Verification of real-time systems design

M. Emilia Cambronero∗,+ Valentín Valero and Gregorio Díaz

Universidad de Castilla-La Mancha, Escuela Superior de Ingeniería Informática de Albacete, Albacete 02071, Spain

SUMMARY

The main objective of this paper is to present an approach to accomplish verification in the early design phases of a system, which allows us to make the system verification easier, specifically for those systems with timing restrictions. For this purpose we use RT-UML sequence diagrams in the design phase and we translate these diagrams into timed automata for performing the verification by using model checking techniques. Specifically, we use the Object Management Group’s UML Profile for Schedulability, Performance, and Time and from the specifications written using this profile we obtain the corresponding timed automata. The ‘RT-UML Profile’ is used in conjunction with a very well-known tool to perform validation and verification of the timing needs, namely, the UPPAAL tool, which is used to simulate and analyze the behaviour of real-time dynamic systems described by timed automata. Copyright © 2009 John Wiley & Sons, Ltd.

Received 27 June 2007; Revised 24 November 2008; Accepted 3 February 2009

KEY WORDS: verification; timed automata; UPPAAL; model checking; RT-UML sequence diagrams; XSLT; real-time systems; design

1. INTRODUCTION

In the classic software life cycle the validation and verification occur after the implementation phase, but in software engineering, the detection of errors is notoriously expensive. Software engineering is forced to repeat every phase, as these errors usually come from the first phases in the software life cycle, thus, increasing the software cost dramatically. For this reason, one of the aims of our work is to accomplish a system design verification in order to detect errors during the early phases of development.

∗Correspondence to: M. Emilia Cambronero, Universidad de Castilla-La Mancha, Escuela Superior de Ingeniería Informática de Albacete, Albacete 02071, Spain.
†E-mail: emicp@dsi.uclm.es

Contract/grant sponsor: Spanish government; contract/grant number: TIN2006-15578-C02-02
Contract/grant sponsor: JCCLM regional project; contract/grant number: PAC06-0008-6995

Copyright © 2009 John Wiley & Sons, Ltd.
The properties of interest that we intend to verify are related to the classical problems of concurrency (deadlock freedom, mutual exclusion, etc.), as well as to some specific aspects of the systems under consideration, such as timing restrictions; for instance, we can think of the failure of a bank to receive a large electronic funds transfer on time, which may result in huge financial losses or we can consider a Seat Reservation System, in which the reservations are maintained for a limited period of time (several days). We see, therefore, that timing aspects are one of the main issues to be considered in the design and implementation of real-time systems.

The main goal of this paper is to present a model-driven development technique oriented to systems with timing restrictions. The work presented here is a part of a methodology that starts with the analysis of requirements, a phase that is covered by the KAOS technique (see Figure 1).

The KAOS goal model [1] allows analysts and specifiers to gather the requirements of software systems in a hierarchical order, i.e. from general and strategic goals to concrete requirements. Goals are objectives that the system under construction must achieve, and with the KAOS technique a structured goal model is constructed as an AND–OR graph. Thus, goals are organized in AND/OR refinement-abstraction hierarchies, where higher-level goals are in general strategic, coarse-grained and involve multiple roles, whereas lower-level goals are in general technical, and involve fewer roles. In such structures, AND-refinement links relate a goal to a set of subgoals possibly conjoined with domain properties; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. OR-refinement links may relate a goal to a set of alternative refinements and, in this case, it is enough to satisfy just one of the subgoals.

![Figure 1. Model-driven methodology.](image-url)
In the analysis phase, for instance, we gather the time requirements, such as deadlines, time-outs and any other constraints where time plays a crucial role.

After the analysis, the design phase starts. To complete this phase we use the information about the different entities that collaborate in the system, the information they exchange and the constraints that have been identified, and then, an RT-UML specification is produced as a first result of the design phase.

Then, this first RT-UML design can be tested by using formal techniques, and specifically timed automata. If we find some mistakes in the proposed design we can redesign it before entering into the implementation phase.

Thus, we present in this paper a part of the methodology shown in Figure 1, specifically the translation from RT-UML sequence diagrams into timed automata, with the purpose of system design verification (bottom arrow in Figure 1).

For this purpose we use UML 2.0 [2], which provides us with some great mechanisms that when combined with Model-Driven Development capabilities, enable the building of complete and fully tested reusable components. These mechanisms are frames, which can be used jointly with the UML profile for Schedulability, Performance and Time [3] (also called RT-UML) in order to describe the control structures and timing restrictions in our systems. This profile was introduced to cover the lack of UML in some key areas that are of particular concern to real-time system designers and developers. Specifically, we use RT-UML sequence diagrams, which are one of the central diagram types used in RT-UML. These allow us to describe the behaviour of a group of entities over time, thus, facilitating the understanding of their collaborative behaviour, by capturing their interactions over time. Likewise, by using timed automata, we have a well-known mathematical formalism that allows us not only to describe but also to analyze the behaviour of discrete dynamical systems with timing restrictions. In principle, a timed system could be directly modelled by timed automata, which would actually allow us the early detection of errors, but this approach only works for small systems, and it can be an exceptionally tiresome task to obtain directly the timed automata for more general and complex systems. Actually, for non-expert modelers the learning cost of using this formalism is high. In this paper, then, we present a method which avoids this learning cost, modelling the dynamic behaviour of a timed system by RT-UML, enabling this description to be then translated automatically to timed automata and verified by using the model-checker of the UPPAAL tool [4–6].

The paper is structured as follows. A discussion of related work is shown in Section 2. In Section 3 we describe timed automata, their semantics and the main features of the tool we use (UPPAAL) to verify some properties of interest. Section 4 shows a description of the UML Profile for Schedulability, Performance and Time, as well as an operational semantics for a meta-model of RT-UML sequence diagrams. Then, in Section 5 we show the translation from RT-UML Sequence Diagrams into Timed Automata. Three Case Studies are presented in Section 6, which illustrate the translation. Finally, the conclusions and future work are presented in Section 7.

2. RELATED WORK

In the literature, we can find some related works: Y. Zhao et al. [7] have defined a formal transformation of UML diagrams into Petri nets based on meta-modelling and graph transformation techniques.
T. Amnell et al. [8] have defined a timed extension of UML statecharts to enable modelling and verification of real-time systems. This extension includes constructions such as clocks, timed guards, invariants and real-time tasks, and the resulting formalism is called hierarchical timed automata. Another interesting work can be found in [9], where Chao Li et al. present a formal specification and validation method, called FORTS, for object-oriented real-time system development using UML, which describes the formal semantics of a timed extension of UML, and provides an automatic transformation of these UML models into timed automata starting from UML sequence diagrams.

E.E. Roubtsova and S.A. Roubtsov [10] have also defined a timed extension of the UML Class, Object and Statechart diagrams and they have defined a semantics for them based on eXtended Timed Graphs (XTG), a timed automata variant. Firley et al. [11] have also used a non-standard timed extension of UML sequence diagrams to describe the behaviour of real-time systems, these descriptions being transferred to timed automata, for the purpose of verification also using the UPPAAL tool. To cover the deficiencies of UML sequence diagrams for describing control structures they introduce some extras, such as loops. By contrast we use a standard timed version of UML, RT-UML, as well as a new feature of sequence diagrams in UML 2.0, namely, the frames, which allow us to describe the different control structures.

In addition, H. Toetenel et al. [12] have extended the UML Class, Object and Statechart diagrams with mechanisms for specifying timing constraints and properties. These extended UML specifications are then translated into the so-called XTG, a timed automata variant, which are used to verify the properties given on the UML specification, by using model checking techniques.

In contrast with these related works, we present an approach that uses an OMG’s recognized UML profile, RT-UML, in order to describe the behaviour of Web Services systems with timing restrictions, and we transform these descriptions into timed automata, in a format supported by the UPPAAL tool. Thus, we provide automatic transformation rules from the RT-UML Sequence diagrams to Timed Automata. We also exemplify this approach by means of three examples, for which we show the formal verification of their functional properties.

3. TIMED AUTOMATA

A timed safety automaton, or simply timed automaton (TA) [13,14] is essentially a finite automaton extended with real-valued variables. These variables model the logical clocks in the system, and are initialized to zero when the system is started. They then increase their value synchronously as time elapses, at the same rate. In the model there are also clock constraints, which are guards on the edges that are used to restrict the behaviour of the automaton, since a transition represented by an edge can only be executed when the clock values make the guard condition to be satisfied. Nevertheless, transitions are not forced to be executed when their guards are true, the automaton can stay at a location without executing any transitions, unless an invariant condition is associated with that location. In this case, the automaton may remain at that same location as long as the invariant condition is satisfied. Additionally, the execution of a transition can be used to reset some clocks of the automaton.

Definition 1 (Timed Automaton). We consider a finite set of real-valued variables $\mathcal{C}$ ranged over by $x, y, \ldots$ standing for clocks, and a finite alphabet $\Sigma$ ranged over by $a, b, \ldots$ standing for actions.
We will use letters \( r, r', \ldots \) to denote sets of clocks. A \textit{clock constraint} is a conjunctive formula of atomic constraints of the form: \( x \sim n \) or \( x - y \sim n \), for \( x, y \in C \), \( \sim \in \{ \leq, <, =, >, \geq \} \) and \( n \in \mathbb{N} \). The set of clock constraints will be denoted by \( \mathcal{C} \), ranged over by \( g, g', \ldots \).

A Timed Automaton is a tuple \( (N, n_0, E, I) \), where

- \( N \) is a finite set of locations (nodes).
- \( n_0 \in N \) is the initial location.
- \( E \subseteq N \times \mathcal{C} \times \Sigma \times 2^\mathcal{C} \times N \) is the set of edges.
- \( I : N \rightarrow \mathcal{C} \) is a function that assigns invariant conditions (which could be empty) to locations.

We will write \( n \xrightarrow{g,a,r} n' \) to denote \( (n, g, a, r, n') \in E \).

The semantics of a timed automaton is defined as a state transition system, where each state represents a location and a clock valuation. We use the following notation: letters \( u, v, \ldots \) will represent clock valuations, i.e. functions that assign non-negative real values to clocks, \( u, v : C \rightarrow \mathbb{R}_0^+ \). By \( u \in g \) we will represent that the clock valuation \( u \) makes \( g \) to be true, where we assume that when \( g \) is empty \( u \in g \) is true, and by \( u + d \) the clock valuation that takes \( u \) and increases the value of every clock by \( d \).

Definition 2 (Timed Automaton Semantics). Let \( \mathcal{A} = (N, n_0, E, I) \) be a timed automaton. The semantics of \( \mathcal{A} \) is defined as the timed labelled transition system \( (Q, q_0, \rightarrow) \), where:

- \( Q \subseteq N \times \mathbb{R}_0^+ \) (set of states).
- \( q_0 = (n_0, 0) \in Q \), is the initial state, where \( 0 \) is the clock valuation that assigns every clock to zero.
- \( \rightarrow \subseteq (Q \times \mathbb{R}_0^+ \times Q) \cup (Q \times \Sigma \times Q) \) (delay and action transitions).

Delay transitions are in the form \((q, d, q')\), for \( d \in \mathbb{R}_0^+ \), denoted by \( q \xrightarrow{d} q' \), and are defined by the following rule:

- \( (n, u) \xrightarrow{d} (n, u + d) \) if and only if \( (u + d') \in I(n) \), for all \( d' \leq d \), \( d' \in \mathbb{R}_0^+ \).

Action transitions are in the form \((q, a, q')\), for \( a \in \Sigma \), denoted by \( q \xrightarrow{a} q' \), and are defined by the following rule:

- \( (n, u) \xrightarrow{a} (n', u') \) if and only if there is an edge \( n \xrightarrow{g,a,r} n' \), such that \( u \in g \), \( u'(x) = u(x) \) for all \( x \notin r \), \( u'(x) = 0 \), for all \( x \in r \), and \( u' \in I(n') \).

A concurrent system is usually modelled by a set of timed automata running in parallel. A \textit{Network of Timed Automata} (NTA) is then defined as a set of timed automata that run simultaneously, using the same set of clocks, and synchronizing on the common actions. Then, in the following definition we distinguish two types of actions: internal and synchronization actions. Internal actions can be executed by the corresponding automata independently, and they will be ranged over the letters \( a, b \ldots \), whereas synchronization actions must be executed simultaneously by two automata. Synchronization actions are ranged over letters \( m, m' \ldots \) and come from the synchronization of two
Definition 3 (Semantics of an NTA). Let \( \mathcal{A}_i = (N_i, n_{0,i}, E_i, I_i) \), \( i = 1, \ldots, k \) be a set of timed automata. A state of the NTA \( \{\mathcal{A}_1, \ldots, \mathcal{A}_k\} \), is a pair \((\pi, u)\), where \( \pi = (n_1, \ldots, n_k) \), with \( n_i \in N_i \), and \( u \) is a clock valuation for the clocks in the system, \( u \in \mathbb{R}_0^+ \).

There are three rules defining the semantics of an NTA:

- \((\pi, u) \xrightarrow{d} (\pi, u+d)\) (delay rule) if and only if \( u+d' \in I_i(n_i) \), for all \( i = 1, \ldots, k \) and for all \( d' \leq d \), \( d' \in \mathbb{R}_0^+ \).
- \((\pi, u) \xrightarrow{a} (\pi', u')\) (internal action rule) if and only if there is an edge \( n_i \xrightarrow{g, a, r} n'_j \), for some \( i \in \{1, \ldots, k\} \), such that \( n'_j = n_j \), for all \( j \neq i \), \( u \in g \), \( u'(x) = u(x) \) for all \( x \in r \), \( u'(x) = 0 \), for all \( x \in r \), and \( u' \in \pi' \cap h=1, \ldots, k I_h(n'_j) \).
- \((\pi, u) \xrightarrow{m} (\pi', u')\) (synchronization rule) if and only if there exist \( i, j, i \neq j \), such that:
  - (i) \( n_h = n_j \), for all \( h \neq i, h \neq j \).
  - (ii) There exist two transitions \( n_i \xrightarrow{g_i, m_i, r_i} n'_i \) and \( n_j \xrightarrow{g_j, m_j, r_j} n'_j \), such that \( u \in g_i \cap g_j \), \( u'(x) = u(x) \), for all \( x \notin r_i \cup r_j \), and \( u'(x) = 0 \), for all \( x \in r_i \cup r_j \).
  - (iii) \( u' \in \pi' \cap h=1, \ldots, k I_h(n'_h) \).

In the following definition we introduce the notion of timed computation, as the timed traces that an NTA can generate according to the previous definition.

Definition 4 (Timed Computations of an NTA). Let \( \mathcal{N} = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\} \) be an NTA, with \( \mathcal{A}_i = (N_i, n_{0,i}, E_i, I_i) \), \( i = 1, \ldots, k \). The initial state is \( q_0 = ((n_{0,1}, \ldots, n_{0,k}), \overline{0}) \). We denote by letters \( s, s', \ldots \) the elements of \((\mathbb{R}_0^+ \cup ACT)^*\) (timed traces), where \( ACT \) is the set of internal and synchronizations actions. Then, we define the timed computations of \( \mathcal{N} \) as follows:

- \( q_0 \xrightarrow{d} q \) is a timed computation if and only if there exists a transition \( q_0 \xrightarrow{d} q \), for \( d \in \mathbb{R}_0^+ \).
- \( q_0 \xrightarrow{a} q \) is a timed computation if and only if there exists a transition \( q_0 \xrightarrow{a} q \), for \( a \in ACT \).
- For any timed computation \( q_0 \xrightarrow{s} q \), if there exists a transition \( q \xrightarrow{d} q' \), for \( d \in \mathbb{R}_0^+ \), then \( q_0 \xrightarrow{s,d} q' \) is a timed computation.
- For any timed computation \( q_0 \xrightarrow{s} q \), if there exists a transition \( q \xrightarrow{a} q' \), for \( a \in ACT \), then \( q_0 \xrightarrow{s,a} q' \) is a timed computation.

3.1. UPPAAL

UPPAAL [6] is a tool for modelling, validation and verification of real-time systems. The validation part is performed by graphical simulations and the verification part by Model-Checking means.

---

4 In the original definition the only internal action is \( \tau \), and synchronizations always yield internal actions.
A system in UPPAAL consists of a network of concurrent processes, each of them modelled as a timed automaton.

Timed automata in UPPAAL are extended with bounded integer variables, which can be read or written by means of expressions labelling the edges, and can be tested in guard conditions. A state of an NTA in UPPAAL is then defined by the locations of all the automata, and the clock and variable valuation. Notice that Definitions 1–4 can be easily adapted in order to include variables in the NTA semantics.

Another important feature of UPPAAL that we use are templates, which are parameterized generic declarations of timed automata, which can later be instantiated.

The simulation step in UPPAAL consists of running the system to check that it works properly in normal conditions. Since simulation does not guarantee system correctness completely, we must use the verifier tool to check some properties of the system that are of interest, and that should be ensured in all conditions. For instance, we can check reachability properties, i.e. if a certain state is reachable or not. This is called Model-Checking and it is basically an exhaustive search that covers all possible dynamic behaviours of the system.

4. UML PROFILE FOR SCHEDULABILITY, PERFORMANCE AND TIME

In this section, we first give a brief description of sequence diagrams and then, we summarize the key elements of this profile, which we use in our translation, in order to model the system time restrictions.

4.1. RT-UML sequence diagrams

A sequence diagram of UML 2.0 [2] depicts the sequence of actions that occur in a dynamical system. The invocation of methods of each object, and the order in which the invocation occurs is captured by a sequence diagram. This makes the sequence diagram a very useful tool to represent the dynamic behaviour of a system in a simple way. Furthermore, we use frames, which are a key element of sequence diagrams, extending their functionality by adding to them some new capabilities, such as the possibility of labelling, nesting and modelling of control structures.

Frames are defined as units of behaviour, and contain, among other things, the set of objects that are in relation, and the sequence of messages between these objects.

4.2. RT-UML

The use of this profile [3] is justified because UML is lacking in some key areas that are of particular concern to real-time dynamical system designers and developers. In particular, the lack of a quantifiable notion of time and resources was an impediment to its broader use in the real-time and embedded domain. It was discovered that UML had all the required mechanisms for addressing these issues, in particular through its extensibility faculties. The UML Profile for Schedulability, Performance, and Time is a standard way of using these capabilities to represent concepts and practices from the real-time domain.
In this work we only use the part of this profile that allows us to model the timing aspects of a system. We have chosen the ‘General Time Modelling’ viewpoint of the profile to model dynamical real-time systems, because this model describes a general framework for representing time and time-related mechanisms that are appropriate for modelling dynamical real-time software systems. It is quite general, but a given application only needs to use a subset of the concepts and semantics that it requires. We have chosen, therefore, a set of the concepts from this viewpoint to model our real-time dynamical systems. We are not dealing then with the resource sharing, schedulability and performance parts of the RT-UML profile.

The concepts of RT-UML that we have used in this paper to capture the timing constraints of systems are the following:

- **Delay**: This represents a null operation for a pre-specified time interval. This action has no side-effects except to delay the action execution that follows it. This is modelled by a model element that is stereotyped as \(<<<\text{RTdelay}>>\>.

- **TimedAction**: This concept is used to represent the execution of an action that has a known duration, and is modelled by applying the \(<<<\text{RTaction}>>\>\) stereotype. Timed events are similar to timed actions, but the action durations can be unknown. In order to define in general the semantics of timed actions and timed events we will consider a basic construction representing the execution of an action with a time limit for it to be executed. Hence, the semantics of a timed action is obtained by concatenating this construction (with time limit 0) with a delay (action duration), whereas a timed event is modelled as a concatenation of a delay (start time) and an action (with the maximum time to be completed).

- **Clock**: A Clock is a kind of timing mechanism that generates a clock interrupt periodically, where a Timing Mechanism is an abstract concept that captures the common features of resources that specialize in performing time measurement and timing-related functions. Clocks are modelled by the stereotype \(<<<\text{RTClock}>>\>\).

- **reset()**: An operation that stops the timing mechanism and sets it back into its initial state. The stereotype \(<<<\text{RTreset}>>\>\) models the reset() operation on a timing mechanism.

### 4.3. RT-UML Semantics

In this subsection we define an operational semantics for the specific subset of RT-UML that we are using. We will use this semantics in order to establish an equivalence with the NTA that we associate with an RT-UML model.

For our purposes an RT-UML sequence diagram can be seen as described by the following syntax:

\[
\mathcal{U} ::= (Frame, \text{Clocks})
\]

where \(\text{Clocks}\) is the set of clocks used in this specification, and \(\text{Frames}\) are defined by the syntax:

\[
\begin{align*}
\text{Frame} & ::= (r_1:A_1, \ldots, r_n:A_n)|\text{Frame};\text{Frame} \\
& \quad |\text{if}(g, \text{Frame}, \text{Frame})|\text{alt}(\text{Frame}, \text{Frame})| \\
& \quad |\text{loop}(g, \text{Frame})|\text{par}(\text{Frame}, \text{Frame})
\end{align*}
\]
where \((r_1 : A_1, \ldots, r_n : A_n)\) stands for a basic description, in which \(r_1, \ldots, r_n\) are the objects or actors involved, and \(A_1, \ldots, A_n\) the actions they make, whose syntax is defined below. By \(Frame\) we represent the sequential execution of both argument frames. The frame \(if (g, Frame_1, Frame_2)\) stands for an \(if\)-then-else construction; when the guard \(g\) is true \(Frame_1\) is executed, otherwise control is transferred to \(Frame_2\). By contrast, \(alt(Frame_1, Frame_2)\) represents a non-deterministic choice, either \(Frame_1\) or \(Frame_2\) can be executed. With \(loop(g, Frame)\) we represent the iterative execution of the indicated frame, as long as the guard \(g\) is true. Finally, \(par(Frame, Frame)\) represents the parallel execution of both argument frames.

For the guards \(g\) in principle we only allow clock constraints\(^\dagger\), which follow the syntax of \(\mathcal{G}\) (see Definition 1), and for the activities we consider the syntax:

\[
A ::= nil|a(d); A|send(i, m); A|recv(i, m); A|delay(d); A|reset(x); A
\]

where \(a \in \text{Alph}\) (alphabet of actions), \(i \in \{1, \ldots, n\}\) (rol identifier), \(m \in \mathcal{M}\) (message identifier), \(d \in \mathbb{R}_0^+\) and \(x \in \text{Clocks}\). The nil operator stands for no action (the object cannot execute any actions), \(a(d); A\) represents that the action \(a\) must be performed at most in \(d\) time units, and afterwards it performs the activity \(A\). For brevity, we will denote immediate actions \(a(0); A\) by \(a; A\). The operator \(send(i, m); A\) represents that the message identified by \(m\) is sent to the object \(r_i\), in a synchronous way, and afterwards, it behaves as \(A\), whereas \(recv(i, m); A\) is used to receive a message (identified by \(m\)) from \(r_i\), also in a synchronous way. Finally, \(delay(d); A\) is used to delay the execution for \(d\) time units, and \(reset(x); A\) is used to reset the clock \(x\) to zero.

The semantics for this language is defined by means of three types of transitions:

- \((F, v) \rightarrow (F', v)\) (null transition)
- \((F, v) \xrightarrow{d} (F', v + d)\) (Delay transition)
- \((F, v) \xrightarrow{b} (F', v')\) (Action transition)

where \(F\) is a frame and \(v\) a clock valuation:

\[
v : \text{Clocks} \rightarrow \mathbb{R}_0^+
\]

We use the notation \(v + d\) to represent the clock valuation that takes \(v\) and increases the value of every clock by \(d\), and \(v[x \rightarrow 0]\) to represent the clock valuation that takes \(v\) and only resets the clock \(x\). Additionally, we denote by \(v \in g\) to mean that \(g\) is true for the clock valuation \(v\).

Null transitions are used to solve the alternatives and loops, depending on the guard evaluation. They are also used in a sequence of frames to activate the second frame when the first has terminated. The rules defining null transitions are shown in Table I. In these rules we denote by \(\overline{nil}\) the frame \((r_1 : \overline{nil}, \ldots, r_n : \overline{nil})\).

In order to define action transitions we first need to define action transitions for the activities inside the frames, which follow the syntax:

\[
(A, v) \xrightarrow{act} (A', v')
\]

where \(act \in \text{Alph} \cup \{reset(x) | x \in \text{Clocks}\} \cup \{snd(i, m), recv(i, m) | i \in \{1, \ldots, n\}, m \in \mathcal{M}\}\).

\(^\dagger\)We will later discuss how to include discrete variables.
Table I. Frame null transition rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NS1)</td>
<td>((nil; F), v) \rightarrow (F, v))</td>
</tr>
<tr>
<td>(IF1)</td>
<td>((if(g,F_1,F_2),v) \rightarrow (F_1,v)) if (v \in g)</td>
</tr>
<tr>
<td>(IF2)</td>
<td>((if(g,F_1,F_2),v) \rightarrow (F_2,v)) if (v \notin g)</td>
</tr>
<tr>
<td>(ALT1)</td>
<td>((alt(F_1,F_2),v) \rightarrow (F_1,v)) if (v \in g)</td>
</tr>
<tr>
<td>(ALT2)</td>
<td>((alt(F_1,F_2),v) \rightarrow (F_2,v)) if (v \notin g)</td>
</tr>
<tr>
<td>(LOO1)</td>
<td>((loop(g,F),v) \rightarrow (F;loop(g,F)),v)) if (v \in g)</td>
</tr>
<tr>
<td>(LOO2)</td>
<td>((loop(g,F),v) \rightarrow (nil,v)) if (v \notin g)</td>
</tr>
<tr>
<td>(PAR)</td>
<td>((par(nil,nil),v) \rightarrow (nil,v))</td>
</tr>
</tbody>
</table>

Table II. Activity action transition rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Seq)</td>
<td>(((a(d); A), v) \xrightarrow{a} (A, v))</td>
</tr>
<tr>
<td>(Reset)</td>
<td>(((reset(x); A), v) \xrightarrow{reset(x)} (A, v[x \rightarrow 0]))</td>
</tr>
<tr>
<td>(Send)</td>
<td>(((send(i,m); A), v) \xrightarrow{snd(i,m)} (A, v))</td>
</tr>
<tr>
<td>(Receive)</td>
<td>(((recv(i,m); A), v) \xrightarrow{rcv(i,m)} (A, v))</td>
</tr>
</tbody>
</table>

The rules defining the action transitions for the activities are shown in Table II. These rules are very intuitive, expressing the sequential evolution of the activities performed by a single object. Notice that even if we have not considered variables in our meta-model, they can be easily introduced. A variable assignment is actually a sort of immediate action, which modifies the value of a variable in an object, and we can see that messages can also be used to send the variable values to other objects in the system. Thus, in the examples we will use the meta-model extended with variables, using a special action for assignments, namely, assign\((variable, value)\), which assigns the indicated local variable of this object to the specified value. We will also abuse the notation using variables as arguments of the communication primitives send and recv, as if they were message identifiers, i.e. with send\((j,v)\) and recv\((i,v)\) we will represent that object \(r_i\) communicates the value of its local variable \(v\) to object \(r_j\), which receives it in a local copy of variable \(v\).

A frame action transition has the following form: \((F, v) \xrightarrow{b} (F', v')\), where \(b \in \text{Alph} \cup \{\text{reset}(x) | x \in \text{Clocks}\} \cup \mathcal{M}\), and they are defined by the rules in Table III. The evolution by executing an action labelled with a message identifier \(m\) is obtained from a synchronization between two objects that communicate using this message identifier (rule \(\text{Syncr}\)). The single evolution of an object is captured by rule \(\text{Obj}\), and rules \(\text{Seq1}\) and \(\text{Par1-2}\) capture the sequential and parallel
Communications are, therefore, synchronous.

\[ \text{(Par1-2)} \]
\[
(F_1, v) \xrightarrow{b} (F'_1, v') \quad \text{and} \quad (F_2, v) \xrightarrow{b} (F'_2, v')
\]

Table III. Frame action transition rules.

<table>
<thead>
<tr>
<th>(Syner)</th>
<th>( (A_i, v) \xrightarrow{snd(i,m)} (A'_i, v) )</th>
<th>( (A_j, v) \xrightarrow{rcv(i,m)} (A'_j, v), i \neq j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (r_1: A_1, \ldots, r_n: A_n), v )</td>
<td>( \xrightarrow{act(i)} (A'_i, v) )</td>
<td>( \forall j \in [1, \ldots, n] )</td>
</tr>
</tbody>
</table>

\[ \text{(Seq1)} \]
\[
(F_1, v) \xrightarrow{b} (F'_1, v') \quad \text{and} \quad (F_2, v) \xrightarrow{b} (F'_2, v')
\]

Table IV. Activity delay transition rules.

<table>
<thead>
<tr>
<th>(D1)</th>
<th>( (\text{nil}, v) \xrightarrow{d} (\text{nil}, v + d) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D2)</td>
<td>( ((\text{delay}(d); A), v) \xrightarrow{d'} ((\text{delay}(d-d'); A), v + d') ) ( d' &lt; d )</td>
</tr>
<tr>
<td>(D3)</td>
<td>( ((\text{delay}(d); A), v) \xrightarrow{d} (A, v + d) )</td>
</tr>
<tr>
<td>(D4)</td>
<td>( ((\text{send}(i, m); A), v) \xrightarrow{d'} ((\text{send}(i, m); A), v + d') ) ( d' \leq d )</td>
</tr>
<tr>
<td>(D5)</td>
<td>( ((\text{recv}(i, m); A), v) \xrightarrow{d} ((\text{recv}(i, m); A), v + d) )</td>
</tr>
</tbody>
</table>

Time elapsing is defined by means of delay transitions. We first define the rules for activity delay transitions (Table IV), and afterwards, the rules for frame delay transitions (Table V). Activity delay transitions have the form \( (A, v) \xrightarrow{d} (A', v + d) \), \( d \in \mathbb{R}_0^+ \), to specify how the activity \( A \) changes when \( d \) time units have elapsed. Time elapsing is allowed for activities except in the case of immediate actions and \textit{reset}; in both cases these actions must be made immediately.

Time elapsing for single frames (rule \textit{DF1}) is possible when all the activities inside it allow the passage of time, and no synchronization is possible\(^1\). For that purpose, the function \textit{Syn} is defined.

\[ \text{Syn}(r_1: A_1, \ldots, r_n: A_n), v \text{ is true if and only if for every pair } i, j, i \neq j, \text{ there are no two activity action transitions } (A_j, v) \xrightarrow{snd(j,m)} (A'_j, v) \text{ and } (A_j, v) \xrightarrow{rcv(i,m)} (A'_j, v) \text{ that could generate a synchronization between the objects } r_i \text{ and } r_j. \]

\(^1\)Communications are, therefore, synchronous.
Table V. Frame delay transition rules.

(DF1) \[
(A_i, v) \xrightarrow{d} (A'_i, v + d), \; i = 1, \ldots, n, \; \neg \text{Syn}(r_1; A_1, \ldots, r_n; A_n, v)
\]

(DF2) \[
(F_1, v) \xrightarrow{d} (F'_1, v + d), \; F_1 \neq \text{nil}
\]

(DF3) \[
(F_i, v) \xrightarrow{d} (F'_i, v + d), \; i = 1, 2
\]

Definition 5 (RT-UML Operational Semantics). Let \( \mathcal{U} = (\text{Frame, Clocks}) \) be an RT-UML sequence diagram. We define its associated labelled transition system, \( \text{lts}(\mathcal{U}) \), as that obtained by the application of the rules of Tables I–V, starting from the state \( q_0 = (\text{Frame}, \bar{0}) \).

Timed computations are then defined as the concatenation of the actions and delays labelling the transitions that can be executed from the initial state.

Definition 6 (Timed computations). Let \( \mathcal{U} = (\text{Frame, Clocks}) \) be an RT-UML sequence diagram and \( \text{lts}(\mathcal{U}) \) its associated labelled transition system. We will denote by \( \xrightarrow{*} \) zero or more sequential applications of frame null transitions.

Then, we define the timed computations of \( \mathcal{U} \) as follows:

- \( q_0 \xrightarrow{d} q \) is a timed computation if and only if there exists a sequence of transitions
  \[
  q_0 \xrightarrow{*} q_1 \xrightarrow{d} q_2 \xrightarrow{*} q
  \]
  for \( d \in \mathbb{R}_0^+ \)

  where \( q_0 \) is the initial state of \( \text{lts}(\mathcal{U}) \).

- \( q_0 \xrightarrow{b} q \) is a timed computation if and only if there exists a sequence of transitions
  \[
  q_0 \xrightarrow{*} q_1 \xrightarrow{b} q_2 \xrightarrow{*} q
  \]

  for \( b \in \text{Alph} \cup \{\text{reset}(x) \mid x \in \text{Clocks}\} \cup \mathcal{M} \), where \( q_0 \) is the initial state of \( \text{lts}(\mathcal{U}) \).

- For any timed computation \( q_0 \xrightarrow{s} q \), if there exists a sequence of transitions
  \[
  q \xrightarrow{*} q_1 \xrightarrow{d} q_2 \xrightarrow{*} q'
  \]

  then \( q_0 \xrightarrow{s,d} q' \) is a timed computation.

- For any timed computation \( q_0 \xrightarrow{s} q \), if there exists a sequence of transitions
  \[
  q \xrightarrow{*} q_1 \xrightarrow{b} q_2 \xrightarrow{*} q'
  \]

  for \( b \in \text{Alph} \cup \{\text{reset}(x) \mid x \in \text{Clocks}\} \cup \mathcal{M} \), then \( q_0 \xrightarrow{s,b} q' \) is a timed computation.
TRANSLATING RT-UML DOCUMENTS INTO TIMED AUTOMATA

As the syntax of RT-UML is so vast, as described in Section 4, we need to restrict ourselves to a subset of it in order to define a translation to NTAs. Thus, only the elements that have been considered in the meta-language presented in Section 4.3 are considered in the translation.

We first define the timed automaton associated with each activity, and afterwards, the NTA associated with every sort of frame. The translation for the different activities is shown in Figure 2. We can observe that all the automata have both one initial and one final location, this property being preserved by all the constructions. Additionally, every translation defines a set of clocks that must be reset when the initial location is reached. This means that all the incoming edges to that location.

<table>
<thead>
<tr>
<th>nil</th>
<th>init\textsubscript{nil} = end\textsubscript{nil}</th>
<th>C\textsubscript{nil} = \emptyset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(d);A</td>
<td>init\textsubscript{A} = \emptyset</td>
<td>C\textsubscript{a} = {d}</td>
</tr>
<tr>
<td>delay(d);A</td>
<td>init\textsubscript{delay(d)} = \emptyset</td>
<td>C\textsubscript{delay(d)} = {d}</td>
</tr>
<tr>
<td>reset(x);A</td>
<td>init\textsubscript{reset(x)} = \emptyset</td>
<td>C\textsubscript{reset(x)} = {x}</td>
</tr>
<tr>
<td>send(m);A</td>
<td>init\textsubscript{send(m)} = \emptyset</td>
<td>C\textsubscript{send(m)} = \emptyset</td>
</tr>
<tr>
<td>recv(m);A</td>
<td>init\textsubscript{recv(m)} = \emptyset</td>
<td>C\textsubscript{recv(m)} = \emptyset</td>
</tr>
</tbody>
</table>

Figure 2. Activity translation.
should include this set within their clocks-to-be-reset part. In this figure we denote by \( ta(A) \) the TA corresponding to the activity \( A \), and by \( \text{init}_A \) and \( \text{end}_A \) its initial and final locations. We also denote by \( C_A \) the set of clocks to be reset when using that timed automaton in a compositional way. Of course, when \( A \) is an activity that is executed from the initial state, this set is useless, because all the clocks are initialized to zero.

The activity translation works as follows:

- **nil**: This corresponds to a single location, and the set of clocks to be reset is empty.
- **\( a(d); A \)**: We first obtain the timed automaton corresponding to \( A \) compositionally, and afterwards, we add a new location (the initial location of \( ta(a; A) \)) and an edge connecting this location with the initial location of \( ta(A) \). We also add a new clock \( x \) that must be reset when using this construction compositionally, with an invariant \( x \leq d \) over the initial location to allow time elapsing up to \( d \) time units, the edge guard being \( x \leq d \) and the action labelling it \( a \). Furthermore, the set of clocks to be reset by this edge is the one associated with \( ta(A) \).
- **delay\( (d); A \)**: This is similar to the previous one, but in this case the edge guard is \( (x = d) \) and the action labelling the edge is an internal action \( \tau \).
- **reset\( (x); A \)**: In this case we take the timed automaton \( ta(A) \) and we add \( x \) to its set of clocks to be reset.
- **\( \text{send}(i, m); A \) and \( \text{recv}(i, m); A \)**: Both cases are similar. We add a new initial location and an edge connecting it with the initial location of \( ta(A) \). The edge guard is true, since it can be executed at any moment, and the actions are \( m! \) for \( \text{send} \) and \( m? \) for \( \text{recv} \). In this edge we also include the set of clocks to be reset of \( ta(A) \).

The translation for the different frames is shown in Figure 4. In this translation, for simplicity, we use a new special kind of channel, which we call *multisynchronization channel*, in which there is no communication, but a synchronization among all the objects. We then use the action \( c! \) to denote such a multisynchronization over channel \( c \), which can only be executed when all the processes (objects) are able to perform it, and in that case it is immediate. Notice that this kind of synchronization is different from a broadcast communication like the one supported by UPPAAL, in which one sender (\( c! \)) can synchronize with an arbitrary number of receivers (\( c? \)); additionally, broadcast sending is never blocking. In contrast to broadcast communication, in a multisynchronization we require all the processes to be able to execute their corresponding action \( c! \) for this action to be simultaneously executed by all of them, which means that multisynchronization is blocking.

A multisynchronization can be encoded by using binary synchronizations, but this is in general a tangled task, as a great number of paths must be considered. However, in the specific cases in which we use the multisynchronization we can encode it in an easy way by using \( 2n \) binary synchronization channels \( c_1, \ldots, c_n, d_1, \ldots, d_n \), a new clock \( x \) and one additional control timed automaton. Figure 3 shows the encoding of a guarded multisynchronization, which is valid in the case that the locations from which the multisynchronization edges leave do not have further edges leaving them, either internal or binary synchronization edges. Thus, as we can see in Figure 3 we may have one or several multisynchronization edges leaving the location, each one labelled with a different multisynchronization channel, and the same guard condition must be associated with each channel for all the objects. These guard conditions are not necessarily exclusive, when some
of them are simultaneously true, the choice is non-deterministic. The proof that this encoding is valid is given in Lemma A.1 in the Appendix.

Frames are then translated in a compositional way (Figure 4). A sequential frame consists of $n$ timed automata, one for each object in the RT-UML description. Together with these timed automata we define the clocks that must be reset when the frame is used in a compositional way, as indicated in this figure\(^1\). Parallel frames are obtained as separate templates, i.e. two frames executing in parallel consist of two sets of $n$ timed automata, but the second set is in some way subordinate to the first (in the sense of activation, as we describe below).

This translation works as follows:

- **Basic frame**: $\langle r_1 : A_1, \ldots , r_n : A_n \rangle$ is translated as the set of timed automata (and set of clocks to be reset) associated with the activities $A_1, \ldots , A_n$.

- **Sequence**: Each timed automaton in $F_1 \cup F_2$ is obtained by concatenating the corresponding timed automata of $F_1$ and $F_2$. The edge that connects both automata is labelled with a new multisynchronization channel $c$ that enforces the termination of all the automata in $F_1$ before starting the automata in $F_2$. The set of clocks to be reset of the corresponding automata in $F_2$ is then taken as set of clocks for this new edge, and the set of clocks to be reset of the automata in $F_1$ is now the set of clocks of the frame sequence.

- **If-then-else**: We take the timed automata corresponding to every object in $F_1$ and $F_2$, and we collapse the final locations of both automata. Furthermore, we add a new initial location and

\(^1\)For a frame executed at the initial state this set of clocks is not used, since all the clocks are initialized to zero.
two edges connecting this location with the initial locations of both automata, one labelled with the guard $g$ to activate the first, and the other with $\neg g$ to activate the second. Furthermore, two new, different multisynchronization channels are used to enforce the simultaneous execution of the corresponding edges, depending on the value of $g$. Finally, each edge has the corresponding set of clocks to be reset of the timed automaton that activates.
• **Alternative:** This is very similar to the previous case. The only difference is that in this case both guards are true, which allows the execution of any alternative.

• **Loop:** In this case two new locations are added to each timed automaton \( ta(A_i) \) and three edges connected and labelled as indicated in the figure. Observe that we also use three new and different multisynchronization channels, \( c, s \) and \( d \). Channel \( c \) is used to activate simultaneously all the timed automata of \( F \); \( d \) is used to synchronize the termination of all of these automata in order to re-evaluate \( g \), whereas \( s \) is used to terminate the loop, when \( g \) is evaluated to be false.

• **Parallel:** As mentioned earlier, the timed automata of the second frame are separate templates that will be activated by their partners in \( F_1 \), by means of binary channels \( s_1, \ldots, s_n \). Thus, once the location \( init_i \) is reached, the binary synchronization over \( s_i \) is immediately executed and both automata run in parallel, up to termination, where both synchronize again over the same binary channel \( s_i \). This causes the timed automaton of \( F_2 \) to return to its initial location (it can be used in a loop).

We now state our main theoretical result, which states that this translation is correct, in the sense that by means of it we obtain the same timed computations both for the RT-UML semantics and for the timed automata semantics.

**Theorem 1.** Let \( \mathcal{U} = (Frame, Clocks) \) be an RT-UML sequence diagram with \( n \) objects and \( \{A_i\}_{i=1}^m \) the corresponding NTA according to the defined translation \((m \geq n)\).

Then, \( s \) is a timed trace of \( \mathcal{U} \) if and only if there exists a timed trace \( t \) of the NTA such that \( \varphi_1(s) = \varphi_2(t) \), where

\[
\varphi_1(s) = \begin{cases} 
  s & \text{if } s \in M \cup R_0^+ \cup \text{Alph} \\
  s_1 \cdot \varphi_1(s_2) & \text{if } s = s_1 \cdot s_2, s_1 \in M \cup R_0^+ \cup \text{Alph} \\
  \varphi_1(s_2) & \text{if } s = \text{reset}(x) \cdot s_2, \text{ with } x \in \text{Clocks}
\end{cases}
\]

\[
\varphi_2(t) = \begin{cases} 
  t & \text{if } t \in M \cup R_0^+ \cup \text{Alph}, t \neq \tau \\
  t_1 \cdot \varphi_2(t_2) & \text{if } t = t_1 \cdot t_2, t_1 \in M \cup R_0^+ \cup \text{Alph} \\
  \varphi_2(t_2) & \text{if } t = \tau \cdot t_2
\end{cases}
\]

**Proof.** In Appendix. \( \square \)

6. **CASE STUDIES**

Three examples are used to illustrate the methodology: an Internet purchase process, an Aero-Electric Management System and a Travel Reservation System. These case studies show how the translation from RT-UML to TA can be used for automatically generating the timed automata for a real-time system from its design (RT-UML diagrams), and verifying that the system fulfils the
properties identified during the analysis phase. If some failures are detected here, we return to the design phase in order to correct the system design.

6.1. Case study: an internet purchase process

We consider a typical purchase process that uses Internet as a business context for a transaction. There are three actors in this example: a customer, a seller and a carrier. The Internet purchase works as follows: ‘A customer wants to buy a product by using the Internet. There are several sellers that offer different products in Internet Servers based on Web-pages. The customer contacts a seller in order to buy the desired product. The seller checks the stock and contacts with a carrier. Finally, the carrier delivers the product to the customer’.

Figure 5 depicts the RT-UML sequence diagram of this purchase process. The behaviour of each participant is defined as follows:

- **Customer**: He contacts the seller to buy a product, sending the information about the product and the payment method to the seller. After the payment, he waits to receive the product from a carrier in the agreed time, 24 h.
- **Seller**: He receives the customer order and the payment method. The seller checks if there is enough stock to deliver the order and sends an acceptance notification to the customer. If there is stock he contacts with a carrier to deliver the product.
- **Carrier**: He picks up the order and the customer information in order to deliver the product to the customer. The product must be delivered in the stipulated time, one day.

6.1.1. Analysis phase

We must identify in this phase the crucial requirements for the Internet purchase process. In this case, we have identified two different kinds of requirements. The first refers to the obligation that
both the seller and the carrier have agreed to deliver the product on time, while the other refers to the quality of service. The time restriction establishes that the seller and carrier have 24 h to deliver the product. This also implies that the seller must prepare the order for the carrier within the interval. The service quality is determined by two different requirements that are closely linked. The service must be rapid and also efficient. Owing to the close relationship between these two requirements, if one of them is fulfilled, the other is fulfilled too.

Figure 6 depicts the KAOS goal-model we have developed for this example. The root goal ‘CorrectInternetPurchase’ is decomposed into two subgoals by an And-refinement, which means that both must be satisfied. The first one, ‘NoDelays’, of type ‘maintain’, is refined by another And-refinement with two leaf goals that inherit the maintain character. The first leaf goal ‘PickupOnTime’ is of type ‘Unbound Respond’. This goal represents that the carrier must pick up the order on time. The second leaf goal ‘DeliverOnTime’ is of type ‘Inevitably’ and specifies that the carrier must deliver the order on time. The second one, ‘SatisfiedCustomer’, of type ‘Achieve’, is formed by two leaf goals. These leaf goals refine the parent goal by an Or-refinement, which means that if one of them is satisfied, the parent goal is satisfied, too. The leaf goal ‘RapidService’ determines that the customer will receive the order on time. The leaf goal ‘EfficientService’ has the behaviour of an ‘Unbounded Response’ requirement. This goal indicates that when the seller accepts the order, then in the future, the customer will receive the order.

6.1.2. Design and verification phases

Figure 7 shows the corresponding description in our meta-model for the RT-UML sequence diagram of Figure 5. Hence, according to the translation defined in Section 5 we obtain the NTA depicted in Figure 8, which consists of three timed automata, each one corresponding to one different participant in this system.
Figure 8. NTA for the Internet purchase process.
We can then use the verifier of UPPAAL to check the properties that have been identified. Notice that these properties must be adapted to consider the particular names of variables and clocks that are used in UPPAAL. For instance, the first property ‘PickupOnTime’ (7) is formalized as follows:

\[
\text{Customer.WaitOrder} \rightarrow (\text{Deliverer.PickUp} \land x < 24)
\] (1)

The second property, ‘DeliverOnTime’ (8) is formalized as:

\[
A <> (\text{Deliver.deliver} \text{ and } x < 24)
\] (2)

The third property ‘SatisfiedCustomer’ (6) is formalized as follows:

\[
E <> (\text{Customer.ReceiveOrder} \text{ and } x < 24)
\] (3)

The fourth property ‘EfficientService’ (5) is formalized as follows:

\[
\text{Seller.AcceptOrder} \rightarrow \text{Customer.ReceiveOrder}
\] (4)

We have checked all of these properties by using the UPPAAL verifier, as well as deadlock freeness, and all of them are satisfied. In the event of some of the identified properties proving to be false in the verification phase, we should return to the design phase (RT-UML diagram), in order to fix the detected error and restart the verification process with the new design.

6.2. Case study: aero-electric management system

This system consists of three components: windmill management system (WMS), productivity management system (PMS) and demand management system (DMS).

The demand management system controls the power requirements for the covered area drawing up a report, which is sent to the productivity management system. This system analyzes these reports and decides how many wind generators should be working.

We focus our attention on one event in the system, namely, a peak in the power demand. The DMS periodically checks the need to increase the production of electric power. In the event of an increase in power demand, this should be seen to within 3 min. The DMS calculates how much power is necessary, and one minute later this system sends a message to the PMS to increase the production.

The PMS then analyzes this request and computes the number of generators that are needed to attend to the new demand, sending a message to the WMS to request how many generators are available to be turned on. The WMS takes at most 2 min to ascertain the number of idle generators. Then, it sends a reply to the PMS with the number of available generators, and when the answer is received, this system decides how many of them need to be turned on. If there are enough generators it sends a message to the WMS for them to be turned on. Otherwise it sends a message to the DMS to indicate that it is not possible to satisfy the new demand (Figure 9).

6.2.1. Analysis phase

Figure 10 depicts the KAOS goal-model that we have developed for this example. We have identified two different kinds of requirements. The first refers to the obligation that both the productivity and
windmill management systems have agreed to attend to the demand on time, and the second refers to the quality of service.

The time restriction establishes that the demand has to be seen to within 3 min. This implies that the PMS must prepare the order for the WMS to turn on the generators within that interval.

Service quality is determined by two different requirements that are closely linked: it must be rapid and efficient. The root goal ‘CorrectSystem’ is decomposed into two subgoals by an And-refinement, which means that both must be fulfilled in order to achieve the root goal. The first, ‘NoDelays’ of type ‘maintain’, is refined by another And-refinement with two leaf goals that inherit the maintain character. The second goal, ‘AttendToDemand’, of ‘Achieve’ type has two leaf goals. These leaf goals refine the parent goal by an Or-refinement, which means that if one of them is satisfied, the parent goal is satisfied too.

6.2.2. Design and verification phases

Figure 11 contains a first version of the RT-UML sequence diagram for the aero-electric system. This diagram corresponds to the checking made periodically for an increase of demand. From this diagram we obtain the corresponding description in our meta-model (Figure 9), and from this description we can use the translation defined in Section 5 in order to obtain the corresponding timed automata (Figure 12).
Figure 11. RT-UML sequence diagram for the increase of power production.

The obtained NTA can then be read and simulated by the UPPAAL tool, and we can also check the properties of interest. Specifically, the properties to check are those corresponding to the goals we have established in the analysis phase:

(i) Ordering on time. This is a property of type ‘Inevitably’, and specifies that the PMS must send the order of turning on the generators on time:

\[ A \leftrightarrow (\text{WindTMS.AvailableG} \rightarrow (\text{ProductMS.OrderTurnOn} \land x < 3)) \]  

(ii) Calculating number of generators on time. This second leaf goal ‘CalculateNTurbinesOn-Time’ is of type ‘Inevitably’, and represents that the WMS must calculate the available number of generators within 2 min. It is formalized as follows:

\[ A \leftrightarrow \text{ProductMS.NuGenerators} \rightarrow (\text{WindTMS.AssignG} \land x < 3) \]  

(iii) The second goal, ‘AttendToDemand’, of type ‘Achieve’, consists of two leaf goals. The leaf goal ‘Rapid Response to Demand’ determines that the WMS will turn on the generators on time. This is represented by a rectangular box, which inherits the ‘Achieve’ behaviour from his father, and it is specified as follows:

\[ E \leftrightarrow (\text{WindTMS.GeneratorsOn} \land x < 3) \]  

(iv) The other leaf goal ‘Satisfied Demand’ has the behaviour of an ‘Unbounded Response’ requirement. This goal represents that when the PMS sends the ‘turn on’ order, then in the future, the WMS will turn the generators on. This goal is formalized as follows:

\[ \text{ProductMS.OrderTurnOn} \rightarrow \rightarrow \text{WindTMS.GeneratorsOn} \]
Using the model-checker of UPPAAL we have verified these properties, and we have concluded that property 7 fails. The problem is that the system cannot turn on the generators on time (in 3 min). This corresponds to a design error in the RT-UML sequence diagram in Figure 11, in which an RTAction has been used with a fixed duration for calculating the number of available generators. This action must be replaced by an RTEvent, in which we establish a maximum duration of 2 min for the detection of idle generators (see Figure 13 for the new RT-UML sequence diagram).

The corresponding meta-model for this new RT-UML sequence diagram can be immediately obtained, as well as the corresponding NTA. Then, checking the properties again, we have obtained that all of them are now fulfilled.

6.3. Case study: travel reservation system

This system consists of three participants: a Traveler, a Travel Agent and an Airline Reservation System, whose behaviour is as follows. A Traveler is planning on taking a trip. Once he has
Figure 13. Corrected RT-UML sequence diagram for the aero-electric system.

decided on the specific trip he wants to make he submits it to a Travel Agent by means of his local Web Service software (Order Trip). The Travel Agent selects the best itinerary according to the criteria established by the Traveler. For each leg of this itinerary, the Travel Agent asks the Airline Reservation System to verify the availability of seats (Verify Seats Availability). The Traveler has then the choice of accepting or rejecting the proposed itinerary, and he can also decide not to take the trip at all.

- If he rejects the proposed itinerary, he may submit the modifications (Change Itinerary), and wait for a new proposal from the Travel Agent.
- If he decides not to take the trip, he informs the Travel Agent (Cancel Itinerary) and the process ends.
- If he decides to accept the proposed itinerary (Reserve Tickets), he will provide the Travel Agent with his Credit Card information to book properly the itinerary.

Once the Traveler has accepted the proposed itinerary, the Travel Agent connects with the Airline Reservation System in order to reserve the seats (Reserve Seats). However, it may occur that at that moment no seat is available for a particular leg of the trip, because some time has elapsed from the moment in which the availability check was made. In that case the Travel Agent is informed by the Airline Reservation System of the situation (No seats), and the Travel Agent informs the Traveler that the itinerary is not possible (Notify of Cancellation).

Once the reservation is made the Travel Agent informs the Traveler (Seats Reserved). However, this reservation is only valid for a period of one day, which means that if a final confirmation has not been received within that period, the seats are unreserved and the Travel Agent is informed. Thus, the Traveler can now either finalize the reservation or cancel it. If he confirms the reservation
Correct Travel Reservation System

SystemBehavesProperly

NoDeadlocks

CustomerSatisfaction

Correct Booking System

Booking Period 24 hrs

Seats Booking When Payment

Correct Seat Reservation

Traveler ReceivItinerary

TASendItinerary On Plan Order

Itinerary When Request

Receive Tickets & Statement

When Request Change Itinerary

When Request Receive Tickets

Figure 14. KAOS goal model for the travel reservation system.

(Book Tickets), the Travel Agent asks the Airline Reservation System finally to book the seats (Book Seats).

6.3.1. Analysis phase

In this phase we identify the main requirements that the system must fulfil. Figure 14 shows the KAOS goal model we have considered for this example. The root goal is called Correct Travel Reservation System, which consists of three subgoals joined with an AND-refinement. One of these subgoals captures deadlock freeness; while the other subgoals capture the correct behaviour of the system and the customer satisfaction.

(a) The ‘System Behaves Properly’ subgoal has a ‘maintenance’ character and consists of two subgoals which check the reservation and booking parts, and both subgoals must be satisfied. For the reservation we have another AND-refinement, with two subgoals, one for the cancellation on demand, and the other for the seat reservation. The Cancel On Demand subgoal has the character of an ‘inevitable’ goal. This goal is specified as follows:

\[ \forall \Box \text{Traveler.} \text{CancelReservation} \rightarrow (\text{TravelAgent.} \text{CancelReservtRcv} \wedge \text{Airline.} \text{PerformCancel} \wedge \text{Airline.} \text{Clock} < 24) \]  

9

The Correct Seat Reservation (CorrectSeatReservt) subgoal has the character of ‘possible always’. This goal captures that if the traveler performs a reservation, the travel agent will send the order to
the Airline and the Airline will perform the seat reservation. It is specified as follows:

\[ \exists \Diamond \text{Traveler}. \text{PerformReservation} \rightarrow (\text{TravelAgent}. \text{SendReservoirOrder} \land \text{Airline}. \text{ReserveSeats} \land \text{Airline}. \text{Clockx} == 0) \] (10)

For the seat reservation part we have two subgoals, joined by an AND-refinement. The first, *Booking Period is 24 h*, inherits the ‘maintenance’ character from *System Behaves Properly*. This requirement establishes that reservations are valid for a period of one day.

\[ \forall \Box (\text{TravelAgent}. \text{Booking} \land \text{Airline}. \text{ReceiveBooking} \land \text{Airline}. \text{ClockX} <= 24) \] (11)

The second subgoal of the seat reservation part is called ‘Seats Booking When Payment’, which has again the ‘maintenance’ character, and determines that the booking is carried out only if the traveler has made the payment.

\[ \forall \Box \text{Traveler}. \text{PerformPayment} \rightarrow \text{Airline}. \text{BookSeats} \land \text{Airline}. \text{ClockX} <= 24 \] (12)

(b) The *Customer Satisfaction* subgoal is refined by an AND-refinement in three subgoals, which inherit the ‘achieve’ character from this subgoal. Its first subgoal, *Itinerary When Request*, is refined by an OR-refinement in two subgoals. The first, *Traveler Receives Itinerary*, has a ‘maintain’ character, and establishes that the traveler receives the itinerary upon asking for a plan order. This is specified as follows:

\[ \forall \Box \text{Traveler}. \text{PlanOrder} \rightarrow \text{TravelAgent}. \text{SendItinerary} \] (13)

The second, *Travel Agent Sends Itinerary On Plan Order*, has the ‘unbounded response’ character, and establishes that the travel agent must send the itinerary when receiving the corresponding travel request:

\[ \text{Traveler}. \text{PlanOrder} \rightarrow \text{TravelAgent}. \text{SendItinerary} \] (14)

The traveler may change the proposed itinerary, which is captured by the subgoal *Change Itinerary When Request*, which inherits the ‘achieve’ character from the *Customer Satisfaction* goal. This is specified as follows:

\[ \exists \Diamond \text{Traveler}. \text{ChangeItinerary} \rightarrow \text{TravelAgent}. \text{PerformChange} \] (15)

Finally, the obligation for the Airline and the Travel Agent to send the tickets and the statement to the traveler once he has performed the booking and payment is captured by the subgoal *Receive Tickets & Statement*, which has the ‘unbounded response’ character. This is written as follows:

\[ \text{Traveler}. \text{PaymentPerform} \rightarrow (\text{Traveler}. \text{Finish} \land \text{Airline}. \text{SnddTckt} \land \text{TravelAgent}. \text{SenddStatement}) \] (16)
6.3.2. Design and verification phases

The RT-UML sequence diagram modelling the communication process between the different entities and objects involved in this system is shown in Figure 15. This diagram has been split into two
diagrams for better understanding. The left-hand side part shows the general structure of the system by using nested frames, whereas in the right-hand side part the message flow and the internal frames are shown.

This sequence diagram starts when the traveler orders a trip to the travel agent, who contacts with the airline to check the ‘Seat Availability’, and sends several possible itineraries as response to the order. This part corresponds to the first three messages exchanged between the three parties.

Once the traveler is aware of the possible itineraries, he may change the default itinerary. This possibility is depicted in the diagram by the ‘Opt: ChangeItinerary’ frame. On the other hand, the traveler may cancel the itinerary (‘Alt1:CancelItinerary’ frame) or may reserve the tickets (‘Alt2: ReserveTickets’). The reservation process is represented by Alt2, and has a temporal constraint associated: ‘The reservation is only available for one day’. This constraint is depicted in the figure by means of two optional frames labelled with ‘within 24 h’ and ‘reached 24 h after reservation’.

In the first case, the traveler has two options ‘cancelreservation’ (Alt2.1) or ‘booktickets’ (Alt2.2). The booking process has a parallel frame structure, in order to send the tickets and the invoice simultaneously. When the traveler has not performed the booking on time, a time-out expires (last Opt frame) and the reservation is annulled.

The corresponding description using the meta-model is shown in Figure 16, where \( r_1 = Traveler \), \( r_2 = TravelAgent \) and \( r_3 = AirlineRS \). In this description frame \( F_1 \) corresponds to a change of itinerary, \( F_2 \) to the cancellation of an itinerary, and \( F_3 \) to the reservation and confirmation. Frame \( F_{311} \) corresponds to the cancellation on user demand, whereas \( F_{312} \) represents the confirmation and payment. \( F_{3121} \) and \( F_{3122} \) represent the sending of the invoice and the tickets, respectively. Finally, \( F_{32} \) stands for the annulation, when the time-out of 24 h has expired.

The NTA corresponding to this system is shown in Figure 17. There are six timed automata, due to the parallel constructions. The separate parallel timed automata obtained for each participant are shown in dotted boxes.

Using then the model-checker of UPPAAL we have concluded that the properties 9, 10, 11, 12, 15 and 16 are satisfied. Concerning the properties 13 and 14, due to the OR-condition in the KAOS goal graph, only one of them must be satisfied, and we have obtained that actually both are satisfied.

7. CONCLUSIONS AND FUTURE WORK

This paper shows how a real-time system description written by using RT-UML sequence diagrams can be automatically translated into a network of timed automata. For that purpose we have defined a meta-model for RT-UML sequence diagrams, and an operational semantics for this meta-model. We have also defined the formal semantics of timed automata and a translation from the meta-model to NTA has been defined, which has been proved to be correct, in the sense that both semantics yield the same timed computations.

Furthermore, as explained throughout the paper, our main goal with this work has been to make the real-time dynamical system verification easier from the early design phases. Once we have obtained the NTA corresponding to some specific RT-UML diagram, we use the UPPAAL tool in order to verify some properties of interest of our dynamical systems, which have been previously identified by using the KAOS technique.
Thus, we consider this work to be a very useful approach to verifying the correctness of the design phase in the software life cycle, since we can detect and correct design errors in an early development phase, which means an important saving of money and time.

As current work, we are implementing a tool that uses this translation and allows us to obtain automatically the NTA system from a given RT-UML sequence diagram. This tool uses XSLT in order to perform these transformations.

APPENDIX A

Lemma A.1. The encoding of Figure 3 is valid in the case that the locations from which the multisynchronization edges leave do not have further edges leaving them, whether internal or binary.
Figure 17. NTA for the travel reservation system.
synchronization edges, and each edge is labelled with a different multisynchronization channel, and the same guard condition must be associated with each channel for all the objects.

Proof. Without loss of generality, we restrict the proof to the (depicted) situation of two multisynchronization channels, the generalization to any number of multisynchronization channels is immediate.

⇒: We must see that every timed trace that can be performed by the left-hand side (L) of the figure, according to the established semantics, can also be performed by the right-hand side (R). In (L) time can elapse indefinitely when some object is not ready for multisynchronization, which is also the case in (R), since the first $n$ binary synchronizations are performed when the corresponding objects are able to do that. The objects that have reached the point of multisynchronization cannot execute any other actions, due to the restriction imposed, so that time elapsing is their only possibility.

The other possible behaviour of (L) is the multisynchronization either on $c$ or $d$ (depending on the guard values), which is captured in (R) by the sequence of synchronizations over the channels $c_1 \ldots c_n c_1 \ldots c_n$ (if $g_1$ is evaluated to true) or $c_1 \ldots c_n d_1 \ldots d_n$ (if $g_2$ is evaluated to true) in time zero.

⇐: For the converse we must see that every timed trace of (R) can also be performed by (L). The initial time elapsing, when some object is not ready for multisynchronization, is possible in both figures, as we have seen in the direct case. Now observe that the first binary synchronizations over $c_1 \ldots c_n$ can be carried out at different instants of time, as long as the different objects are able to perform their corresponding synchronization actions. The synchronization over $c_n$ marks the point in which all the objects are ready for the multisynchronization, and then, in time zero, the control timed automaton checks the guard values in order to decide the channels over which the binary synchronizations must occur. We use the invariant $(x=0)$ to enforce the execution of all these edges in time zero.

It can also be the case that some objects are ready for the multisynchronization (the binary synchronizations over $c_1 \ldots c_i$ have been made), but one object ($o_{i+1}$) never reaches its point of synchronization. This case corresponds in (L) to the situation in which the objects $o_1 \ldots o_i$ (and probably some additional ones) are able to execute the multisynchronizations, but $o_{i+1}$ has not reached its multisynchronization location. The behaviour in this case is the same both in (L) and (R); this object can still execute its own actions, or can even become deadlocked if it tries to perform a synchronization with some other objects.

□

Proof of Theorem 1. Notice first that all the constructions used in Figure 4 fulfil the conditions of Lemma A.1, which allows us to use multisynchronizations as a derived construction. The proof of this theorem is carried out by structural induction on the frame structure:

Base case: The base case is a basic frame $(r_1:A_1, \ldots, r_n:A_n)$. We must now prove that both semantics yield the same timed traces (up to functions $\varphi_1, \varphi_2$) and that the resulting clock valuations** of these timed traces are the same in both cases. We can see how these timed traces are generated movement by movement, in order to establish the equivalence (this can be formalized by an induction reasoning, but the proof would be the same both for the base and the general

**Over the clocks defined in the RT-UML description.
case). Each movement can be either a delay, or an internal action of an activity, or a binary synchronization between two activities.

Then, without loss of generality we can assume that the single movement under consideration is performed by $A_1$, or in the case of a synchronization that it is performed by $A_1$ and $A_2$.

The different syntactical cases we must consider are the following:

- **nil**: The only possible movement is time elapsing, both in the RT-UML activity semantics and in the timed automata semantics.
- **$a(d); A$**: According to the RT-UML semantics the action $a$ must be executed in $d$ time units, and then, it behaves as $A$. The same occurs according to the timed automata semantics, since we are using a new clock $x$ and one invariant that allows time elapsing up to time $d$ on the location from which $a$ is executed. Concerning the clock valuation, if the first activity in $A$ is not a reset over some clock $x$, it follows that no RT-UML clock will change its value in both semantics, because in the timed automata construction the only clocks that can be reset in this case are the new control clocks. Otherwise, when the first activity in $A$ is $\text{reset}(x)$ (or several consecutive reset operators, but for simplicity we consider only one), for some clock $x$, the timed automata construction captures both movements (the execution of $a$ and reset) in a single edge, because the clock $x$ is reset upon the execution of $a$. The same occurs for the remaining activity operators, when one or several reset operators appear at first consecutive activities in $A$, so we will omit this case in the description of these activities.
- **delay($d$); $A$**: Both semantics enforce the passage of $d$ time units before executing the activity $A$. In the case of the timed automaton a $\tau$ action is executed, but this action is hidden by $\varphi_2$, hence, in the timed trace we just see the delay in both cases.
- **reset($x$); $A$**: The reset actions in the RT-UML semantics are hidden by $\varphi_1$, and as we have mentioned before their effect is captured in the timed automata semantics by the third component of the edges that reach the starting location of $\text{ta}(A)$.
- **$A_1$ executes send($2, m$) and $A_2$ recv($1, m$)**: According to the RT-UML semantics the synchronization is immediate, executing the action $m$, and the same occurs in the NTA. Notice that time can elapse when only one of the objects has reached the point of synchronization, and this also occurs in both semantics.

**General case**: For the general case we must distinguish the following subcases:

- **$F_1; F_2$**: The induction hypothesis states that the timed traces are the same (up to $\varphi_1, \varphi_2$) for $F_1$ and $F_2$ in both semantics. The RT-UML semantics for the frame sequence enforces that all the activities inside $F_1$ must have finished before starting the activities in $F_2$ (rule NS1 in Table I). The same occurs in the timed automata construction due to the multisynchronization channel $c$ we use to activate the automata in $F_2$ when all the automata of $F_1$ have terminated. Actually, the multisynchronization is immediately executed, once all the automata of $F_1$ have terminated, and the same occurs with the RT-UML operational semantics (rule DF2 in Table V).

- **$\text{if}(g, F_1, F_2)$**: The RT-UML semantics does not allow time elapsing for this operator. Actually, rules IF1, IF2 in Table I enforce the execution of either $F_1$ or $F_2$, depending on the value of $g$. In the corresponding timed automata (Figure 4) there is a multisynchronization either on channel $c$ or $d$, also depending on the value of $g$, and these multisynchronizations are immediate too. Afterwards, we apply the induction hypothesis to obtain the same timed traces in both semantics.
$alt(F_1, F_2)$: Similar to the previous one, the only difference is that in both semantics either $F_1$ or $F_2$ can be executed, but notice that this (non-deterministic) decision is immediate.

$loop(g, F)$: As in the previous case, the RT-UML semantics does not allow time elapsing for this operator. When $g$ is true, rule LOO1 in Table I enforces the immediate execution of $F$, and then, upon its termination the loop is restarted immediately. Otherwise, when $g$ is false, the loop is immediately abandoned. In the timed automata for the loop operator (Figure 4) there are two multisynchronizations channels, $c$ and $s$, which correspond to the guard evaluation, and these multisynchronizations are immediate. When $g$ is true, the activities of $F$ are executed, and using the induction hypothesis we obtain the same timed traces in both semantics. Once all of these activities have terminated, in the timed automata there is another multisynchronization over channel $d$, which is also immediate.

$par(F_1, F_2)$: According to rules Par1-2 in Table III and rule DF3 in Table V we have an interleaving semantics for two frames running in parallel, and time can elapse when both frames allow this. In the timed automata construction for the parallel operator (Figure 4) we have $n$ binary synchronizations over channels $s_i$, $i = 1, \ldots, n$ that activate the separate template for the second frame. These synchronizations are therefore immediately executed, and thus we have all of these automata running in parallel. Hence, taking into account that the NTA semantics is also an interleaving semantics, and that time elapsing is possible when all the timed automata allow the passage of time, we conclude that both semantics yield the same timed traces. □

ACKNOWLEDGEMENTS

Supported by the Spanish government (co-financed by FEDER funds) with the project TIN2006-15578-C02-02, and the JCCLM regional project PAC06-0008-6995.

REFERENCES


