Principles of Discrete Event System Specification model verification

Hesham Saadawi and Gabriel Wainer

SIMULATION 2013 89: 41 originally published online 23 October 2011
DOI: 10.1177/0037549711424424

Simulation

Simulation: Transactions of the Society of Modeling and Simulation International
89(1) 41–67

DOI: 10.1177/0037549711424424
sim.sagepub.com
SAGE SCS

Principles of Discrete Event System Specification model verification

Hesham Saadawi[1] and Gabriel Wainer[2]

Abstract

Real-time systems modeling and verification is a complex task. In many cases, formal methods have been employed to deal with the complexity of these systems, but checking those models is usually unfeasible. Modeling and simulation methods introduce a means of validating these model's specifications. In particular, Discrete Event System Specification (DEVS) models can be used for this purpose. Here, we introduce a new extension to the DEVS formalism, called the Rational Time-Advance DEVS (RTA-DEVS), which permits modeling the behavior of real-time systems that can be modeled by the classical DEVS; however, RTA-DEVS models can be formally checked with standard model-checking algorithms and tools. In order to do so, we introduce a procedure to create timed automata (TA) models that are behaviorally equivalent to the original RTA-DEVS models. This enables the use of the available TA tools and theories for formal model checking. Further, we introduce a methodology to transform classic DEVS models to RTA-DEVS models, thus enabling formal verification of classic DEVS with an acceptable accuracy.

Keywords

Discrete Event System Specification, model checking, Rational Time-Advance Discrete Event System Specification, real-time systems, timed automata

1. Introduction

Real-time (RT) systems are very advanced computer systems with hardware and software components with timing constraints. In some cases, they have 'soft' timing constraints (i.e. a deadline can be missed without serious consequences). In other cases, the system must satisfy 'hard' timing constraints (and a missed deadline can result in catastrophic consequences). In these highly reactive systems, not only correctness is critical, but also the timeliness of the executing tasks. For instance, if we consider the design decisions made for an autopilot for an aircraft, or a controller for an automated factory, in these cases we need to obtain system responses within well-defined deadlines.

Consequently, RT software development is still time consuming, error prone, and expensive, requiring a difficult and costly testing effort with no guarantee for a bug-free software product. Many techniques have been proposed and used in practice to check RT software; in particular, software testing has been the main methodology for verifying software components.[1] This method has limitations because, in order to guarantee software reliability, we need to apply exhaustive testing to the software component, using all possible input combinations, which is a costly process. Many techniques have been proposed to enable a practical alternative to exhaustive software testing.[2] However, we cannot guarantee a full coverage of all possible execution paths in software component, thus leaving us with limited confidence in our software correctness. Some approaches used modeling and simulation to predict timing behavior of soft RT systems on the multiprocessor platform, as introduced by Florescu et al.[3] and Castro et al.[4]

Formal software analysis use is growing as an alternative, as this technique allows full verification of software components to be free of errors. In recent decades, these techniques have matured to be used in some industrial capacity for software and hardware correctness verification.[5] New theoretical advances in model checking allow one to guarantee certain properties about models of such systems using a formal

[1]School of Computer Science, Carleton University, Ottawa, Canada
[2]Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada

Corresponding author:
Hesham Saadawi, School of Computer Science, 5302 Herzberg Building, 1125 Colonel By Drive, Ottawa, Ontario K1S 5B6, Canada.
Email: hsaadawi@connect.carleton.ca

approach. These techniques can be automated to improve the work of the software engineer. Timed automata (TA) theory,[6] in particular, has provided many practical results in this area. However, there is still a gap between a model that is checked as an abstract entity, and the actual code run on a target platform. Errors could still creep into the final implementation as the programmer translates requirements captured and modeled in TA into code. TA and other formal methods have showed promising results, but are still difficult to apply and have limited power when the complexity of the system under development scales up.

Instead, systems engineers have often relied on the use of modeling and simulation (M&S) in order to make system development tasks manageable. Construction of system models and their analysis through simulation reduces both end costs and risks, while enhancing system capabilities and improving the quality of the final products. M&S let users experiment with 'virtual' systems, allowing them to explore changes, and test dynamic conditions in a risk-free environment. This is a useful approach, moreover considering that testing under actual operating conditions may be impractical and in some cases impossible, see for example use of simulation by Härri et al.[7] and Staub et al.[8] Nevertheless, no practical, automatable approach exists to perform the transition that exists between the modeling and the development phases, and this often results in initial models being abandoned, resulting in increased initial costs that project managers usually try to avoid. Simultaneously, M&S frameworks are not as robust as their formal counterparts.

Here, we show a methodology that would have a higher correctness checking reliability of the actual code executing in the RT system. This is achieved by model checking and simulating models that are described using the Discrete Event System Specification (DEVS).[9] The same models would run on the target platform, using a model-based approach in which the user can move the original models to a target platform to execute them in RT without any changes. In order to guarantee the correctness of the model, the methodology verifies DEVS models with TA theory and tools. TA provides a solid theory and algorithms for model checking, and the many existing tools implementing these algorithms (for instance, UPPAAL, Kronos and others[10,11]). The verified DEVS models would then execute directly on a RT DEVS kernel, eliminating the risk of introducing errors in the final system implementation on the target platform.

The DEVS formalism is based on systems theory, while TA was proposed as a formal way to specify RT reactive systems, and is based on finite automata theory. The DEVS is the most general discrete event specification, and one can build complex models as a composite of different methods for the various components (cellular models, Petri Nets, Timed Finite State Machines, Modelica, Partial Differential Equation (PDEs). and other continuous components) that can be then translated into a DEVS representation. These models can be then simulated safely using DEVS abstract simulation algorithms. TA's main concern is to have a system abstract formal description that is verifiable by model checking, and is not focused on simulating discrete systems. In that way, TA did not consider constructs to build large modular systems out of smaller components as DEVS does with its hierarchical building of coupled models out of atomic models. Existing DEVS simulators provide many functions, which do not exist in UPPAAL, to simulate different systems; however, DEVS simulators lack formal verification capability and this was the motivation to this research.

The proposal we introduce differs from other existing approaches in that it defines a new class of DEVS, called the Rational Time-Advance DEVS (RTA-DEVS), which is close to the classic DEVS in semantics and expressive power. We then define a transformation to obtain a TA that is behaviorally equivalent to the RTA-DEVS. The advantage of doing so is that many classic DEVS models would satisfy the semantics of RTA-DEVS models. Thus, they could be simulated with any DEVS simulator. Likewise, they can be transformed to TA to validate desired properties formally. The RTA-DEVS is close to the general DEVS and adds expressiveness; however, it still restricts the elapsed time in a state used in the external transition function to be a non-negative rational number. When transforming DEVS models to RTA-DEVS models, we make some approximations and abstractions. To assess the accuracy of this approach we introduce a method to estimate if any errors were introduced during the transformation that may affect the verification step, or the validity of the resulting RTA-DEVS model.

2. Background

As discussed in Section 1, we are interested in modeling complex systems with the DEVS, and then to provide analytical validation of the system specification using formal methods. The DEVS is a formal M&S methodology originally defined in the 1970s as a discrete event specification modeling formalism. It is derived from systems theory, and it allows one to define hierarchical modular models that can be easily reused. A real system modeled with the DEVS is described as a composite of sub models, each of them being behavioral (atomic) or structural (coupled). Closure under coupling allows coupled models to be integrated to a model hierarchy.[9] A DEVS atomic model is formally described by

$$M = < X,\ S,\ Y,\ \delta_{\text{int}},\ \delta_{\text{ext}},\ \lambda,\ ta >$$

Each model is seen as having input (X) and output (Y) ports to communicate with other models. The input and output events determine the values to appear in those ports.

The input external events are received in input ports, and the specification of the external transition function (δ_{ext}) defines the behavior under such inputs. The internal transition function (δ_{int}) is activated after the lifetime of the present state has been consumed, which is defined by the time-advance (*ta*) function. Its goal is to produce an internal event, which leads to a state change. The desired results are spread through output ports by the output function (λ), which executes before the internal transition.

A DEVS coupled model is composed of several atomic or coupled sub models. They are formally defined as

$$CM = < X,\ Y,\ D,\ \{M_i\},\ \{I_i\},\ \{Z_{ij}\},\ \text{Select} >$$

The coupled model uses input (X) and output (Y) ports to receive external input events and to generate external output events to communicate with other models. Coupled models are defined as a set (indexed by the finite index D) of basic components. These components M_i (atomic or coupled, with $i \in D$) are interconnected. This interconnection is defined by the translation function (Z_{ij}), which is in charge of converting the outputs of a model into inputs for the others. To do so, an index of influences (I_i) is created for each model. This index defines that the outputs of the model M_i are connected to inputs in the model M_j, where j is an element of I_i. The **Select** function is used to tiebreak the execution of components under simultaneous events.

The CD++ tool[12,13,14] allows one to define models following DEVSs. The tool is built as a hierarchy of models, each of them related with a simulation entity. CD++ includes a graphical specification language to enhance interaction with stakeholders during system specification, while having the advantage of allowing the modeler to think about the problem in a more abstract way.[15] This language uses the DEVS graphs notation to define atomic models' behavior.[16] Each DEVS graph defines the state changes according to internal and external transition functions, and each is translated into an analytical definition. DEVS graphs can be formally defined as[15]

$$GGAD = < X_M,\ S,\ Y_M,\ \delta_{int},\ \delta_{ext},\ \lambda,\ D >$$

where X_M is the $\{(p, v) |\ p \in \text{IPorts},\ v \in X_p\}$ set of input ports, Y_M is the $\{(p, v) |\ p \in \text{OPorts},\ v \in Y_p\}$ set of output ports, S is the $B \times P(V)$ states of the model, B is the $\{b \mid b \in \text{Bubbles}\}$ set of model states, and V is the $\{(v, n) \mid v \in \text{Variables},\ n \in R_0\}$ intermediate state variables of the model and their values.

Here, δ_{int}, δ_{ext}, λ, and D have the same meaning as in traditional DEVS models. Each model is defined by a unique identifier, and it can include a graphical specification or C++ code. When we use the state-based notation, states are represented by bubbles, including an identifier and a state lifetime. When the lifetime is consumed, an internal transition function is executed.

Figure 1 shows a simple atomic model using this notation. The model includes three states: A, B, and C. Dotted lines represent internal transitions, while full lines define external transitions. This graphical notation has a textual representation associated, used for creating simulation models that execute in CD++. The internal transitions use the following syntax:

```
int: source destination [outport!value]* ( { (action;)* } )
```

Here, *source* and *destination* represent the initial and final states associated with the execution of the transition function. As the output function should also execute before the internal transition, an *output* value can be associated with the internal transition. One or more *actions* can be triggered during the execution of the transition (changing the values of state variables).

External transitions are defined as follows:

```
ext: source destination ( { (action;)* } )? EXPRESSION
```

In this case, when the *expression* is true (which includes inputs arriving from input ports), the model will change from state *source* to state *destination*, while also executing one or more actions.

There exist some other tools that support the creation and execution of DEVS models, such as PowerDEVS.[17] PowerDEVS has the ability to execute models on a RT operating system with synchronization of simulation time to a RT clock. With its ability to model continuous systems within DEVS by the Quantized System State (QSS) method, the RT execution of the DEVS allows simulation of physical systems in RT.

Although the DEVS has been used to build RT applications,[15,18,19] there is a need to formally validate and verify these models. The main issue is that traditional techniques rely on simulating the model and a subject matter expert goes through simulation results to validate the model. However, with the strong and precise formal nature of the DEVS, formal validation and verification of its models can be achieved with a high degree of accuracy. In addition, DEVS models are executable directly on varied RT embedded target platforms. Therefore, any formally validated DEVS model would be guaranteed to execute as predicted by the validation, as no human intervention comes between the checked model and the executable system. This advantage would serve not only the simulation community, but also the RT software community, as the DEVS can be used to model controllers that would be simulated, formally validated and then deployed on the target platform. Therefore, these verification and validation tasks are important and they must be carefully planned, in particular when developing RT systems where we need to ensure correctness and reliability of simulation results. However, one of the major challenges to the application of M&S is the lack of a formal validation methodology.[20]

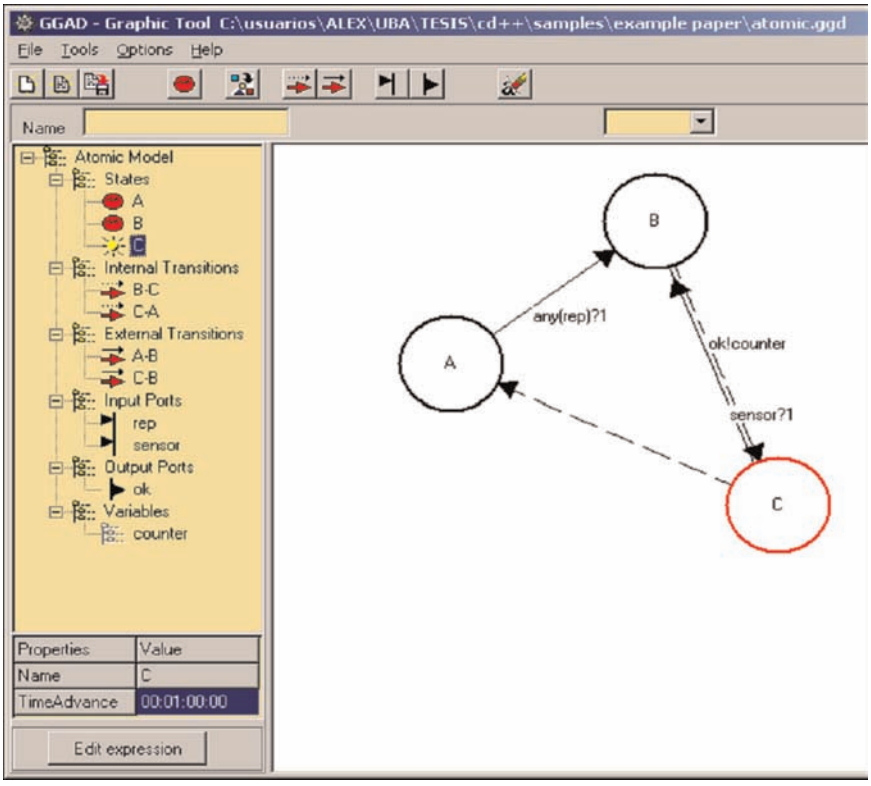

Figure 1. An atomic model defined as a DEVS graph

Varied techniques have been proposed for formal software analysis, and they can be categorized into three broad types,[5] namely Model Checking, Abstract Interpretation, and Deductive Methods. Further, approaches to use formal methods for software correctness vary. There are the Correctness-by-Construction techniques, in which the implementation is generated directly from a model in order to guarantee the final software implementation conformance to its requirements. This generation is done through a series of transformations that are proven formally to preserve the desired properties in the original model. Therefore, the final code generated does not need extensive work to apply formal analysis to prove its conformance to the original model, thus reducing time to market and enabling the average software engineer to produce formally correct software.[21,22,23]

One method of formal software verification is the model-checking approach, which verifies required properties on an abstract model of the software. After verifying the model, an implementation is obtained through manual program coding. In this case, a major concern is to verify the implementation correctness in regard to the verified model. The work presented by Voeten[24] tries to establish a *distance* between the verified model timing properties and the implementation physical time. A distance is defined as a difference between model time and physical time. Once established, RT properties of the implementation can be predicted from the model.

We propose a method based on formal software verification ideas, where we will use a combination of DEVS models with a model-checking approach using TA formal models. A timed automaton can be defined as:[11]

$$A = (N,\ l_o,\ E,\ I)$$

where N is a finite set of locations (or nodes), $l_o \in N$ is the initial location, $E \subseteq N \times \beta(C) \times \Sigma \times 2^C \times N$ is the set of edges, and I: $N \rightarrow \beta(C)$ assigns invariants to locations where $\beta(C)$ denotes a set of clock constraints.

Here, C is a set of clock variables (with x, y, etc. representing clock variables from the set C). We use a, b, etc. to represent actions from a set of finite alphabet Σ. Assume a finite set of real-valued variables C ranged over by clocks x, y, etc. and a finite alphabet Σ (with actions a, b, etc.). Let us call a *clock constraint* a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$ where x, y are clock variables, $\sim$ is one of $\{\leq, <, =, >, \geq\}$ symbols, and n is a natural number. Clock constraints can be used on transitions, where they are called a *guard*, or in a location (state), where they are called *invariant*. Invariants are constraints on the form $x \leq n$, or $x < n$ to restrict time spent in a TA location.

The semantics of a timed automaton are defined as a Timed Transition System where states are pairs $< L, u >$, where L is a location and u is a clock valuation. We write $l \xrightarrow{g,\,a,\,r} l'$ when $(l, g, a, r, l') \in E$ where g is a clock

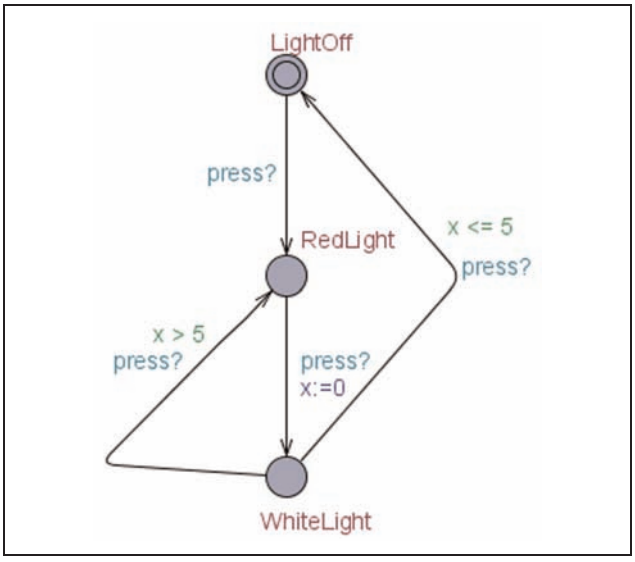

Figure 2. Timed automaton

constraint, a is an action, and r a set of clocks to be reset to zero. TA use two types of transitions:

- *Delay Transition*: $< L, u > \xrightarrow{d} < L, u + d >$: the time passage d causes a transition from the start location to an end location;
- *Action Transition*: $< L, u > \xrightarrow{a} < L', u' >$: an action a causes a transition from the start location to an end location.

An example of a timed automaton similar to the one given by Bengtsson and Yi[11] is shown in Figure 2. This model represents a light that can be either off, operating in red mode, or in white mode. The light changes mode depending on the user pressing the light switch button. From the red mode if the switch is pressed twice within 5 seconds, the light goes to white then to off; otherwise it goes to white then back to red. In order to do that, the TA uses three states *LightOff, RedLight,* and *WhiteLight*. *LightOff* is the initial state. The transition out of *WhiteLight* to *LightOff* has a guard $x \leq 5$ which enables the transition only while clock x value is less than 5 time units, whenever a synchronization signal arrives on channel *press*. States could also have clock constraints called *invariants*. In this case, time is allowed to pass in a state while the clock values satisfy the invariant. Once the invariant is not satisfied, the automaton would leave that state and enable a transition to another state with clock values that would satisfy its invariant.

There have been several proposals to verify DEVS models, including formal model checking of some restricted classes of the DEVS, generation of test traces from DEVS models for testing DEVS simulations, and specification of high-level system requirements in TA (and then verifying the DEVS model against those requirements). Others introduce clock constructs into the DEVS to conform to TA, and it has been shown that DEVS models can be transformed to semantically equivalent TA models, maintaining their original structure and behavior.[25]

Hong et al.[26] introduced RT-DEVS, a formalism with a time-advance function that maps each state to a range with maximum and minimum time values, including an algorithm to build a timed reachability tree to be used for safety analysis with RT-DEVS and a case study of a RT system of train-gate-controller to demonstrate the methodology. Further work on verifying the RT-DEVS is shown in Furfaro and Nigro,[27,28] where TA and UPPAAL[10] are used for verification purposes, and a transformation from RT-DEVS to UPPAAL is shown. This transformation allows weak synchronization between components of the TA model, as RT-DEVS semantics uses weak synchronization. The transformation given, however, did not show timed behavior equivalence formally between the RT-DEVS and the resulting TA models.

Another approach, presented by Hwang and Zeigler,[29] introduced a subclass of the DEVS (called Finite-Deterministic DEVS) in which the time-advance function maps states to rational numbers, and the external transition function cannot use the elapsed time in its domain to produce the next state. In this work, the authors also introduced verification through reachability analysis similar to TA algorithms and techniques.

Han and Huang[30] show a mapping from the DEVS models to TA. The conversion method mapped the DEVS model through its components using the DEVS simulator. The approach suggests trace equivalence as the basis for parallel DEVS and TA behavioral equivalence. In Giambiasi et al.,[31] high-level system specifications are done in TA and then modeled with the DEVS; system requirements are verified through the simulation of the DEVS model.

A result presented by Hernandez and Giambiasi[32] showed that verification of general DEVS models through reachability analysis is undecidable. The authors based their deduction on building a DEVS simulation Turing machine. Since in Turing machines the halting problem is undecidable (i.e. with analysis only, we cannot know in which state a Turing machine would be), they concluded that this is also true for DEVS models: we cannot know if we reach a particular state starting from an initial state, and hence reachability analysis for the general DEVS is impossible. They argue that reachability analysis maybe possible only for restricted classes of DEVSs with finite input/output sets and finite states set. The authors then introduced a new class of DEVSs called Time Constrained DEVS (TC-DEVS) that expanded the classic DEVS atomic model definition with the introduction of multiple clocks incremented independently of other clocks. The classic DEVS atomic models can be seen as having only one clock that keeps track of elapsed time in a state, and is reset on each

transition. The TC-DEVS also added clock constraints similar to TA (to function as guards on external and internal transitions). However, it allows clock constraints in states as state invariants that contain clock differences. The TC-DEVS is then transformed to an UPPAAL TA model. The paper, however, did not explain a transformation of TC-DEVS state invariants to UPPAAL TA when the model has invariants with clock differences, as UPPAAL TA has a restricted type of state invariance without clock differences.

There are other verification techniques proposed for verifying DEVS models.[33,34] The idea is to generate testing sequences from model specifications that are later applied against the model implementation to verify the conformance of implementation to specifications.

As discussed earlier, our proposal, to be discussed in detail in the following sections, differs from the above in that it defines a new class of DEVS, called RTA-DEVS, which is close to the classic DEVS in semantics and expressive power.[35,36] We then define a transformation to obtain a TA that is behaviorally equivalent to the RTA-DEVS. The advantage is that many classic DEVS models satisfy the semantics of RTA-DEVS models and they can be simulated with a DEVS simulator, while being able to be transformed to a TA in order to validate the desired properties formally. The RTA-DEVS follows the FD-DEVS in restricting the time-advance function to non-negative rational numbers, but it also relaxes the restriction of the FD-DEVS on external transition functions, making it closer to the general DEVS. However, the RTA-DEVS still restricts the elapsed time in a state used in the external transition function to be a non-negative rational number. This restriction translates to having non-negative rational constants in guards in the transformed TA model, and ensures termination of the reachability analysis algorithm implemented in UPPAAL,[6,10] as irrational constants in TA guards render reachability analysis undecidable.[37]

We chose UPPAAL as our verification engine, as this tool has many features and new algorithms that have been built into it over more than 15 years of development, many of which focused on enhancing the performance of the model checker. Any reachability analysis algorithm that may work on the RTA-DEVS or other DEVS subclasses would have the same basic timed model-checking algorithm implemented in UPPAAL. Thus, no gain in performance over UPPAAL would result from performing reachability analysis directly on the DEVS. However, UPPAAL uses other optimizations, such as special data structures to store state information and state reduction techniques, that would need to be implemented into any specific DEVS tool to be as efficient as UPPAAL. By translating DEVS models to TA and using UPPAAL, we build a methodology on a reliable tool and a large and stable research community. Moreover, UPPAAL provides the modeler with many optimization techniques that can be exploited with the DEVS to scale the size of DEVS models that can be verified. We believe this would give the best return on effort for DEVS verification and it is still an open area of research.

In the following sections, we build on the work we introduced earlier in Saadawi and Wainer,[36] and we expand it by introducing a methodology to approximate any DEVS model to the RTA-DEVS. We also provide a methodology to estimate the effect on verification results that may exist as a consequence of approximating a DEVS model to an equivalent RTA-DEVS model.

3. Rational Time-Advance Discrete Event System Specification

As in the classical DEVS, we need to define the RTA-DEVS atomic model. The RTA-DEVS changes the definitions of the time-advance function *ta* and the external transition function δ_{ext}. The Atomic Rational Time-Advance is defined as follows:[35,36]

$$AM_{TC} = < X,\ Y,\ S,\ s_0,\ \delta_{int},\ \delta_{ext},\ \lambda,\ ta >$$

where X is the set of external input events, Y is the set of external output events, S is the set of system states, s_0 is the system initial state, - δ_{int}: $S \rightarrow S$ is the internal transition function (the same as in the classic DEVS), δ_{ext}: $TxX \rightarrow S$ with $T = \{(s, e)/s\ 0 \leq e \leq ta(s), e \in Q_{0,+\infty}\}$ is the external transition function (e is the time elapsed since the last transition, which takes a positive rational value), $\lambda\ S \rightarrow Y \cup \varnothing$ is the output function, and *ta:* $S \rightarrow Q_{0,+\infty}$ is the time-advance function that maps each state to a positive rational number.

Coupled RTA-DEVS models are defined exactly as in the classic DEVS Coupled Model *CM* introduced in Section 2. Similarly to DEVS, a coupled RTA-DEVS model *CM* has an equivalent atomic RTA-DEVS model. This is showed by the closure under coupling property described in Appendix A.

3.1 Equivalency of RTA-DEVS and TA

When transforming DEVS models to RTA-DEVS models, we have to make some approximations and abstractions. To assess the accuracy of this transformation, we here introduce a method to estimate if there were any errors introduced during the transformation that could affect the verification step or the validity of the resulting RTA-DEVS model. In order to verify RTA-DEVS models by applying the advanced theoretical results and the multiple tools available for TA, we need to construct TA models from RTA-DEVS models that are behaviorally equivalent.

Generally there are two methods to check if two of the Timed Labeled Transition Systems (TLTSs) are behaviorally equivalent, namely *Trace Equivalence* and *Bisimulation*. In Aceto et al.,[38] it was shown that, for RT systems, trace equivalence is not enough to show the

complete equivalence of two TLTSs. Although one can show the trace equivalency of two TLTSs (based on their acceptance or the generation of event traces), RT systems usually have multiple concurrent components working together. Those components may go into a deadlock state in which no external event is observable. Due to these subtle errors, bisimulation is a better notion for behavior equivalence. Both analyses of RTA-DEVS and TA are based on bisimulation.

The bisimulation between two TLTSs (e.g. systems A and B) establishes the relation between every state in system A and a corresponding one in system B. It also relates every observable transition in A to a corresponding observable transition in B. In Aceto et al.,[38] the concepts of *strong* and *weak-timed bisimilarity* were defined for the behavioral equivalence of systems. The transformation method presented in this section is based on weak-timed bisimilarity equivalence, because its conditions are more general to apply than the conditions for strong bisimulation.

We first define the behavior equivalency based on weak-timed bisimulation. Then, following the conditions of this bisimulation, we construct a TA model for the basic behavior elements of RTA-DEVS (namely, the internal and external transitions). Then, we deduce the required constant values on the TA model to complete the bisimulation equivalence.

3.1.1 Definition: eventual transition relation '⇒'

In a TLTS with states s and t, the *eventual transition relation* defines a transition from state s to state t that may contain one or more direct transitions, labeled with events that are non-observable to the outside world. If we have an observable action a, a non-observable action τ, and a transition label α, then, for any TLTS, the *eventual transition relation* $\Rightarrow$ between s and t on action α (written $s \overset{\alpha}{\Rightarrow} t$) is defined if any of the following conditions is true.

1. $s \overset{\tau}{\Rightarrow} t$: there is a transition from s to t only composed of transitions labeled with non-observable actions. For example, for the non-observable action $\alpha = \tau$, there is a transition $s(\overset{\tau}{\longrightarrow})^* t$ (* defines one or more occurrences of these transitions).
2. $s \overset{a}{\Rightarrow} t$: there is a transition from s to t composed of one transition labeled with an observable action $\alpha = a$, and one or more eventual transitions labeled with non-observable actions. For example, $s \overset{\tau}{\Rightarrow} s_1 \overset{a}{\longrightarrow} s_2 \overset{\tau}{\Rightarrow} t$ for some states s_1 and s_2.
3. $s \overset{d}{\Rightarrow} t$: there is an eventual transition relation from s to t with total delay d (called *eventual delay transition*), which is composed of one or more direct delay transitions combined with some non-observable action transitions. This represents a sequence of transitions with no observable actions whose total delay amounts to d. For example, for action $\alpha = d \in \Re_{\geq 0}$ and $s \overset{\tau}{\Rightarrow} s_1 \overset{d_1}{\longrightarrow} t_1 \ldots\ldots t_{n-1} \overset{\tau}{\Rightarrow} s_n \overset{d_n}{\longrightarrow} t_n \overset{\tau}{\Rightarrow} t$ (with $n \geq 0$), for some intermediate states $s_1 \ldots s_n$, $t_1 \ldots t_n$ and delays $d_1 \ldots d_n$ with $d = \sum_{i=1}^{n} d_i$ (d is the total delay for the eventual transition from s to t; by convention, $d = 0$ when $n = 0$).

3.1.2 Definition: weak-timed bisimulation

The *weak-timed bisimulation* is a binary relation $\boldsymbol{R}$ over a set of states of a TLTS. For example, if we have states s_1, s'_1, s_2, *and* s'_2, then $\boldsymbol{R}$ is a weak-timed bisimulation $s_1 \boldsymbol{R} s_2$ (Aceto et al.[38]) if:

- $s_1 \overset{d}{\longrightarrow} s'_1$, then there is a transition $s_2 \overset{d}{\Rightarrow} s'_2$ such that $s'_1 \boldsymbol{R} s'_2$, as shown in Figure 3;
- $s_1 \overset{a}{\longrightarrow} s'_1$, then there is a transition $s_2 \overset{a}{\Rightarrow} s'_2$ such that $s'_1 \boldsymbol{R} s'_2$, as shown in Figure 4;
- $s_2 \overset{d}{\longrightarrow} s'_2$, then there is a transition $s_1 \overset{d}{\Rightarrow} s'_1$ such that $s'_1 \boldsymbol{R} s'_2$, as shown in Figure 5;
- $s_2 \overset{a}{\longrightarrow} s'_2$, then there is a transition $s_1 \overset{a}{\Rightarrow} s'_1$ such that $s'_1 \boldsymbol{R} s'_2$, as shown in Figure 6.

3.4 RTA-DEVS internal transition semantics

TA expresses the notion of time through clock variables and constraints on them, as shown in the previous section. Here, we present the RTA-DEVS behavior in terms of its transition functions, and we discuss how to obtain a behaviorally equivalent TA according

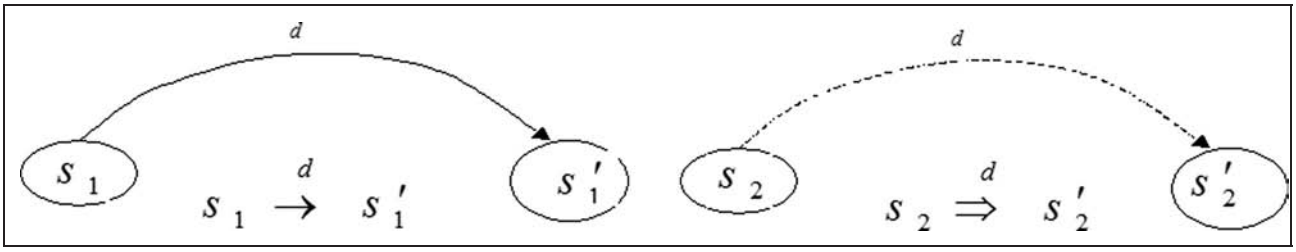

Figure 3. Direct delay transition from s_1 to s'_1 and corresponding eventual delay transition from s_2 to s'_2

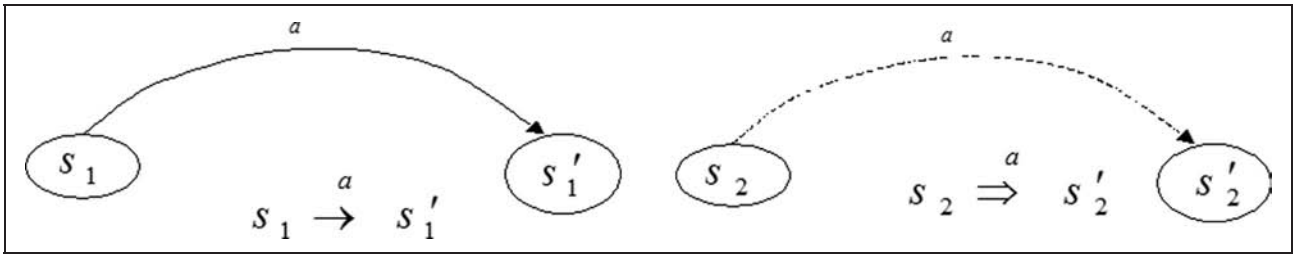

Figure 4. Direct action transition from s_1 to s'_1 and corresponding eventual action transition from s_2 to s'_2

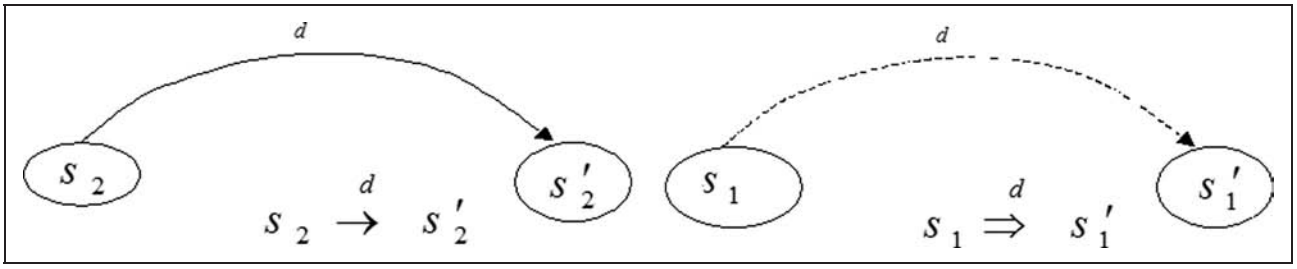

Figure 5. Direct delay transition from s_2 to s'_2 and corresponding eventual delay transition from s_1 to s'_1

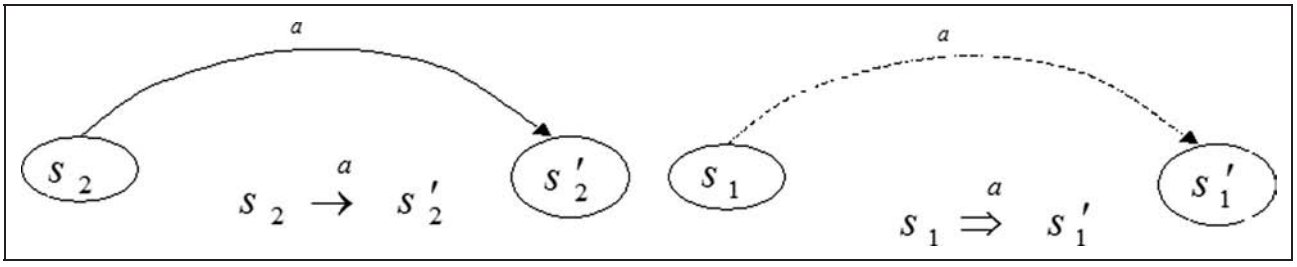

Figure 6. Direct action transition from s_2 to s'_2 and corresponding eventual action transition from s_1 to s'_1

to the previous definition. In doing so, we determine the value of the constant in TA invariants and guards, in order to keep the same behavior of the RTA-DEVS. We will use the same notation of TLTS semantics as defined in the paper for both RTA-DEVS and TA transitions to easily relate RTA-DEVS and TA when we talk about their semantically equivalent transitions.

The RTA-DEVS internal transition semantics is shown as a DEVS graph in Figure 7, and is defined as

1. $\delta_{\text{int}}(s_1, e) = s_3, \quad \text{if } e = ta(s_1) = T$

In RTA-DEVS semantics, this transition means that we move to state s_3 when the elapsed time e in s_1 equals the time-advance value of s_1. In a TLTS, this can be defined as a time-elapsed transition with delay d with the form

$$s_1 \xrightarrow{d} s_3 \text{ if } 0 \le e < ta(s_1) \text{ and } d = ta(s_1) - e$$

which means that if we start at s_1 with time spent e, we need to delay d time units before changing to state s_3:

2. $\delta_{\text{int}}(s_1, e) = s_1 \quad \text{if} \quad 0 \le e < ta(s_1)$

From the RTA-DEVS semantics, this transition means that we stay in the same state s_1 as long as the elapsed time e in that state does not equal or exceed the time-advance value for s_1. This can be defined as a time elapse transition of the form

$$s_1 \xrightarrow{d} s_1 \text{ if } 0 \le e < ta(s_1) \text{ and } 0 \le d < ta(s_1) - e$$

which means that if we start at s_1 with time spent e, as long as time delay d is constrained as above, we stay at s_1.

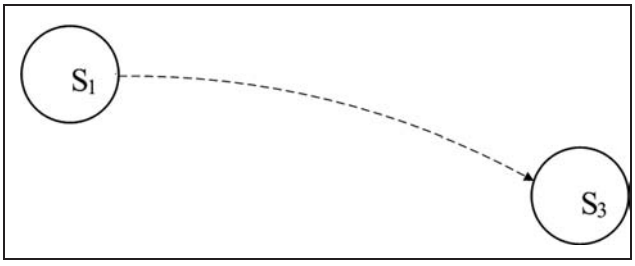

Figure 7. RTA-DEVS internal transition

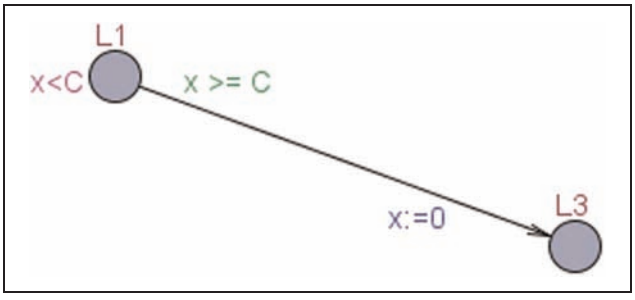

Figure 8. TA model for an internal RTA-DEVS transition

From now on, we will use the operational semantics of UPPAAL for TA, as defined by Behrmann et al.[10] For the graphs in the rest of the paper, we will name RTA-DEVS states as s_i, and the corresponding TA locations as L_i (with i an integer number).

The delay transition for the TA in Figure 8 is defined as

$$(L_1, clock = x) \xrightarrow{d} (L_1, clock = x + d) \text{ for any } d \geq 0$$

This defines the RTA-DEVS delay transition above, in which we stay in the same state with a total delay d less then time advance of s_1. We will show that the TA in Figure 8 is behaviorally equivalent to the DEVS graph model shown in Figure 7 through a timed bisimulation relation. To do this, we will show that the state s_1 is bisimilar to location L_1, and state s_3 is bisimilar to location L_3. This is done showing a weak-timed bisimulation relation (from RTA-DEVS to TA, and from TA to RTA-DEVS). We do this in steps 1 and 2 below.

In this model, two locations are defined (L_1 and L_3), along with a transition from L_1 to L_3. L_1 has an invariant on clock x ($x < C$) that allows the TA to stay in that location as long as the invariant is *true*. The transition from L_1 to L_3 has a guard ($x \geq C$) that must be true for the transition to be enabled, and C is a rational number. The transition also has an update rule for clock variable x to reset it to zero before entering location L_3. We apply the condition above to weak-timed bisimulation, first from RTA-DEVS to TA and then from TA to RTA-DEVS. In doing so, we determine a value for the constant C to preserve the bisimulation relation.

3.4.1 Step 1: from RTA-DEVS to TA

For the bisimulation of the states shown in Figures 7 and 8, we have the following requirements.

- S_1 ***R*** L_1: this is a delay transition from s_1 to itself. If $s_1 \xrightarrow{d} s_1$ for some value of d where $0 \leq e < ta(s_1)$ and $0 \leq d < ta(s_1) - e$, then to satisfy the bisimulation relation we should have $(L_1, x = e) \xrightarrow{d} (L_1, x = e + d)$ for the same value of d. As we have from the invariant of state L_1, $x < C$ then by substituting for x and d, we get

$$ta(s_1) \leq C \qquad \text{(Rule 1)}$$

s_3 ***R*** L_3: bisimilarity between s_3 and L_3: for the delay transition from s_1 to s_3, let us consider the execution of the DEVS internal transition $(s_1, e) \xrightarrow{d} (s_3, 0)$, where $0 \leq e < ta(s_1)$ and $d = ta(s_1) - e$ by the TA transition $(L_1, x = e) \xrightarrow{d} (L_3, x = e + d)$, with a reset statement on the transition x: = 0.

In order for these two delay transitions to be behaviorally equivalent, they need to start from bisimilar states, and after the same delay, they reach two bisimilar states. To achieve this, we use the same value of delay d on both transitions, and we deduce the constant C in the TA clock constraint to give us that condition above for bisimulation. The TA transition above starts from location L_1, with the clock x equal to the same value of elapsed time e at the RTA-DEVS transition above; then, after the same delay d of the RTA-DEVS transition, it transitions to L_3, and the value of clock x increases by d.

We apply the conditions for this delay transition on TA transition above. With the TA guard on the transition out of location L_1: $(x \geq C)$ with the value of clock x: $(x = e + d)$, we get

$$ta(s_1) \geq C \qquad \text{(Rule 2)}$$

This rule means that as long we use a constant C in the guard of the TA transition with a value greater or equal to the time-advance value of s_1, the previous TA transition executes the RTA-DEVS transition above.

The previous two rules give the condition $ta(s_1) = C$ for the TA shown in Figure 8 to execute the DEVS graph as in Figure 7.

This condition guarantees the weak-timed simulation relation from the RTA-DEVS model of an internal transition to a delay transition of the TA. We show the condition of the weak-timed simulation relation from TA to RTA-DEVS in step 2 below.

3.4.2 Step 2: from TA to RTA-DEVS

To satisfy the other direction of the bisimulation relation, we convert the TA in Figure 8 with the RTA-DEVS in Figure 7.

Case 1: TA delay transition $(L_1, x = e) \xrightarrow{d} (L_1, x = e + d)$.

Here, we need the value of clock x to be less than C in order for the L_1 invariant to be true and TA to stay in L_1, that is

$$e + d < C \qquad \text{(Rule 3)}$$

For the RTA-DEVS time delay transition $(s_1, e) \xrightarrow{d} (s_1, e + d)$ to stay in s_1 after d, we need the sum of the elapsed time and delay to be less than the lifetime of s_1:

$$e + d < ta(s_1) \qquad \text{(Rule 4)}$$

Case 2: TA transition $(L_1, x = e) \xrightarrow{d} (L_3, x = 0)$.

We start from location L_1 with a clock x equal to some elapsed time e in L_1. After d, we change to L_3 and clock x is reset. To exit from L_1, the invariant would be false and the guard on the TA transition would need to be true, which gives

$$e + d = C \qquad \text{(Rule 5)}$$

This transition is defined in the RTA-DEVS as $(s_1, e) \xrightarrow{d} (s_3, 0)$, in which we need elapsed time e in s_1 plus a delay d equal to the time advance of s_1 to trigger the internal transition:

$$e + d = ta(s_1) \qquad \text{(Rule 6)}$$

From Rules 3 and 4, we determine $C = ta(s_1)$. With this value, we have a timed simulation relation from TA (shown in Figure 8) to RTA-DEVS (shown in Figure 7). By having a simulation relation in both directions, the RTA-DEVS internal transition shown above is timed bisimilar and behaviorally equivalent to the TA timed transitions shown above if we have the constant C equal to the lifetime of corresponding state in the RTA-DEVS model. This concludes that $s_1 \; \boldsymbol{R} \; L_1$ and $s_3 \; \boldsymbol{R} \; L_3$ by the bisimulation relation $\boldsymbol{R}$.

When we use the previous method to map internal transitions from the RTA-DEVS model to transitions at a TA model and vice versa, we guarantee the resulting transitions to be behaviorally equivalent. We will show the same for RTA-DEVS external transitions in the following section.

3.7 RTA-DEVS external transitions

The RTA-DEVS external transition function is defined as $\delta_{\text{ext}} : V_D \times X \to S$, where $V_D = \{(s, e) : s \in S, 0 \le e \le ta(s)\}$.

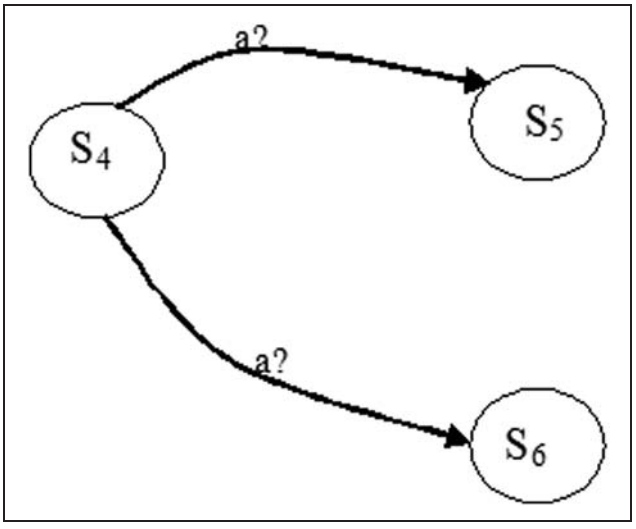

Figure 9. RTA-DEVS external transitions on action *a*

Figure 9 represents the following definitions for the RTA-DEVS external transition function:

$$\delta_{\text{ext}}(s_4, a, e) = s_5 \quad \text{for } 0 < e < 3$$

$$\delta_{\text{ext}}(s_4, a, e) = s_6 \quad \text{for } 3 \le e < ta(s_4)$$

Each of these transitions can be expressed as a time passage and action transitions:

1. $s_4 \xrightarrow[3 \le d < ta(s_4)]{d < 3} s_4$ and $s_4 \xrightarrow{a} s_5$;
2. $s_4 \xrightarrow{} s_4$ and $s_4 \xrightarrow{a} s_6$.

From these expressions, we can represent the external transitions as the TA transitions shown in Figure 10.

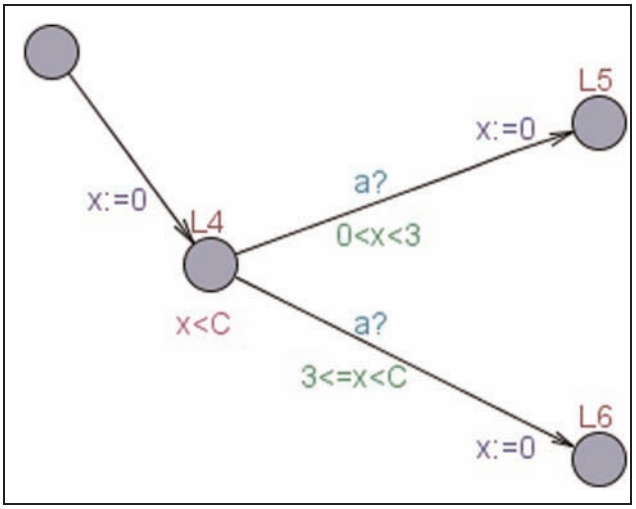

Figure 10. TA model for RTA-DEVS external transition

From TA semantics, each of these transitions is expressed as a time and action transitions as follows.

1. $(L_4, x=0) \xrightarrow{d<3} (L_4, x=3)$ and $(L_4, x=3) \xrightarrow{a} (L_5, x=0)$ for the first RTA-DEVS transition. That is, we stay in L_4 while the elapsed time is less than 3 units, and then with action a, the TA takes a transition to L_5.
2. $(L_4, x=0) \xrightarrow{3 \le d < ta(s_4)} (L_4, x=3)$ and $(L_4, x) \xrightarrow{a} (L_6, x=0)$ for the second RTA-DEVS transition. That is, if the elapsed time in L_4 exceeds 3 units and is less than lifetime of L_4, with action a, the TA transitions to L_6.

This gives us the relation $\boldsymbol{R}$ between RTA-DEVS and TA models, which states $s_4 \; \boldsymbol{R} \; L_4$, $s_5 \; \boldsymbol{R} \; L_5$, and $s_6 \; \boldsymbol{R} \; L_6$.

Conversely, we can show the simulation in the other direction from each of the TA transitions and the RTA-DEVS external transitions above. Hence, this shows a bisimulation relation $\boldsymbol{R}$ between the corresponding DEVS and TA models above.

3.8 Observations: RTA-DEVS to TA transformation.

The transformation examples in the previous section introduce the methodology to transform RTA-DEVS models to TA models. The resulting TA models are a subset of deterministic safety automata used in the UPPAAL model checker. The complete transformation methodology can be summarized as follows.

1. Define a clock variable for each atomic RTA-DEVS model, for example x.
2. Replace every state in RTA-DEVS with a corresponding one in TA, that is, L_1 for source s_1 and L_2 for destination s_2.
3. An RTA-DEVS internal transition $\delta_{\text{int}}(s_1, e) = s_2$ is modeled in TA as follows.
   - A source state L_1 and a destination state L_2.
   - Reset the clock variable on the entry to each location (x: = 0).
   - Put an invariant in the source state derived from the time-advance function for that state as shown above, that is, $x < ta(s_1)$.
   - If the value of the lifetime of state s_1 is zero, then define the corresponding location in TA (L_1) to be a committed location. Then, skip the next step.
   - Define a transition with a guard. This guard should be the complement to the invariant in the source state, as shown in the example transformation above, that is, $x \ge ta(s_1)$.
4. The RTA-DEVS external transition is modeled in TA with the following items:
   - a source state and some destination state(s), that is, L_1 for source s_1 and L_2 for destination s_2;
   - a clock reset on the entry to each location;
   - an invariant in the source state that corresponds to time-advance function for that state, that is, $x < ta(s_1)$;
   - for the external transition(s) with guards of clock constraints, these constraints should be disjointed to obtain a deterministic TA model;
   - an action label on TA transitions corresponds to each RTA-DEVS input event accepted at a source state s_1.

By applying the previous steps, we obtain a TA model that executes every transition defined in the RTA-DEVS model under study. As we know, the RTA-DEVS behavior is completely defined by its transition functions, which defines all the transitions in the RTA-DEVS model. Thus, the resulting TA model executes the RTA-DEVS. This gives us a TA model that is behaviorally equivalent to the RTA-DEVS model, and that can be formally verified with tools, such as UPPAAL, to infer properties about the original RTA-DEVS model. In the following section, we look into a full example of applying this methodology, and we will show how, in case of explosion of states, a DEVS simulator can be used to co-test the model using simulation in parallel with the formal verification steps.

4. A transformation example

From the previous section, we can see that any RTA-DEVS model can be transformed to a behaviorally equivalent TA if we follow the steps shown in Section 3. To show this process in further detail, we introduce an example of an elevator system introduced by Saadawi and Wainer.[35] The system is composed of an elevator and its controller, which interacts with the user to receive button requests from each floor. Then, it makes the elevator to respond to user requests. This is an abstract model of the movement of the elevator. Other details have been ignored, such as door operation, floor display, etc., as we are only interested in controlling the elevator movement with our controller. This is an example of a (soft) RT system with safety and bounded response time requirements. To check for these requirements, we applied the previous transformation rules to the DEVS graph models of the elevator system, and used UPPAAL to check the validity of timing requirements.

The elevator DEVS graph model in Figure 11 includes five external transitions (shown with solid arrows), and three internal transitions (shown with dotted arrows). An external transition is enabled whenever the expression on that transition evaluates to *true* in the RTA-DEVS model. The expression *Value(mover)* returns the value in the

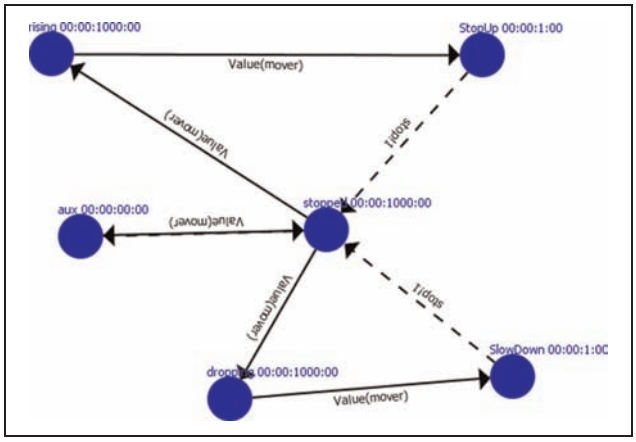

Figure 11. Elevator RTA-DEVS model represented in DEVS graph notation

variable *mover* and compares it to the constant value specified in the external transition function. If this comparison evaluates to *true*, the elevator model takes that external transition. The value of the variable mover is passed from the controller to instruct the elevator to move or stop. The complete textual specification for this graphical model can be found in Figure 12. For instance, we check the external transitions rising→StopUp, stopped→aux, GoingDown→SlowingDown and evaluate to true for the value of 1 at transitions Slowing Down→stopped, StopUp→stopped, stopped→GoingDown and for the value of 2 at transitions stopped→rising.

By taking each transition from the RTA-DEVS model and applying the previous steps defined in Section 3.4, we convert this DEVS Graph into the TA model shown in Figure 13. In our case, the expressions just discussed are translated to a channel reception *move*?, and a variable *direction* that takes values of 0, 1, or 2, as shown in Figure 13. The guard condition on the value of the variable *direction* is equivalent to the condition of the external transition in the RTA-DEVS model. In this case, whenever a value is transmitted on that channel, the transition synchronized on that channel is enabled. In order to model DEVS states with zero lifetimes (i.e. the cases where this state is reached, the output function is executed immediately, and then an internal transition is executed departing from that state), we used *committed* states as defined in UPPAAL's TA. Time does not pass in a committed state and, once we reach it in the TA model, a transition out of that state is enabled immediately. An example of a committed state is the *Aux* state in Figure 13. Likewise, note that the time-advance values for each state in the RTA-DEVS model have been substituted with an equivalent clock invariant in the corresponding state in the TA, and the constant in that invariant equals the state lifetime as indicated on the RTA-DEVS model. By having bisimilar states in the TA model to those of the RTA-DEVS model, we obtain a TA (in Figure 13) that is behaviorally equivalent to an RTA-DEVS model (in Figure 11).

The elevator *Controller* is responsible for interacting with the user and sending commands to the elevator to satisfy the user requests. The RTA-DEVS model for the controller is shown in Figure 14 using DEVS graphs. In this model, we abstract the behavior of the controller to being in one of six possible states, which represent the elevator

```
[controller]
in: button stop sensor
out: move
var: floor cur_floor direction
state: stopping stdbyStop moving Stopped stdbyMov aux1
initial : stdbyStop
int: Stopped stopping move!0
ext: stopping stdbyStop Value(stop)?1
ext: stdbyStop moving Equal(button,cur_floor)?0 {floor =
      button; direction = compare(cur_floor,floor,2,0,1);}
int: moving stdbyMov move!direction
int: aux1 stdbyMov
ext: stdbyMov aux1 Equal(sensor,floor)?0 {cur_floor=sensor;}
ext: stdbyMov Stopped Equal(sensor,floor)?1 {cur_floor = sensor;}
stopping: 00:00:00:00
stdbyStop: 00:00:1000:00
moving: 00:00:00:00
Stopped: 00:00:00:00
stdbyMov: 00:00:1000:00
aux1: 00:00:00:00
floor: 0
cur_floor: 0
direction: 0
```

Figure 12. Controller CD++ model

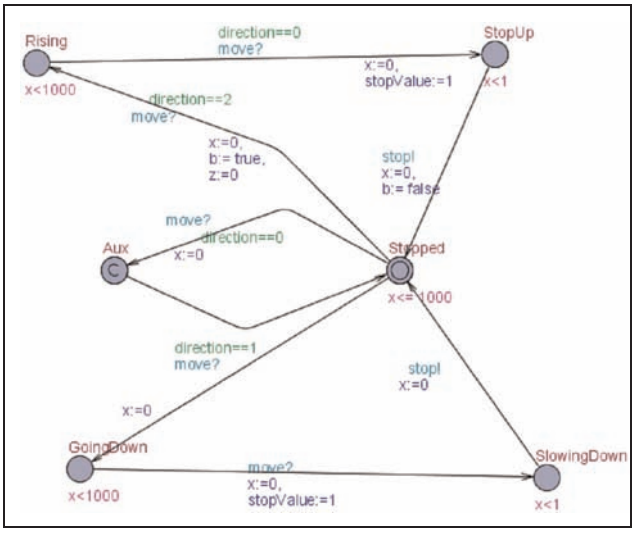

Figure 13. Elevator TA model

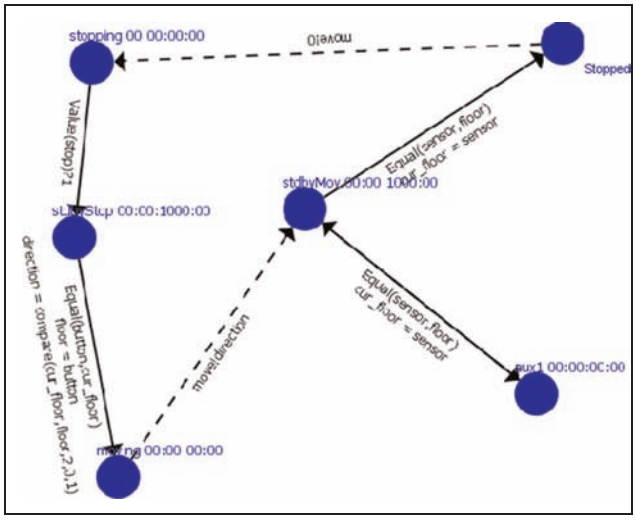

Figure 14. Elevator controller model as DEVS graphs

position, its movement direction, and its acceleration. The state *StdByStop* represents the elevator in a complete stop and ready to move. *Moving* is used when the controller makes a decision to move the elevator based on current floor and the button pressed floor. *StdByMov* corresponds to the elevator moving to the desired floor, and the controller in that state uses sensor signals to decide when to stop the elevator. The state *aux* serves as a 'dummy' state, and its internal transition is executed immediately after reaching that state; its purpose is to enable the test of the sensor value on the external transition to it with the function *equal(sensor, floor)*. The state *Stopped* corresponds to the controller deciding to send a signal to the elevator to slow in preparation to stop, and *Stopping* corresponds to the controller waiting for the elevator to get into complete stop and send a stop signal to the controller.

The idea is that the controller starts in the *StdByStop* state, waiting for a button request. Whenever it receives the *button* request, it triggers an external transition, comparing the button floor with the current floor (*cur_floor*) where the elevator is located. Based on this comparison, the controller determines the direction in which the elevator should move, and stores this info into the *direction* variable. The controller then changes to the *Moving* state, which uses a lifetime of zero time units, making an output function be executed immediately in order to send the direction information through the port *move* to the elevator model. Likewise, an instantaneous internal transition is triggered to change the state into the *StdByMov* state. The

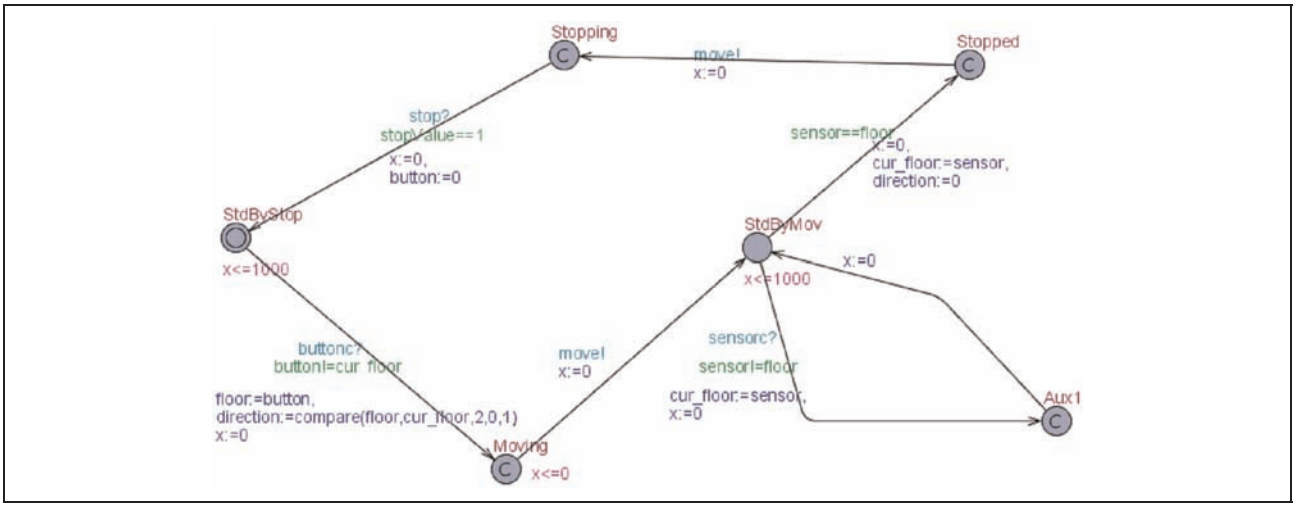

Figure 15. TA controller model in UPPAAL

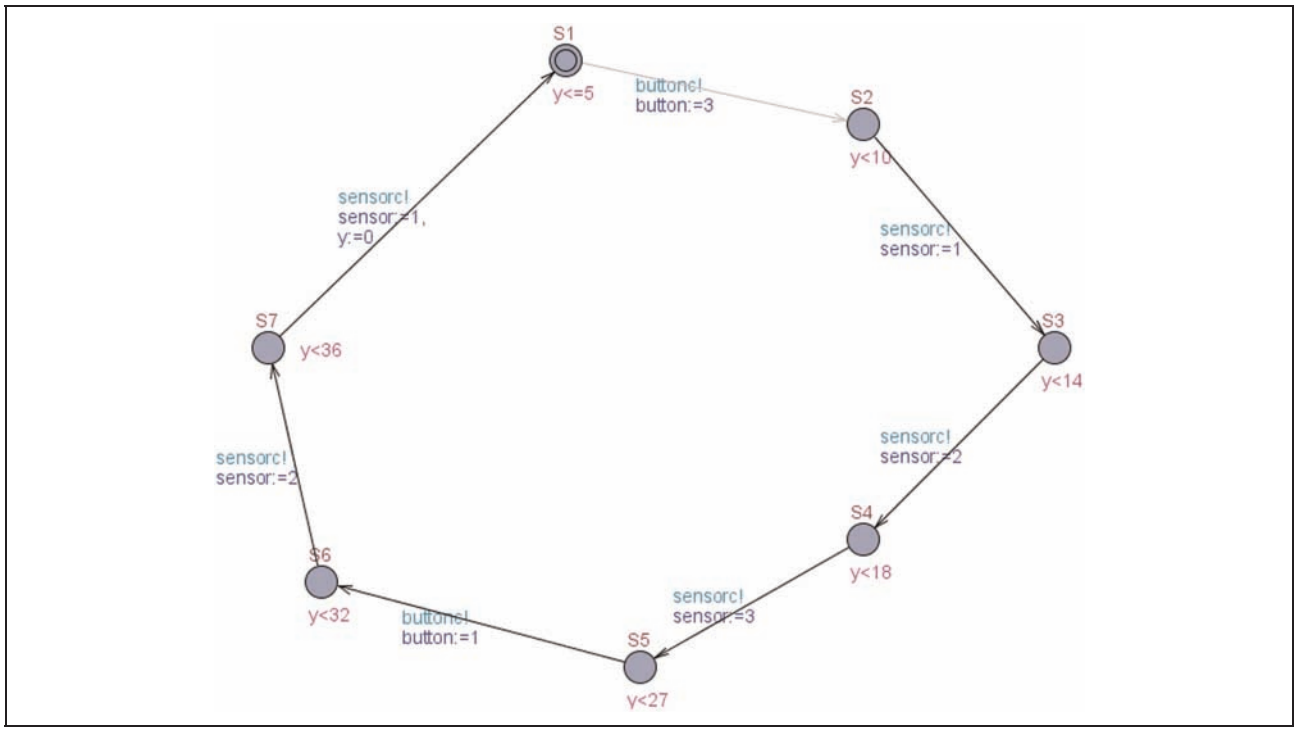

Figure 16. Environment inputs (button and sensor)

controller then decides to change to the *stopped* state if the sensor reading matches the desired floor; otherwise, it loops between the *aux* state and *StdByMov* states, as shown in the figure.

We applied the transformation steps defined in Section 3.4, and we obtained the TA shown in Figure 15. The RTA-DEVS Elevator coupled model is composed of the elevator atomic model and the Controller model. The coupled Elevator model takes as an input a number between 1 and 3 that represents the number of the floor with the button pressed. In order to model inputs to the system, we constructed a new automaton that sends the button and sensor inputs to the controller, as shown in Figure 16. This automaton is necessary to make the TA system under study a closed model, because, in order for model-checking techniques to be able to verify desired properties, they must work on closed systems. The reason for this is that a model checker explores all possible system transitions to determine if the desired property is met or not. Therefore, we need to model the system environment completely, in order to check all the possible system behaviors for all expected environment inputs. In this example, the environment modeled in Figure 16 is responsible for sending button and sensor events to the controller. It starts at S1 state and, after staying in this state for 5 time units, it sets variable button to 3, then synchronizes with the controller TA on channel buttonc. Again, it waits in state S2 until its clock y reaches 10 time units, sets the sensor to 1, and synchronizes with the controller on the channel sensorc. This process continues for the desired input sequences to the controller, and then resets the clock and restarts again at S1. Different environment TAs can be built this way in order to check the system thoroughly.

In Saadawi and Wainer[35] we showed how to verify a number of desired properties for the RTA-DEVS model based on the translation into TA. For instance, UPPAAL (or other model checkers for TA models) can be applied to the transformed model to study deadlock freedom, bounded response time, and safety properties for the original Elevator coupled model. The model checkers are usually based on Computational Tree Logic (CTL), which is used to construct queries with the requirements and submit them to UPPAAL to get an answer and hence verify that requirement. After translating our DEVS model to an equivalent TA model, we can use model checking to answer questions about our original DEVS model behavior (that otherwise needs to be simulated for all possible executions in order to obtain all possible answers). Some of the important questions to address would include the following.

- Does the DEVS model execution stop at one point without being able to progress (i.e. having a deadlock)?
- If no deadlocks are found in the DEVS model, is it always guaranteed that, whenever a user pushes a

floor button, the elevator would eventually reach that floor (normal operation as desired for the elevator system)?

- In case the elevator reaches the requested floor, is there an upper bound for the time between the request and the arrival of the elevator that our model would always guarantee to happen?

In order to answer these questions, we here show how to formulate these questions into formal queries to the TA model, and how UPPAAL responds when applied to the TA models translated from the DEVS graphs we recently discussed.

We started by addressing the first question above, and we applied the UPPAAL verifier to our model, checking for any deadlocks that maybe present in the elevator model To check for that failure, we formulated a simple query to the UPPAAL model. This is expressed in UPPAAL's CTL language as

```
A[] not deadlock
```

which means that, for all paths, there should be no deadlocks. After running the checker, it shows that this property is satisfied, that is, there is no deadlock in the DEVS model:

```
UPPAAL version 4.0.6 (rev. 2986), March 2007 - server.
A[] not deadlock
Property is satisfied.
```

More complex queries could also be formulated to check for more properties, verifying that properties using UPPAAL in a similar fashion. UPPAAL also gives a trace to help the designer get an insight into the system working details. A trace is shown in Figure 17. In this trace, the system starts at initial states for all three components, and then progresses. The composed system state is shown as `(Stopped,StdByStop,S1);` transitions with synchronization are shown as `buttonc: User_sensor_input -> ElevatorController.` That means that `User_sensor_input` synchronizes on the `buttonc` channel with the `ElevatorController`. The new state resulting from this transition is shown below the transition.

This trace result shows the composed state of the model, that is, the *Elevator*, *Controller*, and *User_sensor_input* composed state. The composed state is represented by the tuple (*elevatorState,Controller State,User_Sensor_inputState*). Therefore, in order to compare this UPPAAL trace and a DEVS simulation, we can compare the component trace with its corresponding simulation output, as DEVS simulation output is stored for each component individually. For example, the elevator UPPAAL trace results are in the left-hand side of the tuple. By extracting the elevator trace (`Stopped, Rising, StopUp, Stopped, GoingDown, SlowingDown, Stopped`), we can find it matches the same corresponding states in the simulation results. (A simulation of the same model will be discussed in the following section; the corresponding results can be seen in Figure 18. In that case, the UPPAAL states above correspond to the states `stopped, rising, StopUp, stopped, dropping, SlowDown, and stopped` in the simulation.) As we can see in this trace (and in the corresponding simulated version in Figure 18), the elevator starts at its initial state `Stopped`, and then rises to reach third floor when the third floor button is pressed. The elevator stops there, and it remains in the `stopped` state waiting for the next request. When the first floor button is pressed, the elevator moves down until it reaches the first floor, and it remains there in the `stopped` state, ready for next button request. The same comparison can be done to the controller trace (and we can compare the results with the corresponding DEVS simulation, which is shown in Figure 19).

To answer the second question above, we need to check for the *liveness* property, that is, that something would *eventually* happen. In our case, for the proper operation of the controller within the coupled system, we are interested to check if by pressing a certain floor button, the elevator would *eventually* reach that floor. For example, if the user presses the third floor button, the elevator would *eventually* reach the third floor. This property is expressed in CTL as

```
button == 3 -> ElevatorController.cur_floor == 3
```

that is, whenever the third floor button is pressed, the *cur_floor* variable in the *ElevatorController* eventually reaches that floor. This property was satisfied as in the UPPAAL model checker for the given model. (A detailed trace for the example is found in the table included in the Appendix, in which we can see all the results obtained during the verification process for this and other queries. The table includes the preconditions and state changes are found on the left-hand side of the table; the values of the system variables are shown on the right-hand side of the table.) In this case, the model starts in the composite state at (*Stopped,StdByStop,S1*), then the *User_sensor_input* component has an enabled transition that fires on channel *buttonc* and synchronizes with the *ElevatorController* component on the same channel. This causes the total system state to move to (*Stopped,Moving,S2*). At this moment, the button request for the third floor is pressed (*button* = 3), however the elevator current floor is still on the first floor (*sensor* = 1, *ElevatorController.cur_floor* = 1). The system progresses through execution until the elevator reaches third floor at composite state (*Rising, Aux1, S5*).

However, for the query

```
button == 3 -> ElevatorController.cur_floor == 4
```

the property was not satisfied. When

```
(Stopped,StdByStop,S1)
buttonc: User_sensor_input --> ElevatorController
(Stopped,Moving,S1)
move: ElevatorController --> Elevator
(Rising,StdByMov,S1)
sensorc: User_sensor_input --> ElevatorController
(Rising,Aux,S1)
ElevatorController
(Rising,StdByMov,S3)
sensorc: User_sensor_input --> ElevatorController
(Rising,Aux1,S4)
ElevatorController
(Rising,StdByMov,S4)
sensorc: User_sensor_input --> ElevatorController
(Rising,Aux1,S5)
ElevatorController
(Rising,StdByMov,S5)
ElevatorController
(Rising,Stopped,S5)
move: ElevatorController --> Elevator
(StopUp,Stopped,S5)
stop: Elevator --> ElevatorController
(Stopped,StdByStop,S5)
buttonc: User_sensor_input --> ElevatorController
(Stopped,Moving,S6)
move: ElevatorController --> Elevator
(GoingDown,StdByMov,S6)
sensorc: User_sensor_input --> ElevatorController
(GoingDown,Aux1,S7)
ElevatorController
(GoingDown,StdByMov,S7)
sensorc: User_sensor_input --> ElevatorController
(GoingDown,Aux1,S1)
ElevatorController
(GoingDown,StdByMov,S1)
ElevatorController
(GoingDown,Stopped,S1)
move: ElevatorController --> Elevator
(SlowingDown,Stopped,S1)
stop: Elevator --> ElevatorController
(Stopped,StdByStop,S1)
```

Figure 17. Elevator TA simulation results in UPPAAL

```
C 00:00:00:000 : stopped ,
? 00:00:05:000 : move , 2
E 00:00:05:000 : stopped , rising
? 00:00:18:000 : move , 0
E 00:00:18:000 : rising , StopUp
O 00:00:19:000 : stop , 1
I 00:00:19:000 : StopUp , stopped
? 00:00:27:000 : move , 1
E 00:00:27:000 : stopped , dropping
? 00:00:36:000 : move , 0
E 00:00:36:000 : dropping , SlowDown
O 00:00:37:000 : stop , 1
I 00:00:37:000 : SlowDown , stopped
```

Figure 18. Elevator simulation results

we press the third floor button, if the elevator initially stopped at first floor, there is no way the elevator would reach the fourth floor. This would allow a modeler to revise the original DEVS graph model, redesigning it according to requirements. The system verifies that this would never be satisfied by checking the model with all possible executions until no reachable composite state would satisfy the query. If we run this in UPPAAL, we can see that the trace starts from initial composite state not satisfying the query and, after going through execution, we reach the same initial composite state without satisfying the query in any point during the execution.

To answer the third question (i.e. to find out whether the elevator would reach the requested third floor within some bounded time), we extended the model for bounded time checking by adding a Boolean variable b, and a global clock z, as shown on the Elevator model. The variable b is set to true as long the elevator starts moving, and until it reaches the *Stopped* state again. Therefore, by checking the accumulated time in clock z while b is true, this gives us the property we need to check. That property is expressed as

```
C 00:00:00:000 : stdbystop , (direction=0) (floor=0) (cur_floor=0)
? 00:00:05:000 : button, 3
E 00:00:05:000 : stdbystop , moving(direction=2) (floor=3) (cur_floor=0)
O 00:00:05:000 : move, 2
I 00:00:05:000 : moving, stdbymov (direction=2) (floor=3) (cur_floor=0)
? 00:00:10:000 : floorSensor, 1
E 00:00:10:000 : stdbymov , aux1 (direction=2) (floor=3) (cur_floor=1)
I 00:00:10:000 : aux1 , stdbymov (direction=2) (floor=3) (cur_floor=1)
? 00:00:14:000 : floorSensor, 2
E 00:00:14:000 : stdbymov , aux1 (direction=2) (floor=3) (cur_floor=2)
I 00:00:14:000 : aux1 , stdbymov (direction=2) (floor=3) (cur_floor=2)
? 00:00:18:000 : floorSensor, 3
E 00:00:18:000 : stdbymov , stopped (direction=2) (floor=3) (cur_floor=3)
O 00:00:18:000 : move, 0
I 00:00:18:000 : stopped , stopping(direction=2) (floor=3) (cur_floor=3)
? 00:00:19:000 : stop, 1
E 00:00:19:000 : stopping, stdbystop (direction=2) (floor=3) (cur_floor=3)
? 00:00:27:000 : button, 1
E 00:00:27:000 : stdbystop , moving(direction=1) (floor=1) (cur_floor=3)
O 00:00:27:000 : move, 1
I 00:00:27:000 : moving, stdbymov (direction=1) (floor=1) (cur_floor=3)
? 00:00:32:000 : floorSensor, 2
E 00:00:32:000 : stdbymov , aux1 (direction=1) (floor=1) (cur_floor=2)
I 00:00:32:000 : aux1 , stdbymov (direction=1) (floor=1) (cur_floor=2)
? 00:00:36:000 : floorSensor, 1
E 00:00:36:000 : stdbymov , stopped (direction=1) (floor=1) (cur_floor=1)
O 00:00:36:000 : move, 0
I 00:00:36:000 : stopped , stopping(direction=1) (floor=1) (cur_floor=1)
? 00:00:37:000 : stop, 1
E 00:00:37:000 : stopping, stdbystop (direction=1) (floor=1) (cur_floor=1)
```

Figure 19. Controller simulation output

```
A[] ( b imply z < 27 )
```

which is satisfied, that is, reaching the third floor with less than 27 time units is possible for all possible executions. We can see this in the trace from the initial composite state in the table in the Appendix, until the composite state (*StopUp, Stopping, S5*) where the elevator has stopped in state *StopUp* and current floor is 3 with the clock *z* in the interval [0,27). In this state, the query evaluates to true. However, the query

```
A[] ( b imply z < 26 )
```

is not satisfied. This shows that the elevator would reach the third floor after the request, and it would take no less than 26 time units, but it guaranteed to reach the floor at 27 time units or more. From the same trace presented in the Appendix, when we reach the third floor, clock *z* is in the interval [0,27). This shows that the elevator cannot meet a deadline of less than 26 time units for all executions leading to reach the third floor.

5. A Discrete Event System Specification modeling and simulation example

As discussed earlier, the TA checkers (such as UPPAAL) sometimes cannot solve the questions being asked (or, due to explosion of states, the model checker can take a very long time for every single query). In those cases, running a simulation with individual test cases can provide insight on the quality of the model, even when we cannot prove all the properties available. In order to show this process and the relationship with the formal verification presented in Section 4, we here present a M&S example of the previous models.

As discussed earlier, the elevator model shown in Figure 11 represents the different states of elevator movement and the transitions between these states. In this model, the elevator starts in the *stopped* state and waits for controller commands to move to satisfy a button request from the user. The decisions for proper direction, starting, and stopping the movement are taken by the controller. The DEVS graph in Figure 11 has a corresponding textual representation and the model is shown in Figure 20.

As we can see, the model uses one input and one output port, which are specified with the keywords *in* and *out*, respectively. The *State* keyword defines the list of states on the model, with *initial* as the initial state. External transitions are marked by the keyword *ext*. Internal transitions are marked by the keyword *int*. For both transitions we define source/destination states, along with the input port and value for external transition, and output port and value to that port in the case of internal transition. The lifetimes for each state are represented besides the state name. Similarly, the model textual specification corresponding to the DEVS Graph model of the elevator Controller of Figure 14 is shown in Figure 12.

```
[elevator]
in: mover
out: stop
state: stopped GoingDown SlowingDown aux rising StopUp
initial : stopped
ext: stopped rising Value(mover)?2
ext: rising StopUp Value(mover)?0
ext: stopped aux Value(mover)?0
ext: GoingDown SlowingDown Value(mover)?0
int: aux stopped
int: SlowingDown stopped stop!1
int: StopUp stopped stop!1
ext: stopped GoingDown Value(mover)?1
stopped: 00:00:1000:00
GoingDown: 00:00:1000:00
SlowingDown: 00:00:1:00
aux: 00:00:00:00
rising: 00:00:1000:00
StopUp: 00:00:1:00
```

Figure 20. Elevator CD++ model

Figure 21 shows the coupled model definition for the system, which is composed by the *elevator* and the *controller* models. In the top model, the two components are defined, including the two input ports in the top component: button and sensor. These two ports are *link*ed to the input ports of controller (as seen in lines 6 and 7). The coupling between the *elevator* and *controller* atomic models is shown in lines 4 and 5. The figure also includes a graphical representation of this coupled model, which shows the atomic models of the elevator and controller, the input ports to the coupled model, and the links between all components (both notations are equivalent).

In Figure 22, we show a test case scenario for the elevator top model. These inputs are used to simulate the overall execution of the elevator system. These external events are sent to the model top component defined in Figure 21. The simulator will direct the inputs to the elevator controller as specified in the model definition. In this file, the third floor button is pressed 5 s after the start of the simulation. The *floorSensor* inputs are used to define the signals sent by the elevator sensors to the elevator controller (which includes the floors 1, 2, and 3, which were reached at times 10, 14, and 18 s, respectively). At time 27 s, the first floor button is pressed, and then, the floor sensor sends the corresponding signals at designated times, as shown in the figure.

Figure 19 shows the simulation results for the elevator controller. The character in the first column in the simulation results represents the following:

`C`: the initial state;

`?`: input received by the elevator atomic model;

`E`: external transition executed by the elevator atomic model that is triggered by the reception of an event;

`O`: output caused by invoking the output function;

`I`: internal transition executed.

As we can see, initially, the controller is at state `stdbystop`, with all variables initialized to zero. At 5 time units, the controller receives the third floor button request as specified in the input event file in Figure 22. This input causes the controller to execute the external transition function, and it changes its state from `stdbystop` to `moving` (with the new variable values shown on the simulation trace). At time 5, the output function executes, and it sends the value 2 to port `move`. Then, the internal transition function executes, and it reaches the state `stdbymov`. The simulation continues according to the inputs fed to the model (specified in the input event file in Figure 22).

Similarly, Figure 18 shows the elevator simulation results according to the input/output events received from/sent to the controller.

As we can see, the elevator simulation starts at the `stopped` state at time 00:00. At time 5:00, the elevator receives an input on the `move` port with the value of 2. This causes the elevator to change state to `rising` and wait there for input 0 on the `move` port. At time 18:00, the required input arrives and the elevator changes to state `StopUp`, which its lifetime equals to 1 time units. This state represents the elevator braking in the upward direction preparing to stop. At 19:00, the elevator executes the output function and sends the value of 1 on the output port `stop`, then changes to the `stopped` state. The simulation continues until the model reaches the `stopped` state again in the last line.

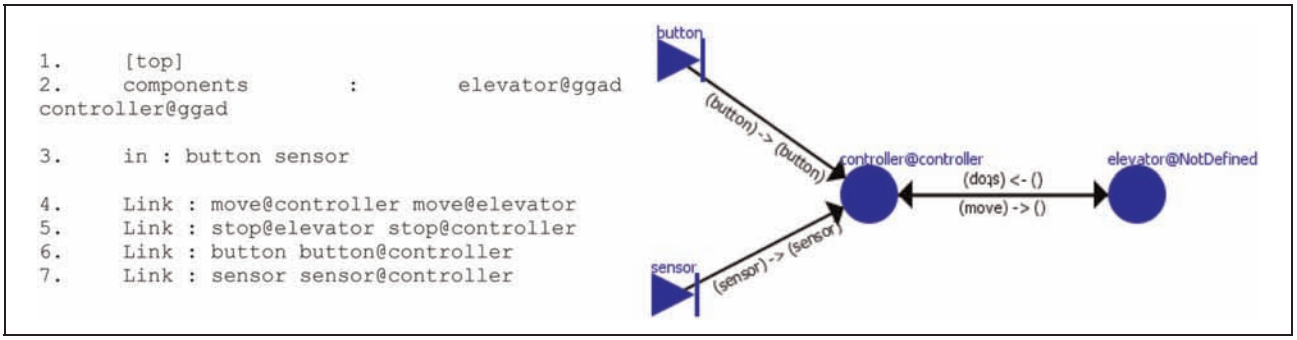

Figure 21. Elevator coupled model definition and corresponding coupled model graph

```
00:00:05:00 button3
00:00:10:00 floorSensor1
00:00:14:00 floorSensor2
00:00:18:00 floorSensor3
00:00:27:00 button1
00:00:32:00 floorSensor2
00:00:36:00 floorSensor1
```

Figure 22. Elevator simulation event file

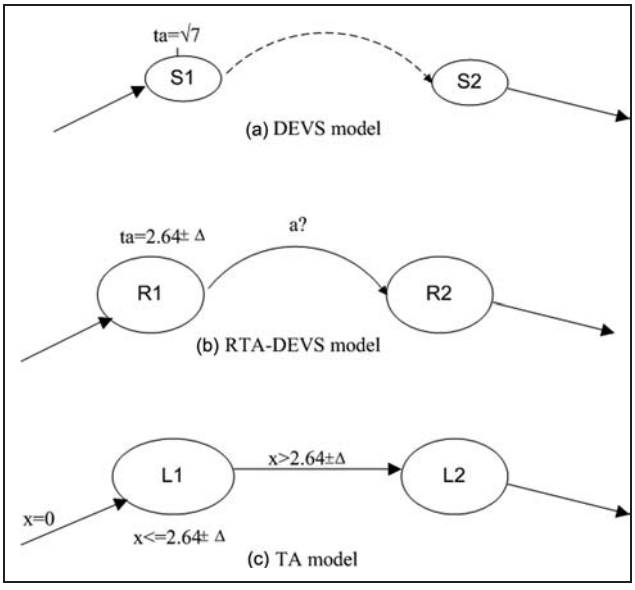

Figure 23. Approximation of irrational time values in internal transition: (a) DEVS; (b) RTA-DEVS; (c) TA

The models just simulated are the same ones that were used earlier (which were transformed into TA, and were verified through model-checking tools). If any problems are found in any of these steps, the modeler can go back directly to the original atomic and coupled models and check any of the multiple results that both the model-checking tools and the simulation software provides. Simultaneously, the same model specification can be used to generate code that can be used on an embedded platform (in this particular case, a microcontroller in charge of moving the elevator). In our case, we have developed a virtual machine that executes in varied platforms that can understand the DEVS formal specifications and execute the models without modification. This provides a sound method for model construction and verification, which can be executed directly without any changes to the original specifications.

6. Model checking of approximated Discrete Event System Specification models.

In the previous sections, we showed a methodology to verify RTA-DEVS models formally using TA, and a method to simulate those models in the case that the verification algorithms do not work. However, classic DEVS models are more expressive than RTA-DEVS models, and therefore it may be difficult if not impossible to directly verify classic DEVS models. In this section, we introduce a method for transforming classic DEVS to RTA-DEVS that would allow us to use the verification methodology presented in the previous sections. However, this method requires a higher level of abstraction and, consequently, some approximations are required.

In this section, we introduce this method and we show what approximations are needed and what effect they would have on the verification results. In the case of a DEVS model with an infinite number of states, we need the modeler to make an abstraction of these to an approximation with a finite number of states. After this, the model can be checked formally by converting, with the necessary approximations, the classic DEVS model to RTA-DEVS. To do so, we need to find a reasonable approximation for any irrational values that may exist in the DEVS model, while building a valid RTA-DEVS.

6.1 Irrational values in the time-advance function

DEVS irrational values in the codomain of the time-advance function *ta* need to be approximated to a rational value when converting to RTA-DEVS. An example of such DEVS model definition is given in Figure 23(a).

In this figure, a DEVS model is approximated with a RTA-DEVS model, shown in (b), and the equivalent and approximated TA model, shown in (c). As we can see, the irrational time-advance value originally found in the DEVS model is converted to a rational value with approximation error Δ. This error propagates on the equivalent TA.

The questions that we need to answer here are as follows. How is this approximation error Δ going to affect the validity of the RTA-DEVS and TA models? Are these valid models? Can we obtain the same conclusions on the original DEVS and the TA?

To answer the first question (i.e. to guarantee building valid RTA-DEVS and TA models), we show an example of the proposed method. Figure 24 shows a piece of a coupled DEVS model in which component A waits in state $S1$ for $\sqrt{7}$ time units, and it then executes the output and internal transition function (which transmits an output event a, and it then changes to the state $S2$). Simultaneously, the component B is in state $S3$ waiting for the event a, and when this is received, it will trigger the execution of the external transition function as follows:

$$\delta_{\text{ext}}(S3, e, a) = (S4, 0) \quad \text{if } \sqrt{7} \leq e \prec \infty$$

$$\delta_{\text{ext}}(S3, e, a) = (S5, 0) \quad \text{if } 0 \prec e \prec \sqrt{7}$$

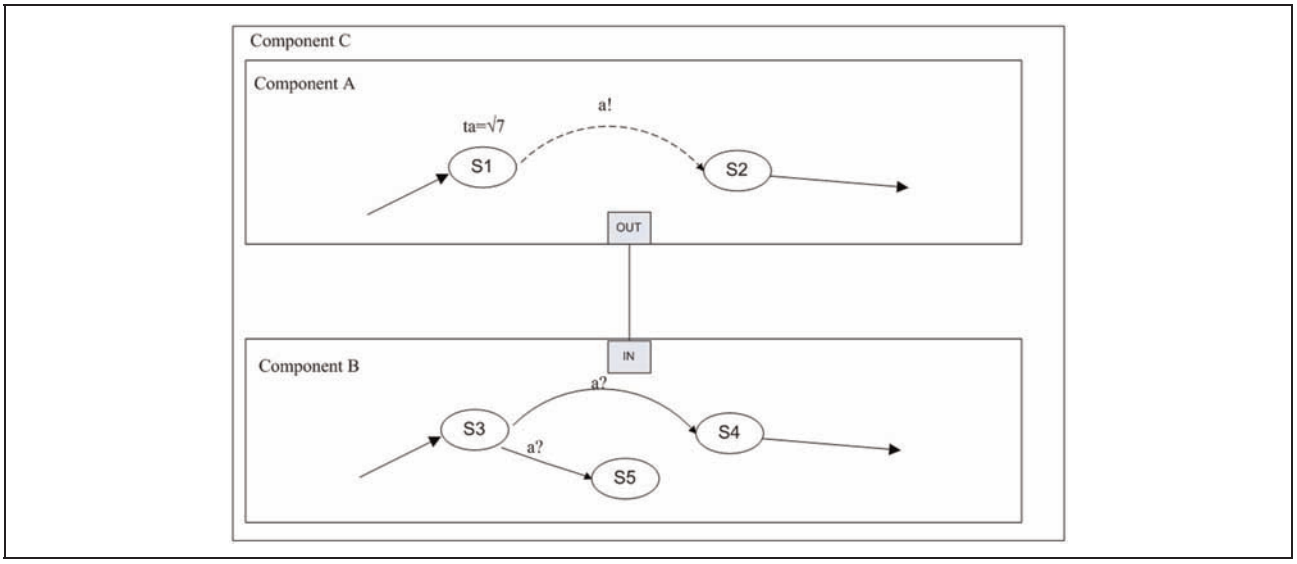

Figure 24. A coupled DEVS model

By coupling components A and B, the total behavior of the coupled DEVS component C would be

$$(S1, S3) \xrightarrow{d=\sqrt{7},\, a} (S2, S4)$$

The coupled system starts in total state of ($S1$, $S3$) and, after a delay of $\sqrt{7}$ time units, model A sends event a to model B, which triggers a transition to the total state ($S2$, $S4$). Based on this, we can construct a behaviorally equivalent RTA-DEVS model (Figure 25) that is approximated to the DEVS shown in Figure 24. In this model, the lifetime of $S1$ is approximated by a rational value with error Δ. The value of Δ depends on the precision chosen; for example, for two decimal digits, $\Delta \leq 0.005$. The external transition function could be approximated as
Approximation 1:

$$\delta_{\text{ext}}(S3, e, a) = (S4, 0) \qquad 2.64 + \Delta \leq e \prec \infty$$
$$\delta_{\text{ext}}(S3, e, a) = (S5, 0) \qquad 0 \prec e \prec 2.64 + \Delta$$

Or Approximation 2:

$$\delta_{\text{ext}}(S3, e, a) = (S4, 0) \qquad 2.64 - \Delta \leq e \prec \infty$$
$$\delta_{\text{ext}}(S3, e, a) = (S5, 0) \qquad 0 \prec e \prec 2.64 - \Delta$$

However, the choice of the approximation would affect the validity of the RTA-DEVS model. For instance, if we approximate the ta of $S1$ with ta =$2.64 - \Delta$, and we choose Approximation 1 for model B, the coupled model C' would have a different behavior from the original DEVS model. Thus, component C' behavior now becomes

$$(S1, S3) \xrightarrow{d=2.64-\Delta,\, a} (S2, S5)$$

Proposition 1: when approximating an irrational value triggering an internal transition that is coupled with an external transition, the choice of approximation value should be consistent for all constants using this irrational number.
Formally, if we have the following defined in the DEVS:

$$\delta^A{}_{\text{int}}(S_i, C_{\text{irr}}) = S_j,\ \lambda^A(S_i) = a,\ ta^A(S_i) = C_{\text{irr}}$$
$$\delta^B{}_{\text{ext}}(S_k, e, a) = (S_l, 0) \quad C_{\text{irr}} \leq e \prec \infty$$
$$\delta^B{}_{\text{ext}}(S_k, e, a) = (S_m, 0) \quad 0 \prec e \prec C_{\text{irr}}$$

it should be approximated in the RTA-DEVS model as

$$\delta^A{}_{\text{int}}(S_i, C_r) = S_j,\ \lambda^A(S_i) = a,\ ta^A(S_i) = C_r$$
$$\delta^B{}_{\text{ext}}(S_k, e, a) = (S_l, 0) \qquad C_r \leq e \prec \infty$$
$$\delta^B{}_{\text{ext}}(S_k, e, a) = (S_m, 0) \qquad 0 \prec e \prec C_r$$

where C_{irr} is an irrational real number, C_r is a rational real number, and $\delta^A{}_{\text{int}}$, λ^A, ta^A are functions defined for component A.

6.2 Irrational values in the external transition function

As seen in the example shown in Figure 26, a modeler may choose different approximations to form component B''.
Approximation 3:

$$\delta_{\text{ext}}(S3, e, a) = (S4, 0) \qquad 2.64 + \Delta \leq e \prec \infty$$
$$\delta_{\text{ext}}(S3, e, a) = (S5, 0) \qquad 0 \prec e \prec 2.64 - \Delta$$

Or Approximation 4:

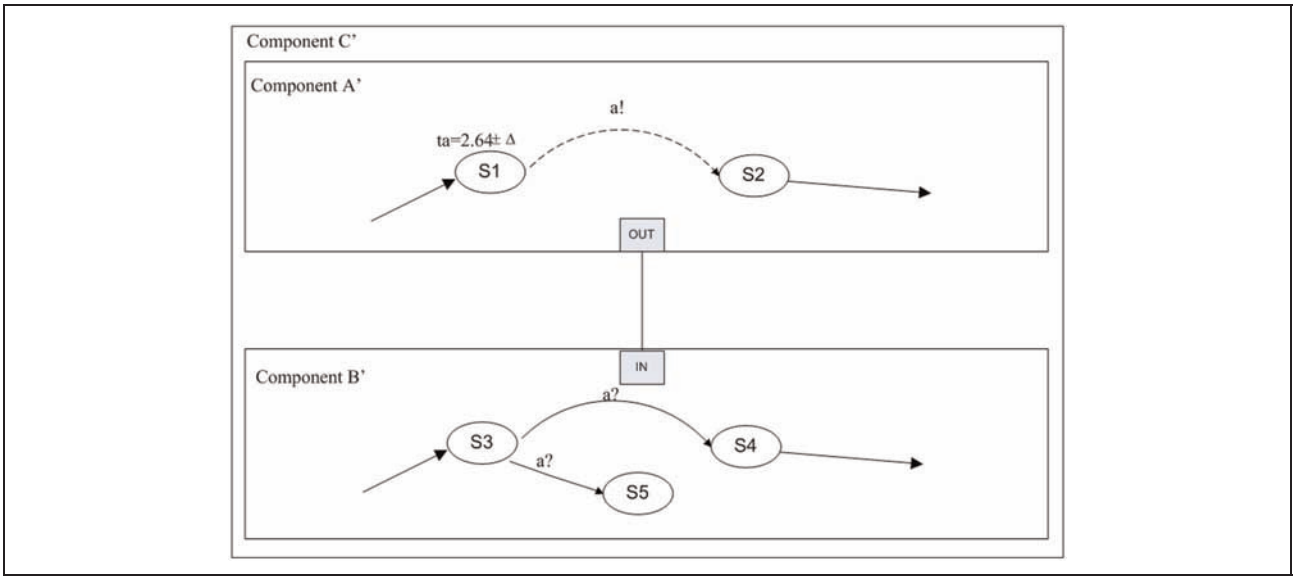

Figure 25. Coupled RTA-DEVS model

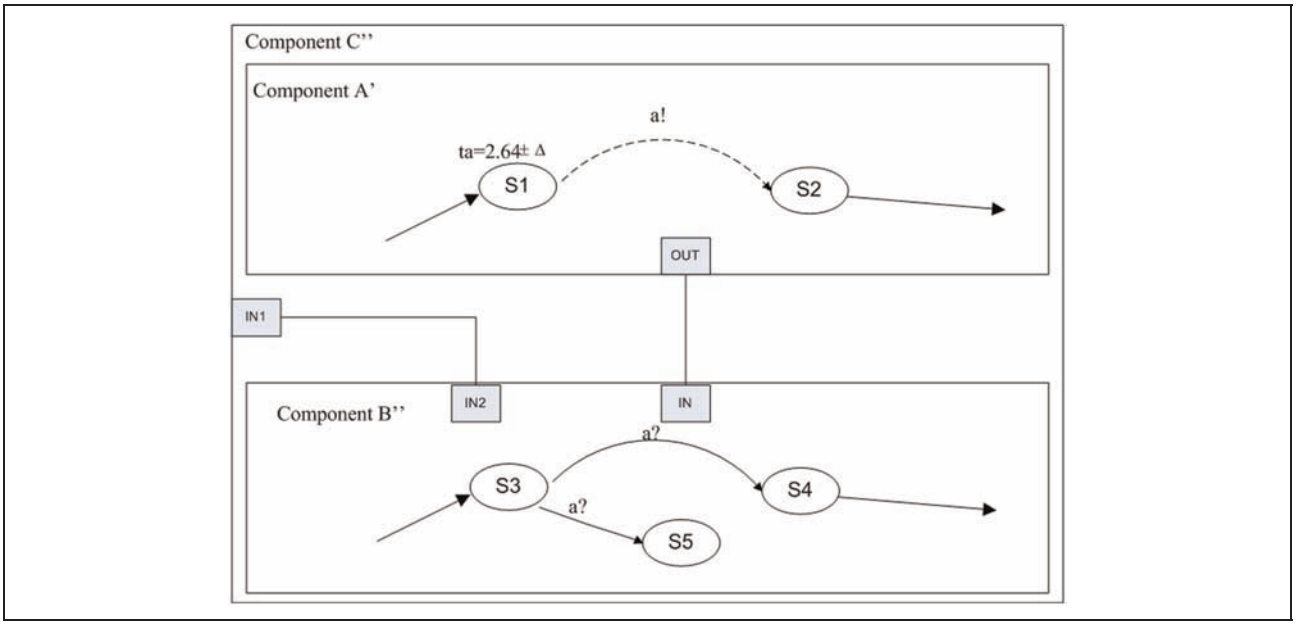

Figure 26. RTA-DEVS component with external input

$$\delta_{\text{ext}}(S3, e, a) = (S4, 0) \qquad 2.64 - \Delta \leq e \prec \infty$$

$$\delta_{\text{ext}}(S3, e, a) = (S5, 0) \qquad 0 \prec e \prec 2.64 + \Delta$$

The component C'' in Figure 26 accepts an external input from its environment on input port IN1, which is connected to component B'' at its port IN2. In this case, the behavior of C'' would not match the behavior of C, in that with Approximation 3, the external transition function is not defined in the interval $2.64 - \Delta \prec e \prec 2.64 + \Delta$. Thus, C'' would contain an action lock,[39] which is a special case of a deadlock in which the system would not progress due to lack of any enabled transitions at the exact point-in-time in which an event occurs. This typically reflects a modeling error (or, in our case, a modeling fault due to the approximation error). Approximation 4 introduces another type of error. This would cause a non-deterministic behavior in the model in the interval $2.64 - \Delta \prec e \prec 2.64 + \Delta$, which differs from the original DEVS model behavior. To avoid such errors, we define the next proposition.

Proposition 2: when approximating an irrational value for elapsed time in external transition function definition, the choice of approximation value should be consistent for all constants using this irrational number.
Formally, if we have the following DEVS definition of external transition function

$$\delta_{\text{ext}}(S_i, e, a) = (S_j, 0) \quad C_{\text{irr}} \leq e \prec \infty$$
$$\delta_{\text{ext}}(S_i, e, a) = (S_k, 0) \quad 0 \prec e \prec C_{\text{irr}}$$

it must be approximated in the RTA-DEVS model on the following form to avoid creating action locks:

$$\delta_{\text{ext}}(S_i, e, a) = (S_j, 0) \quad C_r \leq e \prec \infty$$
$$\delta_{\text{ext}}(S_i, e, a) = (S_k, 0) \quad 0 \prec e \prec C_r$$

6.3 Effect of the approximation error on model-checking results

The consequences of using rational numbers in RTS models would not introduce any errors as long as we use them according to the previous two propositions. This guarantees preserving model behavior when approximating irrational values with rational ones. The approximation we choose for our rational value can be arbitrarily small compared to computer system physical clock precision, to mitigate any practical effect on model behavior.

The next question is how the approximation of irrational constants in *ta* or δ_{ext} affect the formal verification of RTA-DEVS models. Would a result obtained from model checking RTA-DEVS models apply to the original DEVS?

When we approximate an irrational constant C_{irr} with a rational constant C_r, we introduce an error Δ such that $C_{\text{irr}} = C_r \pm \Delta$. This error appears then in constants used for time-advance function or external transition functions. Verification of RTA-DEVS through transforming it to equivalent TA is done with reachability analysis. Would this analysis differ by introducing the error Δ when we move from DEVS to RTA-DEVS?

Answering this question directly would require reachability analysis of the original DEVS with irrational constant values, and for the transformed RTA-DEVS model with the rational values (then, comparing results). This approach, however, is not feasible, as the reachability analysis for timed models with irrational constants has been proven to be undecidable.[35] Therefore, we need to use an approximate approach to estimate the effect of Δ on the reachability analysis.

This problem is equivalent to that of *robustness* of TA.[40] In robust TA, a robust model accepts an input sequence of events within a time interval. This is called a *bundle* of events, which are close in time, and the model still behaves the same with this bundle input. Puri[42] extended the notion of robust TA, and included in the robustness definition those models in which their reachability analysis remains the same with small drifts in clock models. In this definition, a model is not robust if for any small drift in clock rate, the reachability results change. In De Wulf et al.,[41] it was proved that clock drifts in TA are equivalent to having a reaction delay by the implementation that increases guard constants by a small positive value Δ. The robustness problem is then transformed to an implementation problem, in which one needs to find a value Δ that makes the verification results correct. Further work by De Wulf et al.[41] showed a methodology to assess a model for implementability, by using standard TA model-checking tools, and proved that if a model is tolerant to a certain value Δ, it would also be correct with any value Δ' such that $\Delta' < \Delta$.

These results from robustness theory of TA are useful to check if the formal verification results of RTA-DEVS models also apply to the original DEVS model correctly. Given an error Δ introduced by approximation of irrational numbers in DEVS models, we model the possible transition from a state within an enlarged time interval Δ nondeterministically. For example:

$$\delta_{\text{ext}}(S_i, e, a) = (S_j, 0) \quad C_{\text{irr}} \leq e \prec \infty$$
$$\delta_{\text{ext}}(S_i, e, a) = (S_k, 0) \quad 0 \prec e \prec C_{\text{irr}}$$

and $C_{\text{irr}} = C_r \pm \Delta$, then, we enlarge the interval in which the external transition is enabled, that is, to define it as

$$\delta_{\text{ext}}(S_i, e, a) = (S_j, 0) \quad C_r - \Delta \leq e \prec \infty$$
$$\delta_{\text{ext}}(S_i, e, a) = (S_k, 0) \quad 0 \prec e \prec C_r + \Delta$$

This model can be transformed to an equivalent TA, as shown in previous sections, and it can then be checked against the desired properties. With non-determinism in the model, UPPAAL checks the transition as if enabled during the interval, covering the point around the irrational number value. Hence, if the model-checking results were correct, we conclude that the approximation did not introduce errors to the RTA-DEVS model.

6.4 Application to elevator/elevator-controller example

We use the elevator system example shown previously to demonstrate our methodology. The example is extended by changing an irrational value in the controller model, as seen in Figure 27.

As we can see, state *stdbyMov* in Figure 27 uses a *ta* of $\sqrt{1000007} \approx 1000.003$ or $\sqrt{1000007} \approx 1000.004$; $\Delta = 0.001$. The resulting TA model is shown in Figure 28. In this TA model, we added node E and a transition from *StdByMov* to E that is enabled at elapsed time of $x \geq 1000003$. This TA is semantically equivalent to the DEVS

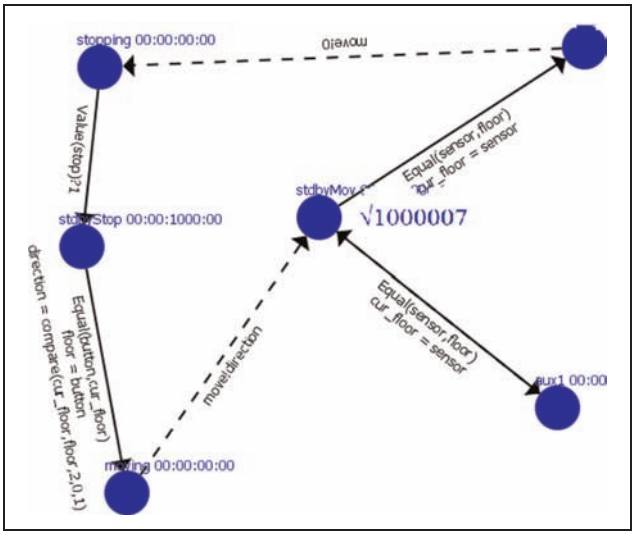

Figure 27. Elevator-controller in DEVS graphs notation

model in Figure 27. However, this TA allows the transition from node *StdByMov* to node *Stopped* to be taken non-deterministically in the interval [0,1000004], while the transition to E is enabled in [1000003, ∞]. This ensures covering the interval [0, √1000007] in UPPAAL's model-checking algorithms.

We ran the model checker to verify the non-deterministic version of the elevator-controller model along with the other components in the elevator system. The results were successful and unchanged from the results we obtained previously as shown in figure 29. The consistency of results in both non-deterministic and deterministic models indicates that the approximation error did not affect the verification results. Hence, for any value smaller than 0.001, the results would not be affected.[43] Although we could not verify the DEVS model in Figure 27 because of the irrational value of the time-advance function, our methodology approximates this model to a behaviorally equivalent RTA-DEVS and then to an equivalent TA, which can be used to verify the equivalent DEVS model.

7. Conclusion

We introduced a series of methods to verify RT models specified with DEVS formally, and discussed some of the problems that prevent general classic DEVS models from being modeled and verified. We introduced the RTA-DEVS to overcome these problems and to offer a subset of DEVS that is still expressive enough for modeling practical problems and that can be transformed to TA for formal verification. The transformed TA can be used to reason about the original RTA-DEVS model, and hence about the real problem being modeled. We also showed a methodology to convert the RTA-DEVS to TA in a systematic way. We finally introduced a methodology based on recent theoretical work that can reveal if a given DEVS model being approximated by RTA-DEVS would have its verification results unaffected by the approximation process. In reality, if a given DEVS model cannot tolerate small approximation errors without changing its formal verification results, this DEVS model would be almost impossible to implement faithfully on a hardware platform, as that platform would never be able to give exact timing due to the nature of digital clocks and transition delays. Inconsistency of verification results in our methodology would be an indication of such a DEVS model.

Validating DEVS models formally with TA model checking can pave the road for solving RT predictability in software systems. DEVS models are also directly

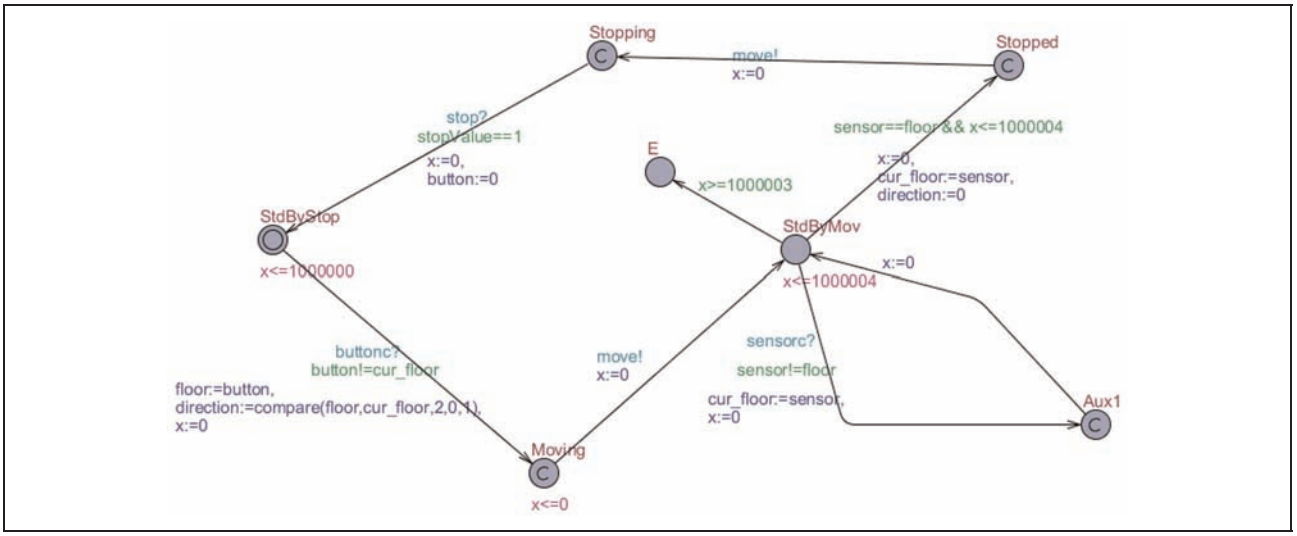

Figure 28. TA model with non-deterministic behavior

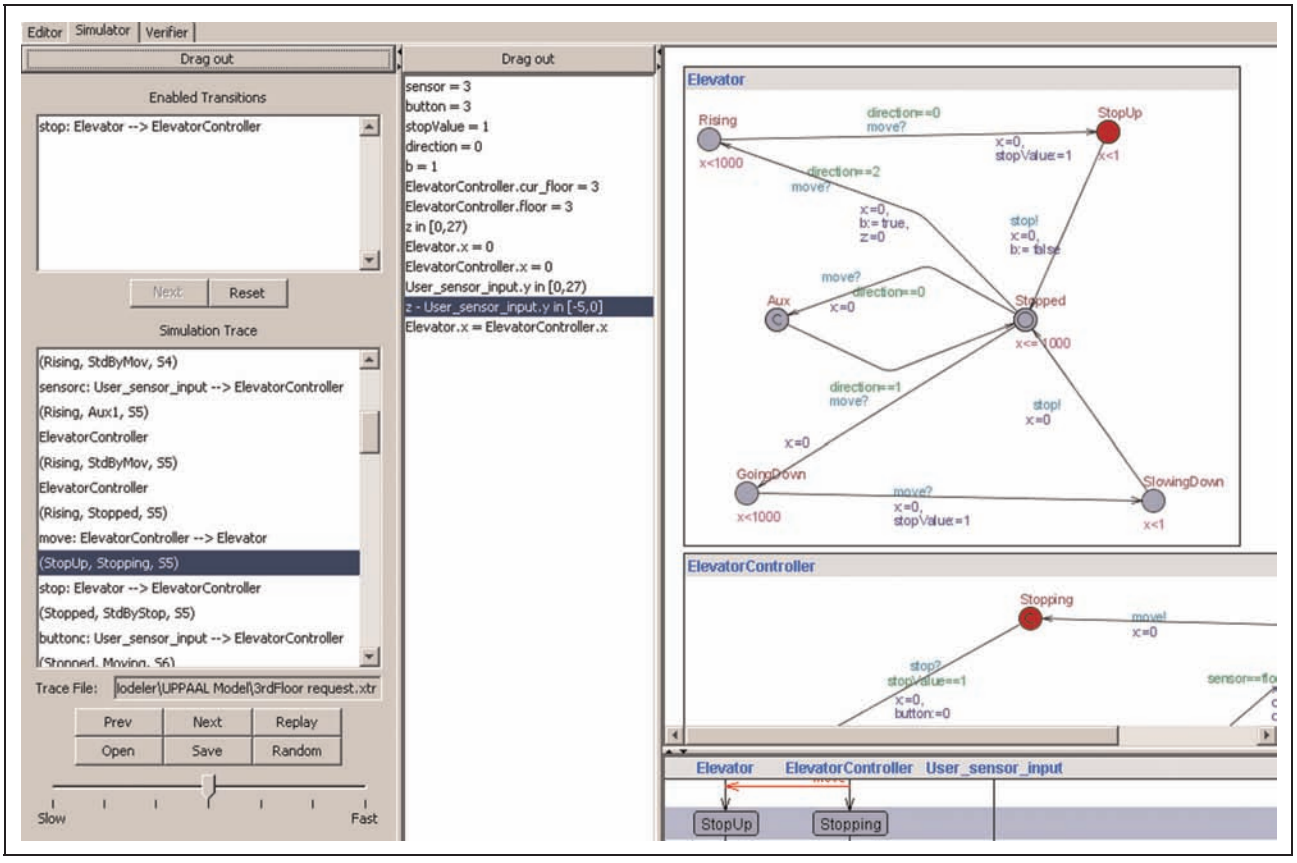

Figure 29. Model verification execution in UPPAAL

executable on a virtual machine executing in embedded RT platforms.[44] With this advantage, any DEVS model validated formally would be guaranteed to execute as predicted by the validation, as no human intervention comes between the checked model and the executable specification. This advantage would serve not only the simulation community, but also the RT software community, as DEVS can be used to model controllers that would be simulated, formally validated, and then deployed on the target platform.

Our approach to using model checking to verify DEVS models is limited with the same limitations imposed on TA model checking, mainly because of the problem of state space explosion. This would limit the methodology to small- and medium-sized DEVS models. However, many real-life applications fall into these boundaries. For larger models, a combination of abstraction and decomposition techniques would be able to reduce the problem size to practical model checking. In addition, in those cases where model checking is not feasible, one can formally verify subcomponents of the whole application while simulating the complete models, in order to gain insight on the software applications.

One side effect of our work is that we obtained an equivalent TA for the RTA-DEVS. This enables us to apply theories and reasoning from existing literature of TA to find specific properties of RTA-DEVS and DEVS formalisms. For example, we can compare the expressiveness of the RTA-DEVS to that of TA. As our transformation showed, the equivalent TA to a RTA-DEVS has no diagonal clock constraints of the form $x - y < C$. This is a limitation of general TA with diagonal constraints. However, this does not reduce the expressiveness of the RTA-DEVS from the general TA formalism, as any TA with diagonal constraints can be expressed with a TA without diagonal constraints, as shown by Bérard et al.[45] However, general TA with diagonal constraints are more concise than the RTA-DEVS (represented by a TA without diagonal constraints), as shown by Bouyer and Chevalier.[46]

We envision that this methodology could influence the development of RT systems in a number of ways, producing models more accurately, and better simulations with less cost and effort. For the validation and verification of simulation models, which are done currently in a manual, error-prone procedure, and usually need a domain expert, this technique can improve such activities.

To overcome the scalability issue, future work would take advantage of different techniques to reduce the state-space during-model checking. These techniques are based on specific properties of DEVS models that can be exploited to simplify the resulting TA models.

Funding

This research was partially funded by NSERC.

References

1. Rehman MJ, Jabeen F, Bertolino A and Polini A. Testing software components for integration: a survey of issues and techniques. *Software Test Verification Reliab* 2007; 17: 95–133.
2. Gerlich R, Gerlich R and Boll T. Random testing: from the classical approach to a global view and full test automation. In: *Proceedings of the 2nd international Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM international Conference on Automated Software Engineering (ASE 2007)*, Atlanta, GA.
3. Florescu O, Voeten J, Theelen B and Corporaal H. Patterns for automatic generation of soft real-time system models. *Simulation* 2009; 85: 709–734.
4. Castro R, Kofman E and Wainer G. A formal framework for stochastic discrete event system specification modeling and simulation. *Simulation* 2010; 86: 587–611.
5. Dwyer MB, Hatcliff J, Robby R, et al. Formal software analysis emerging trends in software model checking. In: *Proceedings of the 2007 Future of Software Engineering (FOSE '07)*, IEEE Computer Society, Washington, DC, pp.120–136.
6. Alur R and Dill DL. A theory of timed automata. *J Theor Comput Sci* 1994; 126: 183–235.
7. Härri J, Fiore M, Filali F, et al. Vehicular mobility simulation with VanetMobiSim, *Simulation* 2011; 87: 275–300.
8. Staub T, Gantenbein R and Braun T. VirtualMesh: an emulation framework for wireless mesh and ad hoc networks in OMNeT ++ . *Simulation* 2011; 87: 66–81.
9. Zeigler B, Kim T and Praehofer H. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. 2nd ed. Academic Press, 2000 San Diego, CA, USA.
10. Behrmann G, David A and Larsen KG. A tutorial on Uppaal. In: *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04), Formal Methods for the Design of Real-Time Systems 2004*, LNCS, vol. 3185, pp.33–35.
11. Bengtsson J and Yi W. Timed automata: semantics, algorithms and tools. In: Reisig W and Rozenberg G (eds) *Lecture Notes on Concurrency and Petri Nets*. Vol. 3098, LNCS Springer, 2004, pp.87–124.
12. Wainer G. *Discrete-event modeling and simulation: a practitioner's approach*. CRC Press, 2009 Boca Raton, FL, USA.
13. Wainer G. CD ++ : a toolkit to define discrete-event models. *Software Pract Ex* 2002; 32: 1261–1306.
14. Wainer G, Glinsky E and Gutierrez-Alcaraz M. Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. *Simulation* 2011; 87: pp.555–580.
15. Christen G, Dobniewski A and Wainer G. Modeling state-based DEVS models in CD ++ . In: *Proceedings of MGA, Advanced Simulation Technologies Conference*, Arlington, VA, 2004.
16. Praehofer H and Pree D. Visual modeling of DEVS-based multiformalism systems based on higraphs. In: *WSC'93: Proceedings of the 25th Conference on Winter Simulation*, Los Angeles, CA, pp.595–603.
17. Bergero F and Kofman E. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation* 2011; 87: 113–132.
18. Song HS and Kim TG. Application of real-time DEVS to analysis of safety-critical embedded control systems: railroad crossing control example. *Simulation* 2005; 81: 119–136.
19. Hu X. From virtual to real – a progressive simulation-based design framework. In: Wainer GA and Mosterman PJ (eds) *Discrete-event modeling and simulation: theory and applications*, CRC Press, 2010 Boca Raton, FL, USA.
20. Pace DK. Modeling and simulation verification and validation challenges. *Johns Hopkins APL Tech Digest* 2004; 25: 163–172.
21. Hemer D and Lindsay PA. Template-based construction of verified software. *IEE Proc Software Eng* 2005; 152: 2–12.
22. Huang J, Voeten J and Corporaal H. Predictable real-time software synthesis. *Real Time Syst* 2007; 36: 159–198.
23. Baleani M, Ferrari A, Mangeruca L, Sangiovanni-Vincentelli AL, Freund U, Schlenker E and Wolff H.-J. 2005. Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development. In Proceedings of the conference on Design, Automation and Test in Europe - Volume 2 (DATE '05), Vol. 2. IEEE Computer Society, Washington, DC, USA, 1044–1049.
24. Voeten J, Florescu O, Huang J and Corporaal H. Error computation for predictable real-time software synthesis. *Simulation* 2011; 87: 334–350.
25. Dacharry HP and Giambiasi N. A formal verification approach for DEVS. In: *Proceedings of the 2007 Summer Computer Simulation Conference*, Society for Computer Simulation International, San Diego, CA, 16–19 July 2007, pp.312–319.
26. Hong JS, Song HS, Kim TG, et al. A real-time discrete event system specification formalism for seamless real-time software development. *Discrete Event Dyn Syst* 1997; 7: 355–375.
27. Furfaro A and Nigro L. Embedded control systems design based on RT-DEVS and temporal analysis using UPPAAL. In: *Proceedings of Computer Science and Information Technology (IMCSIT 2008)*, 20–22 October 2008, pp.601–608.
28. Furfaro A and Nigro L. A development methodology for embedded systems based on RT-DEVS. *Innovat Syst Software Eng* 2009; 5: 117–127.
29. Hwang MH and Zeigler BP. Reachability graph of finite and deterministic DEVS networks. *IEEE Trans Autom Sci Eng* 2009; 6: 468–478.
30. Han S and Huang K. Equivalent semantic translation from parallel DEVS models to time automata. In: *Computational Science – ICCS*, LNCS 2007, volume 4487, pp. 1246–1253.
31. Giambiasi N, Paillet J-L and Châne F. From timed automata to DEVS models. In: *Proceedings of the 2003 Winter Simulation Conference*, 2003, pp.923–931.
32. Hernández A and Giambiasi N. State Reachability for DEVS Models. In: *Proceedings of Argentine Symposium on Software Engineering* (2005), Rosario, Argentina, August 2005: 267–277.

33. Hong KJ and Kim TG. Timed I/O test sequences for discrete event model verification. In: *Artificial Intelligence and Simulation*, 2005, LNCS, volume 3397, pp.275–284.
34. Labiche Y and Wainer G. Towards the verification and validation of DEVS models. In: *Proceedings of the 1st Open International Conference on Modeling & Simulation*, Clermont-Ferrand, France, 2005, pp.295–305.
35. Saadawi H and Wainer G. Verification of real-time DEVS models. In: *Proceedings of SpringSim'09*, San Diego, CA, 2009.
36. Saadawi H and Wainer G. Rational time-advance DEVS (RTA-DEVS). In: *Proceedings of 2010 Symposium on Theory of Modeling and Simulation (DEVS'10)*, Orlando, FL, 2010.
37. Miller J. Decidability and complexity results for timed automata and semi-linear hybrid automata. In: *Hybrid Systems: Computation and Control*, 2000, LNCS; volume 1790, pp.296–309.
38. Aceto L, Anna I, Larsen KG, and Jiri S. Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge, UK, August 2007.
39. Bowman H and Gomez R. *Concurrency theory: calculi and automata for modelling untimed and timed concurrent systems*. 1st ed. Springer, 2006 London, UK.
40. Gupta V, Henzinger TA and Jagadeesan R. Robust timed automata. *Hybrid Real Time Syst* 1997; 1201: 331–345.
41. De Wulf M, Doyen L and Raskin J-F. Almost ASAP Semantics: From Timed Models to Timed Implementations. *Hybrid Systems: Computation and Control*, 2004, LNCS, volume 2993, pp.296–310.
42. Puri A. Dynamical properties of timed automata. *Discrete Event Dyn Syst* 2000; 10: 87–113.
43. De Wulf M, Doyen L and Markey N. Robustness and implementability of timed automata. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems* 2004, LNCS, volume 3253, pp.359–374.
44. Yu YH and Wainer G. eCD++: an engine for executing DEVS models in embedded platforms. In: *Proceedings of the 2007 Summer Computer Simulation Conference*, Society for Computer Simulation International, San Diego, CA, 16–19 July 2007, pp.323–330.
45. Bérard B, Diekert V, Gastin P, et al. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae* 1998; 36: 145–182.
46. Bouyer P and Chevalier F. On conciseness of extensions of timed automata. *J Automata Lang Combinatorics* 2005; 10: 393–405.
47. Wikipedia. 'DEVS Behavior', http://en.wikipedia.org/wiki/Behavior_of_Coupled_DEVS (accessed August 2009).

Author biographies

Hesham Saadawi is currently a PhD candidate at the School of Computer Science, Carleton University, in Ottawa, ON, Canada. He holds the Bachelor of Mechanical Engineering from Ain Shams University 1990, Diploma of Computer and Information Science, ISSR, Cairo University 1992, and master of Computer Science, Carleton University 2003. He has been a reviewer for multiple M&S conferences, and a contract instructor at School of Computer Science, Carleton University. He also has extensive software design and development experience with projects within commercial and public sectors.

Gabriel A Wainer (SMSCS, SMIEEE) received his MSc (1993) and PhD degrees (1998, *with highest honors*) from the University of Buenos Aires (UBA), Argentina and the Université d'Aix-Marseille III, France, respectively. After being Assistant Professor at the Computer Science Department of UBA, he joined the Department of Systems and Computer Engineering at Carleton University, where he is now an Associate Professor. He has been a visiting scholar at ACIMS (The University of Arizona); LSIS/CNRS, University of Nice and INRIA (Sophia-Antipolis), France. He is the author of three books and over 240 research articles; he edited four other books, and helped organize over 110 conferences, including being one of the founders of SIMUTools and SimAUD. He is also the Chair of the Ottawa Center of The McLeod Institute of Simulation Sciences. He is Special Issues Editor of *Simulation*, member of the Editorial Board of *Wireless Networks* (Elsevier), the *Journal of Defense Modeling and Simulation*, and the *International Journal of Simulation and Process Modelling* (Inderscience). Prof Wainer is the Vice-President Publications, and was a member of the Board of Directors of the SCS. He is the head of the Advanced Real-Time Simulation lab, located at Carleton University's Centre for advanced Simulation and Visualization (V-Sim). He has been the recipient of various awards, including the IBM Innovation Award, SCS Leadership Award, and various Best Paper awards. He has been awarded Carleton University's Research Achievement Award (2005–2006), the First Bernard P. Zeigler DEVS Modeling and Simulation Award, and the SCS Outstanding Professional Award (2011). Further information can be found at http://www.sce.carleton.ca/faculty/wainer.

Appendix A: Closure-under-coupling property for the RTA-DEVS

A coupled RTA-DEVS model M can be simulated with an equivalent atomic RTA-DEVS model, whose behavior is defined as follows:[47]

$$M = < X,\ Y,\ S,\ s_0,\ \delta_{\text{ext}},\ \delta_{\text{int}},\ \lambda,\ ta >$$

- X and Y are the input and output event sets, respectively. X is the set of all input events accepted and Y is the set of all output events generated by coupled model M.
- $S = \times_{i \in D} V_i$ is the model state. It is expressed as the Cartesian product of all component states, where V_i is the total state for component i, $V_i = \{(s_i, t_{ei}) | s_i \in S_i, t_{ei} \in [0, ta(s_i)]\}$. Here, t_{ei} denotes the elapsed time in state s_i of component i, and S_i is the set of states of component i.
- $s_0 = \times_{i \in D} v_{0i}$ is the initial system state, with $v_{0i} = (s_{0i}, 0)$ the initial state of component $i \in D$.
- $ta : S \to \mathrm{T}$ is the time-advance function. It is calculated for the global state $s \in S$ of the coupled model as the minimum time remaining for any state among all components, formally

$ta(s) = \min\{(ta(s_i) - t_{ei}) | i \in D\}$, where $s = (\ldots, (s_i, t_{ei}), \ldots)$ is the global total state of the coupled model at some point in time, s_i is the state of component i, and t_{ei} is elapsed time in that state.

- $\delta_{ext} : X \times V \rightarrow S$ is the external transition function for the coupled model, where V is the total state of the coupled model: $V = \{(s, t_e) | s \in S, t_e \in [0, ta(s)]\}$.
- $\delta_{\text{int}} : S \rightarrow S$ is the internal transition function of the coupled model.
- $\lambda : S \rightarrow Y$ is the output function of the coupled model.

Appendix B: Verification trace for the elevator-controller model

Composite system state	System variable values
(Stopped,StdByStop,S1)	• sensor = 1; button = 2 • stopValue = direction = b = 0 • ElevatorController.cur_floor = 1 • ElevatorController.floor = 1 • z in [0,5]
buttonc: User_sensor_input -> ElevatorController (Stopped,Moving,S2)	• sensor = 1; button = 3 • stopValue = 0; direction = 2; b = 0 • ElevatorController.cur_floor = 1 • ElevatorController.floor = 3 • z in [0,5]
move: ElevatorController -> Elevator (Rising, StdByMov, S2)	• sensor = 1; button = 3 • stopValue = 0; direction = 2; b = 1 • ElevatorController.cur_floor = 1 • ElevatorController.floor = 3 • z in [0,10)
sensorc: User_sensor_input -> ElevatorController (Rising, Aux1, S3)	• sensor = 1; button = 3 • stopValue = 0; direction = 2; b = 1 • ElevatorController.cur_floor = 1 • ElevatorController.floor = 3 • z in [0,10)
ElevatorController (Rising, StdByMov, S3)	• sensor = 1; button = 3 • stopValue = 0; direction = 2; b = 1 • ElevatorController.cur_floor = 1 • ElevatorController.floor = 3 • z in [0,14)
sensorc: User_sensor_input -> ElevatorController (Rising, Aux1, S4)	• sensor = 2; button = 3 • stopValue = 0; direction = 2; b = 1 • ElevatorController.cur_floor = 2 • ElevatorController.floor = 3 • z in [0,14)
ElevatorController (Rising, StdByMov, S4)	• sensor = 2; button = 3 • stopValue = 0; direction = 2; b = 1 • ElevatorController.cur_floor = 2 • ElevatorController.floor = 3 • z in [0,18)
sensorc: User_sensor_input -> ElevatorController (Rising, Aux1, S5)	• sensor = 3; button = 3 • stopValue = 0; direction = 2; b = 1 • ElevatorController.cur_floor = 3 • ElevatorController.floor = 3 • z in [0,18)
ElevatorController (Rising, StdByMov, S5)	• sensor = 3; button = 3 • stopValue = 0; direction = 2; b = 1 • ElevatorController.cur_floor = 3 • ElevatorController.floor = 3 • z in [0,27)
ElevatorController (Rising, Stopped, S5)	• sensor = 3; button = 3 • stopValue = direction = 0; b = 1 • ElevatorController.cur_floor = 3 • ElevatorController.floor = 3 • z in [0,27)
ElevatorController (StopUp, Stopping, S5)	• sensor = 3; button = 3 • stopValue = 1; direction = 0; b = 1 • ElevatorController.cur_floor = 3 • ElevatorController.floor = 3 • z in [0,27)