Innovations Syst Softw Eng (2009) 5:117–127
DOI 10.1007/s11334-009-0085-4

ORIGINAL PAPER

A development methodology for embedded systems based on RT-DEVS

Angelo Furfaro · Libero Nigro

Received: 15 March 2009 / Accepted: 20 April 2009 / Published online: 5 May 2009

Abstract This work is concerned with modelling, analysis and implementation of embedded control systems using RT-DEVS, i.e. a specialization of classic discrete event system specification (DEVS) for real-time. RT-DEVS favours model continuity, i.e. the possibility of using the same model for property analysis (by simulation or model checking) and for real time execution. Special case tools are reported in the literature for RT-DEVS model analysis and design. In this work, temporal analysis of a model exploits a translation in UPPAAL timed automata for exhaustive verification. For large models a simulator was realized in Java which directly stems from RT-DEVS operational semantics. The same concerns are at the basis of a real-time executive. The paper describes the proposed RT-DEVS development methodology and clarifies its implementation status. The approach is demonstrated by applying it to an embedded system example which is analyzed through model checking and implemented in Java. Finally, research directions which deserve further work are indicated.

Keywords DEVS · Real-time constraints · Embedded control systems · Model continuity · Temporal analysis · Timed automata · Model checking · Java

1 Introduction

There is a general agreement today about the importance of using formal tools for rigorous development of real-time systems which in general have safety and time critical requirements to fulfil. However, a known hard problem for the developer is how to ensure that a given formal model of a system, preliminarily analyzed from both functional and temporal viewpoints, is correctly reproduced in an implementation.

This paper describes a design methodology for embedded control systems, assisted by modelling, analysis and implementation tools, which makes it possible to experiment with *model continuity* [13,16,17,24], i.e. seamless development where the same model is used, with minimal change, for both property analysis and real time execution.

The modelling language is RT-DEVS [15,25], i.e. a specialization of classic discrete event system specification (DEVS) [27] with a weak synchronous communication model and constructs for expressing timing constraints. RT-DEVS owes to DEVS for both atomic and coupled component formalization and model continuity.

Previous and on-going work on DEVS-based development of real-time systems under model continuity is mainly driven by simulation, e.g. real-time simulation over a distributed context like real-time CORBA [5]. In real-time simulation the controlled environment and other modules are simulated during development and the simulation clock is constrained to advance at the same rate of the physical time. Simulation means are normally adopted due to high-level modelling and general message communication model.

Special case tools are reported in the literature to support specifically the usage of RT-DEVS. An application of RT-DEVS to safety analysis, called timed behavior analysis (TBA), is proposed in [25], where clock matrices are exploited for generating a timed reachability graph of a given model. The achieved graph is then explored for checking whether "unsafe" states can be reached. TBA is used to complement supervisory control techniques [22] for the design of a controller for a given discrete event system.

A. Furfaro · L. Nigro (✉)
Laboratorio di Ingegneria del Software, DEIS,
Università della Calabria, 87036 Rende, CS, Italy
e-mail: l.nigro@unical.it

A. Furfaro
e-mail: a.furfaro@deis.unical.it

Novel in the work described in this paper is:

- a mapping of the fundamental phases of modelling and safety/temporal analysis of RT-DEVS systems in terms of the popular and efficient UPPAAL toolbox with timed automata [4,8,12]. The translation purposely avoids the use of proprietary tools [25] for conducting state-space exploration analysis.
- the achievement of concrete tools in Java for RT-DEVS simulation and final system implementation. The Java-based approach improves applicability and portability of RT-DEVS software.

Prototype implementation tools were built by adapting the existing ActorDEVS agent infrastructure [6,9]. ActorDEVS is an original lean framework which supports DEVS and Parallel DEVS formalisms. A key feature of ActorDEVS is its *control-centric* character. The developer can customize the runtime executive. ActorDEVS was successfully exploited, in a case, for supporting Parallel DEVS components where conflicts among concurrent state transitions [7] are resolved in the runtime, a feature which is beyond standard DEVS [27]. Moreover, an implementation of Parallel DEVS over the High Level Architecture [11] for modelling and simulation of large and *variable structure* systems [18,20] was recently achieved [9]. The control engine of this realization is capable of ensuring logical precedence constraints among simultaneous events through a *tie-breaking* mechanism. In this paper, a specialization of the ActorDEVS control engine is described which stems from the abstract operational semantics of RT-DEVS.

The paper is structured as follows. First RT-DEVS and its operational semantics are described, then a transformation process of RT-DEVS specifications into UPPAAL is suggested for exhaustive verification activities based on model checking. The approach is demonstrated through a realistic embedded control system. The paper goes on by discussing the implementation status of Java-based development tools and the programming style. Finally, conclusions are presented with an indication of directions which deserve further work.

2 Background

2.1 DEVS Basics

DEVS [27] is a widespread modelling formalism for concurrent and timed systems, founded on systems theory concepts. A DEVS system consists of a collection of one or more components. Two types of components exist: *atomic* (or behavioural), and *coupled* (or structural) components. A DEVS atomic component is a tuple AM defined as $AM = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ where:

- X is the set of input values
- S is a set of states
- Y is the set of output values
- $\delta_{int} : S \to S$ is the *internal transition* function
- $\delta_{ext} : Q \times X \to S$ is the *external transition* function, where $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the set of *total states*, e is the *elapsed time* since last transition
- $\lambda : S \to Y$ is the *output function*
- $ta : S \to \mathbb{R}^{+}_{0,\infty}$ is the time *advance function*.

The sets X, S and Y are typically products of other sets. S, in particular, is normally the product of a set of *control states* (also said *phases*) and other sets built over the values of a certain number of variables used to describe the component at hand.

Informal semantics of above definitions are as follows. At any time the component is in some state $s \in S$. The component can remain in s for the time duration (*dwell-time*) $ta(s)$. A state s is said to be *transitory* in the case $ta(s) = 0$ and it is said *passive* if $ta(s) = \infty$, i.e. when the component can remain forever in s if no external event interrupts.

Provided no external event arrives, at the end of (supposed finite) time value $ta(s)$, the component moves to its next state $s' = \delta_{int}(s)$ determined by the internal transition function δ_{int}. In addition, just *before* making the internal transition, the component produces the output computed by the output function $\lambda(s)$ (see Fig. 1a). During its stay in s, the component can receive an external event x which can cause s to be exited earlier than $ta(s)$. Let $e \leq ta(s)$ be the elapsed time since the entering time in s. The component then exits state s moving to next state $s'' = \delta_{ext}(s, e, x)$ determined by the external transition function (see Fig. 1b). As a particular case, the external event x can arrive when $e = ta(s)$. In this (*collision*) case two events occur simultaneously: the internal transition event and the external transition event. A collision resolution rule is responsible for ranking the two events and determining the next state. After entering next state ns, the new time advance value $ta(ns)$ is computed and the same story continues. It should be noted that there is no way to directly generate an output from an external transition. To achieve this effect a transitory phase, used as destination of the external transition and whose lambda function generates the desired output, can be introduced (see Fig. 6).

In practice, an atomic component receives its inputs from typed *input ports* and similarly, generates outputs through

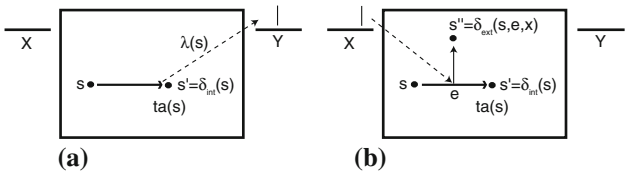


Fig. 1 State transitions. **a** *ta* exhaustion; **b** external event handling

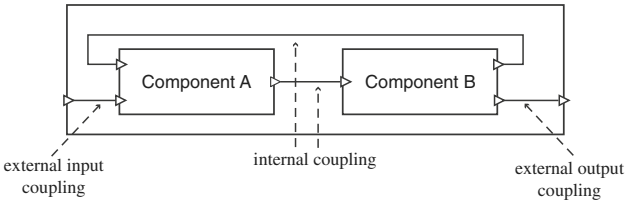


Fig. 2 A coupled component

typed *output ports*. Actually X is a set of pairs $< inp, v >$ where inp is an input port and v the type of values which can flow through inp. Y is a set of pairs $< outp, v >$ where $outp$ is an output port. Ports are architectural elements which enable modular system design. A component refers only to its interface ports. It has no knowledge about the identity of cooperating partners.

A coupled component (subnet) is an interconnection of existing atomic or coupled (hierarchical) components (see Fig. 2). Formally, it is a structure CM defined as $CM = (X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC)$, where:

- X and Y are input and output sets of the coupled component
- D is a set of (sub) component identifiers (or names)
- M is a set of (sub) DEVS components whose interconnection gives rise to the coupled model
- EIC is the external to internal coupling function (for routing external events to internal components)
- EOC is the internal to external coupling function (for routing internally generated events to the external environment of the coupled component)
- IC is the internal to internal coupling function.

2.2 RT-DEVS concepts

RT-DEVS [25] refines basic DEVS with the following concepts.

1. The dwell-time $ta(s)$ in a state now mirrors the execution time of an *activity* associated with the state. In particular, the execution time is specified by a (dense and static) time interval $[lb, ub]$, where lower and upper bounds $lb, ub \in \mathbb{R}^{+}_{0,\infty}$, $0 \leq lb \leq ub$, express uncertainty in the activity duration. Default interval of passive states is $[\infty, \infty]$ and can be omitted. Transitory (or immediate) states have interval $[0, 0]$.
2. Non determinism is assumed as collision resolution rule.
3. The communication model is weak synchronous, i.e. non blocking with (possible) message loss. At any communication, an output event is always immediately consumed. If the receiver is not ready, the message is lost. If both sender and receiver are ready to communicate, the output event is converted into an input event which is instantly received.

A time interval $[lb, ub]$ is made absolute at the instant in time τ the corresponding state s is entered. An internal transition outgoing s can occur at any time greater than or equal $\tau + lb$ but, to avoid a timing violation, before or at $\tau + ub$. An external transition fires upon synchronization on an input event independently of the dwell-time of current phase. It is assumed that a self-loop external transition does not restart timing in current phase. Pre-emption and restarting of current timing, when desired, can be simulated with the help of an transitory phase. Graphically (see e.g. Fig. 4), an internal transition is depicted by a thin oriented edge terminating with a dashed arrow which specifies the execution of the lambda (output) function, which can be void. An external transition is instead drawn by a thick oriented edge. Sending event ev through outport OP is denoted by the syntax $OP!ev$. Similarly, readiness to accept event ev through input port IP is expressed by $IP?ev$. The abstract executor of RT-DEVS initializes current time to 0 and iterates the following two basic steps.

1. The next minimal time at which new internal transitions can fire is determined and it becomes the current time.
2. All candidate internal transitions which can occur at current time are determined. Let C_i be an atomic component with one such a transition. Let the lambda function of current state of C_i consist of $OP!ev$. Let C_j be a component coupled with C_i where input port IP matches output port OP of C_i. Provided C_j has an outgoing transition from current state annotated with $IP?ev$, the two transitions (internal in C_i and external in C_j) are immediately executed with the event sent by C_i synchronously transmitted to C_j. In the case C_j is not ready to receive C_i event, the output transition in C_i is still made but the event gets lost. The above activity is repeated for each candidate internal transition. When the candidate set empties, the executor goes back to step 1.

It is worthy of note that while weak synchronization is a useful feature in general real time systems (e.g., a message with a sensor reading can be lost for a missing synchronization, in which case a controller can use previous sensor data), it increases the burden of the RT-DEVS modeller when the system cannot tolerate synchronization losses. Model validation through simulation or verification can help in assessing correct system behaviour.

3 A traffic light controller

The following describes the modelling of a traffic light control system (TLC) [21]. In the proposed scenario, the traffic

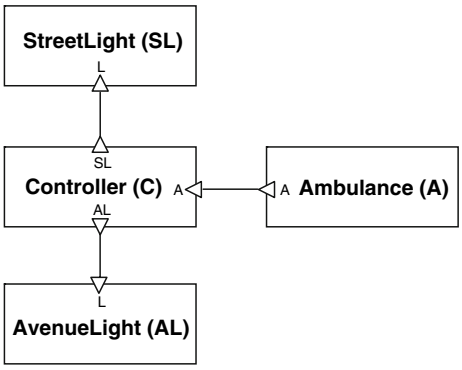


Fig. 3 Traffic light coupled model

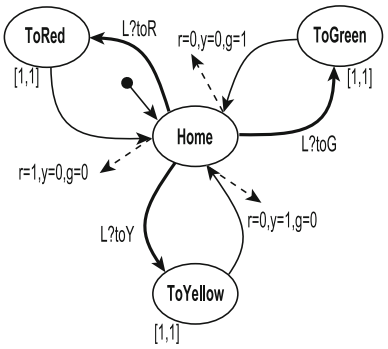


Fig. 4 Light behaviour

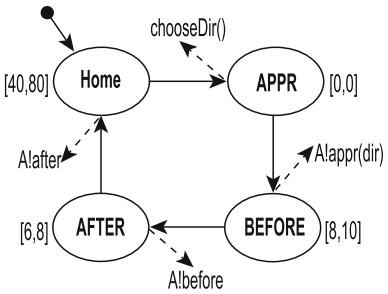


Fig. 5 Ambulance behaviour

flow at an intersection between an avenue and a street is regulated by two traffic lights. The lights are operated by a control device (controller) that, in normal conditions, alternates in a periodic way the traffic flow in the two directions. In addition, the controller is able to detect the arrival of an ambulance and to handle this exceptional situation by allowing the ambulance crossing as soon as possible and in a safe way. For the sake of simplicity, it is assumed that at most one ambulance can be in the closeness of the intersection at a given time. During normal operation conditions, the sequence green-yellow-red is alternated on the two directions with the light held green for 45 time units (tu), yellow for 5 tu and red on both directions for 1 tu. The intersection is equipped with sensors able to detect the presence of an ambulance at three different positions during its crossing. As soon as the ambulance arrival is detected, a signal named "approaching" is sent to the controller. When the ambulance reaches the nearness of the intersection the signal "before" is issued. After the ambulance completes the crossing the signal "after" is generated. The controller reacts to the "approaching" event by leading the intersection to a safe state, i.e. bringing both lights on red.

When the signal "before" is received, the controller switches to green the light on the ambulance's arrival direction. After the ambulance leaves the intersection ("after" event) the controller turns the green light to red and resumes its normal sequence. Figure 3 illustrates an RT-DEVS coupled model of the TLC system which is made of four connected components: there are two instances of the Light component, which respectively correspond to the light on the avenue and that on the street, one Ambulance component, which models the behaviour of the sensing equipments of the intersection, and one Controller component which implements the above described control logic. Couplings in Fig. 3 are realized between matching input/output ports. X/Y sets for the Controller are as follows:

```
X={<A,appr>,<A,before>,<A,after>}
Y={<SL,toR>,<SL,toY>,<SL,toG>,<AL,toR>,
  <AL,toY>,<AL,toG>}
```

Component behaviour is specified in Figs. 4, 5 and 6 where an oval box represents a phase of the component. The complete state set S obviously depends also on the component local variables. For instance, the Controller has a dir variable whose value indicates the avenue or the street, and logical variable amb where information about an arriving ambulance is maintained when current phase of the controller cannot be pre-empted. Similarly, light components keep the light status in the three logical variables r,y, and g. A light component (Fig. 4) is normally in the Home phase with default interval $[\infty, \infty]$. The arrival of a toR, toY or toG event causes an external transition respectively to toRed, toYellow or toGreen phase which is then exited after 1 time unit by an internal transition reaching again Home. The lambda function associated with the internal transitions specifies the required state changes in the light.

Behaviour of the ambulance (Fig. 6) is cyclic. After a non deterministic time in [40, 80], the ambulance announces itself by choosing an arriving direction and sending the appr event to the controller. From the BEFORE phase and after a time in [8, 10] the ambulance sends a before event to the controller. Finally, form the AFTER phase the ambulance signals its passage through the intersection by sending an after event with an elapsed time in [6, 8]. In Fig. 6, the normal and exceptional behaviours of the controller can be distinguished. The initial phase is BR1 (both lights reds). Under normal behaviour, the controller steps through a light cycle (e.g. from BR1, to AV to AY to BR2 for the

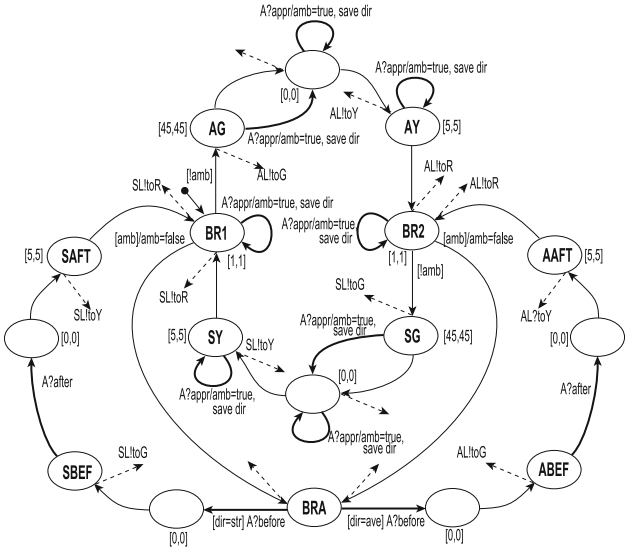


Fig. 6 Controller behaviour

avenue, and from BR2 to SG, to SY to BR1 for the street). It should be noted that a "both reds" condition (BR1 or BR2) is always maintained for 1 time unit. Avenue and street cycles strictly alternate. A normal cycle is started provided no ambulance is sensed. During a light cycle the arrival of ambulance pre-empts normal behaviour. In particular, a green phase (AG or SG) is immediately abandoned by anticipating the next yellow phase and then finishing the cycle. However, current yellow phase is never pre-empted. All of this guarantees the duration of the yellow phase (in the example in [21] it was erroneously made possible, in worst case conditions, that a yellow phase doubles its duration). It should be noted the efforts taken in Fig. 6 for not losing the approaching signals. As soon as an ambulance is sensed, the logical variable amb is set to true. At cycle end, the presence of an ambulance requires an exceptional behaviour to be executed by first reaching the BRA (both reds with ambulance) phase. From BRA, and depending on the arriving direction of the ambulance, the controller senses events from the ambulance and commands accordingly the light by turning it first green, then yellow after ambulance passage and finally red. Ambulance events (e.g. before and after) are processed by external transitions. Light control is instead realized through internal transitions. Following an exceptional behaviour, the controller restarts the normal cycle by giving the turn to the other direction.

3.1 Property requirements for the TLC

The TLC has safety and (bounded) liveness (e.g. deadline) properties, besides absence of deadlock or livelock conditions.

1. *Traffic must never be allowed in both directions simultaneously*. For safety reasons it is required that the status of the traffic lights be consistent at all times. To avoid accidents among vehicles crossing the intersection, when on a direction the light is green or yellow, thus allowing traffic in the direction, the light on the opposite direction must be red.
2. *Lights should be both reds at a before event*. No vehicle should be allowed to cross the intersection at a before event.
3. *Deadline of 3 tu for turning green the light after a before event*. Assuming that it takes at least 4 tu for the ambulance to reach the intersection from the time instant of the before signal, it follows that there exists a deadline of 3 tu for the controller to turn green the light on the arriving direction, also considering that a light takes 1 tu for changing its status.
4. *Correct sequencing of the lights on each direction*. A correct behaviour requires that only transitions from red to green, from green to yellow and from yellow to red should be allowed. A transition out of this sequence denotes a wrong sequence.
5. *The ambulance must be live*. In particular, after signalling an approach, it must be guaranteed that the ambulance model comes back to its Home phase.

4 Temporal analysis using UPPAAL

Weak synchronization and message losses increase the need for functional, safety and temporal analysis of an RT-DEVS model. In this work an RT-DEVS model is preliminary transformed into UPPAAL [4] for model checking. UPPAAL was chosen because it supports data variables and weak synchronization through broadcast channels [26].

4.1 An overview to UPPAAL

UPPAAL allows to verify systems modelled as networks of timed automata (TA) [3], i.e. timed finite state machines. TA synchronize to one another (*rendezvous*) by CSP-like channels which do not carry data values.

States (*locations*) of an automaton are linked by a set of edges (*transitions*). Time is handled by means of *clock* variables. Clocks have only a *reset* operation of the form $x = v$ where v is a non negative integer value. In addition a clock x can be compared against a non negative integer constant. Clocks of a system are automatically increased following the rate of advancement of the (hidden and dense) system time. UPPAAL extends basic TA with integer (and boolean) variables and arrays of integers, clocks and channels. TA processes are parametrized *template processes*. Global data variables can be used for process synchronization. State

transitions admit three (optional) components: (i) a *guard*, (ii) a *synchronization* operation (? for input and ! for output) on a channel, and (iii) an *action-part* consisting of a set of clock resets and variable assignments. The action part of an output command is executed before that of the matching input command. A guard (true if omitted) is a boolean expression built over data variables and clock constraints. It defines the transition enabling condition. For bounded delay in a location s, a clock *invariant* can be attached to s as a *progress* condition. UPPAAL supports also *committed* locations which must be exited immediately (without passage of time), and *urgent channels* whose synchronizations must be fired as soon as possible.

A channel can also be declared as *broadcast* for allowing a synchronous communication among multiple timed automata where one automaton plays the sender role and a (possibly empty) set of automata acts as receivers. Communications by broadcast channels are *weak* because the sender executes its state transition even if there is no receiver ready to synchronize.

UPPAAL consists of a graphical editor, a simulator and a verifier. The simulator executes a specification and visually documents the reached execution state by traversing the model state graph. When branches are encountered (multiple transitions which can fire starting from current state) one choice can be made interactively by the user or the simulator can be instructed to proceed randomly. The simulator is useful for model debugging and for examining a diagnostic trace built by the verifier. For systematic property assessment, though, the verifier must be used which tries to build the reachability graph of the model, where execution states are organized into equivalence classes (zones).

4.2 Query specifications

Safety (e.g., absence of deadlock) and bounded liveness (e.g. an end-to-end time constraint) properties can be verified by reachability analysis using a subset of TCTL formula [2,4] for networks of timed automata. Admitted formulas refer to local state properties, i.e. boolean expressions over predicates on locations and integer variables and clock constraints.

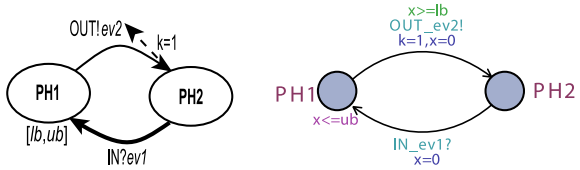
$E <> \varphi$ means *possibly* φ (i.e., a state can be reached in which φ holds).

$A[]\varphi$ means that φ holds *invariantly*,i.e. in all the reachable states.

$E[]\varphi$ means that φ holds *potentially always*, i.e. there exists at least one path where φ holds in all the reached states.

$A <> \varphi$ means that φ holds *always eventually* (which is equivalent to not $E[](\text{not } \varphi)$).

$\varphi \dashrightarrow \psi$ means that φ *always leads to* ψ, i.e. each path departing from a reachable state where φ holds leads to at least one state where ψ holds (which is equivalent to $A[](\varphi \text{ imply } A <> \psi)$.



(a) RT-DEVS **(b)** UPPAAL

Fig. 7 Translation schema from RT-DEVS to UPPAAL

4.3 Mapping RT-DEVS onto UPPAAL

The following describes the translation rules which are used to map an RT-DEVS model into a network of Timed Automata. A translation summary is provided in Fig. 7.

- An RT-DEVS component is mapped onto an UPPAAL template, which has a local clock x.
- Phases of the source component correspond one-to-one to locations of the template.
- Each pair of matching ports (e.g. the output port A of Ambulance and the input port A of Controller) together with a data/control symbol, is mapped on to a broadcast channel. For instance, broadcast channels A_appr, A_before and A_after are shared between Ambulance and Controller etc.
- Templates receive as parameters the broadcast channels corresponding to used input/output ports.
- Shared communication data, e.g. the dir variable used by Ambulance and Controller, become global declarations.
- A strict time interval $[lb, ub]$ of a phase PH of an RT-DEVS component implies the invariant $x \leq ub$ is added to location PH (see Fig. 7). Default time interval $[\infty, \infty]$ is implicit. Time interval $[0, 0]$ of a transitory phase is mapped on the invariant $x \leq 0$. UPPAAL requires bounds of a time interval to be expressed by naturals.
- An internal transition of the RT-DEVS model is associated with a timed edge having the guard $x \geq lb$. The update portion of the command on the edge contains the effect of the output function. An external transition is associated with an untimed edge which in turn relates to an input synchronization with a broadcast channel.

The above rules were applied to obtain the models in Figs. 8, 9 and 10 which depict the UPPAAL version of RT-DEVS TLC components. In Fig. 9, random choice of the ambulance arriving direction is simply achieved by non deterministic selection, on the edge between Home and

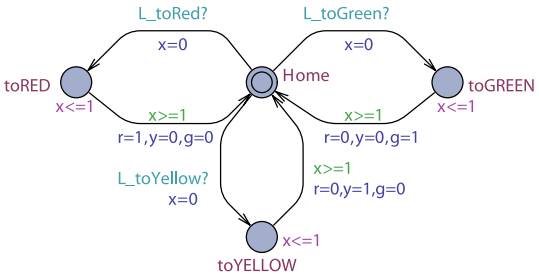


Fig. 8 Light template

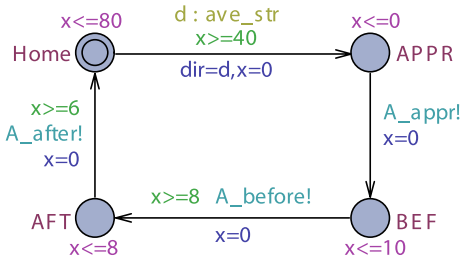


Fig. 9 Ambulance template

APPR locations, of the value of the local variable d between ave and str values (type ave_str is an alias of int[ave,str]).

As one can see, the UPPAAL templates correspond as close as possible to source RT-DEVS components. Therefore, the translation can be easily automated. The resultant UPPAAL system model is the parallel composition of one instance of the Controller template, two instances of the Light template and one instance of the Ambulance.

4.4 Verification of the TLC

The timed automata model of the TLC was verified using the UPPAAL version 4.1.0. Table 1 illustrates some TCTL queries issued to the UPPAAL verifier used for property analysis.

The first query checks for the absence of deadlock in the model. It was found to be satisfied thus confirming that despite weak synchronization and (possibly) message losses the system does not ever reach a deadlock situation.

The second and the third queries are used to check that the unsafe state where the traffic is allowed in both directions is never reached. The second query ask if invariantly whenever the light along the avenue is green or yellow (i.e. the traffic is allowed along this direction) this implies that the light on the street is red. The third query is the dual of the second.

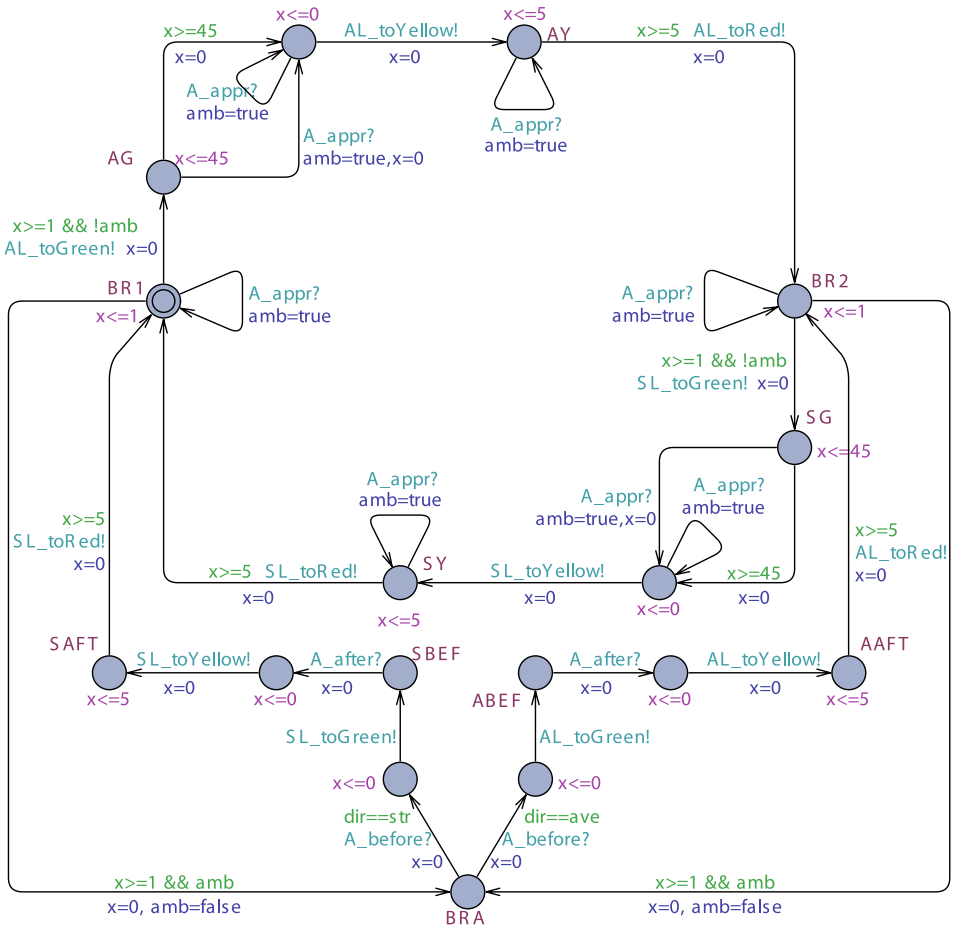


Fig. 10 Controller template

Table 1 UPPAAL queries for property analysis of TLC

Property	Query	Result
Absence of deadlocks	A[] !deadlock	Satisfied
Lights consistency	A[] (AL.g==1||AL.y==1) imply SL.r==1	Satisfied
Lights consistency	A[] (SL.g==1||SL.y==1) imply AL.r==1	Satisfied
Ambulance is live	A.APPR–>A.Home	Satisfied
Deadline checking	A[] flag imply z<=3	Satisfied

Because both were found satisfied the model is safe from this point of view.

Correct sequencing of lights was verified by introducing three additional variables in the Light template for storing the previous status of the light, by changing the Light behaviour so as to conserve previous status at any new assignment, and by checking that it is always true that a green status is preceded by the red status etc. These details and queries are omitted for simplicity.

Liveness of the ambulance is simply verified by checking that a global state where Ambulance find itself in the APPR location leads to a state where it find itself in the Home location, thus confirming that whenever an ambulance is going to reach the intersection it will eventually complete the crossing.

Finally, a few additional words relate to deadline checking. The UPPAAL model was decorated by introducing the global logical variable flag and the extra clock z. Variable flag is set to true in the Ambulance template when the before event is sent to controller, and reset in the Light template (therefore in both instances of the template) when the green status is installed (on the exiting edge from the toGreen location in Fig. 8). Clock z is reset by the transition from BEF to AFT in the Ambulance. It was found that not only the required deadline is fulfilled but that in reality 1 tu is always sufficient for the controller, following a before signal, to turn green the light in the arriving direction of the ambulance.

5 Implementation status

RT-DEVS was prototyped in Java using an adaptation of ActorDEVS minimal agent-based framework [6,9]. The underlying agent computational model is a light-weight variant of the Actor model [1]. An actor is a threadless object which has a public message interface, a set of hidden data variables and a behaviour modelled as a finite state machine. An actor is at rest until a message arrives. Actors communicate one with another by asynchronous message passing. Message processing is atomic and is the responsibility of the `handler()` method (Fig. 11) which implements the actor dynamic behaviour (the `become()` method changes current actor state). For distributed execution, actors can be parti-

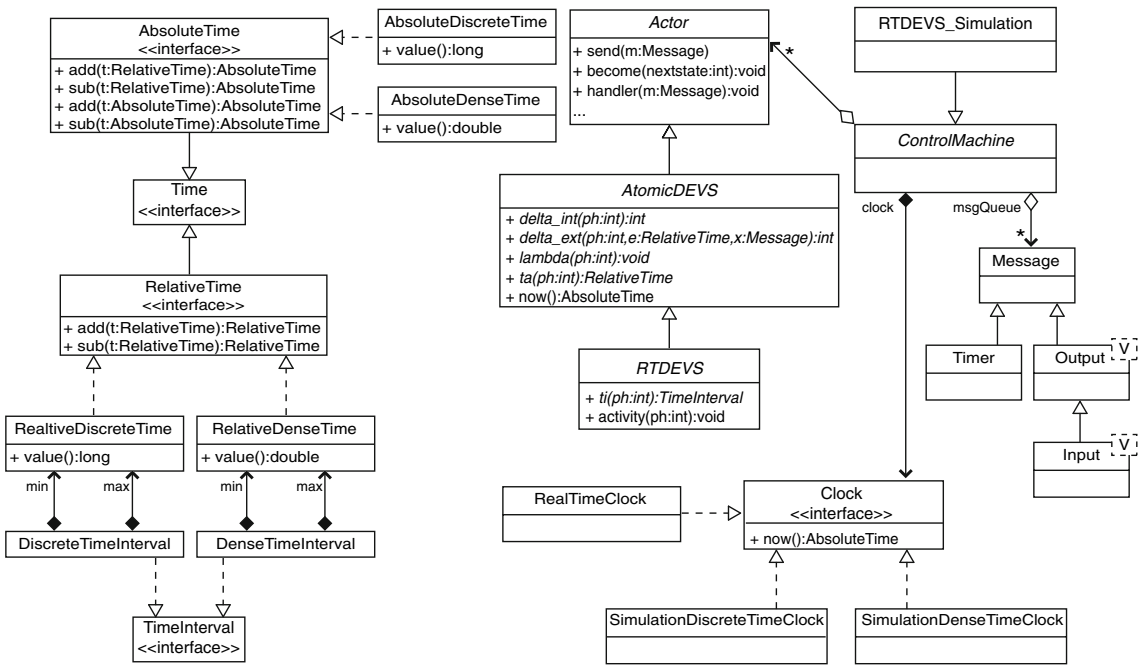


Fig. 11 Simplified UML class diagram of RT-DEVS framework

```
public abstract int delta_int( int phase );
public abstract int delta_ext( int phase, RelativeTime e, Message x );
public abstract void lambda( int phase );
public abstract RelativeTime ta( int phase );
public abstract TimeInterval ti( int phase );
public void activity( int phase ){}
public AbsoluteTime now();
```

Fig. 12 Programming interface of RTDEVS atomic components

tioned into execution theatres [10]. Within a theatre, actors are transparently orchestrated by a control machine which furnishes basic message-based time-sensitive scheduling and dispatching services. These services can be customized.

ActorDEVS was built over actors by encapsulating DEVS semantics within the `handler()` method of the `AtomicDEVS` abstract base class (see Fig. 11). A specific DEVS component derives from `AtomicDEVS` and focuses only on a redefinition of DEVS basic functions (`delta_int()`, `delta_ext()` etc.). Typed input/output ports are supported respectively by parametric classes `Input<V>` and `Output<V>` which are extensions of the `Message` base class. Typically, `V` is a user defined class which specifies the data/control symbols which can flow through the port. `Input` is a subclass of `Output`. Each component exports its input port types. Output ports are instantiated by a configuration method (e.g. the `main()` method) and passed to relevant components e.g. at construction time. This method is also in charge of linking matching ports for establishing a coupled model.

RT-DEVS specializes the DEVS behaviour provided by `AtomicDEVS` according to new time model and weak synchronous communication model.

Both discrete and dense time models are supported. The interfaces `AbsoluteTime` and `RelativeTime`, heir of the base interface `Time`, respectively model absolute and relative time notions. The framework furnishes concrete implementation classes for the dense and discrete cases. In particular, a concrete time object has a `value()` method which returns a `long` for discrete time, and a `double` for dense time. The interface `TimeInterval` abstracts the notion of a time interval and suitable concrete implementations of it are given.

An RT-DEVS atomic component is programmed as a class which derives directly or indirectly from the `RTDEVS` abstract base class, which exposes the contract of operations (see also the extract in Fig. 12).

A specific component must implement the abstract methods of `RTDEVS` in order to specify its behaviour. For simulation purposes the `activity()` method can be left to its default (no-operation) body. Phases are coded as integers. Internal and external transitions return the `int` value which identifies the next phase. It should be noted that component methods have direct access to the whole state by accessing the

```
public class LightEvent {
    public static enum Symbol{ TO_RED, TO_YELLOW, TO_GREEN };
    private Symbol symbol;
    public Symbol getSymbol(){ return symbol; }
    public void setSymbol( Symbol symbol ){ this.symbol=symbol; }
}//LightEvent
```

Fig. 13 Class of light events

```
public class Light extends RTDEVS{
    //message interface
    public static class L extends Input<LightEvent>{}
    //phases
    private static final byte Home=0, ToRED=1, ToYELLOW=2, ToGREEN=3;
    //state variables
    private byte r=1, y=0, g=0, id;
    private Monitor m;
    public Light( byte id, Monitor m ){ this.id=id; this.m=m; initialPhase(Home); }
    public int delta_int( int phase ){
        if( phase!=Home ) phase=Home;
        return phase;
    }//elta_int
    public int delta_ext( int phase, RelativeTime e, Message x ){
        if( phase==Home ){
            LightEvent le=((L)x).get();
            if( le.getSymbol()==LightEvent.Symbol.TO_RED ) phase=ToRED;
            else if( le.getSymbol()==LightEvent.Symbol.TO_YELLOW ) phase=ToYELLOW;
            else phase=ToGREEN;
        }
    return phase;
    }//delta_ext
    public RelativeTime ta( int phase ){
        if( phase==Home ) return RelativeDenseTime.INFINITY;
        return new RelativeDenseTime(1);
    }//ta
    public TimeInterval ti( int phase ){
        if( phase==Home ) return new DenseTimeInterval();//[infty,infty]
        return new DenseTimeInterval(1,1);
    }//ti
    public void lambda( int phase ){
        if( phase!=Home ){
            switch( phase ){
                case ToRED: r=1; y=0; g=0; break;
                case ToYELLOW: r=0; y=1; g=0; break;
                case ToGREEN: r=0; y=0; g=1; break;
                default: throw new RuntimeException("Illegal phase");
            }
            m.light( id, r, y, g, ((AbsoluteDenseTime)now()).value() );//to monitor
        }
    }//lambda
    protected boolean acceptable( Message x ){ return x instanceof L; }//acceptable
}//Light
```

Fig. 14 Class light of the TLC

component local data variables. The `ti()` method returns the (dense or discrete) time interval associated with the given state. Method `now()` returns the `AbsoluteTime` value of current time.

The resulting programming style is exemplified by showing details of the Light atomic component. Light events were modelled as instances of the `LightEvent` class (Fig. 13). The Light component, shaped for prototyping and simulation purposes, is illustrated in Fig. 14.

For components with non punctual time intervals (e.g. Ambulance and Controller) the `ta()` method returns a number uniformly distributed in the time interval of current phase.

Java TLC model was executed using dense time and the `RTDEVS_Simulation` control engine which mimics the

RT-DEVS operational semantics. `RTDEVS_Simulation` receives the (`AbsoluteDenseTime`) simulation time limit (e.g. 10^7) and a simulation clock (here a `SimulationDenseTimeClock`). `RTDEVS` maintains a priority queue of timers ranked by ascending fire times (absolutized `ta` values). A `Timer` object is a timed message. Before timer expiration, both the remaining time or the elapsed time since its setting can be checked.

The engine fires most imminent internal transitions one at a time and updates the simulation clock to the fire time accordingly. The output function then sends synchronously its message to the coupled component (actually, the `RTDEVS handler()` is directly invoked with the message as an argument, without involving the control machine). In the case the partner component is not ready for synchronization, the sent message is simply lost.

During the TLC simulation, a `Monitor` object (transducer) gets informed of event occurrences and checks system properties (e.g. it counts the number of times an hazardous state, e.g. green-green of the two lights, is reached, and measures the maximal time distance between the occurrence time of the green light in the arrival direction of the ambulance, and that of the immediately preceding before event, etc.). Also under simulation, the TLC was found to be temporally correct.

For real-time execution, RT-DEVS naturally requires a multi-processor implementation (each component runs on its own processor, as was assumed by temporal analysis). The `ta()` function is no longer useful. The `activity()` method should be programmed with the (sub)algorithms to be carried out in each phase of the component. All other methods remain unchanged. Of course, a real-time executive has to possibly compensate for violations of activity durations. An activity can terminate earlier than its lower bound duration or after its upper bound. In the first case the engine can delay the firing of the internal transition until the real time clock reaches the lower bound. In the latter case activity interruption and concepts of adaptive scheduling and imprecise computation [14] could help. As a particular scenario, an RT-DEVS model could be analyzed and executed on a single processor, by ensuring atomicity and mutual exclusion of activities.

6 Conclusions

In this paper, a development methodology is proposed which enables specification, analysis and Java implementation of RT-DEVS based embedded control systems operated under model continuity, i.e. where the same model is reused with minimal change during different design stages—from model design to functional/non functional property evaluation down to final system execution. The approach makes it possible to replace the runtime control engine used for executing a model, which can be sensitive to simulation or real-time.

A properly abstracted model, mainly focussing on timing aspects, can be thoroughly studied by exhaustive verification techniques. In particular, model checking activities rest on a translation of an RT-DEVS model onto the timed automata of UPPAAL. All of this avoids the use of proprietary analysis tools as described in [25], and opens to the exploitation of efficient algorithms and data structures of the popular UPPAAL toolbox [26]. Of course, in the case of large models which can be difficult to analyze with model checking, the approach depends on simulation.

Current implementation tools are based on an adaptation of a lean and customizable agent framework [6,9] in Java.

On-going and future work is directed at:

- experimenting with real-time executives using the Real-time Specification for Java platform [23]
- extending the approach to the distributed context using standard middleware like HLA/RTI or real-time CORBA [19]
- building development tools for visual modelling, prototyping/simulation, and automatic generation of Java code and UPPAAL XML code.

Another interesting and challenging direction of research will be concerned with a possible exploitation in RT-DEVS systems of the parallelism offered by modern multi-core architectures.

References

1. Agha G (1986) Actors: a model for concurrent computation in distributed systems. The MIT Press, Cambridge
2. Alur R, Courcoubetis C, Dill DL (1993) Model checking in dense real-time. Inf. Comput. 104(1):2–34
3. Alur R, Dill DL (1994) A theory of timed automata. Theor. Comput. Sci. 126(2):183–235
4. Behrmann G, David A, Larsen KG (2004) A tutorial on UPPAAL. In: Bernardo M, Corradini F (eds) Formal methods for the design of real-time systems, LNCS, vol 3185. Springer, Heidelberg, pp 200–236
5. Cho Y, Hu X, Zeigler B (2003) The RTDEVS/CORBA environment for simulation-based design of distributed real-time systems. Simulation 79(4):197–210
6. Cicirelli F, Furfaro A, Nigro L (2006) A DEVS M&S framework based on Java and actors. In: Proceedings of second European modeling and simulation symposium (EMSS'06). Barcelona, Spain
7. Cicirelli F, Furfaro A, Nigro L (2007) Conflict management in PDEVS: An experience in modelling and simulation of time Petri nets. In: Proceedings of summer computer simulation conference (SCSC'07), pp 349–356
8. Cicirelli F, Furfaro A, Nigro L (2007) Using TPN/Designer and UPPAAL for modular modelling and analysis of time-critical systems. Int. J. Simul. Syst. Sci. Technol. 8(4):8–20
9. Cicirelli F, Furfaro A, Nigro L (2008) Actor-based simulation of PDEVS systems over HLA. In: Proceedings of 41st annual simulation symposium (ANSS'08), pp 229–236

10. Cicirelli F, Furfaro A, Nigro L (2008) An agent infrastructure over HLA for distributed simulation of reconfigurable systems and its application to UAV coordination. Simulation: transactions of the society for modeling and simulation international. doi:10.1177/0037549708100187
11. Defense Modeling and Simulation Office: HLA-RTI. http://www.dmso.mil/public/transition/hla
12. Furfaro A, Nigro L (2007) Modelling and schedulability analysis of real-time sequence patterns using time Petri nets and UPPAAL. In: Proceedings of international workshop on real time software (RTS'07), pp 821–835
13. Furfaro A, Nigro L, Pupo F (2006) Modular design of real-time systems using Hierarchical Communicating Real-time State Machines. Real Time Syst 32(1/2):105–123
14. Halang WA (1992) Load adaptive dynamic scheduling of tasks with hard deadlines useful for industrial applications. Computing 47:199–213
15. Hong J, Song H, Kim T, Park K (1997) A real-time discrete-event system specification formalism for seamless real-time software development. Discrete Event Syst Theory Appl 7:355–375
16. Hu X, Zeigler B (2004) Model continuity to support software development for distributed robotic systems: A team formation example. J Intell Robot Syst 39(1):71–87
17. Hu X, Zeigler B (2005) Model continuity in the design of dynamic distributed real-time systems. IEEE Trans Syst Man Cybern A Syst Hum 35(6):867–878
18. Hu X, Zeigler B, Mittal S (2005) Variable structure in devs component-based modelling and simulation. Simulation 81(2):91–102
19. Object Management Group: RealTime-CORBA Specification. http://www.omg.org/docs/formal/03-11-01.pdf
20. Posse E, Vangheluwe H (2007) Kiltera: a simulation language for timed, dynamic structure systems. In: Proceedings of 40th annual simulation symposium (ANSS'07), pp 293–300
21. Raju SCV, Shaw AC (1994) A prototyping environment for specifying and checking Communicating Real-time State Machines. Softw Pract Exp 24(2):175–195
22. Ramadge RJ, Wonham WM (1987) Supervisory control of a class of discrete event processes. SIAM J Control Optim 25(1):206–230
23. RTSJ. http://jcp.org/aboutJava/communityprocess/first/jsr001/rtj.pdf
24. Shang H, Wainer G (2008) Dynamic structure devs: improving the real-time embedded systems simulation and design. In: Proceedings of 41st annual simulation symposium (ANSS'08), pp 271–278
25. Song H, Kim T (2005) Application of real-time DEVS to analysis of safety-critical embedded control systems: railroad-crossing example. Simulation 81(2):119–136
26. Uppaal. http://www.uppaal.com
27. Zeigler BP, Praehofer H, Kim T (2000) Theory of modeling and simulation, 2nd edn. Academic Press, New York