On the Verification of Hybrid DEVS Models

Hesham Saadawi[1] Gabriel Wainer[2]

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

Hybrid systems (those where continuous and discrete phenomena interact) can be found in many natural and artificial systems. For example, real-time embedded systems usually include discrete-event controllers interacting with a continuous plant. Verifying such real-time systems for correctness is of utmost importance, as results of incorrect behavior could be catastrophic. Although Modeling and Simulation is one of most used tools to study such hybrid real-time systems, they lack of a robust formal mechanism for checking the correctness of the system. Here, we introduce a new verification method, based on RTA-DEVS, hybrid Timed Automata and the QSS method, which allows verifying real-time hybrid systems modeled by DEVS formalism.

I. 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. Embedded Real-Time (RT) software systems are increasingly used in mission critical applications, where a failure of the system to deliver its function can be catastrophic. For instance, if we consider the design decisions made for an aircraft autopilot, or a controller for an automated factory, we need to obtain system responses within well-defined deadlines. Great care must be taken when developing RT systems to guarantee their functional correctness along with non-functional correctness such as timing constraints.

Because of the growing complexity of RT systems and their need for high reliability, RT software development is still time consuming, error prone, and expensive, requiring a difficult and costly development effort with no guarantee for a bug-free software product. Many techniques have been proposed to check correctness of RT software. Current RT Engineering methodologies use *modeling* as a method to study and evaluate different system designs before building the real application. In this way, RT systems can have very high predictability and reliability. To do so, a designer must abstract the physical system at hand and build a model for it, then combine this with a model of the proposed controller design. Then, different techniques can be used to reason about these models and gain confidence in its correctness. *Informal methods* usually rely on extensive testing of the systems based on system specification [1]. These methods have limitations because 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 practical testing methods [2]. However, we cannot guarantee full coverage of all possible execution paths in software, thus leaving us with limited confidence in about correctness. These informal techniques can reveal errors, but cannot prove model's correctness.

Formal analysis is growing as an alternative, as it allows the full verification of the software components, which can be proved as being free of errors. In last decades, these techniques have matured, and they have been used in some industrial capacity [3]. Nevertheless, these formal methods are still constrained in their application, as they do not scale up well. Likewise, the designers need a high level of expertise in applying these techniques. Another drawback of formal techniques is their need to be applied to an *abstract model of the real system*. However, in doing so, what is being verified is not the target system. Even if the abstract designed model is proven correct, there is a risk that some errors creep during the development process through the manual implementation of the design into executable code [1] .

Formal verification techniques are of two main types, *deductive* or *algorithmic* [13]. Deductive techniques rely on representing the system and its specification with logic rules, and then try to deduct a proof of system correctness. Algorithmic techniques rely on modelling the system in a graphical form, and coding specifications in logical queries. Then, an algorithm for *reachability analysis* searches the graph space for nodes reachable from an initial system configuration that satisfies the specification queries. This method is also called *model checking*. New theoretical advances in model checking allow guaranteeing certain properties about models of such systems using a formal approach. Model checking techniques can be automated, and Timed Automata (TA) theory [4], in particular, has provided many practical results in this area. However, there is still a gap between a system model that is checked as an abstract entity, and the actual system implementation code run on the target platform. Errors can creep into the final implementation (when the programmer translates requirements captured in TA into code). Also, though formal methods have showed promising results, they are difficult to apply, and do not scale up well.

A different approach considers using Modeling and Simulation (M&S) to gain confidence about the model correctness. The use of M&S is not new, and systems Engineers often rely on these methods in order to improve the study of experimental conditions during model definition. 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 or even impossible. Nevertheless, no practical, automated 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 are.

If the models used for M&S are formal, their correctness would also be verifiable, and a designer could see the system evolution and its inner workings even before starting a simulation [5]. Another advantage of executable models is that they can be deployed to the target platform, thus giving the opportunity to use the model not only for simulations, but also as the actual implementation deployed on the target hardware. This avoids any new errors that would appear during the implementation from transformation of the verified models into an implementation, thus guaranteeing a high degree of correctness and reliability.

The objective of this paper is to introduce a methodology enabling formal verification of hybrid RT systems modeled with DEVS formalism. This methodology would add the benefit of rigorous formal correctness check to the current practice of simulating RT hybrid systems. The main contribution is to show a transformation method from continuous systems modeled with QSS to an equivalent TA model. This method would deal with issues of infinite continuous state space, abstraction and preservation of critical model properties through the transformation.

II. RELATED WORK

A. Hybrid DEVS models

Hybrid models are important particularly in modeling control systems where the controlled environment obeys the laws of physics, while the controller is a digital discrete system. The study of such systems requires the verification of the resulting hybrid system.

A Major problem in verification of hybrid systems is the lack of a unified theory to model and solve both continuous and discrete components together [6]. As a result, modeling and simulation is still one of the most useful methods to verify this kind of systems [7][8][9]. Hybrid systems simulation was enabled within DEVS formalism by using a method, called Quantizes State Systems (QSS) that will be covered in section B, which allows modeling continuous components [10][11][12]. However, simulation does not guarantee the absence of defects from the system under study. Simulation verifies the system for particular scenarios chosen by the system tester. Formal methods can then be used to provide an absence-of-defects guarantee. In doing so, a hybrid system needs to be modeled and verified within a formal framework.

To use the algorithmic method (model checking through reachability analysis) to verify hybrid systems, the focus would be to find a suitable *finite abstraction* of the hybrid system that can be verified and hence reachability algorithm is guaranteed to terminate. Different types of labeled transition systems were proposed to model hybrid systems abstractions including Petri Nets [14], hybrid automata and TA [13].

However, as Henzinger et al. shows in [16], Hybrid TA verification through reachability analysis is not decidable in general. For this reason, recent research has concentrated on modeling the hybrid system in some form with a decidable verification such as TA. In doing so, a technique must be used to model the continuous component in a discrete finite form. As continuous system variables are real values, their state space could be infinite. An approximation to a finite representation is needed to enable the decidability and termination of reachability analysis. Many techniques have been proposed to approximate continuous-time systems into a discrete representation of TA [17] [18][19][20].

This paper uses another innovative technique to represent the continuous system in discrete format using DEVS formalism. Although DEVS is a discrete-event system specification, some methods are used to represent continuous systems in a discrete format that can be simulated with DEVS. One of these methods is Quantized State Systems (QSS) method [11]. This method enables modeling and simulation of hybrid systems with DEVS formalism.

B. Quantized State Systems (QSS) method

In this section, we introduce the QSS method [10][11]. The QSS is an approximation method to model and simulate continuous systems, which are usually modeled with Ordinary Differential Equations (ODE) and Algebraic Equations. Obtaining a detailed description of system behavior entails solving these equations simultaneously. In doing so, many different techniques of numerical integration are used to solve ODEs such as Euler, Runge-Kutta, etc. These methods approximate the solution of ODEs, and they limit the error to an acceptable range based on the choice of its discrete integration step. All these methods rely on discrete-time integration of ODEs. In this way, time is allowed to progress in small steps, and at each step, an approximation is computed for ODEs solution. When a system modeled by ODEs has a discontinuity (i.e. sudden jumps in its variables values with regard to time), the numerical integration method may produce unacceptable errors [23]. These kinds of discontinuity are normal properties in hybrid systems, which can be seen as operating in different modes each described with a specific ODE. An example of such a system would be a heating system with an on-off thermostat switch.

A different method for approximation is called Quantized State Systems QSS, a quantization-based method that models hybrid systems as discrete-event systems and not as discrete-time. This solves the above problem around discontinuities while solving hybrid system as discussed in [11]. Consider a continuous system modeled by some time-invariant Ordinary differential equation (ODE) and it is in its State Equation System (SES) representation:

$$x^{\cdot}(t) = f[x(t), u(t))] \qquad (Eq.II.1)$$

Where $x(t) \in \boldsymbol{R}^{n}$ represents the system state vector and $u(t) \in \boldsymbol{R}^{m}$ represents an input vector, which is a known piecewise constant function, and $\boldsymbol{R}$ is the set of Real numbers. With the

QSS method, we simulate an approximate system, which is called Quantized State System:

$$\dot{x}(t) = f[q(t), u(t)] \qquad (Eq.II.2)$$

Where $q(t)$ is a vector of quantized variables which are obtained with quantization function q from the state variables $x(t)$. Each component of $q(t)$ may be related with the corresponding component of $x(t)$ by a hysteretic quantization function, as given in [11]. A hysteresis function approximates a continuous linear function $x_i(t)$ by outputting a number of discrete levels. Each level is called a quantization level Q_i. The difference between two successive quantization levels (Q_i, Q_{i+1}) is called the *quantum* (dq) and it is usually constant. The crossing of the continuous function to a quantization level generates an output.

An example of simulating a continuous system with QSS can be shown by using the exponential decay formula which is modeled as follows, using an ODE:

$$dx/dt = -x(t) \qquad (Eq.II.3)$$

Which has the analytical solution $x(t) = e^{-t}$, with the initial condition x(0) =1. Figure 1 shows a graph of the exact analytical solution of the exponential decay formula $x(t) = 10\ e^{-t}$ where $x(0) = 10$.

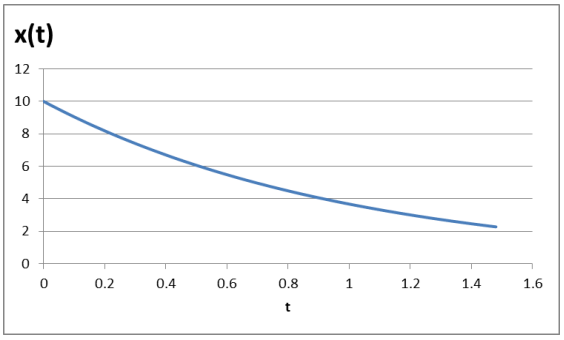

Figure 1: Exact solution for exponential decay formula.

The solution of *(Eq.II.3)* is approximated in discrete-event form by the following QSS DEVS model:

$$AMD = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \qquad (Eq.II.4)$$

$X = \emptyset$; $S=\{s \mid s=(q,\sigma)\}$; $ta(s) = ta(q,\sigma) = \sigma$

$\delta_{int}(s) = \delta_{int}(q,\sigma) = (q\text{-}0.1,\ 0.1/q)$; $\lambda(q,\sigma) = q$

q: is a quantized variable related to the $x(t)$ system variable by a quantization function.

Figure 2 shows the quantized representation of the decay formula as a result of simulating this QSS model.

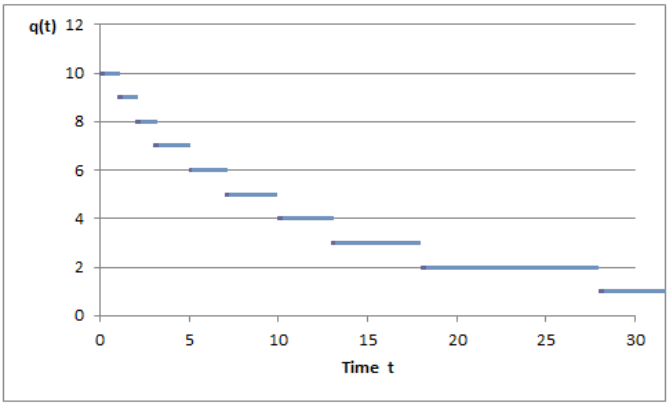

Figure 2: Quantized representation of Exponential Decay.

III. Mapping QSS to TA

The main contribution of this paper is a novel approach to transform QSS model to TA, and hence enabling formal verification of hybrid models within DEVS formalism. To transform a DEVS model *(Eq.II.4)* to a TA, we need to solve the following issues:

1. The TA variables can only be of bounded integer type, in order to guarantee the finiteness of state space and hence the termination of the reachability algorithm. However, in QSS, state variables are real numbers and thus have infinite values.
2. Time (σ) of next quantum event is approximated to an integer number. However in doing so we need to preserve original behavior of QSS and hence the properties we need to formally verify.

The first issue is handled by converting rational real numbers to integers by multiplying all values by the least common multiple of all the denominators. For any irrational values, we can use a technique we introduced in [29]. For the second issue, we use abstraction by over-approximation [26]. With this technique, we approximate the real value of the event time t_i with a bounded time interval such that $t_c \in [T_L, T_H]$. This interval is bounded by floor(t_i) and ceiling(t_i) respectively. This guarantees that the resulting TA would include all possible event timings in that interval. Hence, verification of TA would apply to the real value produced in QSS model, as proved in [21] for robust timed automata. figure 3 shows a transformation from QSS to TA for the QSS model of *(Eq.II.4)*. Over-approximation also preserves safety properties, i.e. any proof of a safe over-approximation implies the original system is also safe, however as over-approximation contains more behaviors than the original system, it verification may produce safety property violations that does not exist in the original system. In this case, any violation scenario should also be checked against the original system to confirm it is a real safety violation [22].

The semantics of the QSS models a loop as follows:

1. Initial values are assigned to q=1 and $\sigma=0$, s_o=(1,0).
2. After a time elapse of $e=\sigma$, the output function is triggered to send value q, and δ_{int} is triggered to calculate the next state, composed of new q and σ, s = (q-0.1, 0.1/q).
3. Repeat step 2 in the loop until s = (1,10).

To obtain a TA that contains all the behavior of QSS model, we need to a simulation relation with QSS model (i.e., TA simulates QSS). To do so, each state in QSS would be simulated by a corresponding state in TA and each target state in QSS is simulated by a corresponding target state in TA. Inspecting the TA model figure 3, we can see the following simulation to the QSS model after multiplying by scale of 10 to remove fractional parts:

1. The TA starts in the initial state S1, and moves to S2. On this initial transition, the total state variables are initialized as sigmaL=0, sigmaH=0, and q=10.
2. After time elapse t where $\lfloor\sigma\rfloor \le t \le \lceil\sigma\rceil$, the transition S2→S3 is executed, calculating new value of $q=q\text{-}1$.
3. S3 is a committed state, causing transition S3→S2 to be taken immediately, calculating new values for sigmaL =

$\lfloor\sigma\rfloor$ and sigmaH = $\lceil\sigma\rceil$. The total state at S2 is (q-1, $\lfloor\sigma\rfloor \le t \le \lceil\sigma\rceil$).

4. Steps 2 and 3 are repeated until the total state = (q=1, sigmaL=sigmaH=10).

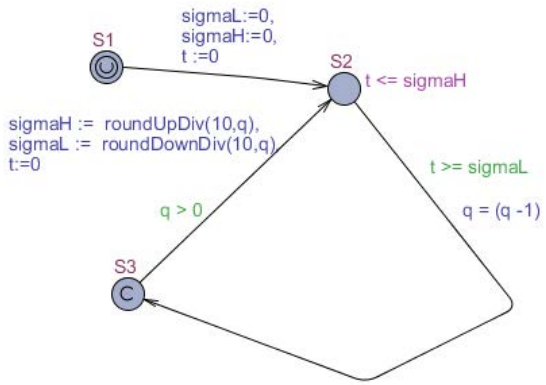

Figure 3: TA representation of QSS exponential decay.

This shows that the TA of figure 3 simulates the above QSS model.

A. *Verification of discrete DEVS models*

There have been several proposals to verify discrete DEVS models. In this paper, we use the methodology we introduced in [27][28][29]. The idea is to transform DEVS models formally into behaviorally equivalent TA, which are then verified against system requirements by TA model checking tools (such as UPPAAL). For our purpose of verification, we assume all to-be-verified DEVS models are finite in terms of the number of states, input events, and output events. For a coupled DEVS model on the form

CM = $<X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, SELECT>$ *(Eq. III.1)*

The algorithm below is used to model it with TA:

1. Declare a set of clocks C = $\{x_i \mid 1 \le i \le |D|\}$, where i is the index of component $d \in D$
2. Convert the rational numbers defined in RTA-DEVS to integers, as described in section II.B.
3. Define a TA location for each RTA-DEVS state and define location invariant if necessary:

   For each $d \in D$ *do*

   $N_d = \{l_j \mid \exists s_j \in S_d, d \in D\}$ *//component N of the TA corresponding to component d*

   $\beta(C)_{lj} = \{ x_i \le ta(s_j) \mid \exists s_j \in S_d, 1 \le i \le |D|, ta(s_j) < \infty \}$
4. Define a set of channels for communication between TA components: $H = \{a_j \mid j \in IC\}$
5. Define set of TA transitions for RTA-DEVS internal transitions:

   set $E = \emptyset$; *//Initialize the set of TA Transitions*

   For each $d \in D$ *do*

   For each $s_j \in S_d$ *do*

   If ($ta(s_j) < \infty$ && $\delta_{int}(s_j) = s_k$ && $\lambda(s_j) = a$) *then*

   $E = E \cup (l_j, x_d \ge ta(s_j), a!, x_d := 0, l_k)$;
6. Add TA transitions corresponding to RTA-DEVS external transitions:

   For each $d \in D$ *do*

   For each $s_j \in S_d$ *do*

   If ($\delta_{ext}(s_j, a, cond(e)_m) = s_{k\,m}$) *then*

   $E = E \cup (l_j, cond(x_d)_m), a?, x_d := 0, l_{km})$;

// whenever: $\delta_{ext}(s_j,a,e) = s_{k1}$ $cond(e)_1$: $0 \le e < c_1$

// $= s_{k2}$ $cond(e)_2$: $c_1 \le e < c_2$

// ... $= s_{km}$ $cond(e)_m$: $c_{m-1} \le e < c_m$

// we define $cond(x_d)_m = c_{m-1} \le x_d < c_m$, *where* $cond(x_d)_1$, *// cond*$(x_d)_2, \ldots$, *cond*$(x_d)_m$ *are convex polyhedra (i.e. described // by a finite number of linear inequalities)*

IV. Case Study: A Hybrid Elevator Control System.

To show our methodology for verification of hybrid DEVS models, we modified an example originally introduced in [27]. That example defines an elevator system composed of an Elevator, the Elevator Controller and an Environment that represents a user pressing different buttons. The elevator controller interacts with the user to receive button requests from each floor. Then, it makes the elevator move to respond to the user requests. This is an example of a (soft) Real-Time System with safety and bounded response time requirements. This example was transformed from RTA-DEVS to TA, and verified to work correctly in UPPAAL. A summary of this case study, originally presented in [28], is given below.

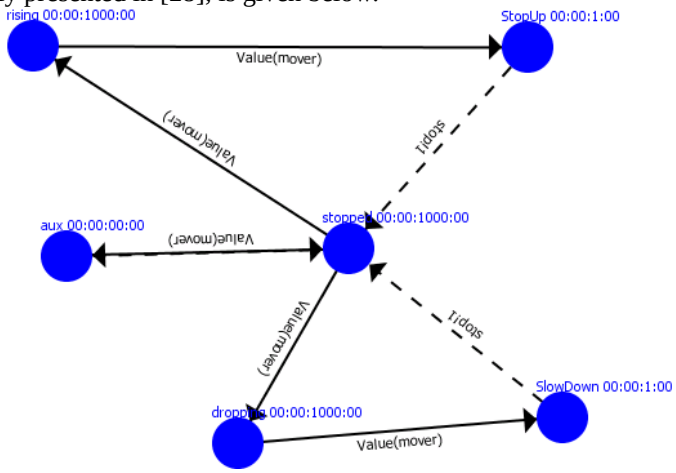

Figure 4: Elevator RTA-DEVS Model.

The elevator model shown in figure 4 represents the different states of the elevator movement and transitions between these states. This is an abstract model of the elevator where some details like door operation, floor display, etc. have been ignored (as we only interested to control the elevator movement). The elevator starts in the *stopped* state and waits for the controller commands to move (satisfying a button request from the user). The controller takes the decisions for direction, start and stop of the motors.

The elevator DEVS graph model in figure 4 has 5 external transitions, shown with solid arrows; and three internal transitions, shown with dotted arrows. Note that an external transition is enabled whenever the expression on that transition evaluates to *true* in RTA-DEVS model.

The expression *Value(mover)* evaluates to *true* whenever the elevator model receives a value in *mover* variable equals to *1*. This expression is translated to a channel reception *move?* as shown in TA model in figure 5. By following the algorithm in section III.A, we obtain the TA model shown in figure 5. Details of the transformation and interaction of this model with other components are described in [28] and [29].

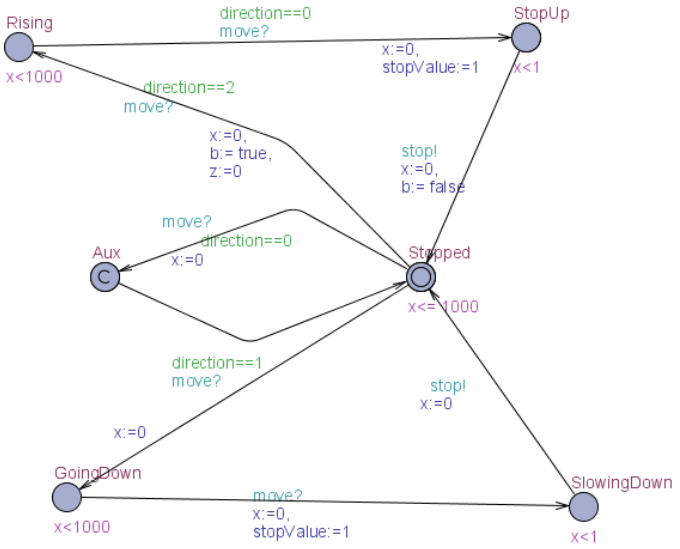

Figure 5: Elevator TA model.

The elevator controller is also responsible to interact with the user, and to send commands to the elevator to satisfy the user requests. The controller RTA-DEVS model is shown in figure 6, represented in DEVS graphs notation. We abstracted the behaviour of the controller to being in one of possible 6 states, representing the elevator's direction and acceleration. *StdBy-Stop* represents the elevator in a complete stop and ready to move for any coming requests. *Moving* is 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 receiving sensor signals to decide when to stop the elevator. *Aux* is an intermediate state with an instantaneous internal transition (to enable the test of the sensor value on the external transition with the function *equal(sensor,floor)*). *Stopped* corresponds to the controller deciding to send a signal to the elevator to slow in preparation to stop. *Stopping* corresponds to the controller waiting for the elevator to get into complete stop and sending a stop signal to the controller.

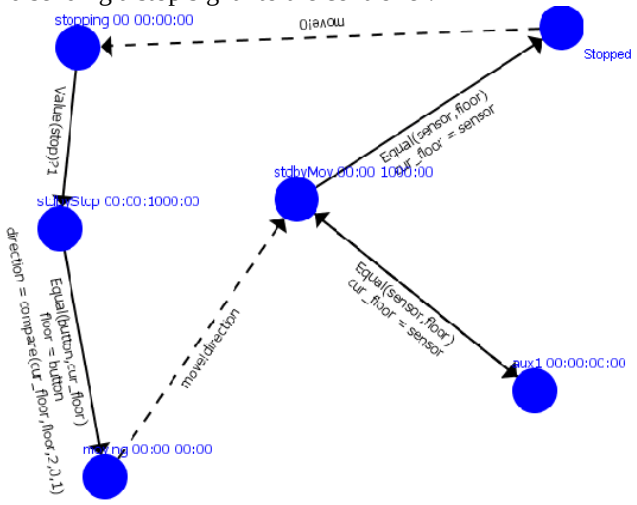

Figure 6: Elevator Controller Model as DEVS Graphs.

After applying the transformation steps in [28] to the controller RTA-DEVS model, we obtain TA shown in figure 7.

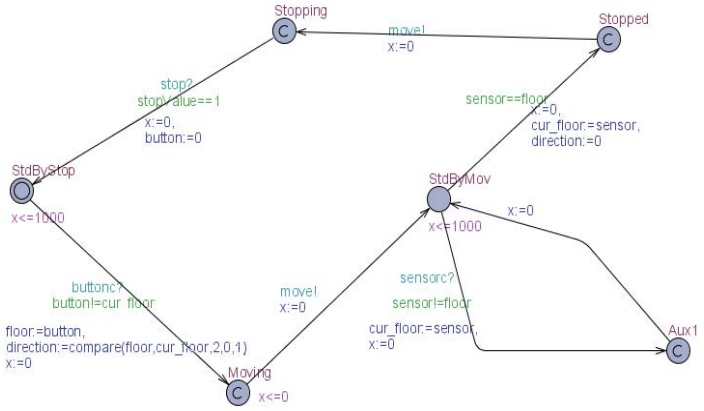

Figure 7: TA Controller model in UPPAAL.

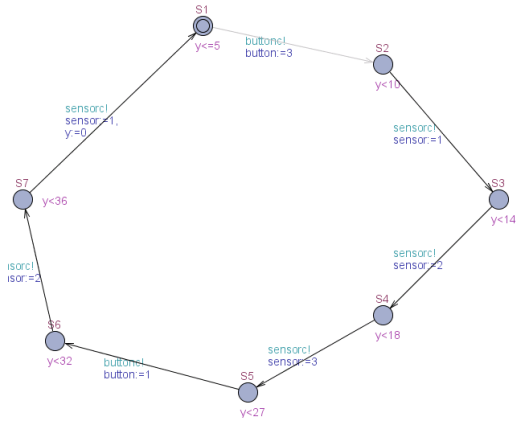

Figure 8: Environment inputs (Button and Sensor).

A. *Continuous Model of Elevator Braking*

In this paper we extend the case study by introducing a model of the elevator de-acceleration motion due to applying the brakes that can be described by a differential equation as:

$$\frac{dv}{dt} = a \qquad (Eq.IV.1)$$

Where v is the elevator speed, and a is a constant acceleration which would be a negative value in case of braking or a positive value for moving out of rest. Figure 9 shows a negative acceleration representing braking action on the elevator.

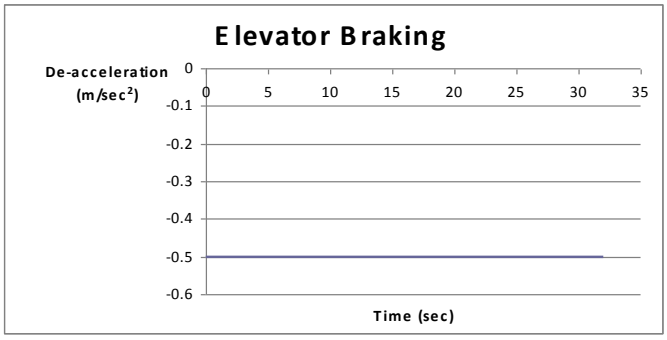

Figure 9: Elevator braking. De-acceleration: 0.5 m/s^2.

The speed of the elevator at any point in time t can then be obtained as:

$$v = at + v_i \qquad (Eq.IV.2)$$

With v_i the initial elevator speed before applying the brakes. Figure 10 shows the change in elevator speed during braking. The elevator brakes are applied while its speed is 4 m/s, and the subsequent reduction of speed is shown until complete stop.

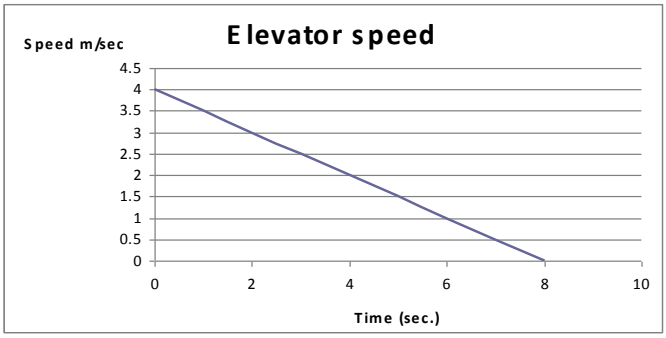

Figure 10: Elevator speed under braking.

With this continuous model of the elevator motion, our overall elevator model becomes a hybrid between discrete and continuous components. To simulate and then verify this hybrid model, we must obtain discrete representation of the elevator braking model. This can be done within DEVS formalism using QSS method. A QSS model to represent elevator speed under braking (Eq.IV.2) can be described as follows:

$AM_D = <X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta>$ (*Eq.IV.3*)

$X = \emptyset;$ $S=\{s \mid s=(q, \sigma)\};$

$ta(s) = ta(q, \sigma) = \sigma;$ $\delta_{int}(s) = \delta_{int}(q, \sigma) = (q-0.5, 0.5/a)$

$\lambda(q,\sigma) = q$

q: is a quantized variable related to $v(t)$ system variable by quantization function as shown in section II.B. Figure 11 shows the quantized output of this QSS model.

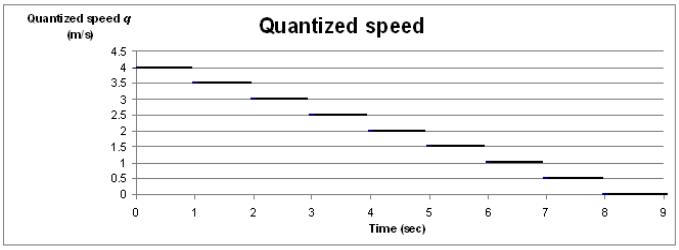

Figure 11: Quantized braking-elevator speed.

To enable the formal verification within UPPAAL to the combined hybrid elevator model, we transform the QSS model of (Eq.IV.3) to an equivalent TA model as per the method shown in III resulting in the TA shown in figure 12.

On this model, location *S1* represents the initial elevator speed (4 m/s), the quantum value is dQ = 0.5 m/s, and *sigma* represents the time interval between the outputs of two successive quantized values. When this model receives a synchronization event on the *applyBrake* channel, it moves to the state *S2* to start the main loop *S2-S3-S2-...*, calculating the next quantized output q and the next values of *sigmaL* and *sigmaH*. When the quantized speed q reaches zero, the model moves back to *S1* and waits for another *applyBrake* event.

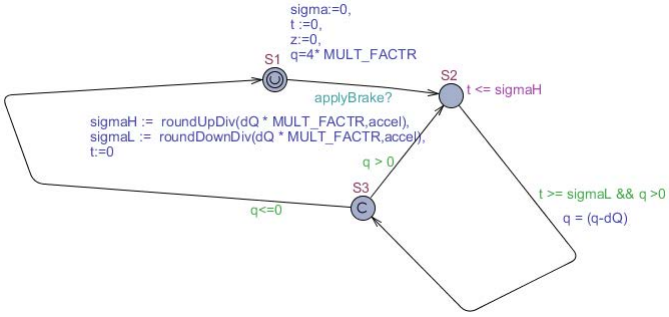

Figure 12: TA model of braking elevator motion.

Sigma is calculated as per the definition of (Eq.IV.3) within δ_{int} function. However, sigma is defined as a real value in QSS model. Therefore, sigma value is over-approximated with an integer interval $\sigma \in [sigmaL, sigmaH]$ as described in section III. Thus, TA model behavior includes all trajectories (q,t) where q is the quantized state, and $t \in [sigmaL, sigmaH]$. In addition, since TA deals with only integer type variables, we multiplied all values of the QSS model by a factor of 100 to convert all fractions to integers. This multiplication is done on all TA components to scale the time evenly of all component models.

Another modification to the original elevator model was necessary to enable the elevator communicate with the QSS model through the synchronization channel *applyBrake*, and for the elevator to be in *Stopped* location only when the quantized speed value q reaches zero. The modified elevator model is shown in figure 13.

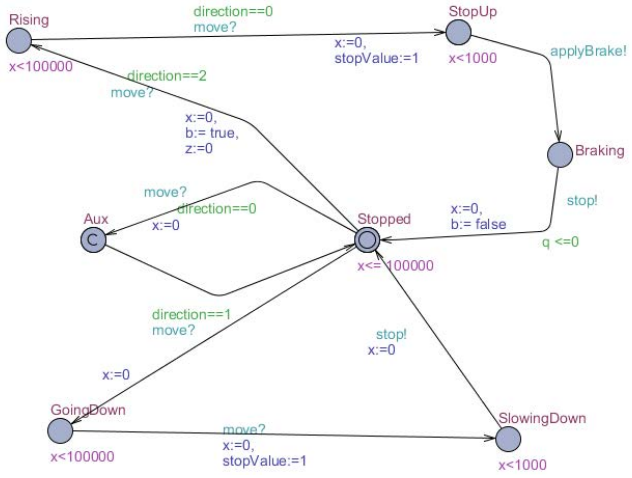

Figure 13: Modified elevator TA model.

In this model, whenever the elevator receives the command from the controller to stop, it synchronizes with the braking elevator motion model with sending *applyBrake!*. This would start the computation of quantized speed output by the TA shown in figure 12. The elevator waits in state *Braking* for the quantized speed q value to reach zero. Once elevator speed reaches zero, the transition from *Braking* to *Stopped* would be enabled and executed, then the elevator sends *stop!* to the elevator-controller. The rest of the model executes exactly as shown before. This hybrid model allows the designer to verify the control system with different parameters of the elevator physical system such as different braking values of de-

accelerations, different elevator initial speeds, or other parameters in a more detailed QSS model. This is an important addition to the elevator system verification as relevant physical factors to the controller performance can be identified and formally verified during design phase.

V. Verification Examples

In [27][28], we showed how to verify a number of desired properties for the DEVS model such as deadlock freedom, bounded response time, and safety properties for the elevator coupled model. We used Computational Tree Logic (CTL) to construct queries with the requirements and submitted it to UPPAAL to get an answer and hence verify that requirement. We check one of these required properties here with the hybrid system modeled in the previous section. We start with an elevator model as the one described in figure 12 with its speed decrease as in figure 11.

One such requirement is the freedom of deadlocks expressed in CTL as `A[] not deadlock`. This means for all paths, there should be no deadlocks.

After running the checker, it shows that this property is satisfied, i.e. there is no deadlock in the DEVS model:

```
(Academic) UPPAAL version 4.0.13 (rev. 4577), September 2010 -- server.
A[] not deadlock
Property is satisfied.
```

In another example, a second elevator with braking deacceleration equals -0.12 m/s^2 has its continuous and quantized speeds described by graphs shown in figure 14 and figure 15.

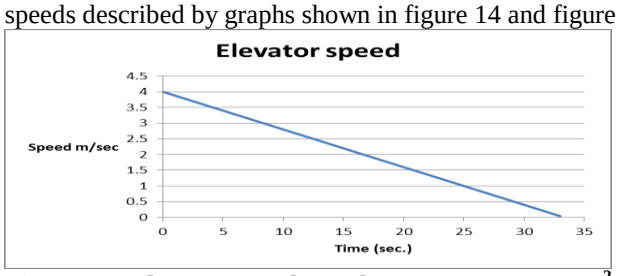

Figure 14: Elevator speed, acceleration= - 0.12 m/sec^2.

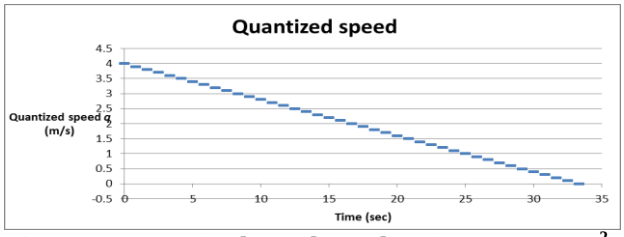

Figure 15: Quantized speed, acceleration= - 0.12 m/s^2.

To verify the elevator-controller with this second version, we changed the parameters of the elevator acceleration in its TA model and re-verified again. The results are shown as follows:

```
A[] not deadlock
Property is not satisfied.
```

In this case, the time needed for the elevator to stop is approximately 33 seconds. This would contradict with the user requirements model shown in figure 8. In this model, the user expects the elevator to reach 3rd floor within 27 seconds at most, and after this time the requirement for the elevator controller to be ready to accept another as shown on the transition *S5* → *S6*. However, the slow-braking elevator would not be able to fulfill the second request in time, hence we have a time lock [31] and the model cannot progress beyond *S5*.

VI. Conclusion

We showed a methodology to verify hybrid DEVS models. This is an extension of previous results verifying discrete DEVS [27][28][29], and this was obtained by using QSS method to model continuous components in a discrete representation. We believe this methodology would enable real-time system designers not only study their systems by simulation, but also to be able to formally verify system requirements within simulation models.

References

[1] M. J. Rehman, F. Jabeen, A. Bertolino, and A. Polini. 2007. "Testing Software Components for Integration: a Survey of Issues and Techniques". *Software Testing,Verification and Reliability* 17(2): 95–133.

[2] R. Gerlich, R. Gerlich, T. Boll. 2007. "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, Georgia.

[3] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, and W. Visser. 2007. "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, pages 120-136.

[4] R. Alur, D. Dill. "Theory of Timed Automata". Theoretical Computer Science, volume 126, pg. 183-235, 1994.

[5] G. Wainer, E. Glinsky, and P. MacSween. 2005. "A Model-Driven Technique for Development of Embedded Systems Based on the DEVS Formalism". In *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, edited by S. Beydeda and V. Gruhn. Springer-Verlag.

[6] M. S Branicky. 2005. "Introduction to Hybrid Systems" D. Hristu-Varsakelis and W.S. Levine (eds.), *Handbook of Networked and Embedded Control Systems*, 91-116. Boston: Birkhauser.

[7] A. Donzé, O. Maler. 2007. "Systematic simulation using sensitivity analysis". In *Proceedings of the 10th international conference on Hybrid systems: computation and control (HSCC'07)* :174-189.

[8] A. Donzé. 2007. "Trajectory-Based Verication and Controller Synthesys for Continuous and Hybrid Systems". PhD thesis, University Joseph Fourier.

[9] A. Donzé, B. Krogh, and A. Rajhans. 2009. "Parameter synthesis for hybrid systems with an application to simulink models". In *Proceedings of the 12th International Conference on Hybrid Systems : Computation and Control (HSCC'09)*, San Francisco, CA, USA, April 13-15, 2009.

[10] E. Kofman, S. Junco. 2001."Quantized State Systems. A DEVS Approach for Continuous Systems Simulation". Transactions of SCS. 18(3): 123-132.

[11] E. Kofman. 2004. "Discrete Event Simulation of Hybrid Systems". SIAM Journal on Scientific Computing 25(5): 1771-1797.

[12] M. Otter, F. Cellier. 1996. The Control Handbook, chapter Software for Modeling and Simulating Control Systems, 415–428. CRC Press, Boca Raton, FL.

[13] S. Kowalewski. 2002. "Introduction to the Analysis and Verification of Hybrid Systems". *Modelling, Analysis, and Design of Hybrid Systems*. Lecture Notes in Control and Information Sciences, 279: 153-171.

[14] G Decknatel, R. Slovák, E. Schnieder. 2002. "Definition of a Type of Continuous-Discrete High-Level Petri Nets and Its Application to the Performance Analysis of Train Protection Systems In S. Engell, G. Frehse, and E. Schnieder (Eds.), *Modelling, Analysis, and Design of Hybrid Systems, Lecture Notes in Control and Information Sciences* 279: 355–367.

[15] T. Henzinger. 1996. "The Theory of Hybrid Automata". *Lecture Notes in Computer Science* 278.

[16] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. 1995. "What's decidable about hybrid automata?". In *Proceedings of the twenty-*

seventh annual ACM symposium on Theory of computing, STOC '95, New York, NY, USA, 373–382.
[17] J. Lunze and J. Raisch. 2002. "Discrete Models for Hybrid Systems. Modelling, Analysis, and Design of Hybrid Systems". *Lecture Notes in Control and Information Sciences*, 279: 67-80.
[18] R. Alur, T.A. Henzinger, G. Lafferriere, G.J. Pappas. 2000. "Discrete abstractions of hybrid systems". *Proceedings of the IEEE*, 88(7): 971-984.
[19] E. Barke, D. Grabowski, H. Graeb, L. Hedrich, S. Heinen, R. Popp, S. Steinhorst, and Y. Wang. 2009. "Formal approaches to analog circuit verification". In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '09)*, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 724-729.
[20] Oded Maler and Grégory Batt. 2008. "Approximating Continuous Systems by Timed Automata". In *Proceedings of the 1st international workshop on Formal Methods in Systems Biology (FMSB '08)*, Cambridge, UK, Jasmin Fisher (Ed.). Springer-Verlag, Berlin, Heidelberg, 77-89.
[21] M. De Wulf, L. Doyen, N. Markey. 2004."Robustness and Implementability of Timed Automata" Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems : 359-374.
[22] B. Berard, , M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen and P. McKenzie. 2001. Systems and Software Verification: Model-Checking Techniques and Tools. Springer Verlag.
[23] M. Otter, F. Cellier. 1996. The Control Handbook, chapter Software for Modeling and Simulating Control Systems, 415–428. CRC Press, Boca Raton, FL.
[24] G. Wainer, L. Morihama, and V. Passuello. 2002. "Automatic verification of DEVS models", In *Proceedings of SISO Spring Interoperability Workshop*, Orlando, FL. U.S.A. March 10-15.
[25] B. P. Zeigler, T. Kim, and H. Praehofer. 2000 . *Theory of Modeling and Simulation*. San Diego, CA: Academic Press, ISBN-10: 0127784551.
[26] L. Aceto, A. Ingólfsdóttir, K. Guldstrand Larsen, J. Srba. 2007. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press.
[27] H. Saadawi, G. Wainer. 2009. "Verification of real-time DEVS models". In *Proceedings of DEVS Symposium 2009*. San Diego, CA, March 22 – 27.
[28] H. Saadawi, G. Wainer. 2010. "Rational time-advance DEVS (RTA-DEVS). In *Proceedings of DEVS Symposium 2010*, Orlando, FL., April 11-15.
[29] Hesham Saadawi and Gabriel Wainer. 2010. "From DEVS to RTA-DEVS". In Proceedings of the 2010 IEEE/ACM 14th International Symposium on Distributed Simulation and Real Time Applications (DS-RT '10). IEEE Computer Society, Washington, DC, USA, 207-210.
[30] J. Bengtsson, W. Yi. "Timed Automata: Semantics, Algorithms and Tools". Lectures on Concurrency and Petri Nets, Vol. 3098. 2004.
[31] H. Bowman, R. Gomez. *Concurrency Theory: Calculi and Automata for Modelling Untimed and Timed Concurrent Systems*. Springer-Verlag London 2006.

VII. APPENDIX

In this section, we give a quick reference to the concepts used in this paper. Hence, we cover a basic introduction to RTA-DEVS and Timed Automata. The reader may consult the referenced material for more detailed background.

A. RTA-DEVS

RTA-DEVS [28] atomic model is defines as in classical DEVS [25]. RTA-DEVS changes the definitions of time advance function *ta* and the external transition function δ_{ext} as follows. The Atomic Rational Time-Advance is defined as:

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

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

Coupled RTA-DEVS model are defined exactly as in classic DEVS Coupled RTA-DEVS models are composed of atomic or other coupled RTA-DEVS models:

$$CM \equiv \left\langle X, Y, D, \{M_i\}, C_X, C_Y, Select \right\rangle$$

X: Set of external input events.

Y: Set of external output events.

D: Finite index of sub-components.

$\{M_i\}$: The set of sub-components. A sub-component may be an atomic or coupled. $i \in D$ is the index of the component.

C_x: Set of input couplings.

C_y: Set of output couplings.

Select: $2^D \rightarrow D$ is a tie-breaking function, which defines how to select an event from asset of simultaneous events.

A coupled RTA-DEVS model M can be simulated with an equivalent atomic RTA-DEVS model, whose behavior is defined as follows [29]:

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

- 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, $V_i = \{(s_i, t_{ei}) \mid s_i \in S_i, t_{ei} \in [0, ta(s_i)]\}$. 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 $v_{0i} = (s_{0i}, 0)$ is the initial state of component $i \in D$.
- $ta: S \rightarrow T$ 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: $ta(s) = \min\{(ta(s_i) - t_{ei}) \mid i \in D\}$ where $s = (\ldots, (s_i, t_{ei}), \ldots)$ is the global total state of coupled model at some point in time, s_i is the state of component i, 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 total state of the coupled model: $V = \{(s, t_e) \mid s \in S,\ t_e \in [0, ta(s)]\}$.
- $\delta_{\text{int}} : S \rightarrow S$ is the internal transition function of the coupled model.

$\lambda : S \rightarrow Y$ is the output function of the coupled model.

UPPAAL added more features to ease the process of modeling using TA as variables, committed states, communication channels, etc. These additions do not change the basic semantics of TA and are described in [30].