Proceedings of the International Multiconference on
Computer Science and Information Technology pp. 601–608
ISBN 978-83-60810-14-9
ISSN 1896-7094

# Embedded Control Systems Design based on RT-DEVS and Temporal Analysis using UPPAAL

Angelo Furfaro and Libero Nigro
Laboratorio di Ingegneria del Software
Dipartimento di Elettronica Informatica e Sistemistica
Universitá della Calabria, 87036 Rende (CS) Italy
http://www.lis.deis.unical.it
Email: a.furfaro@deis.unical.it, l.nigro@unical.it

***Abstract*—This work is concerned with modelling, analysis and implementation of embedded control systems using RT-DEVS, i.e. a specialization of classic DEVS (Discrete Event System Specification) 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 proposed in the literature for RT-DEVS model analysis and design. In this work, temporal analysis exploits an efficient translation in UPPAAL timed automata. The paper shows an embedded control system model and its 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 discusses the implementation status and, finally, indicates research directions which deserve further work.**

***Index Terms*—DEVS, real-time constraints, embedded control systems, model continuity, temporal analysis, timed automata, model checking, Java.**

## I. 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 some work aimed to the realization of tools for modelling, analysis and implementation of embedded control systems, specifically for experimenting with model continuity [1], [2], i.e. seamless development where the same model is used both for property analysis (through simulation or model checking) and for real time execution. The modelling language is RT-DEVS [3], [4], i.e. is a specialization of classic DEVS (Discrete Event System Specification) [5] 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. Special case tools are reported in the literature [4] to support a development methodology for RT-DEVS.

The original contribution of this work is twofold:

- proposing 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 [6], [7], [8]
- building concrete tools in Java for RT-DEVS simulation and final system implementation. The Java-based approach aims to improve applicability and portability of RT-DEVS software.

This paper introduces RT-DEVS and its operational semantics, 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. After that, current implementation status of Java-based development tools and programming style are clarified. Prototype tools were achieved by adapting existing tools for ActorDEVS [9], [10]. Finally, conclusions are presented with an indication of directions of further work.

## II. RT-DEVS DEFINITIONS

### A. DEVS Basics

DEVS [5] 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 $AM =< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$ where:

- $X$ is the set of input values
- $S$ is a set of states
- $Y$ is the set of output values
- $\delta_{int} : S \rightarrow S$ is the *internal transition* function
- $\delta_{ext} : Q \times X \rightarrow S$ 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
- $\lambda : S \rightarrow Y$ is the *output function*
- $ta : S \rightarrow \mathbb{R}^+_{[0,\infty]}$ 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*) $ta(s)$. $ta(s)$ can be 0, in which case $s$ is said a transitory state, or it can be $\infty$, in which case it is said a passive state because 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 $\delta_{int}$. In addition, just *before* making the internal transition, the component produces the output computed by the output function $\lambda(s)$. 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 $\delta_{ext}$. 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 state $s'$, the new time advance value $ta(s')$ 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. 4).

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

### B. RT-DEVS Concepts

RT-DEVS [4] 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 $\tau$ 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. 2), 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 become 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.

## III. A TRAFFIC LIGHT CONTROLLER

The following describes the modelling of a Traffic Light Control system (TLC) [11]. In the proposed scenario, the traffic 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.

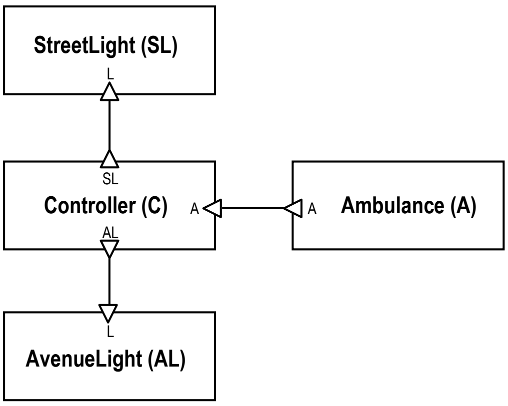


Fig. 1. Traffic light coupled model

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. Fig. 1 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. 1 are realized between matching input/output ports. $X/Y$ sets for the Controller are as follows:

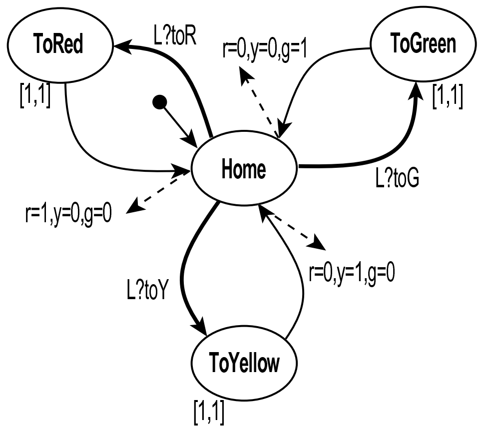


Fig. 2. Light behaviour

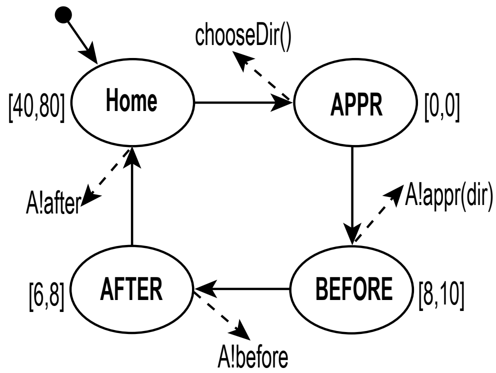


Fig. 3. Ambulance behaviour

```
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 Figg. from 2 to 4 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 indicate 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. 2) 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. 4) 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. 4 the normal and exceptional

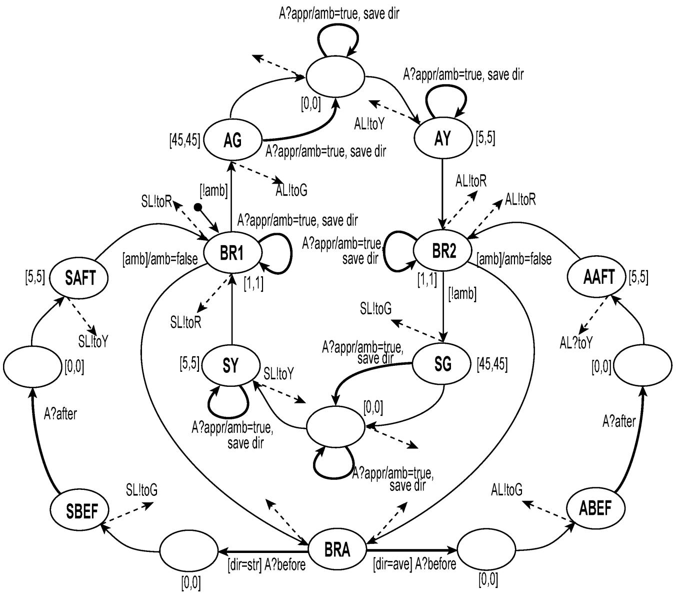

Fig. 4. Controller behaviour

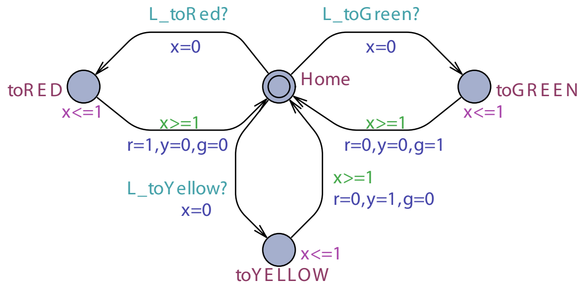

Fig. 5. Light template

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 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 [11] 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. 4 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.

### A. 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.

## IV. 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 [6] for model checking. Uppaal was chosen because it supports data variables and weak synchronization through broadcast channels [12]. The following summarizes the translation rules.

- 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.

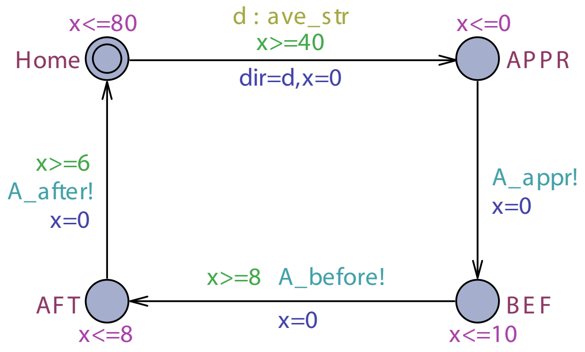


Fig. 6. Ambulance template

TABLE I
UPPAAL QUERIES FOR PROPERTY ANALISYS 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 |
| Lights must never be both green | E<> SL.g==1 && AL.g==1 | not satisfied |
| Ambulance is live | A.APPR --> A.Home | satisfied |
| Deadline checking | A[] flag imply z<=3 | satisfied |

- 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. 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 Figg. from 5 to 7 which depict the UPPAAL version of RT-DEVS TLC components. In Fig. 6, random choice of the ambulance arriving direction is simply achieved by non-deterministic selection, on the edge between Home and 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.

### A. Verification of the TLC

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

Absence of deadlocks confirms the TLC model correctly behave despite weak synchronization and (possibly) message loss. That the unsafe state of both lights green is never reached is checked by asking the verifier if there exists a state in the state graph where the g data of both lights is 1. In addition, it was verified that whenever the traffic is allowed in one direction (the light is green or yellow), the light is red on the other direction.

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.

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. 5). 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.

## V. IMPLEMENTATION STATUS

RT-DEVS was prototyped in Java using an adaptation of the ActorDEVS lean agent-based framework [9], [10]. The following provides some implementation hints and gives a flavour of the programming style. Both discrete and dense time models are supported, through the class hierarchy (interfaces are underlined): <u>`Time`</u>, <u>`AbsoluteTime`</u>, <u>`RelativeTime`</u>, <u>`TimeInterval`</u>, `AbsoluteDiscreteTime`, `AbsoluteDenseTime`, `RelativeDiscreteTime`, `RelativeDenseTime`, `DiscreteTimeInterval`, `DenseTimeInterval`. A concrete time object has a `value()` method which returns a `long` for discrete time, and a `double` for dense time. An RT-DEVS atomic component must be programmed as a class which derives directly or indirectly from the `RTDEVS` abstract base class, which provides the contract of operations (see the extract in Fig. 8) and basic behaviour.

A specific component must implement the abstract methods of `RTDEVS` in order to specify its specific 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 of the next phase. It should be noted that component methods have direct access to the whole state by accessing the 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.

Typed input/output ports are supported respectively by parametric classes `Input<V>` and `Output<V>`. Typically, `V` is a

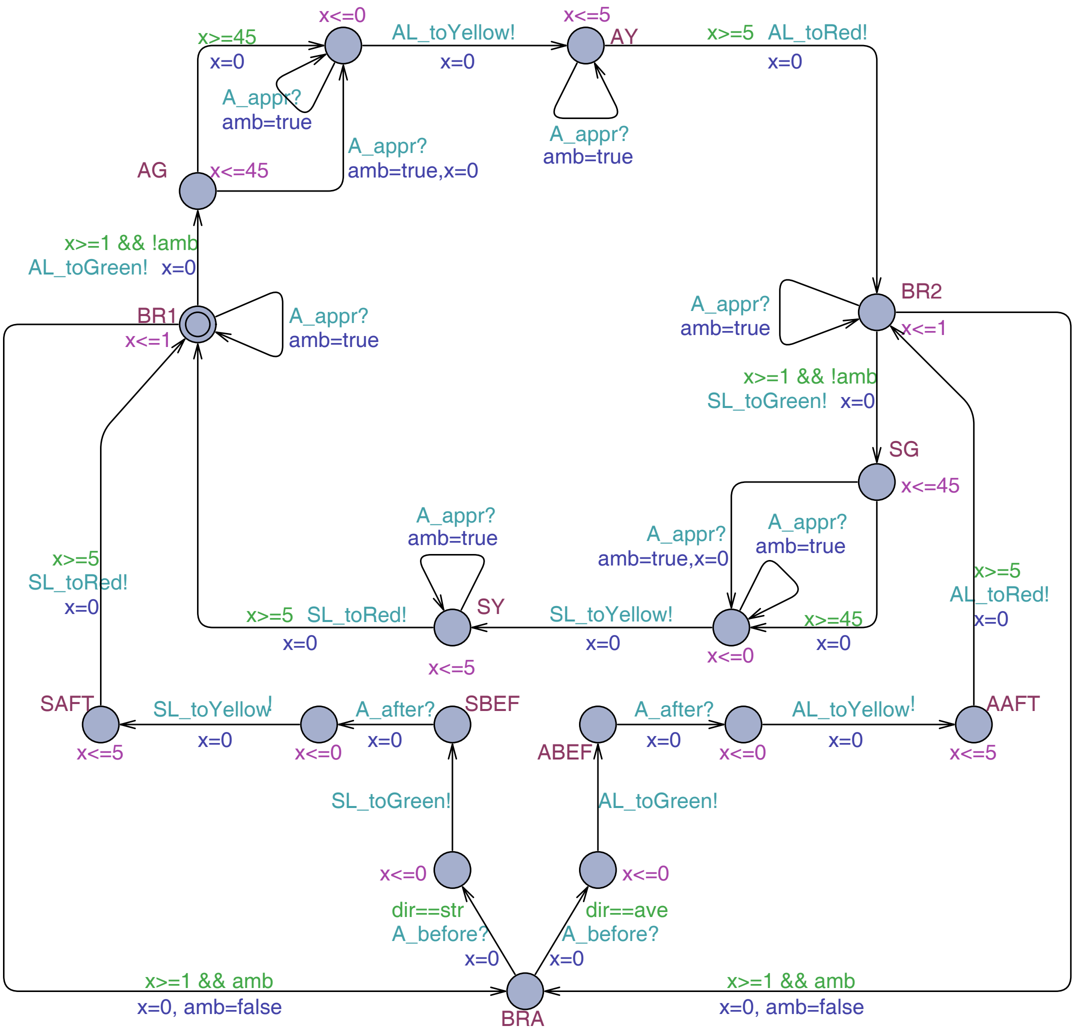


Fig. 7. Controller template

```
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. 8. An extract of RTDEVS atomic components programming interface

```
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. 9. Class of light events

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 created by a configurer (e.g. the `main()` method) and passed to relevant components e.g. at construction time. The configurer is also in charge of linking matching ports for establishing a coupled model. Programming style is exemplified by showing details of the Light atomic component. Light events were modelled as instances of the `LightEvent` class (Fig. 9). The Light component, shaped for prototyping and simulation purposes, is illustrated in Fig. 10.

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). The engine fires most imminent

```
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. 10. Class Light of the TLC

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. In the case the partner component is not ready for synchronization, the sent message is simply lost. During simulation, a `Monitor` object (transducer) gets informed of event occurrences and checks system properties (e.g. it counts the number of times the bad state green-green of the two lights is reached, and measures the maximal time distance between the occurrence time of the green light in the arriving 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 [13] 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.

## VI. Conclusion

This paper reports about specification, analysis and Java implementation of RT-DEVS systems operated under model continuity. Model checking is enabled by a translation onto timed automata of UPPAAL. For large models an achieved discrete-event simulation tool can be exploited. Java implementations rely on a minimal, efficient and customizable agent framework [9], [10].

On-going and future work is directed at:

- experimenting with real-time executives using the Real-time Specification for Java platform [14]
- extending the approach to the distributed context using standard middleware like HLA/RTI or real-time CORBA
- building development tools for visual modelling, prototyping/simulation, and automatic generation of Java code and UPPAAL XML code.

## References

[1] X. Hu and B. Zeigler, "Model continuity to support software development for distributed robotic systems: A team formation example," *Journal of Intelligent and Robotic Systems*, vol. 39, no. 1, pp. 71–87, 2004.

[2] ——, "Model continuity in the design of dynamic distributed real-time systems," *IEEE Trans. Syst., Man, Cybern. A*, vol. 35, no. 6, pp. 867–878, 2005.

[3] J. Hong, H. Song, T. Kim, and K. Park, "A real-time discrete-event system specification formalism for seamless real-time software development," *Discrete Event Systems: Theory and Applications*, vol. 7, pp. 355–375, 1997.

[4] H. Song and T. Kim, "Application of real-time DEVS to analysis of safety-critical embedded control systems: railroad-crossing example," *Simulation*, vol. 81, no. 2, pp. 119–136, 2005.

[5] B. P. Zeigler, H. Praehofer, and T. Kim, *Theory of modeling and simulation*, 2nd ed. New York: Academic Press., 2000.

[6] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems*, ser. LNCS 3185, M. Bernardo and F. Corradini, Eds. Springer, 2004, pp. 200–236.

[7] F. Cicirelli, A. Furfaro, and L. Nigro, "Using TPN/Designer and UPPAAL for modular modelling and analysis of time-critical systems," *International Journal of Simulation Systems, Science & Technology*, vol. 8, no. 4, pp. 8–20, 2007, special Issue on Frameworks and Applications in Science and Engineering.

[8] A. Furfaro and L. Nigro, "Modelling and schedulability analysis of real-time sequence patterns using time Petri nets and UPPAAL," in *Proc. of International Workshop on Real Time Software (RTS'07)*, October 16 2007, pp. 821–835.

[9] F. Cicirelli, A. Furfaro, and L. Nigro, "A DEVS M&S framework based on Java and actors," in *Proc. of 2nd European Modeling and Simulation Symposium (EMSS'06)*, Barcelona, Spain, October 4-6 2006.

[10] ——, "Actor-based simulation of PDEVS systems over HLA," in *Proc. 41st Annual Simulation Symposium (ANSS'08)*, 2008, pp. 229–236.

[11] S. C. V. Raju and A. C. Shaw, "A prototyping environment for specifying and checking Communicating Real-time State Machines," *Software–Practice and Experience*, vol. 24, no. 2, pp. 175–195, 1994.

[12] Uppaal. [Online]. Available: http://www.uppaal.com

[13] W. A. Halang, “Load adaptive dynamic scheduling of tasks with hard deadlines useful for industrial applications,” *Computing*, vol. 47, pp. 199–213, 1992.

[14] RTSJ. [Online]. Available: http://jcp.org/aboutJava/communityprocess/first/jsr001/rtj.pdf