires.
Under *stochastic semantics*, at its enabling, a transition gets a sample d (duration) from its associated *pdf* which must be: $EFT \leq d \leq EFT$, then it resets its timer. The transition is fireable when the $timer == d$, provided the transition does not loose its enabling in the meanwhile. When multiple stochastic transitions are fireable, one of them is chosen non-deterministically and it is fired.

A.3 Mapping sTPN onto UPPAAL

sTPN are supported by the TPN Designer toolbox [56]. This way an *sTPN* model can be graphically edited and preliminarily simulated. An *sTPN* model can then be translated into the terms of the timed automata of UPPAAL for model checking. A translated *sTPN* model can be decorated for it to be more convenient for the analysis.
The availability of a high-level modelling language with basic types (*int* and *bool*), arrays and structures of basic types (under statistical model checking is also permitted the type *double*), and C-like functions, greatly facilitated the reduction process. The main points of the translation are summarized in the following:

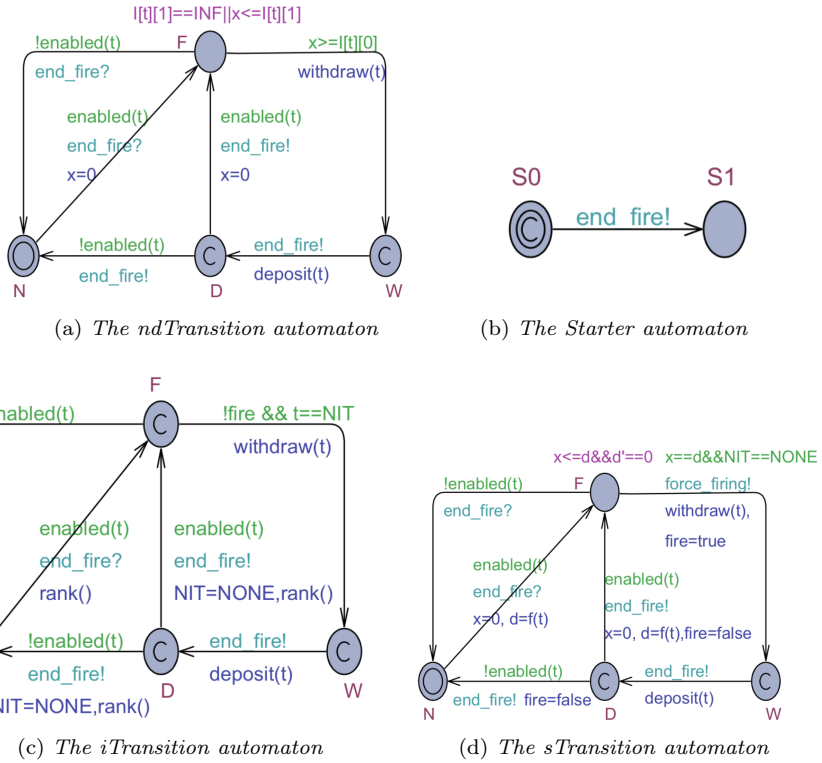
- The number of places (P) and the number of transitions (T) are determined. In particular the number ST of stochastic transitions and the number IT of immediate transitions are defined, with $T = ST + IT$. In addition, constant names for places and transitions are introduced as in the source model.

- The B and F functions of formal syntax are realized by two corresponding constant matrices whose elements are pairs of a place *id* and its associated weight of the input arc or output arc. An inhibitor arc has weight 0.
- The $[EFT, LFT]$ intervals of all the transitions are collected into a (constant) matrix $I : T \times 2$. If t is a transition *id*, $I[t][0]$ holds the EFT and $I[t][1]$ stores the LFT of t. An infinite bound for LFT is coded by the constant $INF = -1$.
- The *pdf* of timed transitions are implemented in a *double* $f(stid)$ which receives the *id* of a stochastic transition and returns a sample of the corresponding *pdf*, constrained into the associated $[EFT, LFT]$ support interval.
- The random switch in a not empty candidate set C_i^m of enabled immediate transitions in current marking, is realized by a function $rank()$ which returns the *id* of the *Next Immediate Transition* (NIT) to fire. A stochastic transition can fire provided $NIT == NONE$, that is there are no immediate transitions to fire.
- The enabling, withdraw and deposit operations of transitions are respectively implemented by the *bool enabled*(*const tid t*), *void withdraw*(*const tid t*), and *void deposit*(*const tid t*) functions. Such fundamental functions receive as parameter the id of a generic transition.

A.3.1 Timed Automata For Transitions

The active part of a reduced $sTPN$ model into UPPAAL is constituted by timed automata each one corresponding to a distinct transition. Basic automata are: $ndTransition(consttidt)$, $sTransition(conststidt)$ and $iTransition(constitidt)$ whose parameter is the unique *id* of the transition. The $ndTransition$ is used when an $sTPN$ model is analyzed by the exhaustive symbolic model checker of UPPAAL. The $stTransition$ and $iTransition$ are instead used when an $sTPN$ model is evaluated with the statistical model checker.
Fig. A.1a depicts the $ndTransition$ automaton of a non-deterministic Time Petri Net transition [175], which captures the basic behavior of any $sTPN$ transition. An $ndTransition$ owns a locally declared clock x which implements the timer explained above. The transition starts in the N (Not enabled) location. From N it moves to the F (under Firing) location as soon as the transition finds itself enabled. When moving from N to F the clock x is reset to measure the time-to-fire. In F the transition can stay as permitted by the LFT time. In the case LFT is infinite, the dwell-time in F is arbitrary. A finite stay in F is imposed by the *invariant* $x <= I[t][1]$ attached to F, in the case of a strict $[EFT, LFT]$ interval. At any instant in time, the transition moves from F to N as it finds itself disabled.
As soon as the clock x reaches the earliest firing time EFT, and the transition keeps continually enabled, the transition can terminate its firing by initiating the withdraw sub-phase and switching to the W location. The complete firing process is achieved by a pair of synchronizations using the end_fire broadcast channel, raised respectively from the committed locations W and D (a committed location has to be left immediately, without time passage; committed

locations have priority over urgent locations, for example, the F locations of transitions which have reached the LFT time). Being broadcast, end_fire is heard by *all* the remaining transitions in the model, which thus can evaluate their enabling status following, respectively, the withdraw and the deposit sub-phase of the transition which is completing its firing.
A subtle point in Fig. A.1a refers to the fact that at the first end_fire synchronization (following the withdraw sub-phase), all the remaining transitions call *enabled*() as a guard and check effectively their enabling status in the intermediate marking established by the withdrawal of tokens. In the second end_fire synchronization, the influenced transitions evaluate their status in the final marking reached after the deposit sub-phase. From D the transition will move to N if it is not enabled, or come back to F, by resetting the clock x, would it be still enabled.
For bootstrapping purposes, a *Starter* automaton (see Fig. A.1b) is used which initially launches a first end_fire synchronization and let transitions to reach the F location or remain in the N location would they be respectively enabled or disabled in the initial marking. After entering the $S1$ location, the Starter will take no further part in the model behavior.



(a) *The ndTransition automaton*

(b) *The Starter automaton*

(c) *The iTransition automaton*

(d) *The sTransition automaton*

Figure A.1: Transition template

Fig. A.1c and Fig. A.1d show respectively the $iTransition$ and the $sTransition$ automata whose basic behavior coincides with that described for the $ndTransition$. The F location in $iTransition$ is a committed location, meaning that an enabled

immediate transition has to complete its firing without time passage. In order to guarantee the atomicity of the firing process of a stochastic transition st, it is fundamental to forbid immediate transitions to exit F before the completion of the firing process of st. The global bool $fire$ variable is set to true by a transition st at its exiting from F (see Fig. A.1d) and put to false at the end of the firing. But an immediate transition can conclude its firing only when it is selected by the $rank()$ function which applies probability weights and realizes the random switch. The automata in Fig. A.1c and A.1d assume that transition conflicts are always *homogeneous*, that is they are composed by the same type of transitions: immediate or stochastic.
The $stTransition$ uses two clocks: x and d (delay). The clock d is assigned the next sample of the pdf of the stochastic transition, returned by the $f(t)$ function. In F the transition remains until x reaches d. During this time, the clock d is frozen by putting its first derivative to 0. The pattern exploited in Fig. A.1d is suggested at page 13, Fig. 22, of the UPPAAL SMC tutorial [90].
Since UPPAAL SMC can have problems exiting the F location of a stochastic transition whose pdf is, e.g., deterministic, a $force\_firing$ urgent broadcast channel is used in Fig. A.1d. This way as soon as the delay is elapsed, F is forced to be exited. The $force\_firing$ broadcast synchronization is non-blocking and it is heard by no one. Only its urgent character is exploited. Both designs in Fig. A.1c and Fig. A.1d improve the work described in [70].
The automata in Fig. A.1c and Fig. A.1d implement in a natural way the semantics of transitions above discussed.

A.4 First Example

In the following, the realistic distributed probabilistic and timed system described in Section 7.2.3, adapted from [196, 129] and concerning the behavior of a sensor network, is considered. In [196, 129] the model was achieved by timed actors and asynchronous message passing.
There is a lab wherein a scientist is working. In the environment of the lab a gas level can grow so as to become toxic. One or multiple sensors are used to monitor the gas level. In the case a toxic gas level is sensed, the life of the scientist is threatened and thus she/he has to be immediately asked to abandon the lab. However, due to non-deterministic and probabilistic behavior, the request to abandon the lab can possibly not occur at all (the sensor can be faulty), or even that the dangerous situation is correctly sensed, the scientist can be in a position of not hearing the request. As a consequence, following a given deadline for the scientist to acknowledge the request-to-abandon the lab, a rescue team is asked to go and reach the lab so as to possibly save the scientist. There is a deadline, tied to the dangerous character of the gas, measured from the time a toxic gas level occurs, for the scientist to be saved. Failing to rescue the scientist, in a way or another, within the deadline causes the scientist to die.
The case study was modelled as an $sTPN$ as shown in Fig. A.2 together with its initial marking. Black bars denote immediate transitions. White bars indicate timed/stochastic transitions. In the model in Fig. A.2 all the stochastic transitions are deterministic. The model was designed to be a *good* abstraction [163] of the chosen system, in the sense that only the relevant actions are reproduced. Not essential aspects are omitted. In particular, the model focusses

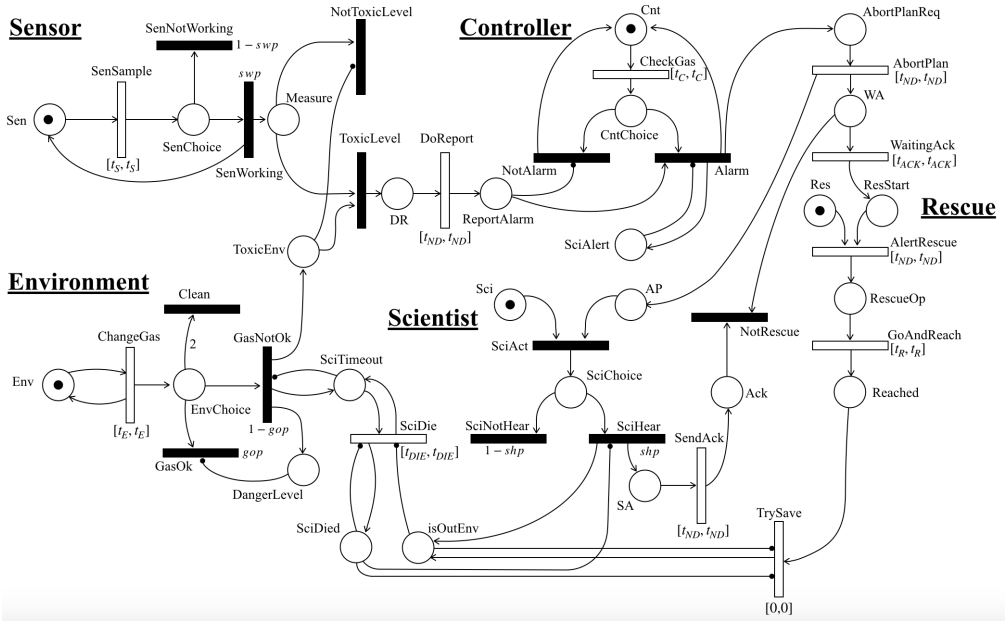


Figure A.2: An $sTPN$ model for a sensor network real-time system

on the reaction of the system to one *single* toxic gas level. The model consists of the following components: the *Environment*, the *Sensor* (can be multiple instantiated), the *Controller* and the *Rescue* and the *Scientist*. Each component is mirrored by a distinct place (*Env*, *Sen*, *Cnt*, *Sci* and *Res*) whose token represents its ability to perform actions.

At each period t_E, the environment choices if the gas level is ok or not ok. In the case the gas is not ok, one token is generated in the place $ToxicEnv$ and in the place *SciTimeout*. Transition *SciDie*, with time t_{DIE}, activates the main deadline for the scientist saving process. Would *SciDie* fire, the scientist dies (one token is generated in the place *SciDied*). The Sensor, with period t_S, samples the environment gas. However, with probability *swp* (sensor working probability) the sensor is actually working, and with probability $1 - swp$ it can become not working. A not working sensor remains faulty forever and will not be able to inform the controller about a toxic gas. All of this was achieved by a *random switch* between the conflicting $SenWorking$ and $SenNotWorking$ immediate transitions of the sensor. In a similar way, the environment decides between gas ok or gas not ok during its operation. It should be noted, though, that following the first firing of $GasNotOk$, the transition will no longer become enabled. A toxic gas level is reported by the sensor to the controller through a firing of the $ToxicLevel$ transition (which can fire only one time), which is followed by a firing of the *DoReport* transition whose timing expresses the net communication delay (t_{ND}). One token in the *ReportAlarm* place, causes an alarm event to be signalled to the controller. However, also the controller has a cyclic behavior with period t_C. Hence, an *Alarm* is actually heard at the *next* period of the controller. Only one firing of the *Alarm* transition can occur during a model reaction. *Alarm* deposits one token in the *AbortPlanReq* which,

after one net delay, triggers an *AbortPlan* event thus depositing one token in the AP place and in the $WaitingAck$ place. Following a t_{ACK} time, the controller knows the scientist was not responding to the abort request. Therefore, the rescue is alerted to go and reach the scientist (the *GoAndReach* transition fires). For realistic modelling, the scientist *can* hear a request to abort plan with probability *shp* and with probability $1 - shp$ she/he cannot hear the request. If the scientist is hearing, then an ack is sent to the controller through the transition *SendAck* which definitely causes the rescue team to be *not* activated. In addition, hearing the abort plan request, determines the scientist to generate one token in the $IsOutEnv$ which mirrors the scientist exited the lab and she/he is saved. In the case the scientist is not hearing, the rescue will try to save the scientist by a firing of the $TrySave$ transition. Such a transition does not fire if the scientist was already saved or she/he died. As a subtle point, $TrySave$ was made timed with $[0, 0]$ time interval, to give priority to the event of responding to the abort plan request (see $NotRescue$) would the ack be generated at the same time.
As a final remark, generating one token in $SciDied$ or $IsOutEnv$ terminates the response of the system to the environmental stimulus of a gas toxic level. The model in Fig. A.2 can easily be extended to accommodate multiple sensors. It is sufficient to replicate the *SenSample* transition and to adjust the initial marking of the *Sen* place to reflect the required number of sensors. All the *SenSample* transitions share *Sen* as the input place and *SenChoice* as the output place.

A.5 Experimental analysis

A model like that in Fig. A.2 naturally requires to be quantitatively analyzed, e.g., by a statistical model checker [9] in order to estimate, for example, the probability for the scientist to be not saved in time when a toxic gas level occurs, when some scenario parameters like those shown in Table A.1 are assumed. It is important to note that the EFT and LFT bounds of transitions *must* be integral values when the non-deterministic model checker of UPPAAL is used.
Some preliminary experiments were carried out using the non-deterministic symbolic model checker of UPPAAL [27], which ignores probability and pdfs. A perfect sensor was assumed (the $SenNotWorking$ transition in Fig. A.2 was omitted). However, the scientist was kept capable of perceiving or not an abort plan request, as well as the environment can choice at each period if the gas level is ok or not. It is worthy of note that by ignoring probability weights, alternate model paths which could occur with very different probabilities, are handled non-deterministically, in the sense that the model checker considers and visits them as occurring with the same probability.

A.5.1 Non-deterministic analysis

The $sTPN$ model of Fig. A.2 reduced to UPPAAL, and with a perfect never faulty sensor, was configured using only the $ndTransition$ template of Fig. A.1a as follows (implicit instantiations of processes occur):

```
system Starter, ndTransition;
```

Table A.1: Scenario parameters for the $sTPN$ model of Fig. A.2

Parameter	Name	Value
Sensor period	t_S	2
Controller period	t_C	3
Environment period	t_E	5
Scientist saving period	t_{DIE}	14
Network delay	t_{ND}	1
Scientist ack deadline	t_{ACK}	2
Rescue time	t_R	3
Sensor working probability	swp	0.99
Gas ok probability	gop	0.98
Scientist hearing probability	shp	0.90

Then the exhaustive model checker of UPPAAL was used which relies on the construc- tion of the model state graph. Properties are specified in the supported subset of the TCTL temporal logic [27].
An important concern was checking that the system does not admit deadlock states:

```
A[] !deadlock
```

This query was found satisfied. This in turn also proved that the model is 2-bounded. In fact, except for the *EnvChoice* place of the Environment, all the other places will have at most 1 token during system evolution. This property was checked by the (sat- isfied) query:

```
A[] forall(p:pid)(p==EnvChoice || M[p]<=1)
```

Since the use of a not faulty sensor, the next checked property was knowing if the intrinsic timing behavior of the model (see parameters in Table A.1) can guarantee the scientist can always be saved when a toxic gas level occurs. The following queries (both satisfied) were used:

```
ndTransition(GasNotOk).W --> M[isOutEnv]==1
```

```
A[] M[SciDied] == 0
```

The first one, based on the $leads-to$ operator, checks if starting from a firing of the $GasNotOk$ transition, it inevitably follows that one token will be generated in the place $IsOutEnv$ (i.e., the scientist is saved). The second query, similarly, checks if *invariantly*, that is in all the states of the state graph, the marking of the $SciDied$ place is without tokens.
Since in the assumed operating conditions, the scientist gets saved, the following query (satisfied) was used to assess that the saving is effectively performed by the *AbortPlan* request or through the rescue team (see the $TrySave$ transition in Fig. A.2):

```
ndTransition(GasNotOk).W --> ndTransition(SciHear).W
  || ndTransition(TrySave).W
```

A critical issue in the scenario parameters in Table A.1 is the *scientist die deadline* (t_{DIE}) which obviously depends e.g. on the sensor period. Therefore, by changing the sensor period from 1 to 15, and keeping unchanged all the other parameters except for the t_{DIE} value which was set to 30, it was determined

the maximum *end-to-end delay* (EED) between the occurrence of a toxic gas level and the completion of the system reaction which saves the scientist. To this purposes, a decoration clock z was added to the UPPAAL model, which is reset (in the $deposit(tid)$ function) when the $GasNotOk$ transition concludes its firing. Then the clock z was checked when either the $SciHear$ or the $TrySave$ transition concludes its firing, by a query like the following:

```
A[] (ndTransition(SciHear).W || ndTransition(TrySave).W) &&
(M[SciDied]==0 && M[IsOutEnv]==0) imply z op bound
```

where *op* can be >= or <= and the corresponding *bound* is the lower bound or the upper bound of the EED. More precisely, the lower bound (best case response time) is assessed by the greatest value lb which satisfies the above query with the constraint $z >= lb$. Similarly, the lowest value ub which satisfies the above query with the constraint $z <= ub$, establishes the upper bound (worst case response time). Fig. A.3 shows the observed lb and ub values for the monitored EED. The results coincide with those achieved in [196] using actors for modelling the case study.
Some further checks were carried on the non-deterministic model with the sensor which can fail, and thus is not able to inform the Controller about a toxic gas level. It is observed that it is the logic of exhaustive verification that of checking all the state paths, then also the path where the sensor fails:

```
E<> ndTransition(SenNotWorking).W
```

This query is satisfied. As one expected, even assuming a t_{DIE} value greater than the worst case value of the EED emerged, for a given sensor period, in the analysis on the optimistic model, the scientist can now die.
The query:

```
E<> M[SciDied]==1
```

is satisfied and clearly indicates that exists at least one state where the place *SciDied* holds one token. For correctness of the model, the following query was also used to check that in no case the scientist can be both saved *and* died:

```
E<> M[SciDied]==1 && M[IsOutEnv]==1
```

Such query is *not* satisfied. From the non-deterministic analysis it emerges that the $sTPN$ model of the case study is compliant with the actor model developed in [196]. This in turn talks about correctness of the $sTPN$ reduction into UPPAAL.
The analysis confirms the scientist *can* possibly be not saved in the event of a dangerous gas level. This fact raises the important concern of estimating a probability measure for the scientist to be effectively saved under different operating conditions.

A.5.2 Quantitative analysis

The statistical model checker (SMC) of UPPAAL rests on a stochastic interpretation of timed automata. Properties are specified using the Metric Interval Temporal Logic (MITL) and its weighted extension (WMITL) [90]. An $sTPN$ model can be naturally analyzed under SMC because it depends on broadcast synchronizations only [90].

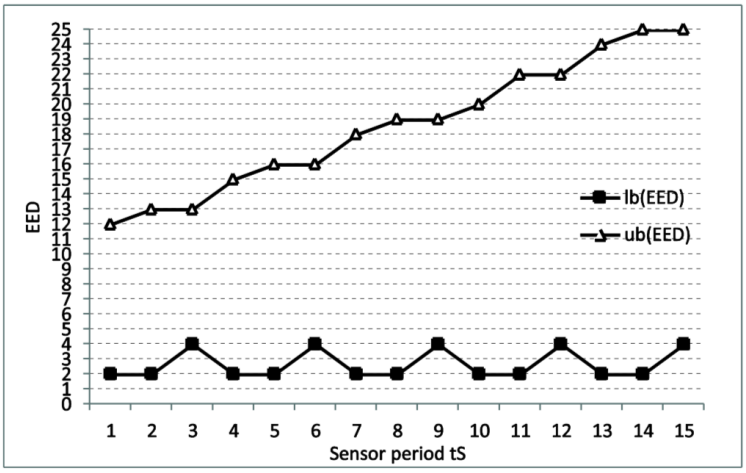


Figure A.3: Observed EED vs. sensor period

Basically, SMC uses simulation runs whose number is dynamically adjusted according to the property to check. Important is the number of time units assigned to each experiment for reaching a conclusion. Such time units should guarantee a dangerous gas level occurs in the environment and sufficient time exists to produce a system response, a sensor not working can happen and the scientist receiving an abort plan request can possibly be not hearing it. The *sTPN* model in Fig. A.2 was configured to exploit stochastic and immediate transitions, thus:

```
system Starter, sTransition, iTransition;
```

The following query:

```
Pr[<=1000] (<> iTransition(GasNotOk).W)
```

asks to quantify the probability of occurrence of a toxic gas by using a certain number of experiments each one lasting after (at maximum) 1000 time units (tu). Each run is actually stopped as soon as the event occurs. By setting the uncertainty error of a confidence interval (CI) as $\epsilon = 0.005$, 3013 runs were used with parameters as in Table A.1. Then the following CI (0.95 confidence degree) was proposed $[0.975932, 0.98593]$ which confirms the expected probability of the event. The query:

```
Pr[<=1000] (<> M[SciDied]==1 && M[IsOutEnv]==1)
```

checks that never should happen that the scientist can be found (absurd) both saved and died. After 368 runs, UPPAAL SMC suggests a CI of $[0, 0.00997405]$ thus witnessing the event is almost impossible. For the sake of simplicity, in the subsequent SMC analysis work, the default error of $\epsilon = 0.05$ was adopted, which implies fewer runs but anyway an acceptable level of accuracy in the results.
In the hypothesis that the toxic gas requires the scientist to be saved within a deadline of 10 tu, the following query based on the until operator [90], with the sensor period varied from 1 to 15 and other parameters as in Table A.1, was used to estimate the probability of the event: *"Would an instant in time in* $[0, 1000]$ *exists where a token is deposited in the SciTimeout place and no*

token is present in the IsOutEnv place, will it happen that a token is put in the IsOutEnv place within the next 10 time units?".

```
Pr[<=1000] (( M[ScTimeout]==1 && M[IsOutEnv]==0)
U[0,10] M[IsOutEnv]==1))
```

Each execution of the query uses 738 runs. The observed confidence interval bounds, when only one sensor is used, are collected in Fig. A.4 and are in good agreement with similar results reported in [196].
As one can see from Fig. A.4, the scientist saving probability is low for small and for high sensor periods. In fact, the more frequent is the sensor sampling the more likely the sensor can be faulty, and thus unable to notify the controller about a dangerous gas level. On the other hand, a sensor with a high period can capture late a toxic gas level thus delaying the controller intervention and putting at a severe risk the life of the scientist. The scientist saving probability

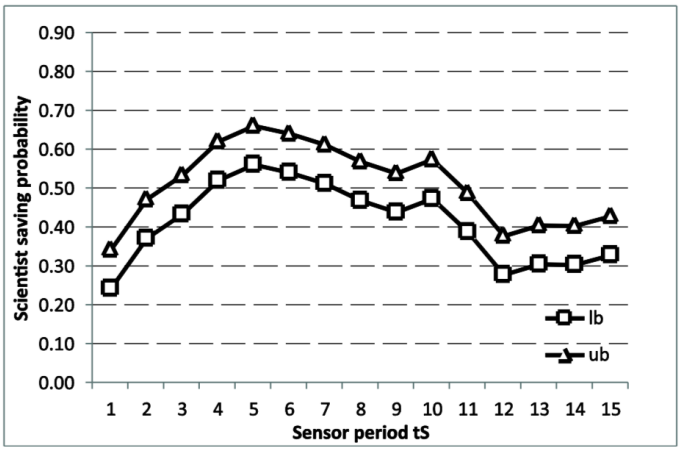


Figure A.4: Scientist saving probability vs. sensor period (one sensor)

increases as the sensor period augments by taking a maximum when the period reaches the value 5. Other local maxima occur at 10 and 15 and so forth. These maximum points correspond to *parameter alignment situations* (the sensor period is a multiple of the environment changing period t_E, see Table A.1), also observed in [196], when the sensor can perceive a bad gas level at the same moment the environment signals it. Near to local maxima, the saving probability keeps high because although the sensor can capture with a small delay a toxic gas, sufficient time remains for the controller to execute the rescue operations.
Fig. A.5 portrays the scientist saving probability when two sensors are used. This experiment was not carried in [196]. Fig. A.5 confirms the expectation that the more are the sensors, the less likely is the circumstance that all the sensors become faulty simultaneously. The same behavior with multiple maxima observed in Fig. A.4 is also present in Fig. A.5 although now the scientist saving probability is higher. The following queries evaluate specifically the probability that the rescue operations are completed respectively by the abort plan request of the controller or through the rescue team intervention. The model is configured with two sensors, the sensor period is $t_S = 5$ and the scientist die deadline is $t_{DIE} = 10$. All the other parameters are as in Table A.1.

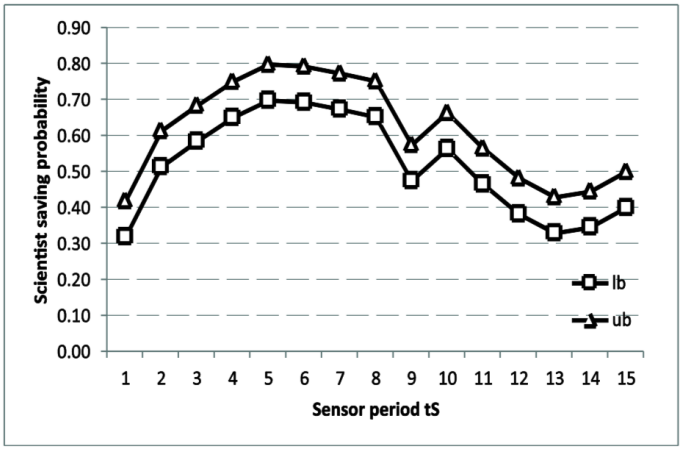


Figure A.5: Scientist saving probability vs. sensor period (two sensors)

```
Pr(<>[0,1000]((M[SciTimeout]==1 && M[IsOutEnv]==0)
U[0,10 iTransition(SciHear).W))

Pr(<>[0,1000]((M[SciTimeout]==1 && M[IsOutEnv]==0)
U[0,10 sTransition(TrySave).W))
```

The first query, after 738 runs, generates a CI of $[0.645122, 0.745122]$. The second query proposes a CI of $[0, 0.0838753]$ thus confirming that the scientist is mostly saved through the first abort plan request issued by the controller.
All the experiments were carried out on a Win 7 station with 4GB RAM using the UPPAAL version 4.1.19 and the 4.1.20.beta25 development version.

A.6 Second Example

In the following an sTPN model of the Fisher's time-based mutual exclusion protocol [108] is shown. The formalism adopted is a light extension of the one reported previously, in which three functions are supposed to be available (*Enabling*, $Withdraw$ and *Deposit*), that allows to investigate if in a place instead of token, there is a number (and eventually, its own value).
The model was adapted from [206] and retains the same timing information for validation purposes. In Fig. A.6 three processes are assumed, whose id are 1, 2 and 3. Shared place *id* holds an integer as its token, whose value can be 0 when no process is interested in entering its critical section. The value of *id* is $i = 1, 2$ or 3 when process i intends to enter its critical section. Place *id* is handled through the $ExtE$, $ExtW$, and $ExtD$ functions.
The $Arrival_i$ transition has an interval $[0, \infty]$ and its pdf is an exponential distribution whose parameter λ is $0.01s^{-1}$. All the other transitions have, by default, a uniform distribution in the associated temporal interval. In particular, $ReadEmpty_i$, $ReadSelf_i$, $Reset_i$, $ReadOther_i$ and $Wait_i$ have a pdf which reduces to a deterministic constant value which is 1.1 for $Wait_i$ and 0 for the other transitions.
Initially, each process is in its non-critical section (NCS) modelled by one token

in the $Idle_i$ place. The dwell-time in NCS is arbitrary as reflected by the $[0, \infty]$ time interval of the $Arrival_i$ transition. After a firing of $Arrival_i$, the process raises its interest to compete for entering its critical section by putting one token in the $Ready_i$ place. After that, the process continues by reading the value in the id place. In the case $id == 0$ the process can go on. If $id \neq 0$, another process assigned its id to the shared place id, and the current process has to remain in the ready status (transition $ReadEmpty_i$ is *not* enabled). When $id == 0$ the process continues by tentatively writing its id to the shared place id. The writing time (W) was supposed to be a non-deterministic value in the interval $[0, 1]$. A key point of the protocol consists now in forcing the process to wait for a time just greater than W. In Fig. A.6 the waiting time is the constant 1.1. After the waiting phase, the process checks again (reading phase) if its id is still in the id place or not. A different value would testify another process attempted the same writing concurrently and succeeded. Therefore the process has to come back into its ready status by a firing of the $ReadOther_i$ transition. When its own id is found in id, the process (through a firing of $ReadSelf_i$) eventually enters its critical section (one token is deposited in the CS_i place). The duration of the critical section is modelled by the time interval of the $Service_i$ transition. After its firing, $Service_i$ puts one token in the $Completed_i$ place and then, through $Reset_i$, the process frees the shared place id by putting 0 in it, and returns to its non-critical section ($Idle_i$ place).
The number of times the $ReadOther_i$ transition fires represents the cases where this process is overtaken by competing processes. Hopefully, the competing time before entering the critical section should be bounded.
The following properties should be checked when proving the correctness of a mutual exclusion, in general untimed, algorithm (see e.g. [75]):

- The algorithm should be deadlock free (*safety*);
- Only one process at a time can be in its critical section (*safety*);
- The waiting time due to competing should be bounded (absence of starvation, *bounded liveness*);
- A not-competing process should not forbid a competing one to enter its critical section (*liveness*);
- No hypothesis should be made about the relative speed of the processes.

It is worth noting that in the literature there are mutual exclusion algorithms which e.g. satisfy the constraint 2, but not the constraint 3. A notable example is the Dijkstra's untimed mutual exclusion algorithm for N processes [95] which it can be proved it does not satisfy the constraint bullet 3.
The analysis work described later aims at showing the interplay between non-deterministic behavior and the stochastic one, by confirming, in particular, that the stochastic behavior can positively contribute to the proper operation of the Fisher's algorithm.

A.7 Analysis of the Fisher's protocol

The Fisher's algorithm in Fig. A.6 was used in [206]for an in-depth analysis of its stochastic behavior, using numerical methods and stochastic state enumeration.

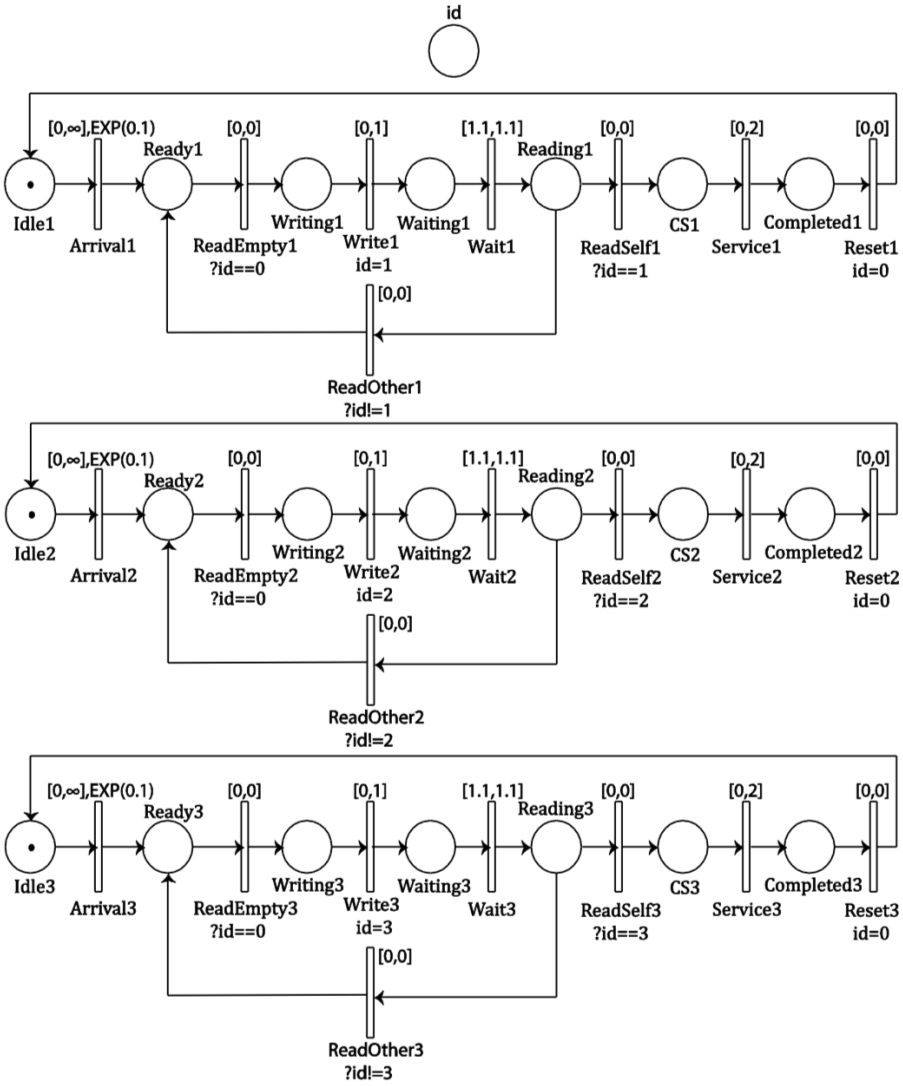


Figure A.6: An $sTPN$ model for the Fisher's timed-based mutual exclusion algorithm, adapted from [206]

The reduction of the psTPN version of the Fisher model in UPPAAL can be thoroughly analyzed under both the non-deterministic interpretation and the stochastic interpretation. Non-deterministic analysis ignores stochasticity and exploits the symbolic model checker of UPPAAL which builds the (hopefully finite) model state graph and permits exhaustively to investigate properties specified in the subset of the TCTL temporal logic language supported by the toolbox [27]. Stochastic analysis depends on the UPPAAL Statistical Model Checker (SMC) which does not build the state graph and uses simulation runs and statistical inference for quantitatively estimating probability measures of event occurrences. The SMC rests on a weighted extension of the Metric Interval Temporal Logic (MITL) for formally specifying system properties.

A.7.1 Non-deterministic analysis

The non-deterministic version of the psTPN Fisher model, adapted so as to convert temporal interval bounds of transitions to integer constants, was extensively studied for *qualitative* behavior: detecting that something *can* happen. Since the three processes are identical, properties can be checked with a reference to one process, e.g. the process 1.
A first concern was to check for the absence of deadlocks in any execution state, with the query:

```
A[] !deadlock
```

The property is satisfied. Then, the mutual exclusion property was verified as follows:

```
A[] M[CS1]+M[CS2]+M[CS3] <= 1
```

This property too is satisfied. Other equivalent queries could be used to assess the mutual exclusion, e.g.:

```
A[] M[CS1]==1 imply M[CS2]==0 && M[CS3] == 0
```

which ensures that if process 1 is in its critical section, necessarily the other processes are not in their critical section. The property is satisfied.
As a further check, the following query gets not satisfied:

```
E<> M[CS1]+M[CS2]+M[CS3] > 1
```

The last queries need to be accompanied by checks which can guarantee that effectively each process can possibly enter its critical section:

```
E<> M[CS1]==1
```

Such existential query is satisfied. The next query checks the property that processes which are in the non-critical-section do not forbid another process to enter its critical section:

```
E<> M[CS1]==1 && M[Idle2]==1 && M[Idle3]==1
```

The query is satisfied. Finally, the absence of starvation was checked with the queries:

```
E<> M[Ready1]==1 satisfied
M[Ready1]==1 --> M[CS1]==1 not satisfied
```

The last query makes use of the *leads-to* operator $--\!>$, which asks if starting from a marking where process 1 is competing, inevitably follows that the process effectively enters its critical section. Unfortunately, UPPAAL says the property is *not* satisfied. Therefore, a competing process can experiment unbounded waiting. This property was not verified in [206].
Considering that during model checking UPPAAL investigates exhaustively *all* the process interleavings, it can be concluded that the Fisher model, under non-deterministic analysis, satisfies all the requirements stated at the end of Section A.6, except for the constraint 3, about starvation which is not fulfilled.
It is useful to note that although each $ndTransition$ owns its local clock, the complexity of model checking does not depend on the total number of clocks but, time to time, on the number of *active* clocks. An active clock is one that, after being reset, is actively referenced in guards or invariants. Therefore, it is the parallelism degree, i.e., the number of simultaneous under firing transitions, which really affects the efficiency of the model checking procedure on a psTPN model.

A.7.2 Stochastic Analysis

Whereas the non-deterministic analysis can indicate something *can* happen, the stochastic analysis can be used to quantify a probability measure of event occurrence. In particular, since the chosen Fisher model does not exclude the starvation of a waiting/competing process, it is interesting to have a quantitative measure of the waiting time for processes wishing to enter their critical section.
As discussed in [206], the stochastic aspects of a model like that of Fisher can positively contribute to improving the properties of the algorithm.
As a preparatory phase of a model to be studied with statistical model checking (SMC), the duration of each experiment (a simulation run) should be estimated. SMC will execute, and also adapt, a number of experiments in order to ensure a certain degree of accuracy in the results. Roughly speaking, the SMC evaluates the number of experiments in which an event occurs and extrapolates a probability measure by a proportion. As soon as the event occurs, the experiment is interrupted.
A first query was used to estimate the mean arrival time of a process, that is the mean time to abandon the non-critical section (place $Idle_i$). Such a time has obviously to tend to the mean time of the exponential distribution of rate 0.1, which is $1/0.1 = 10$ time units. The Fisher model was first *decorated* by adding a double variable s which accumulates the samples generated by the distribution (in the $f(t)$ function) and also counting the number of samples in another variable ns. The decoration function $mean()$ was then prepared to furnish, time by time, the ratio s/ns (at the time 0, when $ns == 0$, $mean()$ returns 0). The following query shows the trend of the $mean()$ value:

```
simulate [<=50000] { mean() }
```

The picture in Fig. A.7 was directly proposed by UPPAAL SMC and confirms that 5×10^4 is a sufficient time for the exponential to converge to its mean time. As a consequence 5×10^4 was used as the limit time of experiments. Liveness and safety properties of the Fisher model were re-checked with SMC as follows:

```
Pr[<=50000] (<>M[CS1]==1)
```

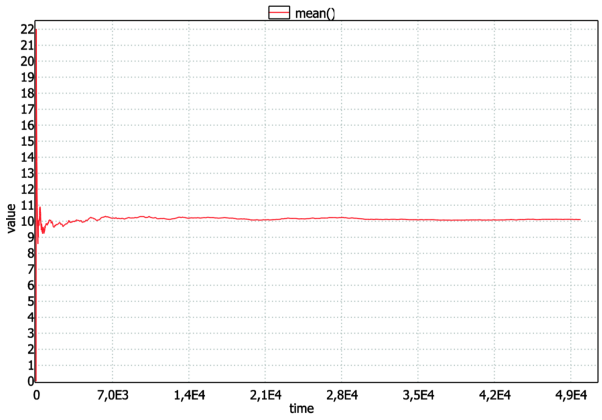


Figure A.7: Monitored mean arrival time of processes

Uppaal SMC, using 299 runs, proposes a confidence interval (CI) for the event "process 1 possibly enters its critical section" of $[0.990031, 1]$ with confidence 0.95. The result was achieved by setting the option $\epsilon = 0.005$ as the error in the evaluation of a probability (the default value is 0.05). The result confirms the event is "almost certain". This option was kept also for the other queries.

Pr[<=50000] (<>M[CS1]+M[CS2]+M[CS3]>1)

Again using 299 runs, Uppaal SMC proposes a CI of $[0, 0.00996915]$ 95% confidence, to suggest the event is "almost impossible" to occur.
To check specifically the fairness of the model, the following decorations were introduced in the model:

- a clock z which is reset at each firing of an $Arrival_1$ transition and each firing of the corresponding $ReadSelf_1$ (when process 1 definitely enters its critical section). Such reset operations are purposely accomplished in the $deposit(tid)$ function;
- a double variable wt to annotate the waiting time when process 1 is competing; more in particular, at each firing of $ReadSelf_1$, the value of clock z is assigned to wt (this type of operation is only allowed in the SMC) before being reset;
- a double variable mwt which is updated, during each experiment, with the maximum value of the observed waiting time held by wt.

The query

E[50000;100] (max:mwt)

asks the SMC to estimate the maximum value of mwt, using 100 experiments each lasting 50000 time units. It is worth noting that in this case, each experiment executes completely for the 50000 time units. Uppaal suggests a value of about 15.868 ± 0.433615. This result implicitly emerges also in the work described in [206]. It confirms that the stochastic behavior of the Fisher protocol helps toward fairness and starvation absence.

Table A.2: Service pdf types used in Fig. A.8b

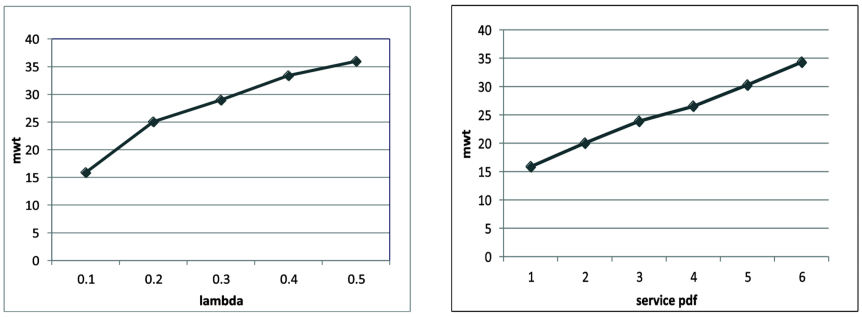
Service type	Service Uniform pdf
1	$[0, 2]$
2	$[0, 3]$
3	$[0, 4]$
4	$[1, 4]$
5	$[2, 4]$
6	$[3, 4]$

Fairness was also studied from a different perspective. The model was further decorated by adding a boolean variable *competing* which takes the value *true* at the firing of $Arrival_1$, and the value $false$ at the firing of $ReadSelf_1$. The following query estimates (using extended weighted MITL) the probability of the event that within the time window $[0, 50000]$, it there exists an instant from which *competing* remains true for at most 13 consecutive time units:

```
Pr(<>[0,50000] ([] [0,13] competing ) )
```

UPPAAL proposes a CI of $[0.897154, 0.997154]$ 95% confidence.
Other experiments were carried out to observe how the estimation of the maximum waiting time (mwt) changes when the parameter λ of the exponential distribution of the *Arrival* transitions, is varied from $0.1, 0.2, 0.3, 0.4$ and 0.5. As expected, a greater process contention caused by higher values of λ, augments the waiting time as shown in Fig. A.8a.
Fig. A.8b shows how mwt varies when the service uniform pdf, which provides the sojourn time in the critical section, is varied from type 1 to type 6 as indicated in the Table A.2. In this scenario too, augmenting the average service time causes an increase in the mwt.

In the light of the reported stochastic analysis, it appears that the chosen



(a) *Maximum waiting time vs.* λ (b) *Maximum waiting time vs. service pdf*

Figure A.8: Query results

Fisher model now fulfills all the requirements of a mutual exclusion algorithm.

A.8 Conclusions

Modelling and formal verification are fundamental tools for the development of distributed and probabilistic timed systems [129, 196]. In this appendix, the

Stochastic Time Petri Nets ($sTPN$) modelling language [253, 51, 70] is adopted. A reduction of $sTPN$ onto the Uppaal model checkers [27, 90] is developed which enables both non-deterministic ex- haustive analysis, and quantitative evaluation of system properties through statistical model checking [9].
Two non-trivial case study concerning a distributed and dependable real-time sensor network and a mutual exclusion protocol are introduced, modelled in $sTPN$, and thoroughly verified in the paper.
Prosecution of the research aims to:

- Building and making available in Uppaal SMC, a library of recurrent pdfs.
- Developing a structural approach to the operational semantics [210] of $sTPN$ (see, e.g., [129, 196]) and formally showing the correctness of the $sTPN$ reduction into Uppaal.
- Exploiting $sTPN$ for performance prediction of general complex timed and stochastic systems. In fact, the particular adopted syntax of $sTPN$ is close to the Generalized Stochastic Petri Nets (GSPN) formalism [173], when the default sup- port interval $[0, \infty]$ is attached to each timed transition and a general distribution probability function (pdf) is chosen.
- Establishing a formal transformation of distributed probabilistic timed actors [129, 196] into $sTPN$ so as to leverage the modelling and verification activities afford- ed by Uppaal. A transformed timed actor model, indeed, as demonstrated through the case study presented in this paper, can be more amenable to analysis due to its abstraction level [163] and greater efficiency and scalability during verification.

Appendix B

A layered IoT-based architecture for distributed Structural Health Monitoring System [1]

Structural Health Monitoring (SHM) is in charge to identify techniques and to prototype systems performing a state diagnosis of structures. Its aim is to prevent a sudden civil infrastructures failures as a result of several not visible damages. Since structural damages often are caused by ground phenomena involving circumscribed geographical areas, it becomes useful to extend the SHM systems to allow the exchange of information among nearby buildings and then to increase the timeliness of the alerts. In this appendix a SHM based on the Internet-of-Things paradigm is proposed (SHM-IoT). SHM-IoT carries out both a *localized monitoring*, on a single building, that using information collected by several sensors correlated in time aims to identify a potentially dangerous damage, and to perform a *wider monitoring* on a group of building in order to alert a larger number of peoples. Hardware and software architectures are presented together whit the synchronization requirements and the methods to satisfy them. Experimental results, obtained are implemented in real scenarios, to validate the proposed system.

B.1 Introduction

The structural collapse of buildings can be caused by the sum of minor damages caused by ageing phenomena prolonged pressure to the structural load curve limit, and much other stress causes that if promptly identified, could prevent loss of human life [232]. With this aim, several inspections, and buildings maintenance programmes are carried out and regulated by law, depending on importance, ownership, use, risk and hazard [44]. These solutions are not sufficient to guarantee high safety levels, because the periodicity of the human inspections carried out are generally unrelated with the unpredictable time with

[1] The material in this chapter is related to publications [147]

which the structure damage events can occur.
The need to carry out continuous and more accurate monitoring has led to the development of the Structural Health Monitoring (SHM) research field [111, 179]. It is in charge to develop methods, techniques and systems voted to perform real-time and automated monitoring of buildings [257] detecting the occurrence of structural damaging events, such as a crack in a concrete pillar, that could provoke the structural failure.
The sensing part of an SHM system is generally constituted by a set of wired or wireless sensors, typically based on the fibre Bragg gratings [245] or MEMS (Micro-Electro-Mechanical-Systems) [44]. Both estimate the impact of static and dynamic loads on a pillar, measuring the structural vibration response. Nowadays, techniques based on the use of the Acoustic Emission (AEs) associated with a crack event shows interesting results in the pillar damage detection. AEs are elastic radiations generated by the release of energy within the material [101], converted to voltage signals through the use of piezoelectric sensors, applied on the different faces of the pillar. The acquired signal contains information about fracture and plastic deformations, impacts, friction, corrosive film rupture, and other ageing processes [55, 205].
The overall system, that represents an improvement of [149], can be decomposed into two sub-sections: the first deals with the monitoring of the state of a single structure, the second, instead, is in charge to send a notification alarm not only within the building but also to the competent authorities and to the other buildings within the neighbourhood. In fact, if a subsoil event has involved a certain area, it is likely that more than a single building may have been damaged. The communication among structures is a feature useful to improve the reliability of the SHM system by cover possible failures in the detection of damaging events occurring in the building. In fact, if a structure receives a critical event notification from a neighbour, the preventive alert is performed, even if its sensing part have not detected any event.
According to the Internet-of-Things (IoT) paradigm [22], the proposed SHM system is designed as a distributed measurement system, that follows a layered implementation. Each layer has a specific role and requires the fulfilment of specific temporal constraints to work properly, depending on the goals that it has to achieve.

B.2 SHM Based On Multi-Agent IoT

The proposed system considers the SHM system as a network of interconnected Smart Objects (SO), that can be installed in a non-invasive way on the pillars of a structure, to carry out the real-time structural monitoring. Each SO is modelled following a hierarchical layered architecture (see Fig. B.1), which is given by the composition of:

- a physical part that includes all the sensors and actuators;
- a cyber part that includes all the algorithm and software protocols.

Challenging in this design is to ensure that all the time constraints on the synchronization accuracy, required by each layer to work properly, are satisfied.

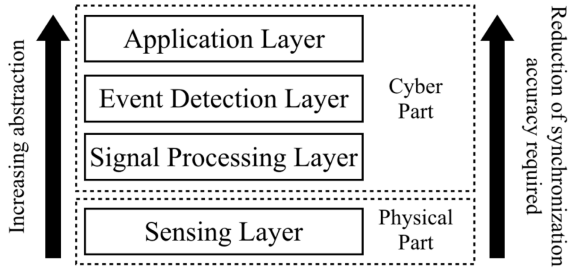


Figure B.1: Hierarchical layered implementation of a SHM system

Moreover, although going up in the hierarchy the growing level of abstraction is coupled with the relaxing of the time constraints, problems could arise from the synchronization functioning among different layers (for example, in operations such as data exchange), which limit the reactivity of the overall system, highlighting a separate block, non-holistic functioning.

By collecting, processing and combining the information coming from the individual SOs, it is possible to realize complex monitoring applications, which are not only in charge of structural damage detection. In particular, the operation computed by each SO, are grouped into two groups. The first group includes all the tasks finalized to the local detection of a damage, such as:

- acquisition of AE signals;
- processing of acquired signals to classify them as associated with a critical event or not;
- correlation of multiple critical events, to infer if a dangerous damage event has occurred.

After the identification of a critical event or the reception of a dangerous event notification, the second group of tasks performs operations aimed at guaranteeing people's safety. Exploiting the IoT paradigm, SOs can communicate with each other through the network [170] and the actions performed include:

- the propagation of an alarm throughout the building in order to evacuate it;
- the sending of a remote message to advise the competent authorities;
- the sending of a remote notification to the other buildings included in a fixed neighbourhood, to suggest their evacuation.

The last feature described, is introduced to increase the reliability of a group of SO installed on the same building and to extend the implementation of an SHM system on a set of structures. In fact, since events generated in the subsoil generally affect a wide geographic area, it is probably that not only a single structure has been damaged, but also its neighbours. So if a building detects its own damage, it spreads the news towards its neighbours, to propose their preventive evacuation, useful to avoid risks due to the eventual failure in the detection of a dangerous event.

All local and remote communication operations performed by the SOs, are carried out exploiting the agent programming paradigm. An agent, as defined in [259], is a software entity capable to perform the autonomous operation in order to reach an established behavioural goal. Each SO is equipped with an agent that, through its properties such as reactivity and proactiveness [259], supervises and manages the monitoring operations, choosing autonomously when it is necessary to interact with other SOs. The main properties that agents offer, useful to model the functioning and the dynamics of a distributed system [259], are:

- sociality and mobility, that simplify the implementation of data exchange and offer a distributed view of the application;
- concurrent data processing that permits to obtain a high computational throughput;
- extensibility that permits the addition of new features to the application or updates the technologies used, avoiding any redesign cost;
- easy detection of malfunctions by simple problem isolation;
- roles decomposition that permits to scale the system.

The whole SHM system can be seen as a federation of interacting agents, also called Multi Agent System (MAS). In this kind of system, critical aspects regard the implementation and synchronization issues related to the developing of a single SO. These aspects, based on THEATRE actors (see chapters 7. and 8.), will be addressed by analysing in detail each layer of Fig. B.1.

B.2.1 Sensing layer

To detect the structural damage, avoiding the use of invasive techniques, as well known in literature, the proposed architecture uses the measurement of the AE generated in a pillar. As shown in Fig. B.2 if a pillar is over-stressed, a set of micro-cracks will be generated within it. If the pressure exceeds for a long time, or if it increases its intensity, these micro-cracks tend to be generated at multiple points inside the pillar. Consequently, if that micro-crack will couple each other, a very critical damage can be originated inside the pillar, that can be recognized detecting the occurrence of the crack in a time window.
As shown in Fig. B.1 the lowest level of the proposed hierarchy is the Sensing Layer. That layer is in charge of carrying out a continuous real-time monitoring of the structure and to spread the alarm if a critical event occurs, exploiting the different SOs installed on the different pillars. The low-level AEs acquisition is achieved through the use of a set of piezoelectric sensors (PSs), equipped on each SO. The main problems resolved in this layer concern with:

- the identification of the signals of interest, distinguishing it among environmental noise and sounds produced by mini-crack;
- the storage and acquisition only of signals associated to potentially critical events, in order to save computational resource;
- the synchronization of measurements coming from different SO, in order to ensure that they are related to the same phenomenon.

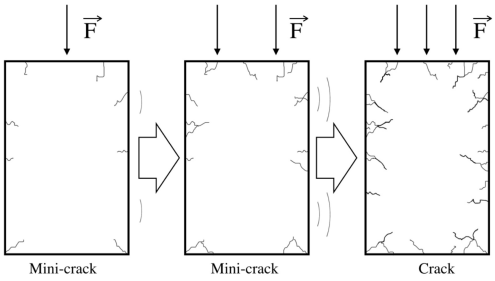


Figure B.2: Example of crack generation

As shown in Fig. B.3, in the proposed architecture, all these issues have been solved in hardware, including the Logic Flat Amplifier and Trigger (L-FAT) component, described in [149]. L-FAT works coupled with the Data AcQuisition system (DAQ) extending its capabilities, and guarantying no loss of signal and no waste of storage memory.
The L-FAT component is in charge to condition the input signals coming from PSs, amplifying their values, and to manage the beginning of the acquisition operations by sending a trigger to the DAQ. That trigger is generated only when one of the signals perceived on the input channels, exceed an experimental fixed threshold. By appropriately calibrating that threshold, it is possible to store and process only signals associated with a potential event of interest. To avoid the loss of signal of interest samples due to the trigger propagation delay (estimated in the order of ns), it is possible to set DAQ functioning in a pre-trigger mode. In this way, the board is enabled to acquire a fixed number of samples before the trigger occurs, estimated for the AEs using the Hsu-Nielsen test [19]. All the operations involved in the acquisition phase are time-critical and a synchronization accuracy in the order of the μs, is needed to establish that the different measurements are related to a same event/phenomenon. The parallel channels architecture of the L-FAT component is able to satisfy that constraint, introducing a propagation delay among the acquired signal estimated in the order of 20 ns, with an uncertainty of few ns [149] and can be considered as negligible.

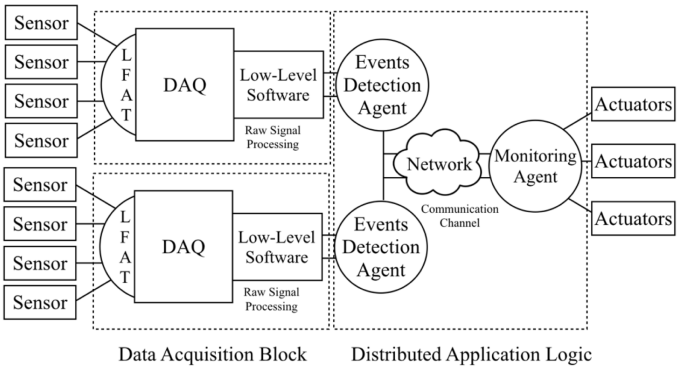


Figure B.3: Distributed Structural Health Monitoring Architecture

B.2.2 Signal Processing Layer

Processing Layer includes the algorithms needed to perform a low-level processing of the acquired signal. In particular, while the Sensing Layer using the L-FAT acquires the signals related to crack, this level acts as a filter, discarding among the signals received all those with an intensity not sufficient to be considered associated to the occurrence of dangerous events.
To achieve this goal, the Signal Processing Layer implements all the mathematical operations and the procedures finalized to:

- realize the processing of the acquired signals;
- make available to the higher software levels, the information about the number of the crack identified as dangerous.

Exploiting the similarity, known in literature [217], among the crack AE signals and those associated to earthquakes, it is possible to use the Gutenberg Richter (GBR) law, in order to estimate the crack intensity and to determine the level damage. Using a variant of the GBR law [113], reported in the equation (1), critical cracks are characterized by a value of b in the neighbor of 1 [55]

$$Log(N) = a - bA_{dm} \tag{B.1}$$

where N is the number of the hits higher than the threshold noise, experimentally fixed at $40dB$. The A_{dm} variable represents the maximum amplitude peak of AE signal. The a and b parameters are two constants fixed experimentally, using techniques reported in [55].
Since that signal law-processing is only a set of algorithmic operations, the processing time cannot be estimated, because it depends on the available computing power.
As shown in Fig. B.4, the Signal Processing Layer keeps track of the number of potentially dangerous cracks, within a Counter Variable (CV), appropriately increased. The value of CV is needed by the higher software level into the hierarchy, to carry out its own operations. To make the system as reactive as possible, the decoupling among the updating time of the Signal Processing Layer (depending from the event occurrence) from the periodicity associated to the Event Detection Layer is required. To perform the data exchange non-blocking mechanism are used, in particular to increase the parallelism and to improve the performances, the CV variable is stored in a shared memory location, accessible in Windows using different software, using the dynamic-link library. Consequently, the main problem in this layer lies in synchronizing the read and write access, between the different software components, to that the shared variable. To overcome this issue, the Dekker mutual exclusion algorithm was used [77].

B.2.3 Event Detection Layer

Continuing to rise in the hierarchy of Fig. B.1, the next level is occupied by the Event Detection Layer. It represents a front-end between the functioning of the single Smart Object and the remaining part of the system and implements operations:

- that allow to identify the occurrence of an effective dangerous damage in the pillar monitored locally by the specific SO;

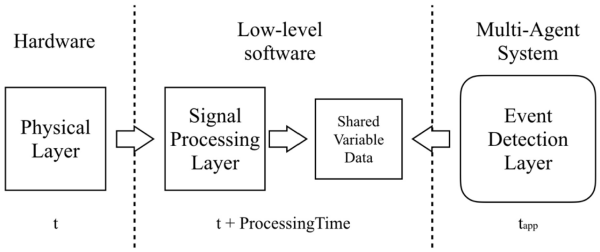


Figure B.4: Layer interactions to perform the low-level crack identification

- that send, through the LAN, of a notification to the node hosting the Application Layer.

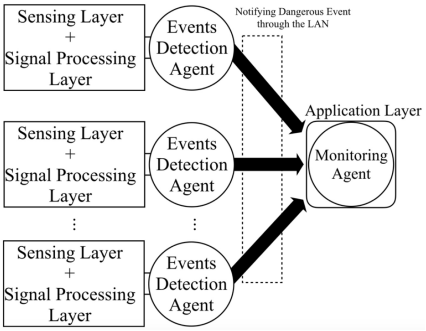


Figure B.5: Event Detection Layer functioning

The operation of this tier and the algorithms needed to achieve its goals are implemented within the behaviour of a software entity called the Event Detection Agent (EDA). Each SO is equipped with one and only one EDA, that exploits the agent message passing protocol to exchange data with the next layer. To detect a dangerous damage within a pillar, the first operation that the EDA algorithm performs is the continuous monitoring of the CV value, according to an observation period TO=1s. This periodicity is acceptable and it is not required a greater resolution, because TO is compatible with the time constraints needed by the higher layer, to perform its operations. Since recent literature [110] assesses that the structure can be considered dangerously damaged, if in a time interval of 60s are detected at least 3 events of interest, if the EDA detects a CV variation equals to 3, it automatically realizes the situation risk and sends the appropriate notifications.

B.2.4 Application Layer

When the Event Detection Layer identifies a dangerous damage, it sends a message to the Monitoring Agent (MA), that implements the high-level operations of the Application Layer (see Fig. B.5). That agent is in charge to:

- enable the actuators (such as alarms) to command the evacuation of the building, sending an alert message to all the SOs widespread in the structure, using the LAN;

- send two kinds of remote notifications using the WAN:
  - the first to the authorities (such as firefighters), to require their intervention
  - the second to the Monitoring Agents installed on the structures included in the neighbourhood.

No strict time constraints are required to implement the notification mechanism via LAN, and globally, via WAN, because any propagation delay, estimated in hundreds of ms [110], is negligible. The reaction-actions that will be performed, in fact, do not require hard the fulfilment of hard real-time constraint. In fact, the addition of a few ms has no impact on the time needed to carry out an evacuation or the implementation or on the arrival of the authorities for the inspection.
The only critical constraint that must be satisfied, is to ensure that both remote and local notification messages will arrive at their destination.
In order to implement the propagation of the warning also to nearby structures, it is necessary that the MA knows and can contact the MAs installed on them. To solve this problem, the proposed SHM system offers two types of solutions. Using the first, each MA statically memorizes information about the neighbours' IPs in a map, appropriately initialized during configuration. The advantage of this type of solution lies in the fact that no further network communication is required to discover the neighbourhood. The disadvantage lies in the fact that if a new structure is built, or if the neighbourhood radius would be extented, a manual reconfiguration is needed. The second proposed solution is a compromise among the two previous considerations. Following the agent philosophy, a single MA can periodically update its list of neighbouring nodes, requesting it to a *yellow pages service*, that acts as a *directory facilitator*. Issues concerning fault tolerance and possible reduction of incoming traffic on the node that offers this service, are resolved with node replication mechanisms.

B.2.5 Remote Transmission Protocol

If all the agents are on a same SO, they can exchange information locally exploiting their sociality propriety, through the local message passing. Instead, if they are deployed on different SOs, the data exchange requires the use of the network and mechanisms that support the correct message delivery in a distributed environment.
The MAS architecture proposed in [78] includes the Gateway component, which has the task of managing the distributed data exchange, exploiting different protocols. In particular, it exposes to the agents the basic *read* and *write* operations needed to interact with physical devices or other remote cyber components, hiding all the details about the communication protocols used. To develop the proposed SHM, the Gateway component has been improved with the introduction of a new module, that enables the data exchange through the Internet.
According to the IoT paradigm, to avoid the active waiting due to the polling cycle between sender and receiver agents (i.e. different MAs or among EDA and MA) and to guarantee the no loss of packets in a distributed communication, the Message Queue Telemetry Transport protocol (MQTT) is used [244]. MQTT is based on the publish-subscribe communication paradigm. The architecture

does not allow a direct data exchange between publisher and subscriber, but provide an entity called broker, which acts as a mediator. Subscribers register themselves to the broker, specifying the topic of the data that they want to receive. When the publisher makes available a data it dispatches that information to the broker, adding a topic string label, that summarizes their content. When the broker receives the data it forwards and delivers them to the appropriate receivers if and only if the associated topic is the same as the one requested. It is worth to note that through the use of the broker, publishers ignore details related to the subscribers' location and vice-versa, guaranteeing the operations a-synchronicity.
MQTT was also chosen because it is designed for networks with low bandwidth and high latency. It uses reduced header and payload for the packet transmission, estimating the transmission upper bounds delays in the order of 56 ms [92]. Furthermore, MQTT offers also three Quality-of-Service (QoS) levels for the reliability of message delivery, summarized as follow:

- Level 0: it guarantees the best effort performance, because a message is delivered at most once and no acknowledgement of reception is required. This level ensures lower transmission times, but no reliability of delivery;
- Level 1: every message is delivered at least once to the receiver and confirmation of message reception is required. This level ensures that the message arrives to the receiver but duplicates can occur;
- Level 2: through a four-way handshake mechanism this level guarantees that each message is received only once by the receiver. It is the safest and also the slowest quality of service level and ensures that delivery occurs avoiding network congestion and packets duplication.

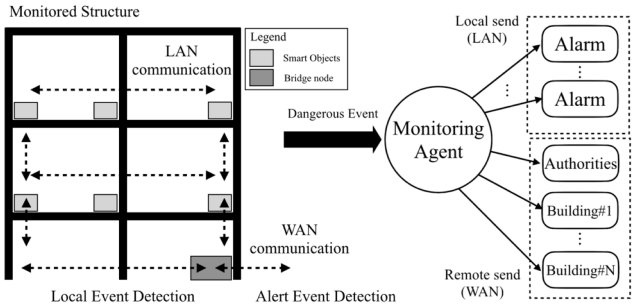


Figure B.6: Application Layer communication

B.2.6 Experimentation validation

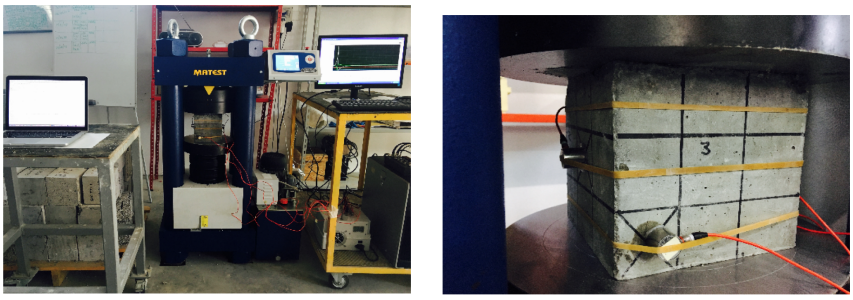
Experiments were carried out to validate the proposed SHM system. Fig. B.7a, shows the experimental test bed. It is composed by:

- Compression system:
  - Matest high stiffness compression machines with load control (Mod. YIMC109NS);

  - PC

- Monitoring system:
  - Sensing Layer:
    - four AE transducers R15α, operating in the frequency range $[50, 200]$ kHz, with a peak sensitivity of 69V/(m/s), resonant frequency 150kHz, and directionality ± 1.5 dB; item the L-FAT component with four input channels;
    - the data acquisition board DAQ is the NI 6110 PCI, allowing a sampling frequency of 5MS/s for each input channel and a resolution of 12-bit;
  - Signal Processing Layer:
    - Hp PC-Desktop, 2Gb, Windows XP, equipped with the DAQ;
  - Application Layer:
    - Hp PC-Desktop, 2Gb, Windows XP, equipped with the DAQ;
    - Macbook Pro Intel Core i5, 2.9GHz, 16GB, OS High Sierra.

Fig. B.7b shows the detail of the sensors positioning over the concrete specimen. In particular, the four sensors were placed on the free faces of the specimen. According to [54] the b-value acceptability parameter, for the detection algorithm, is selected in the range $[0.9 - 1.2]$. The threshold of the L-FAT to send the



(a) *Overview*

(b) *Details of the sensors positioning over the concrete specimen*

Figure B.7: Experimental test bed

acquisition trigger is established with the Hsu-Nielsen [19] test and it is 0.7 V. The number of pre-trigger samples settled in the DAQ is 1000.
Tests were conducted on six specimens, four are characterized by typical resistance used in concrete structures. Two of them are characterized by a very high resistance with respect to the typical values. The results show that, in the case of typical resistance, the three dangerous cracks in the time interval of 60 s have been detected around the 80% of the maximum load curve (Fig. 8). In the other two cases, however, the ISHM system identified only two cracks, instead of three. This may be due to the a and b value used in the identification of the events that are established on the basis of typical resistance values.
The validation of the sensing layer respect to the time constraint is performed by comparing, for each event, the delay among the p-wave of the signals acquired

by the four sensors and by verifying that, according to the speed of the wave in the concrete, this delay is compatible with a position of the crack inside the specimen.
In order to evaluate the one-way delay from publisher to subscriber, it is considered that the IoT devices operate in the same LAN. In this scenario, the one-way delay is evaluated by executing multiple instances of MQTT publishers and subscribers. The subscribers provide multiple messages that flow through the broker to the subscribers.
Fig. B.9 depicts the architecture of the testbed. Several instances of MQTT

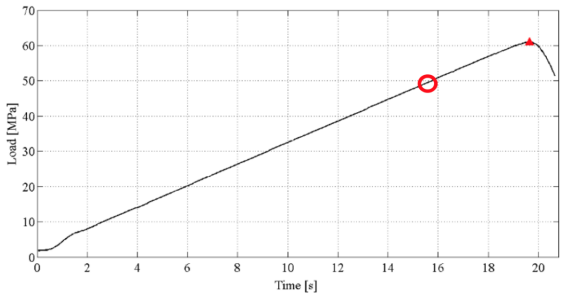


Figure B.8: Load vs Time. Round marker highlights the 80% of the load and triangular marker indicates the maximum load

publishers run on the HP computer. One instance of the selected MQTT broker (Mosquitto) is executed on $PC\#2$. Several instances of MQTT subscribers run on MAC computer. The delay measurement system has been installed on $PC\#4$. The delay measurement system deploys the open source network analyser tool Wireshark [5] to: (i) capture network packets in real time, (ii) select only the packet exchanged by the agents running on the HP and MAC computers, (iii) saving the acquired information, in human-readable format, together with the acquisition timestamp, and (iv) evaluating the one-way delay from the transmission of a packets up to its reception. It is worth to note that the packets are timestamped by Wireshark by using the clock equipping $PC\#4$. This solution does not require the use of protocols to synchronize the clocks equipping the HP and MAC computers to evaluate the packet delay [54]. As a consequence, it is avoided that the synchronization uncertainty characterizing the actual realization of such protocols would degrade the accuracy of the packet delay measurement [146, 148]. All computers in the testbed are connected together to a network hub. This choice allows to capture each packet as soon as it is sent by HP and MAC computers and then to consider the one-way delay values as function of the message flows, only.
Diring the test, HP computer has been used for all the publishers and the Macbook computer for all the subscribers. This does not happen in the actual applicative scenario, where a dedicated computer is typically used for each publisher and subscriber. However, this represents a worst case because: (i) all the publishers/subscribers, share the same computational resources, and (ii) the messages sent by all the publishers are enqueued to the same network interface. Table B.1 shows the results obtained by the experimental testbed considering different numbers of message flows produced by the publishers and received by the subscribers.

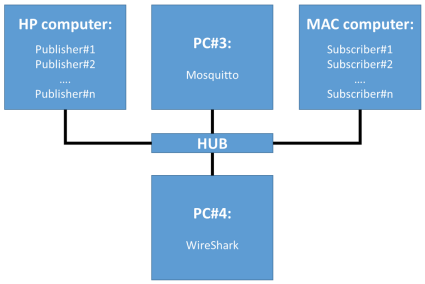


Figure B.9: Application Layer communication

As expected, the values of mean μ and standard deviation σ increase by in-

$\#Publisher$	$\#Subscriber$	$Max[ms]$	$Min[ms]$	$\mu[ms]$	$\sigma[ms]$
1	1	14.62	0.01	0.09	0.82
1	3	13.25	0.01	0.04	0.59
3	3	13.05	0.01	0.43	2.07
5	10	24.38	0.01	3.59	4.39
10	5	21.04	0.01	5.27	4.38
10	10	20.52	0.01	7.41	6.30

Table B.1: Packet delays with respect to multiple publishers and subscribers

creasing the number of message flows. Maximum delay in the order of tens ms is acceptable to correlate in time dangerous events registered on different components of the same structure. The evaluation of the one-way delay from publisher to a subscriber through the Internet is not useful since it is related to human reaction in the case of an event. So delay in the order of some seconds, typically in the case internet is used, are fully acceptable at this level. Vice versa, the guarantee of the reception of the alarm by the authorities is a requirement. Such need is satisfied by using the MQTT with acknowledgement.

Acknowledgements

"E mancano sempre le giuste parole, peró ci sarebbe parecchio da dire. Se vivi la vita in punta di piedi, d'accordo non corri, peró quasi voli".
Si dice che la musica possa aiutare ad esprimere il groviglio di emozioni che ci si porta dentro, cui spesso non si riesce a dare un nome. Condivido a pieno. Questo verso, tratto da una canzone di Marco Mengoni, riassume perfettamente ció che provo qui di fronte questa pagina bianca, che attende solo di essere riempita.
Mi piacerebbe scrivere qualcosa di leggero, capace di associare a quest'altra tappa del dottorato, la stessa semplicitá e naturalezza vissuta nei precedenti traguardi accademici, ma sarebbe una bugia. Al termine di un percorso particolarmente intenso, guardando a me stesso, mi rendo conto di quanto il Paolo di oggi, sia molto diverso da quello di quasi quattro anni fa. Si cresce e si invecchia, fa parte della biologia, ma i processi di maturazione sono altra cosa e, talvolta, per essere innescati, bisogna percorrere strade cosí improbabili, che lí per lí non ci si riesce a spiegare o dargli un senso, ma che col senno del poi, si riesce ad apprezzare. Un discorso degno di chi si sta avvicinando ai 30 anni, non c'é che dire, ma lo scrivo perché lo penso. Veramente. Per quanto mi sia sempre piaciuto guardare il cielo e le nuvole, ho capito che bisogna essere ben provvisti anche di praticitá e disincanto, perché, altrimenti, camminando per strada, stando troppo con il naso all'insú, si finirebbe inevitabilmente dritti contro un palo. In questo senso, ho avuto conferme del fatto che la pretesa ingegneristica dell'associare una logicitá a tutto, puó essere decisamente fuori portata: l'essere umano é complicato ed imprevedibile, gli equilibri nelle relazioni non sono sempre chiarissimi (specialmente quando entrano in gioco diverse sensibilitá) ed a volte puó bastare anche l'innocente citofonata del postino, per rendere una giornata storta. E quando vivi determinate cose, se non vuoi commiserarti o lasciarti schiacciare dal peso delle prove, davanti hai solo una scelta: uscire fuori un po' di carattere, resistere. Non abbiamo un libretto di istruzioni per non rimanerci male, cosí come non abbiamo la garanzia che ci giri sempre bene. Eppure, anche se stanchi, ho capito che ci si puó comunque ritenere fortunati, se ad affrontare le giornate ed i pensieri non si é lasciati da soli, ma si ha qualcuno a fianco. Anche se la vita é simile a delle porte girevoli, con annesso continuo andirivieni di gente, in questi anni mi sono scoperto ricco,

perché ho la consapevolezza di avere avuto accanto persone di valore, capaci di volermi bene, rinforzarmi, sostenermi, darmi l'esempio. Dunque, ecco che dopo questo lungo preambolo, arriva il momento di ringraziare uno per uno, chi a suo modo ho ritenuto determinante per farmi fare quest'ulteriore balzo in avanti. Ci vorranno diverse righe, quindi armatevi di pazienza e... cominciamo!
Alla mia famiglia; perché nonostante sarebbe stato facile disgregarsi a fronte del susseguirsi di eventi degli ultimi due anni, non solo non si é sfilacciata, ma mi ha insegnato che basta non mettere in discussione il bene che ci si vuole l'un l'altro, per trovare sempre il modo di rialzarsi e rimettersi in carreggiata, anche quando la notte puó essere particolarmente buia, o il sole eclissarsi dietro qualche nuvola.
A Mamma perché tra tutti forse é quella che ha dovuto fare un po' piú di lavoro su se stessa, non solo per capirmi, ma anche per imparare a lasciarmi andare. L'amore vero non trattiene e gioisce del fatto che l'altro trovi il suo posto nel mondo, anche se questo dovesse portarlo lontano, ma costa fatica. Tanta. Vivere questo proposito dopo che per 25 anni hai avuto tra le scatole uno che da mattina a sera ti annoiava con domande, ripetizioni di interrogazioni/lezioni/esami e musica a tutto volume, infatti, non é facile, perché ti ritrovi a dover fare i conti con un improvviso silenzio, che a poco a poco si trasforma in vuoti ed una nuova routine, cui puó essere difficile abituarsi (chi l'avrebbe mai detto che saresti stata capace di andare a dormire guardando Striscia la notizia?!). Grazie perché con la libertá che hai saputo accordarmi (conquistata dopo qualche discussione/incomprensione), mi hai ridato un'altra volta la vita e la possibilitá di tracciare una nuova strada, mia.
A Babbo perché mi ha insegnato concretamente cosa significhi non tirarsi indietro, quando per il bene di chi ami ti ritrovi a dover fare ció che é necessario (compreso ingoiare dei rospi), ma che non coincide con quello che vorresti. So che da tifoso del Napoli sei abituato a molta sofferenza e gioie solo sfiorate, ma vedere la serenitá con cui porti avanti i tuoi impegni e la responsabilitá che ti anima ed aiuta a vincere quella stanchezza che logorerebbe chiunque al tuo posto, mi é di prezioso esempio, per coltivare la determinazione necessaria per spendermi in ció in cui credo, senza lasciare che vada sciupato. Grazie, perché mi hai trasmesso "serietá" e spirito di sacrificio, fondamentali nel passaggio da ragazzino ad uomo.
A Pomí perché é sempre stata piú felice ed orgogliosa dei miei successi, di quanto lo fossi io stesso. É bello avere una "tifosa", ma lo é ancora di piú se l'orgoglio e la stima che nutre sono sinceri. Con percorsi ed esperienze diverse, siamo cresciuti entrambi ed ormai non siamo piú i bambini che la notte a cavallo tra 5 e 6 gennaio, si davano continuamente il cambio, in attesa di trovare nel corridoio i tanto sospirati regali portati dalla befana. Tuttavia, mi auguro che tu riesca a conservare un po' di quella sana ingenuitá, necessaria per continuare a sognare. Il tuo esempio mi ha aiutato a capire quanto la vita possa essere difficile, ma tu stessa sei la prova che con un po' di coraggio, ci si puó sempre rialzare. Ci sono eventi particolari che custodiremo sempre dentro di noi; alcuni li hai anche trasformati in tatuaggi, ma non dimenticare che *dai diamanti non nasce niente,é dal letame che nascono i fiori*. Sei la persona che mi convince ancora a credere alle seconde possibilitá; non fermarti mai.
A p. Ciro perché oltre ad essermi stato di aiuto e sostegno in piú circostanze, ha saputo essere piú sognatore e lungimirante di me nei confronti di questo dottorato, spronandomi sia a coglierne il valore, che a non interromperlo pre-

maturamente. Avevi ragione, é stata un'esperienza fruttuosa, non solo perché mi ha aiutato a sviluppare capacitá che effettivamente sono parte di me, ma, soprattutto, perché mi ha messo in condizione di imparare cose banali, che prima non sarei stato in grado di fare, come muovermi da solo in una capitale europea, armato del mio mezzo e sgangherato inglese. Ci sarebbe tanto altro da dire, ma gran parte lo tralascio ed il resto provo a condensarlo in una battuta. Grazie per avermi insegnato l'arte della disponibilitá, anche quando essa costa tempo, sacrifici ed incomprensioni; grazie per avermi fatto capire che per quanto si possa essere circondati di persone, la solitudine é un sentimento umano da tenere in conto e con cui presto o tardi ci si deve misurare; grazie per aver rafforzato ancor di piú la convinzione di quanto sia importante fare la propria parte (anche a costo di consumarsi), pur di voler far funzionare ció cui si dá valore, vincendo la pigrizia del "ma chi me lo fa fare!", con il "sono stanco" di chi ne ha viste tante, ma non si é tirato indietro.
A questo punto, primo colpo di scena: grazie a Davide Perri, perché oltre ad essere stato coinquilino generoso e premuroso (di quelli che all'improvviso entrano in stanza e ti offrono la cioccolata, o che prima di andare a dormire, a fine giornata bussano alla tua porta per darti la buonanotte e chiedere com'é andata), é stato amico sincero, di quelli capaci di intravedere quella piccola macchia, che magari cerchi di nascondere ai piú dietro ad un sorriso. Mi ritengo fortunato ad averti conosciuto, nonostante dopo le 23:30 di sera, sia sempre stato molto difficile discutere insieme in maniera lucida. Adesso studia e laureati... cosí capisci quanta fatica ci vuole per trovare queste parole, che spero ti abbiano strappato un qualcosa a metá tra un sorriso ed una lacrimuccia.
Al Prof. Libero Nigro perché oltre che tutor e riferimento nel mondo accademico, mi ha insegnato una cosa importantissima: l'amore e la passione per quel che si fa. Sembrerá sciocco, ma porteró sempre nel cuore gli scambi di mail dopo mezzanotte, quando ci aggiornavamo sui risultati o sulle ultime correzioni delle bozze dei vari paper: chi al suo posto l'avrebbe fatto? Grazie per questo esempio, che mi ha spinto ad interrogarmi molto anche circa il mio futuro; qualunque strada percorreró spero di riuscire a mettere sempre il 100% di me stesso, proprio come fa lei. Ah, ultima nota (non meno rilevante di quanto detto): grazie per il bene paterno dimostratomi e per tutta la stima che ha sempre riposto in me. Spero di esserne stato all'altezza.
Al Prof. Domenico Grimaldi che purtroppo non é piú qui, e non puó vedere il completamento di questo lavoro. Mi auguro che da lassú sia orgoglioso di quanto scritto, anche se sono certo che qualche modifica all'inglese l'avrebbe data :) ! Grazie anche a lei per avermi fatto sentire figlio e per l'opportunitá che mi ha dato di lavorare "alla pari", offrendomi sempre occasione e modo di dire la mia, nella massima libertá, senza timori. Mi ha sempre messo a mio agio, facendomi capire che con educazione é giusto far sentire la propria voce, a prescindere dal chi si ha davanti, perché nessuno é da meno degli altri, anche se ti puó capitare il saccente di turno che voglia convincerti del contrario (proprio la fiducia nei propri mezzi ed il non lasciarsi impressionare dagli altri, era stato il succo del discorso che mi aveva fatto, qualche giorno prima della mia primissima conferenza!). Grazie anche per avermi spronato a fare l'esercitatore (nonostante qualche mio timore), dandomi carta bianca su tutta la gestione: é stato un segno di fiducia, che ho apprezzato molto ed é stata l'occasione per scoprire (e confermare) una cosa di cui ero mezzo consapevole: insegnare mi piace.

Grazie a Luca e Lavinia che con la loro presenza hanno riempito il mio ufficio, facendolo sembrare meno vuoto, rallegrandomi le giornate.
Agli amici del Santuario (Viviana, Jessica, Giovanni, Lucrezia, Sonia, Gerardo, Federica, Davide, Luca, Daniele e tutti gli altri), perché da semplice gruppo "pu 4 magg", siamo diventati compagni di viaggio ed abbiamo condiviso esperienze cui spero riusciate a dare seguito (no, non mi riferisco alle nostre emozionanti performance nel presepe vivente o alla sfilata dei carri di carnevale, né alle mangiate o ai nostri bans... penso piú a cose belle come il tempo che abbiamo passato alla Stella del Mare). Avete un gran cuore, oltre che belle capacitá: fatele fruttificare!
Ai Giovanissimi (anzi, ormai giovani) della parrocchia San Paolo apostolo di Rende, perché sono stati la "prima volta" in cui mi sono dovuto prendere cura di qualcuno, nelle vesti di educatore/fratello maggiore. Grazie a voi ho sia gustato quanto possa riempire il cuore vedere qualcuno cui vuoi bene crescere, maturare e fare le scelte giuste; che imparato quanto possa essere complicato l'affetto, nel momento in cui si é chiamati a doversi fare da parte, pur di lasciare l'altro libero, facendogli fare con tranquillitá il suo percorso.
In particolare, grazie a Simone e Lollo perché hanno cercato di capire il mio silenzio, oltre che continuato a farmi sorprese e visite, nonostante le vicissitudini ci avessero un po' separato.
Alla comunitá dei Padri Dehoniani di Roma per l'accoglienza datami e l'avermi fatto sentire fin da subito a casa. Grazie per il nuovo inizio e la disponibilitá al volermi conoscere, oltre che all'avermi dato la giusta serenitá nell'ultima parte di questo percorso universitario, sopportando le mie ripetizioni avanti e indietro nel corridoio ed in sala tv.
Ai giovani della parrocchia Ascensione di N.S.G.C. perché in poco tempo mi hanno accolto tra loro, dimostrandomi un affetto tutt'altro che scontato o prevedibile. Benché li conosca da poco, da loro ho giá potuto imparare tre cose: non cedere alle scelte apparentemente piú convenienti, ma che portano a corromperti; il valore del lavoro e della fatica pur di non gravare/campare sulle spalle di mammina e papino; il fatto che ci si puó distinguere da ció che ci circonda, senza dover per forza lasciarsene assorbire. Siete capitati nel momento giusto e con la vostra spontaneitá, mi state anche aiutando a smussare qualcosina di me stesso (PS: grazie anche per avermi fatto mettere a frutto le mie conoscenze, con ripetizioni, interrogazioni e cose del genere: non ci crederete, ma mi state aiutando a trovare dei perché a tutto un percorso fatto, di cui non avevo capito il senso).
A Rosa e Mimmo perché é vero che la stima faccia sempre piacere, ma avere qualcuno che addirittura sceglie concretamente di rimanere al tuo fianco in un periodo buio (anche quando non sarebbe tenuto a farlo!), perché crede sinceramente nel buono che c'é in te, é qualcosa di incredibile ed inestimabile. Le telefonate scambiate sono state importantissime e mai banali, perché sempre utili a vedere il bicchiere mezzo pieno. Grazie per la discrezione, l'affetto e... l'essere "davvero contenti per me", che mi avete ripetuto piú volte con voce convita e carica di gioia; so che é sempre stato sincero. E Non é poco.
A Francesco Lamonaca perché se non si fosse messo nelle orecchie, l'idea del dottorato non mi sarebbe passata per la mente. Grazie per aver creduto nelle mie capacitá ed essermi stato amico/confidente in qualche momento delicato. Peccato per la distanza.
A Domenico Luca Carní perché é stato il migliore vicino d'ufficio che mi potesse

capitare. Grazie per il buonumore, la pazienza e l'avermi messo a mio agio. Peró sii sincero: non ti mancherá neanche un po' il mio romperti le scatole?!
A Vincenzo Inzillo collega ed amico in questo percorso di dottorato, nonché uomo del "senti ci facciamo una passeggiata?". Grazie per le pause, i confronti, il non avermi lasciato annegare nelle scartoffie burocratiche e nei moduli da compilare e per avermi dato ospitalitá in quel di Malaga! Ti auguro il meglio perché te lo meriti e sei un grande: nelle tue stesse condizioni, non credo che avrei reso quanto hai reso tu.
A Scogny perché essere amici non significa vedersi/sentirsi necessariamente tutti i giorni, ma esserci. E chi l'avrebbe mai detto che una di Bologna, conosciuta per caso in una lontana estate del 2013, sarebbe stata a distanza un riferimento?
A p. Luigi per l'inaspettata (ma super piacevole) premura avuta, nello starmi vicino in uno dei periodi piú difficili della mia vita. Forse abbiamo perso un po' di tempo ed occasioni, ma ci rifaremo.
A Trenitalia per la grande compagnia che ha saputo farmi con i suoi mezzi e le sue risorse. Le stazioni sono crocevia di storie e speranze ed hanno molto da raccontare agli occhi di chi seduto in attesa del treno, con le cuffie nelle orecchie, é lí a guardare. Ed anche lí ho saputo prendere ció che c'era di buono: dalla mamma che con cura pulisce il viso della figlia sporco di gelato, alla coppia di fidanzati che fa la corsa per arrivare in tempo al binario, salvo poi sciogliersi in un abbraccio nel vedere sullo schermo quei 7 minuti di ritardo, passando per la coppia di anziani, dove il marito si china per cambiare le scarpe alla moglie, che con il bastone non riusciva a reggersi in piedi, mettendole le ciabatte per farla stare piú comoda.
Alla Comunitá dei padri dehoniani di Rende, per l'ospitalitá datami nei primi due anni di dottorato e la fiducia accordatami nel portare avanti diversi impegni, che mi é servita per imparare a gestirmi, dividendomi tra mondo accademico e resto della mia vita.
In ultimo, non per importanza, ma piú come "meglio da conservare per la fine", proprio come si fa con il dessert ad un pranzo o con le patatine fritte mischiate ad altri cibi in uno stesso piatto, a Dio perché in questi ultimi anni si é dimostrato essere particolarmente creativo e capace di stupirmi. Ti sono grato per la forza che mi hai aiutato a sviluppare e per la tranquillitá, mista a leggerezza ed abbandono, con cui mi hai fatto vivere questa esperienza ed insegnato a far fronte alle tensioni ed ai problemi. Grazie per avermi aiutato a preferire il diventare costruttore, piú che distruttore ed avermi fatto aprire gli occhi, per capire cosa non diventare. Tu che sei Padre, continua a farmi sentire figlio, perché il bello deve ancora venire.