Verification of Real-Time DEVS Models

Hesham Saadawi [1], Gabriel Wainer[2]

[1]School of Computer Science, Carleton University, Ottawa, ON . K1S 5B6, Canada, hsaadawi@connect.carleton.ca
[2] Dept. of Systems and Computer Engineering, Carleton University, Ottawa, ON . K1S 5B6, Canada, gwainer@sce.carleton.ca

Discrete Event System Specification (DEVS) has been widely used to describe hierarchical models of discrete systems. DEVS has also been used successfully to model with Real-Time constraints. In this paper, we introduce a methodology to verify Real-Time DEVS models, and describe the methodology by using a case study of a DEVS model of an elevator system. Our methodology applies recent advances in theoretical model checking to DEVS models. The methodology also handles the cases where theoretical approach is not feasible to cross the gap between abstract Timed Automata models and the complexity of the DEVS Real-time implementation by empirical software engineering methods. The case study is a system composed of an elevator along an elevator controller, and we show how the methodology can be applied to a real case like this one in order to improve the quality of such real-time applications.

Keywords: DEVS, Formal methods verification, Real-Time software, Timed automata.

I. INTRODUCTION

Real-time Systems are very advanced computer systems with hardware and software components, which must satisfy "hard" timing constraints. In these highly reactive systems, design decisions can lead to catastrophic consequences; hence, not only correctness is critical, but also the timeliness of the executing tasks. For instance, if we consider an autopilot for an aircraft, or a controller for an automated factory, we need to obtain system responses within well-defined deadlines. Although advances in computing technology made possible to build very advanced RTS, the software development tasks for this kind of systems is still time consuming, error prone, and expensive, requiring a difficult and costly testing effort with no guarantee for a bug-free software product.

Software correctness deals with the verification methods to ensure that a piece of software is doing what it is designed to do. There are many approaches been proposed and used in practice to do this task. Testing has been the main methodology for verifying software components [1], but this technique has its own limitations. In order to guarantee software reliability, we need to apply an exhaustive testing to the software component, using all possible input combinations. 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.

Formal software analysis use is growing as an alternative, as this technique allows full verification of software components to be free of errors. In last decades, these techniques have matured to be used in some industrial capacity for software and hardware correctness verification [3]. Recent trends in formal software analysis can be categorized into three broad types [3], namely Model Checking, Abstract Interpretation, and Deductive Methods. In this paper, we would cover more of the Model Checking approach.

Further, approaches to use formal methods for software correctness vary. There are the Correctness-by-Construction techniques in which to guarantee the final software implementation conformance to its requirements the implementation is generated directly from the model. This generation is done through a series of transformations that are proven formally to preserve the desired properties in the original model. The final generated code, in this case, does not need an 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 [4][5][6]. Our work is a step in this direction.

Formal specification techniques still have limited power when the complexity of the system scales up. Instead, systems engineers have often relied on the use of modeling and simulation (M&S) techniques 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. Nevertheless, no practical, automatable approach exists to perform the transition that exists between the modeling and the development phases, and this often results in model artifacts 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.

New theoretical advances in model checking allow guaranteeing certain properties about models of such systems using a formal approach. These techniques can be automated to improve the work of the software engineer. Timed automata (TA) theory [7], 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 are still difficult to apply when the complexity of the system under development scales up.

In this paper, we propose a methodology that would have a higher correctness checking reliability of the actual code executing in the real-time system. This is achieved by model-checking a DEVS model [8] that would run on the target plat-

form, using a model-based approach in which the user can move simulated models to a target platform that will execute them in real-time. 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 [9],[10]. DEVS models can be transformed to semantically equivalent TA models maintaining its original structure and behaviour [11]. The verified DEVS models would then execute directly on a Real-time DEVS simulator, eliminating the risk of introducing errors in the final system implementation on the target platform.

II. Background

DEVS was originally defined in the '70s as a discrete-event modeling specification mechanism. It was derived from systems theory, and it allows one to define hierarchical modular models that can be easily reused. A real system modeled with DEVS is described as a composite of sub models, each of them being behavioural (atomic) or structural (coupled). Closure under coupling allows coupled models to be integrated to a model hierarchy [8]. Each model is defined by a time base, inputs, states, outputs, and functions to compute the next states and outputs. A DEVS atomic model is formally described by:

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

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

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

CD++ [12] allows defining models following these specifications. 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. This notation (named DEVS Graphs), allows defining atomic models' behavior [13]. Each DEVS graph is translated into an analytical definition. DEVS graphs can be formally defined as [14]:

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

$X_M = \{(p,v) | p \in IPorts, v \in X_p\}$ set of input ports;
$Y_M = \{(p,v) | p \in OPorts, v \in Y_p\}$ set of output ports;
$S = B \times P(V)$ states of the model,
$B = \{ b | b \in Bubbles \}$ set of model states.
$V = \{ (v,n) | v \in Variables, n \in R_0 \}$ 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. States are represented by bubbles including an identifier and a state lifetime. 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.

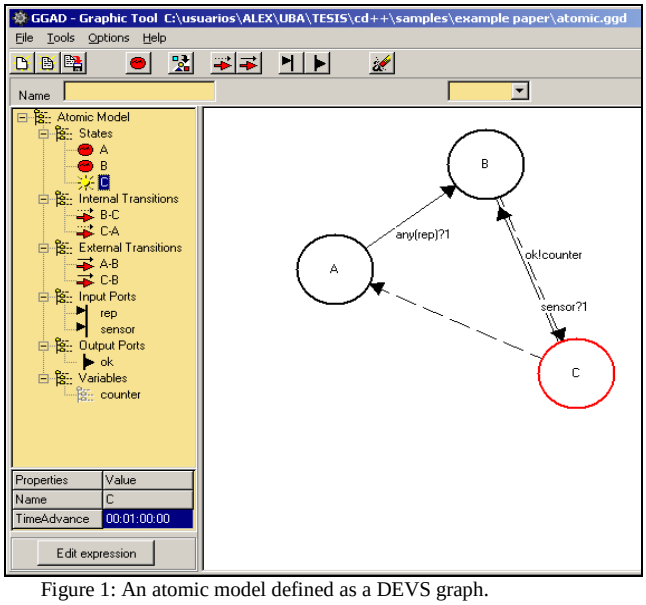

Figure 1: An atomic model defined as a DEVS graph.

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.

eCD++ is an extension to CD++ that allows real-time execution of DEVS models on a Single Board Computer (SBC) [15]. It allows also interaction between the simulator and the surrounding environment: inputs of eCD++ can be received by ports connected to real input devices such as sensors, timers, thermometers, or data collected from human interaction, while outputs can be sent through output ports connected to devices such as motors, transducers, valves, etc.

In our case study, the target system would be an eCD++ platform executing on an embedded system. On that target system, the *ElevatorController* component, as defined later, has its own set of timing constraints and would run on the eCD++ platform in real-time.

III. Timed Automata

Timed automata (TA) [16], [17] is an extension to IO automata with timing information. Time is being tracked in TA with

clocks that hold real numbers and increment their values with time advance. Locations in TA could have constraints on the time spent in that location, using expressions on clocks that called invariants. Transitions from one location to another are guarded with clock constraints and synchronization channels. Transitions could also have actions to reset values of the clocks.

We introduce a formal definition of TA as defined in [10]:

- A finite set of real-valued variables C ranged over by *x,y, etc.* standing for clocks and a finite alphabet Σ ranged over by *a, b, etc.* standing for actions.

- *Clock Constraints:*

A clock constraint is a conjunctive formula of atomic constraints of the form $x \sim n$ or $x\text{-}y \sim n$ for $x, y \in C$, $\sim \in \{<, >, =, \leq, \geq\}$ and $n \in \tilde{N}$, where $\tilde{N}$ is the set of natural numbers. Clock constraints will be used as guards for timed automata. *B(C)* denotes the set of clock constraints, ranged over by *g*.

(Timed Automaton) *A timed automaton $\acute{A}$ is a tuple <N,l_0,E,I> where:*

- *N is a finite set of locations (or nodes),*
- *$l_0 \in N$ is the initial location,*
- *$E \subseteq N \times B(C) \times \Sigma \times 2^C \times N$ is the set of edges*
- *I: N→B(C) assigns invariants to locations*

Where: Σ is set of actions, 2^C are selection of clocks to be reset to zero.

We write $l \xrightarrow{g,a,r} l'$ when $(l, g, a, r, l') \in E$

The semantics of a timed automaton is defined as a transition system where a state or configuration consists of the current location and the current values of clocks. There are two types of transitions between states. The automaton may either delay for some time (a delay transition), or follow an enabled edge (an action transition).

(Operational Semantics) *The semantics of a timed automaton is a transition system (also known as a timed transition system) where states are pairs <l,u> and transitions are defined by the rules:*

- $\langle l, u \rangle \xrightarrow{d} \langle l, u + d \rangle$ *if* $u \in I(l)$ *and* $(u + d) \in I(l)$ *for a non-negative real* $d \in R_+$
- $\langle l, u \rangle \xrightarrow{a} \langle l', u' \rangle$ *if* $l \xrightarrow{g,a,r} l', u \in g, u' = [r \rightarrow 0]u$ *and* $u' \in I(l')$

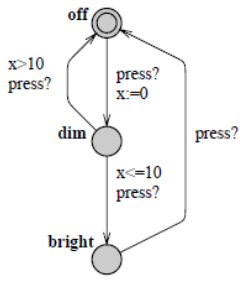

Figure 2: Timed Automaton [11].

An example of a timed automaton as given in [10] is shown in Figure 2. The TA has three states. *Off* is the initial state. The transition out of *dim* has the a guard $x \leq 10$ which enables the transition only while clock *x* value stays less than 10 time units, when 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 that clock values would satisfy its invariant.

TA are suitable for modeling discrete systems with continuous time. These systems could be composed of single TA model, or multiple models that interact together. The latter case is called *network* of TA.

TA model checking for finite state machines [18], [19], [20] has been extended by employing symbolic model checking techniques [21],[22] to build a finite reachability graph with continuous time. However, state explosion problem still limits the size of actual problems that can be solved. Recent techniques to reduce this problem have been proposed and these results were implemented in a number of tools for TA model checking with success to check models of increasing sizes. One of these tools is UPPAAL [9],[10] which has extended TA with integer variables, urgent channels and user defined functions. These extensions increase the conciseness of the model, but not the expressiveness power as shown in [23]. UPPAAL uses a subset of CTL (Computation Tree Logic) [22] to specify queries for properties in the TA model.

In our methodology, if UPPAAL (or any other model checker) faces a problem of state explosion, and no answers can be obtained in finite time, the user can switch to simulated mode and exhaustively test the models using DEVS simulation. Subcomponents can be verified using TA model checkers, improving the overall quality of the system.

IV. A Case Study: an Elevator system

We will show the effective use of TA model checking techniques for verification of DEVS Graphs models, through a concrete example. This case study shows an Elevator system composed of an elevator and a computer controller modeled in DEVS Graphs notation with a model that abstracts the elevator and controller behaviors. This abstraction is necessary to study only properties of concern and to simplify the modeling task.

A. DEVS Model definition and Simulation

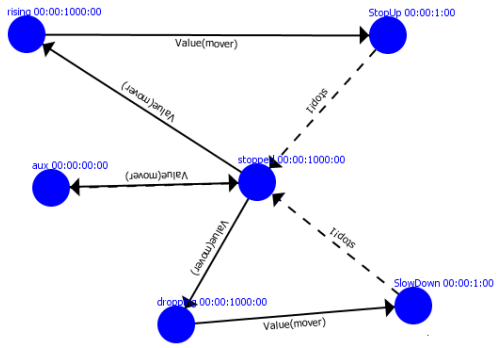

Figure 3: Elevator Model in DEVS Graph notation.

The elevator model shown in Figure 3 represents the different states of an 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 with our controller). The elevator starts in *stopped* state and waits for controller commands to move (to satisfy a button request from the user). The controller takes the decisions for direction, start and stop of the motors. Figure 4 shows the translation of this DEVS Graph model to a textual definition.

```
[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 4: Elevator CD++ model.

Input and output ports 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 keyword *ext*. Internal transitions are marked by keyword *int*. For both transitions we define source/ destination states, input ports/values for external transitions, and output ports/values for internal transitions. The lifetimes for each state are represented beside the state name.

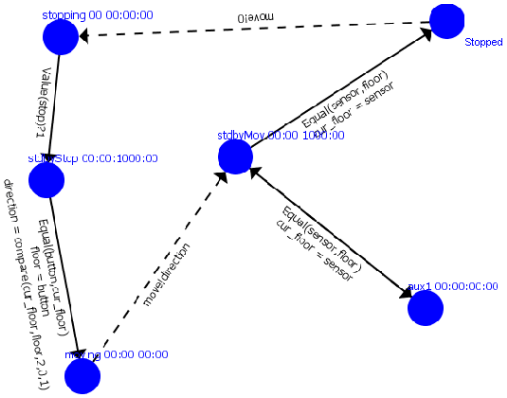

Figure 5: Elevator Controller Model in DEVS Graphs notation

The DEVS Graph model of the elevator Controller is shown in Figure 5. In this model, we abstract the behavior of the controller to being in one of possible 6 states. These states represent the elevator in regards to its movement direction and its acceleration. The states are *StdByStop* that represents the elevator in a complete stop and ready to move for any coming requests, *Moving* in which 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* which serve as a dummy state with internal transition that is executed immediately after reaching that state, the state purpose is to enable the test of the sensor value on the external transition to it with the function *equal(sensor,floor)*, *Stopped* which 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.

In this model, the controller would be in *StdByStop* state waiting for a button request to move. Whenever it receives the *button* request, it would trigger an external transition, and compare the button floor to the *cur_floor* of the elevator. Based on this comparison, the controller would determine the direction in which the elevator should travel to, and stores this info into the *direction* variable. The controller then reaches *Moving* state that has a lifetime of zero time units. Therefore, an output function is executed to send the direction information through the port *move* to the elevator model, and an instantaneous (with no time advancing in *Moving* state) internal transition would be triggered to change the state into *StdByMov* state. The controller then decides to change to *stopped* state if the sensor reading matches the desired floor; otherwise it would loop between *aux* state and *StdByMov* states as shown in the figure. The corresponding CD++ model textual specification is shown in Figure 6.

```
[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 6: Controller CD++ model.

Figure 7 shows the coupled model definition for the total system composed of the Elevator and the Controller. In the `top` component, these two components are defined, with input ports to the top component `button` and `sensor`. These ports are *linked* to the input ports of controller in lines 6 and 7.

```
1.  components : elevator@ggad controller@ggad
2.  in : button sensor
3.  Link : move@controller move@elevator
4.  Link : stop@elevator stop@controller
5.  Link : button button@controller
6.  Link : sensor sensor@controller
7.  [elevator]
8.  source : elevator.CDD
9.  [controller]
10. source : controller.cdd
```

Figure 7: Elevator coupled model definition.

The coupling between the elevator and Controller atomic models is shown on lines 4 and 5. Lines 8 to11 specifies the files names for the atomic components. A DEVS graph representing the coupled model is also shown in Figure 8. This graph shows the atomic models of the elevator and controller, input ports to the coupled model, and links between all components.

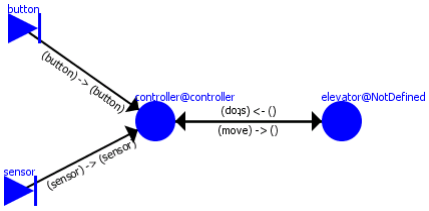

Figure 8: The elevator-Controller coupled model graph

In Figure 9, we show a test case scenario for the elevator top model is shown. This file would direct the simulation, and its events would be sent to the model top component as defined in Figure 7. This top component would direct the input to the elevator controller as specified in the model definition. In this file, third floor button is pressed at 5 time units. The floorSensor sends signals to the elevator controller with floors visited 1, 2, and 3 at times 10, 14, and 18 respectively. At time 27, first floor button is pressed and then floor sensor sends signals at designated times as shown in the figure.

```
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 9: Elevator Simulation event file

In Figure 10, simulation results are shown for the elevator controller. Initially, the controller is at `stdbystop` state, with all variables initialized to zeros. At 5 time units, the controller receives the third floor button request as specified in the input event file shown in Figure 9. This input causes the controller to execute the external transition and change its state from `stdbystop` to `moving`, with variable values as shown on the third line from top. At time 5, output function executes sending 2 to `move` port, then internal transition executes to reach the state `stdbymov`. The simulation continues with inputs at lines marked with question mark "?", Output with "O", Internal transitions with "I", and External transitions with "E" letters.

```
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 10: Controller Simulation output.

Figure 11 shows the elevator simulation results with receiving and sending input/output from the 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.

To describe the simulation results in Figure 11, the elevator simulation starts at `stopped` state at time 00:00. At time 5:00 the elevator receives an input on `move` port with value 2. This causes the elevator to change state to `rising` and wait there for input 0 on `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 execute the output function and sends value of 1 on the output port `stop`, then changes to `stopped` state. The simulation continues until the model reaches `stopped` state again in last line.

```
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 11: Elevator simulation results

B. Translating the Elevator DEVS Graph to Timed Automata

In order to formally verify the operation of our models and hence our controller implementation on ECD++ platform, we converted the previous DEVS models to equivalent TA that we can check it with UPPAAL model checker. DEVS graphs notation matches the definition on TCDEVS as defined in [11]. TCDEVS is a subset of DEVS formalism such that TCDEVS models are deterministic. In that work, it shown that TCDEVS models can be translated into equivalent TA models that maintain the same behavior and properties. By applying this translation method to our DEVS models, we obtain the TA models shown in Figure 12 and Figure 13.

Figure 12 shows the elevator equivalent TA model with clock constraints in locations that represent the time-life values in corresponding DEVS model. Values sent through DEVS

ports are modeled with shared variables in UPPAAL, sending and receiving messages in DEVS are modeled with channel synchronization on the corresponding transitions in the TA model. Each state in the DEVS model has a corresponding one with the same name in the TA model. Only one clock variable is sufficient for each TA to model an atomic DEVS graph model. This clock is reset to zero whenever the automaton enters a state. In Figure 12, *x* is the clock variable, and at each state a constraint with *x* is formed to limit the time spend in that state to the state lifetime as defined in the DEVS model. An internal transition in DEVS model is represented with transitions in TA with output synchronization channel and assigns the output value to a shared variable. External transitions are represented with transition with input synchronization channel and shared variable in its guard. For example the transition from `StopUp` to `Rising` is synchronized on `move` channel, and is enabled only if value of the shared variable `direction` equals to zero.

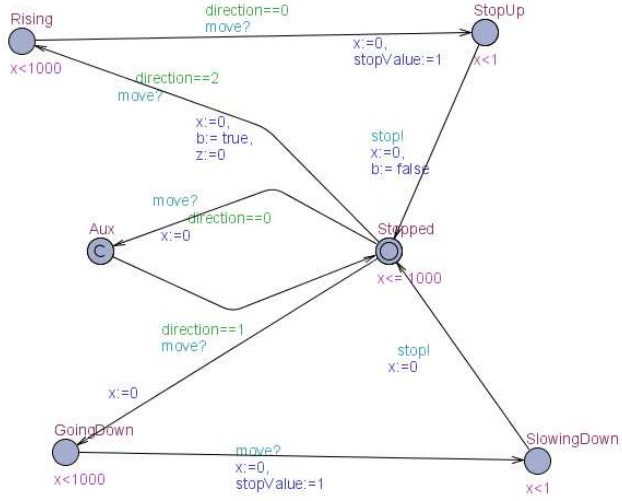

Figure 12: Elevator TA model in UPPAAL.

Figure 13 shows the translated TA model from the DEVS controller model. We used the same transformations as in the elevator model, however in this one, we converted the DEVS function *compare()* to an equivalent user defined function in UPPAAL. This function is used on the corresponding transitions in TA model as its equivalent in the DEVS model.

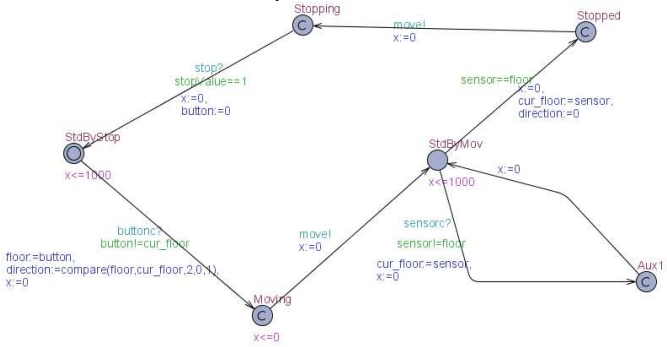

Figure 13: TA Controller model in UPPAAL.

In order to model DEVS states with zero lifetimes, i.e. once this state is reached, its output function is executed, then an internal transition happens out of that state, we used *committed* states as defined in UPPAAL timed automata. Time does not pass in a committed state, and once we reach it in TA model, a transition out of that state is enabled immediately. Example of a committed state is Aux state in Figure 12.

In order to model input to the system as per the event file shown for the DEVS model in Figure 9, we construct an automaton that would send the button and sensor inputs to the controller as in Figure 14 This automaton is necessary to make the TA system under study a closed model. In order for model checking techniques to be able to verify desired properties, they must work on closed systems as model checking would explore all possible system transitions to be able to determine if the desired property is met or not. Therefore, a good modeling of the system environment is also necessary to completely check all possible system behaviors for all expected environment inputs.

The environment modeled in Figure 14 is responsible for sending button and sensor events to the controller. It starts at `S1` state, after staying in this state for 5 time units, it sets variable `button` to 3, then synchronizes with controller TA on channel `buttonc`. Again, waits in state `S2` until its clock `y` reaches 10 time units, sets `sensor` to 1, and synchronizes with the controller on channel `sensorc`. This process continues for the desired inputs sequences to the controller, and then resets the clock and restarts again at S1.

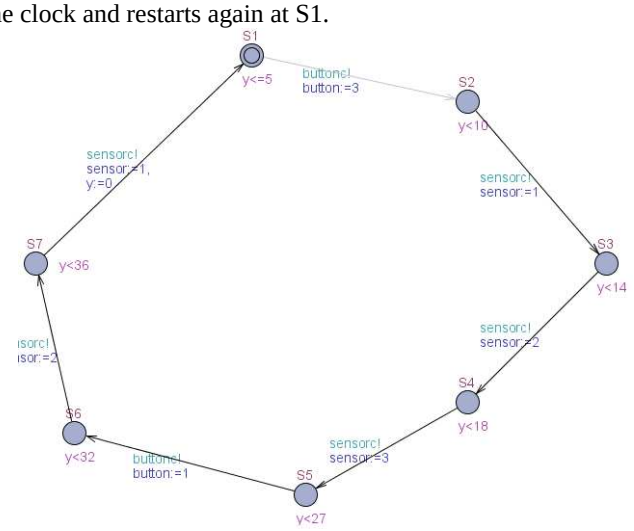

Figure 14: Environment inputs (Button and Sensor).

After translating our DEVS model to an equivalent TA model, we can use model checking to answer questions about our DEVS model behavior that otherwise would have needed to fully simulate all possible executions of the DEVS model to get the answers.

Some of the important questions would be:

a. Does our DEVS model execution stop at one point without being able to progress (having a deadlock)?

b. If no deadlocks are found in the DEVS model, is it always guaranteed whenever a user pushes a floor button that the elevator would reach that floor (normal operation as desired for the elevator system)?

c. In case the elevator eventually reaches the requested floor, is there a time upper bound between the request and the arrival

of the elevator that our model would always guarantee to happen?

In order to answer these questions, we formulated these questions into formal queries to the TA model.

For the first question, we applied the UPPAAL verifier to our model to check for any deadlocks that maybe present in the elevator model To check for that failure, we had formulated a simple query to UPPAAL model. It is expressed in UPPAAL CTL language as:

```
A[] not deadlock
```

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

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

Figure 15: Elevator Verification Results in UPPAAL

To answer the second question, we need to check for the *liveness* property, i.e. 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 3rd floor button, the elevator would *eventually* reach the 3rd floor. This property is expressed in CTL as:

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

i.e. whenever a user input for the third floor button happens, the cur_floor variable in the ElevatorController would eventually reach that floor. This property was satisfied as well in UPPAAL model checker for the given model. However, for a query as:

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

The property is not satisfied as we expect. By pressing 3rd floor button, given the elevator initially stopped at 1st floor, there is no way the elevator would reach the 4th floor.

To answer the third question, i.e. to know if the elevator would reach the requested 3rd floor within some bounded time. We extend the model for bounded time checking by adding boolean variable b, and a global clock z as shown on the Elevator model. The variable b would be set to true as long the elevator starts traveling and until it reaches the Stopped state again. Therefore, by checking the accumulated time while b is true, it would give us the property we need to check. Then, the property would be expressed with the following query:

`A[] ( b imply z < 27 )` which is satisfied. However, the query `A[] ( b imply z < 26 )` is not satisfied.

This shows that the elevator would reach the 3rd floor after requested there by no less than 26 time units, but guaranteed to be there at 27 time units or more. More complex queries to check for more properties could also be formulated and verified by UPPAAL in case that we have a more complex DEVS model.

UPPAAL tool can also give a trace to help the designer get an insight into the system working details. A trace is shown in Figure 16. 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)`, and transitions with synchronization is shown as `buttonc:User_sensor_input --> ElevatorController`. That means that `User_sensor_input` synchronizes on `buttonc` channel with the `ElevatorController`. The new state resulting from this transition is shown below the transition.

```
(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 16: Elevator TA Simulation Results in UPPAAL.

This trace result shows the composed state of the model, i.e. Elevator, Controller and User_sensor_input composed state. The composed state is represented by a tuple (elevatorState, ControllerState, User_Sensor_inputState). Therefore, to compare this UPPAAL trace to the previous DEVS simulation, we compare the component trace with its corresponding DEVS simulation output, as DEVS simulation output is stored for each component individually. For example, the elevator UPPAAL trace results are in the left side of the tuple. By extracting the elevator trace `(Stopped, Rising, StopUp, Stopped, GoingDown, SlowingDown, Stopped)`, we find it matching the same corresponding states in the simulation results shown in Figure 11 `(stopped,rising, StopUp, stopped, dropping, SlowDown, stopped`. In both of these traces, the elevator starts at its initial state `Stopped`, and then rises to reach third floor as 3rd floor button is pressed, until it stops there and stays in `stopped` state waiting for a next request. When first floor button is pressed, the elevator moves down until it reaches first floor and stops there in `stopped` state, ready for next button request. The same comparison can be done to the controller trace and to see it matches the DEVS simulation as shown in Figure 10.

V. Conclusion and Future Work.

We showed an effective methodology for Real-Time systems development using DEVS modeling and simulation tools combined with the power of formal model checking tool as timed Automata. In this methodology, we showed translation

of DEVS Graph to TA models details, and how to satisfy the requirement to have a closed system be able to fully check DEVS models.

In order to use Timed automata to verify DEVS models, we need an accurate modeling of the environment in which the modeled system would work and interact. This is not always a straightforward task, as many assumptions may be needed to model the environment. Future work would be needed to deduce patterns for modeling complex environment behaviors to be able to completely check DEVS models.

Validating DEVS models formally with TA model checking is paving the road for solving real-time predictability for software systems. DEVS models are executable directly, without the need for compilation, on eCD++ embedded platform in a real-time. With this advantage, any formally validated DEVS model would be guaranteed to execute exactly as predicted by the validation, as no human intervention comes between the checked model and the executable system. This advantage would serve not only simulation community, but also real-time software community as well, as DEVS can be used to model controllers that would be simulated, formally validated and then deployed on target platform.

Our approach to use model checking to verify DEVS models is limited with the same limitations imposed on TA model checking, mainly because of state explosion problem. This would limit the methodology to small and medium size DEVS models. However, many real-life applications fall into these boundaries.

We see our methodology scaling up to include more complex and general DEVS models that may represent a challenge for model checking tools because of the state explosion problem. In this case, other decomposition and abstraction techniques would be applied on the problem on hand. These techniques would simplify the generated TA models to the point that it is practical for model checking.

We intend to expand this methodology to enable more efficient methodology for building DEVS components. In this expanded methodology, the system analyst would concentrate on system modeling tasks for the system on hand, for example the elevator, and the surrounding environment. This model would then be verified and validated by the designer to make sure it captures all necessary system and environment details. The methodology would use existing formal methods to generate DEVS component (for example the elevator controller) that interacts with the modeled system to achieve the desired overall system properties, i.e. lack of deadlocks, safety, liveness, and bounded-time-response. This would automate the building of a considerable part of final coupled system, and eliminate the need to formally check the generated component, as it would be correct by the correct-by-construction technique.

REFERENCES

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

[2] R. Gerlich, R. Gerlich, T. Boll. “Random Testing: From the Classical Approach to a Global View and Full Test Automation”. *Proc. of 2nd international Workshop on Random Testing*. Atlanta, GA, 2007.

[3] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, W. Visser, "Formal Software Analysis Emerging Trends in Software Model Checking," *In 2007 Future of Software Engineering*. ICSE. Minneapolis, MN. 2007.

[4] D. Hemer and P.A. Lindsay. “Template-based construction of verified software”. *IEE Proceedings-Software Engineering*, Vol. 152, No. 1, 2005.

[5] M.Baleani,A.Ferrari, L.Mangeruca, A.L.Sangiovanni-Vincentelli, U.Freund, E.Schlenker, H.-J.Wolff. “Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development”. *Design, Automation and Test in Europe Conference and Exhibition*. Munich, Germany. 2005.

[6] J. Huang, J. Voeten, H. Corporaal. “Predictable Real-time Software Synthesis”. *Real-time systems.* (36) 3, pp.159-198, 2007.

[7] R. Alur and D. L. Dill. “A Theory of Timed Automata”. *Journal of Theoretical Computer Science*, 126(2):183–235, 1994.

[8] B. Zeigler, T. Kim, H. Praehofer, *Theory of Modeling and Simulation*. 2nd Edition. Academic Press. 2000.

[9] G. Behrmann, A. David, K. G. Larsen. “A Tutorial on Uppaal”. *In Proceedings of 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*. Forli, Italy. LNCS 3185. 2004.

[10] J. Bengtsson , W. Yi. “Timed Automata: Semantics, Algorithms and Tools”. In *Lecture Notes on Concurrency and Petri Nets. W. Reisig and G. Rozenberg (eds.)*, LNCS 3098, 2004.

[11] H. Dacharry, N. Giambiasi. “A Formal Verification Approach for DEVS”. *In Proceedings of Summer Computer Simulation Conference*. San Diego, CA, 2007.

[12] G. Wainer. "CD++: a Toolkit to Define Discrete-Event Models". *Software, Practice and Experience*. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002.

[13] B. Zeigler, H. Song, T. Kim, H. Praehofer. "DEVS Framework for Modelling, Simulation, Analysis, and Design of Hybrid Systems". *Proceedings of HSAC*, LNCS, Vol. 999. Ithaca, NY. 1995.

[14] G. Christen, A. Dobniewski, G. Wainer. "Modeling State-Based DEVS Models in CD++ ". *In Proceedings of MGA*, Advanced Simulation Technologies Conference. Arlington, VA. U.S.A. 2004.

[15] Y. H. Yu, G. Wainer. “eCD++: An Engine for Executing DEVS Models in Embedded Platforms”. In *Proceedings of the 2007 Summer Computer Simulation Conference*. San Diego, CA. 2007.

[16] R. Alur, D. L. Dill. “Automata for Modeling Real-Time Systems”. *In Proc. of Int. Colloquium on Algorithms, Languages, and Programming*. Warwick University, UK. LNCS 443, 1990.

[17] J. E. Hopcroft, J. D. Ullman. *Introduction of Automata Theory, Languages, and Computation*. Addison Wesley, 2001.

[18] L. Aceto, A. Bergueno, and K. G. Larsen. “Model Checking via Reachability Testing for Timed Automata”. In *Proceedings, Fourth Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Lisbon, Portugal, LNCS 1384. 1998.

[19] R. Alur, C. Courcoubetis, and D. L. Dill. “Model-Checking for Real-Time Systems”. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science.* Philadelphia, PA. 1990.

[20] R. Alur, C. Courcoubetis, d. Dill. “Model-Checking in Dense Real-Time”. *J. of Information and Computation*, 104(1):2–34, 1993.

[21] T. A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine. “Symbolic Model Checking for Real-Time Systems”. *Proc. of 7th Annual IEEE Symposium on Logic in Computer Science*, Santa Cruz, CA. 1992.

[22] T. A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine. “Symbolic Model Checking for Real-Time Systems”. *Journal of Information and Computation*, 111(2):193–244, 1994.

[23] P. Bouyer and F. Laroussinie. “Model Checking Timed Automata”. *Modeling and Verification of Real-Time Systems*, pages 111-140. ISTE Ltd. - John Wiley & Sons, Ltd., 2008.