Proceedings of the 2009 Winter Simulation Conference
M. Moallemi and G. Wainer

A SIMPLIFIED REAL-TIME EMBEDDED DEVS APPROACH TOWARDS EMBEDDED AND CONTROL DESIGN

Mohammad Moallemi Gabriel Wainer

Dept. of Systems and Computer Engineering
Carleton University Centre of Visualization and Simulation (V-Sim),
1125 Colonel By Dr. Ottawa, ON, Canada.

ABSTRACT

The development of embedded systems with real-time constraints has received the thorough study of the software engineering community in the last 20 years. We propose a model-driven real-time simplified P-DEVS (Parallel Discrete EVent Simulation system) method to develop this kind of applications based on formal DEVS, a formal technique originally created for modeling and simulation of discrete event systems. We will explain how to use this framework to incrementally develop embedded applications, and to seamlessly integrate simulation models with hardware components. The use of this methodology shortens the development cycle and reduces its cost, making it possible to reuse DEVS and P-DEVS models for real-time and embedded applications with almost no modifications, improving quality and reliability of the final product and making it portable on different hardware. Our approach does not impose any change on the actual DEVS model to convert them to real-time DEVS, providing flexibility to the overall process.

1 INTRODUCTION

One particular use of modeling and simulations tools is in the development of embedded systems, usually these systems have time constraints in which case they are also called real-time Systems. real-time Systems must provide reliable outputs to external inputs within a time limit. Depending on the strictness of the time limit, the systems are usually separated in soft or hard real-time systems.

Embedded real-time software construction has usually posed interesting challenges due to the complexity of the tasks executed. Most methods are either hard to scale up for large systems, or require a difficult testing effort with no guarantee for bug-free software products. Formal methods have showed promising results, nevertheless, they are difficult to apply 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) techniques in order to make system development tasks manageable. Integration of M&S techniques with real-time system design must provide seamless transportability of a virtual system model to a real-time and finally embedded system. This approach is one of the most beneficial and time saving applications of M&S. This way engineers are able to model their final embedded systems using reliable formal methods and then test them under virtual time simulation conditions to save cost and time. After that, the model can be test in real-time conditions using test data and finally it can be ported on hardware for final verification and production. However that, many issues are raised in real-time systems that do not exist in virtual time simulations. Nevertheless, this approach makes the design phase easier, faster and more reliable as it is based on formal mathematical based methods. It also provides both virtual and real-time simulation environments for the model to be verified with much less cost, time and 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.

For engineering in particular, Modeling and Simulation (M&S) of embedded systems is of utmost importance. For example, engineers and scientists make heavy use of simulation tools when a process is difficult to replicate (because of the cost involved, or if the environmental conditions for the experiment are difficult to replicate or the danger is too high) or when the simulation of a natural process is many times faster than the real process. By using different techniques for modeling, we can predict the behavior of simple or complicated phenomena with, most of the time, a high degree of certainty. For systems that interact with real data, the preferred method for modeling is the use of continuous differential equations. However, one layer higher in the interaction between systems and the real world we deal with a different nature of modeling and control which is usually easy to model using discrete event modeling methods.

2 BACKGROUND AND MOTIVATIONS

DEVS (Zeigler 1984; Zeigler 1990) is an increasingly accepted framework for understanding and supporting the activities of modeling and simulation. DEVS is a sound formal framework based on generic dynamic systems, including well-defined coupling of components, hierarchical, modular construction, support for discrete event approximation of continuous systems and support for repository reuse. DEVS theory provides a rigorous methodology for representing models, and it does present an abstract way of thinking about the world with independence of the simulation mechanisms, underlying hardware and middleware. A real system modeled with DEVS is described as a composite of sub-models, each of them being behavioral (atomic) or structural (coupled).

A DEVS atomic model is formally defined by:

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

, where

X: a set of external input event types

S: a sequential state set

Y: an output set

δ_{ext}: $Q \times X \rightarrow S$, an external transition function

Where Q is the total state set of $M = \{(s, e) | s \in S \text{ and } 0 \leq e \leq ta(s)\}$

δ_{int}: $S \rightarrow S$, an internal transition function

λ: $S \rightarrow Y$, an output function

ta: $S \rightarrow R^{+}_{0,\infty}$, a time advance function

Where the $R^{+}_{0,\infty}$ is the non-negative real numbers with ∞ adjoined.

An atomic model AM is a model which is affected by external input events X and which in turn generates output events Y. The state set S represents the unique description of the model. The internal transition function δ_{int} and the external transition function δ_{ext} compute the next state of the model. If an external event arrives at elapsed time e which is less than or equal to ta(s) specified by the time advance function ta, a new state s′ is computed by the external transition function δ_{ext}. Then, a new ta(s′) is computed, and the elapsed time e is set to zero. Otherwise, a new state s′ is computed by the internal transition function δ_{int}. In the case of an internal event, the output specified by the output function λ is produced based on the state s. As before, a new ta(s′) is computed, and the elapsed time e is set to zero.

Figure 1 illustrates the state transition of an atomic model. An atomic model is in state s for a specified time ta(s). If the atomic model passes this time without interruption it will produce an output y at the end of this time and will change state based on its δ_{int} function (complete transition) and continues the same behavior. But, if it receives an input x during its ta(s) time it will change its state which is determined by its δ_{ext} function and does not produce an output (incomplete transition).

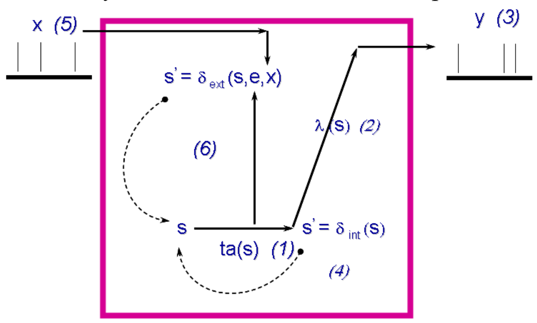

Figure 1: DEVS atomic model state transition sequence

A coupled model connects the basic models together in order to form a new model. This model can itself be employed as a component in a larger coupled model, thereby allowing the hierarchical construction of complex models. The coupled model is defined as:

$CM = <X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, \mathit{Select}>$, where:

$X = \{(p, v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p, v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

D is the set of the component names

The component couplings are subject to the following requirements:

External input coupling (EIC) connects external inputs to component inputs,

$EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$;

External output coupling (EOC) connects component outputs to external outputs,

$EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\}$;

Internal coupling (IC) connects component outputs to component inputs,

$IC \subseteq \{((a, op_a), (b, ip_b)) \mid a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$;

SELECT: 2M − Á → M, the tie-breaking selector

A coupled model CM consists of components Mi, which are atomic models and/or coupled models. The influences Ii and the i-to-j output translation Zi,j define three types of coupling specification as follows. The external input coupling connects the input events of the coupled model itself to one or more of the input events of its components. The external output coupling connects the output events of the components to the output events of the coupled model itself. The internal coupling connects the output events of the components to the input events of other components. The SELECT function is used to order the processing of the simultaneous events for sequential events. Thus, all the events with the same time in the system can be ordered with this function.

Figure 2 shows a hierarchical DEVS model. This model is composed of two atomic models (Generator, Buffer and Processor) and two coupled models: the top model that contains generator atomic model and BUF-PROC coupled mode, the BUF-PROC coupled model includes tow atomic models: BUF and PROC. The port connections are also visible in the figure. For example the output port "out" of atomic model PROC is connected to the "done" input port of BUF atomic model within the same coupled model and also is connected to the output port of the its parent coupled model which connects this output to the Top model output port.

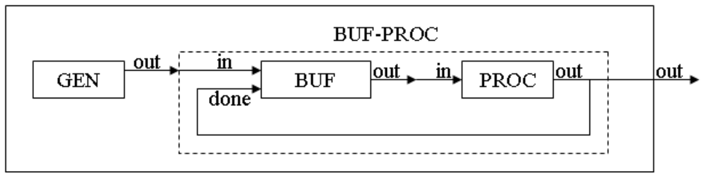

Figure 2: Generator-Buffer-Processor hierarchical DEVS model

As will be mentioned in the next section, there has been some efforts to modify DEVS formalism to make it suitable for real-time application. Many existing techniques that have been widely used for the development of embedded and real-time systems, also mapped into DEVS models. Many state-based approaches, such as Verilog (Thomas et al. 2008), VHDL (Navabi Z. 1997), Petri Nets (Bobeanu et al. 2004) and Timed Petri Nets (Wang J. 1998), Timed Automata (Raskin et al. 2007), State Charts (Harel D. 1986) and Finite State Machines (Gill A. 2007) have their DEVS equivalents. This permits sharing information at the level of the model, and different sub-models can be specified using different techniques, while keeping independence at the level of the execution engine. Existing DEVS tools have showed their ability to execute such wide variety of models with high performance in standalone or distributed environments. In this way, we count with a mathematical technique that can be used to describe different modeling techniques and prove properties about general aspects of the system, while having a general technique for sharing model information using different approaches, and being able to apply the right technique to each part of the application development process.

3 PREVIOUS WORKS

A Parallel DEVS (P-DEVS) model presented in (Chow et al. 1994) is described as a set of basic and coupled models. Atomic models are still the most basic constructions, which can be combined with other models into coupled models. The P-DEVS atomic model has the following structure:

$AM = < X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta >$, where:

$X_M = \{(p, v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y_M = \{(p, v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

S is the set of sequential states;

$\delta_{ext}: Q \times X_M^b \rightarrow S$ is the external state transition function;

$\delta_{int}: S \rightarrow S$ is the internal state transition function;

$\delta_{con}: Q \times X_M^b \rightarrow S$ is the confluent transition function;

$\lambda : S \rightarrow Y_M^b$ is the output function;

ta: $S \rightarrow R^{+}_{0,\infty}$ is the time advance function; with

Q: $= \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ the set of total states.

The semantics of the P-DEVS definition are as follows. At any given time, a basic model is in a state *s*. And in the absence of external events, it will remain in that state for a period of time as defined by ta(s). When an internal transition takes place, the system outputs the value λ(s), and changes to state $\delta_{int}(s)$. If one or more external events $E = \{x_1 \ldots x_n / x \in X_M\}$ occurs before

ta(s) expires, i.e., when the system is in the state (s, e) with $e \leq ta(s)$, the new state will be given by $\delta_{ext}(s, e, E)$. Suppose that an external and an internal transition collide, i.e., an external event E arrives when e = ta(s), the new system's state could either be given by $\delta_{ext}(\delta_{int}(s), e, E)$ or $\delta_{int}(\delta_{ext}(s, e, E))$. The modeler can define the most appropriate behavior with the δ_{conf} function. As a result, the new system's state will be the one defined by $\delta_{conf}(s, E)$.
A P-DEVS coupled model (CM) is defined the same as DEVS model except that there is no tie breaking function (SELECT), as this problem is solved within the atomic model using δ_{conf} function.
The real-time DEVS formalism (Hong et al. 1997) is an extension of the DEVS formalism for real-time systems simulation. An atomic model in RT-DEVS formalism (RTAM), is defined as:

RTAM=<X, S, Y, δ_{ext}, δ_{int}, λ, ta, ti, □, A >, Where:
X, S, Y, δ_{int}, λ and ta are the same as original DEVS.
δ_{ext}: Q x X→S, an external transition function, where Q is the total state set of M= $\{(s, e)|s \in S \text{ and } 0 \leq e \leq ti(s)|_{max}\}$
ti: a time interval function,
□: an activity mapping function,
A: a set of activities,

With constraints:

□: S→A
ti: $S \rightarrow R^{+}_{0,\infty} \times R^{+}_{0,\infty}$,
Where $ti(s)|_{min} \leq t(a) \leq ti(s)|_{max}$, $ti(s)|_{min} \leq ta(s) \leq ti(s)|_{max}$, $s \in S$, a = □(s)∈A and t(a) is the execution time of an activity a.
A= $\{a|\ t(a) \in R^{+}_{0,\infty},\ a \notin \{X?, Y!, S=\}\}$
Where: X? is the action of receiving data from X, Y! is the action of sending data from Y and S= is the action of modifying a state in S.

In RT-DEVS an activity mapping function □ and an activity set A are defined to advance virtual time with an executable activity associated with an event. The regular ta time advance function only verifying the correctness of activity mapping time constraints and compensate time discrepancy problems. The time bound of each activity are specified by ti function.
A coupled model within the RT-DEVS formalism is defined the same way as in the original DEVS formalism with an exception. The exception is that there is no SELECT function in RT-DEVS, which has been defined in the DEVS formalism to break ties for simultaneous events scheduling. This is because such simultaneous events cannot be happened in a real-time simulation environment. In real-time simulation with one processor, only one event at a time can be physically processed even if more than one event occurred from the external environment.

In (Cho et al. 1998) RT-DEVS has been used with slight modification and addition of the concept of driver for hardware interaction. The definition for a real-time driver model, which is an interface between a simulation model and a real world object, is as follows:

RTDM=<X, Y, T_{ME}, T_{EM}>, where:
$X= X_M \cup X_E$: an input events set
X_M: input events from model
X_E: input events from environment
$Y= Y_M \cup Y_E$: an output events set
Y_M: output events to models
Y_E: output events to environment
T_{ME}: $X_M \rightarrow Y_E$: an event translation function from a model to an environment
T_{EM}: $X_E \rightarrow Y_M$: an event translation function from an environment to a model

The main function of the driver model is the translation of input and output events to and from RT-DEVS environment to the hardware environment.

4 PROPOSED REAL-TIME APPROACH

Here we propose a rather more efficient real-time extension to P-DEVS formalism that does not change the formalism and also define driver object for hardware interaction. The RT-DEVS formalism modifies DEVS formalism and adds time interval function, activity mapping function and set of activities. Each state is reflected to the hardware while the state change is happening. Thus, the time advance function is responsible of verifying the hardware reaction time to compensate the time discrepancy problem. The RT-DEVS formalism does not scale well for control applications and its main application is for real-time simulation.

4.1 Time Advance and State Change Reflection

In the DEVS and P-DEVS formalisms, virtual simulation time advances only when a simulator calls the time advance function ta of an atomic model. The RT-DEVS formalism replaces virtual time advance by real-time advance. The actual advance of simulation time is the real execution time of δ_{ext} and δ_{int}.
In our proposed approach we rely on P-DEVS formalism with some modification:

- The time advance function (ta) will count the wall clock time, hence the simulation/control will proceed with real-time clock and events will be processed at the real-time point that they are supposed to be injected to the model. The model will also listen to the hardware and accept hardware inputs during the simulation time and forwards them to the atomic model that they are supposed to go. Unlike RT-DEVS we do not map activities to states and control the behavior of the hardware against the model state. Instead, we wait for inputs from hardware and signal activities on the hardware.
- Here we introduce the concept of state reflection to hardware using the output function (λ). In our approach the output function is responsible of reflecting the state change result of the model to the actual hardware. Therefore, each hardware needs to have its own atomic model to generate hardware control signals. Whenever an atomic model finishes its ta(s), it produces an output to the hardware which informs the hardware about the state change and then the internal transition function will change the state based on the current state. All the hardware control signals will be produced in the output function. In this way, whenever the model receives an input it needs to pass a dummy state with $0< ta(s) < \varepsilon$ to invoke output function and signal the hardware for the next state. This dummy state will not have any burden on the model and is as small as possible that does not interfere with the hardware functionality or response-time to an input.

Thus, the atomic model is formally defined by:
RTAM = $< X, S, Y, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta >$, where:
X, S, Y, δ_{ext}, δ_{int}, δ_{conf} and λ are the same as P-DEVS
ta: $S \rightarrow R^{+}_{0,\infty}$, a time advance function which works with actual wall clock time

The coupled model definition will be the same as P-DEVS in which the only difference from DEVS is the omission of SELECT.

Our Approach does not modify the definition of P-DEVS models and we can reuse the same models that have been developed for virtual-time simulations, in real-time simulation/verification and then use them on the hardware platform.

4.2 Hardware Interface and Deadline

As discussed in previous works section, (Cho et al. 1998) presented a modification to RT-DEVS formalism in which they added the driver model to RT-DEVS formalism. In our approach we use the idea to translate the input and output signals of the model to the hardware commands. In this approach, all the input and output ports of only the Top coupled model own a driver object which is an interface between the model and the real world hardware. This way the model can be portable on different hardware and only the driver object will change on the target hardware platform.
The most critical attributes of real-time systems is the availability of output within the deadline specified or otherwise ignoring the output. RT-DEVS verifies the time bound of each state change to make sure the deadline of each state is met. Here we embed the concept of deadline in the driver object of the Top model. Assuming any input that comes to the system will finally produce an output to the hardware, we define deadline for each input from hardware.
Our definition of driver model will be limited to the Top coupled model, thus the P-DEVS notation of Top coupled model will be modified as follows:

TOPCM = $<X, Y, OS, IS, DX, DY, D, \{M_d \mid d \in D\}, EIC, EOC, IC>$, where:

X, Y, D, M_d, EIC, EOC and IC are the same as P-DEVS

IS = {(is, iy, dl) | is $\in$ Input Signals from Hardware, iy$\in$Y output port which the result of incoming signal will be produced at, dl$\in R^{+}_{0,\infty}$ deadline for the input signal} is the set of hardware input signals and associated deadlines

OS = {(os, oy, pt) | os $\in$ Output Signals to Hardware, oy$\in$Y output port that the signal will be submitted to, pt$\in R^{+}_{0,\infty}$ processing time from when the associated is signal has been received} is the set of hardware output signals

DX: IS $\rightarrow$Xv: converts external hardware inputs signals to input port value (Xv)

DY: Yv $\rightarrow$OS: converts output port value to external hardware outputs signals (Yv) with constraint ($\forall$ iy = oy $\rightarrow$ pt $\leq$ dl)

Note that every input and output to and from hardware to atomic model is routed through coupling connections from the Top coupled model to atomic model or from atomic model to the Top coupled model. The DY function of the driver object checks the output port oy of each output and looks for the deadline definition from any previous input signal to compare the processing time and the predefined deadline for that output. If the processing time is less than or equal to the deadline, then the output signal will be produced, otherwise it will be ignored. The port checking for output signal deadlines uses first come first serve algorithm. The first deadline for the output port will be compared against the current output processing time. Figure 3 shows the hierarchical structure of a model in our RT-DEVS. Only the ports of the coupled model are connected to the hardware.

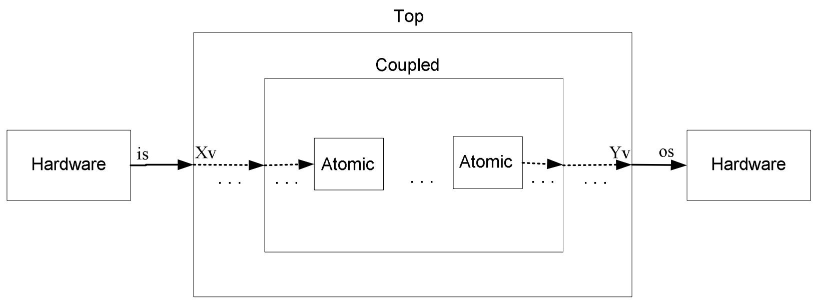

Figure 3: Model hierarchy with hardware connection

4.3 Internal Time Management

Our approach does not define activity mapping time constraint, instead we use deadline for outputs produced. Thus we do not need to verify ta(s) time constraints. The time stamp of the messages transferred for triggering state changes and events do not get updated with real-time clock and are the same during the lifetime of the message. Because of that, we still need the δ_{conf} tie breaking function for atomic models. As a result, we can consider the whole model as a black box that receives the input from hardware, processes the input and performs state changes in real-time and produces output within the acceptable pre-specified deadline. Because the states are not tied with hardware activities, there is no need to check their durations, therefore the simulator is responsible to initiate internal events at the end of ta(s) of a state.

4.4 Execution Algorithm

Here we present the execution algorithms for the main loop, internal transition and external transition in our formalism.

The main loop of the simulation will work as follows:

```
1.  main():
2.  s =s0
3.  tN = ta(s0);
4.  forever for each RT-DEVS model /* main loop */
5.      wait for is signals from hardware or eventfile events
6.      if an external event then
7.          x = DX(IS)
8.          when_rcv_(x, t)
9.      else if an internal time out then
10.         when_rcv_(*, t)
11.         OS = DY(y)
12.         if (OS.oy = IS.iy) then
13.             if (OS.pt $\le$ IS.dl) then
14.                 send os signal to hardware
15.             end if
16.         end if
17.     else if both external and internal event then
18.         when_rcv_ (*, t)
19.         when_rcv_ (x, t)
20.     end if
21. end forever
```

Note that line 7 and 11 is the driver object functionality in converting inputs and outputs and lines 12 to 14 is the process of deadline verification. Line 17 is the confluent function case in which we first serve internal event and then external event.

The following pseudo code represents the handling of external event at each atomic model:

```
1.  when_rcv_(x, t):
2.  if ($T_L \leq t \leq TN|_{max}$) then
3.      e =t – $t_L$
4.      s = $\delta_{ext}(s,e,x)$
5.      $t_L$ = t
6.      $t_N$ =[$t_L$ + ta(s)$|_{min}$ – time($\delta_{ext}$), $t_L$+ta(s)$|_{max}$ – time($\delta_{ext}$)]
7.  else
8.      error
9.  end if
```

The next algorithm shows how an internal event is managed in an atomic model:

```
1.  when_rcv__(*, t):
2.    if $t_N|_{min} \leq t \leq t_N|_{max}$ then
3.      y = $\lambda$(s)
4.      send (y, t) to associated ports
5.      signal them
6.      s = $\delta_{int}(s)$
7.      $t_L$= t
8.      $t_N$ =[$t_L$ + ta(s)$|_{min}$ – time($\delta_{int}$), $t_L$+ ta(s)$|_{max}$ – time($\delta_{int}$)]
9.    else
10.     error
11.   end if
```

5 DESIGN CONSIDERATIONS

There are some considerations that must be in to account while designing a real-time model using our approach. There is no one to one mapping of the activities between the hardware and the atomic models. Though, the designer can draw this mapping with the design. One of the advantages of our approach is that we are not forced to map a state to an activity in hardware. Thus, some atomic models or states can only be responsible for processing, using state changes. To keep track of hardware behavior, an atomic model can be defined for each hardware (sensors, motors, actuators …).

Matching deadlines is another important issue, as the designer must be careful about deadline definition in the driver object of the Top model. Each input to the system whether from hardware or from event file, specifies the output port in which the hardware output is produced, and the deadlines associated with each output ports are kept in a queue and served in first come first serve order. Thus, the deadlines must be specific and close to expected processing time, in order not to interfere with other output deadlines. If there is no deadline for an input, the deadline can be set as infinity.

5.1 Interruptive Inputs

Interruptive inputs from hardware (e.g. touch sensor) are the regular inputs that happen randomly whenever a sensor detects something. While working with real-time hardware, in some circumstances (e.g. a robotic car has been blocked by an obstacle and the touch sensor is kept touched against the obstacle) the hardware might detect a recursive input that locks the model because the time interval between two consequent inputs is too small compared with the ta(s) of the model state to finish and produce output. Thus, a deadlock will happen. To overcome this problem, the model must ignore any input during ta(s) to complete the state and produce output. The output will signal an activity on the hardware and resumes the sensor to its normal condition.

5.2 Periodic Inputs

Periodic inputs are those which happen at certain periods of time (e.g. distance sensor). A model that receives these kinds of inputs must avoid deadlocks that happen because of period < ta(s). There are two strategies to avoid deadlock with these inputs:

- If the model is sensitive to some ranges of data received by a periodic input device, then ta(s) can be greater than the input period and the model must ignore incoming inputs while it is in ta(s). Because usually certain ranges of data are received close to each other in terms of time. Though, the model gets flooded with inputs while it is in ta(s) and won't be able to produce output. (e.g. the robotic car has become close to an obstacle and the distance sensor is sending small ranges of values which the model is sensitive to them and must react)

- If the model must react to each input values of a periodic input device, then, ta(s) < period must be satisfied. This ensures that before the next input is received, the model produces an output therefore reacts to the input. The period must be long enough for the hardware to react to the input that receives from model.

6 CASE STUDY

As a case study we will study a real-time controller model for a robotic vehicle. The model is composed of one coupled model which is responsible for controlling the car's motion. It is composed of three atomic models: *Movement Controller*, *Speed Controller* and *Motor Controller*.

There are four sensor controllers involved which every one of them is an atomic model: *Touch sensor*, *Sound sensor*, *Light sensor* and *Sonar sensor*. There are also two atomic models that control the right and left wheels of the car.

The car is initially stopped. When it receives a sound command or an event from event file or sees a green light, it starts moving forward.

While it is moving forward it listens to all sensors and also accepts inputs from eventfile:

- If there is a sound it stops.
- If it sees a red light it stops.
- It periodically (every 500ms) receives sonar distances from sonar sensor which show the distance between the car and the obstacle ahead. If the sonar distance is less than 20cm the car starts turning 90◦ to either directions (direction toggles). If the distance is between 20cm and 100cm it moves to second gear and speeds up. If the car is in second gear and the distance is between 100cm to 200cm it moves to third gear and speeds up again. If the car is in third gear and the distance becomes 20cm to 100cm it moves back to the second gear.
- If it touches something, it starts moving back for 2 seconds and then turns 90◦ to either directions (direction toggles). While it is turning it also listens to sonar sensor and if the distance is still less than 20cm it keeps turning until the distance becomes more than 20cm.
- If it receives a stop command from event file it stops.

When it is stopped it starts moving forward if one of bellow happens:

- It hears a sound.
- It receives a move forward input from the event file.
- It sees a green light.

As was mentioned above there are 9 atomic models and one coupled model in this design. **Figure 4** shows the hierarchical structure diagram of the design. Note that the top model contains all the models shown in the figure.

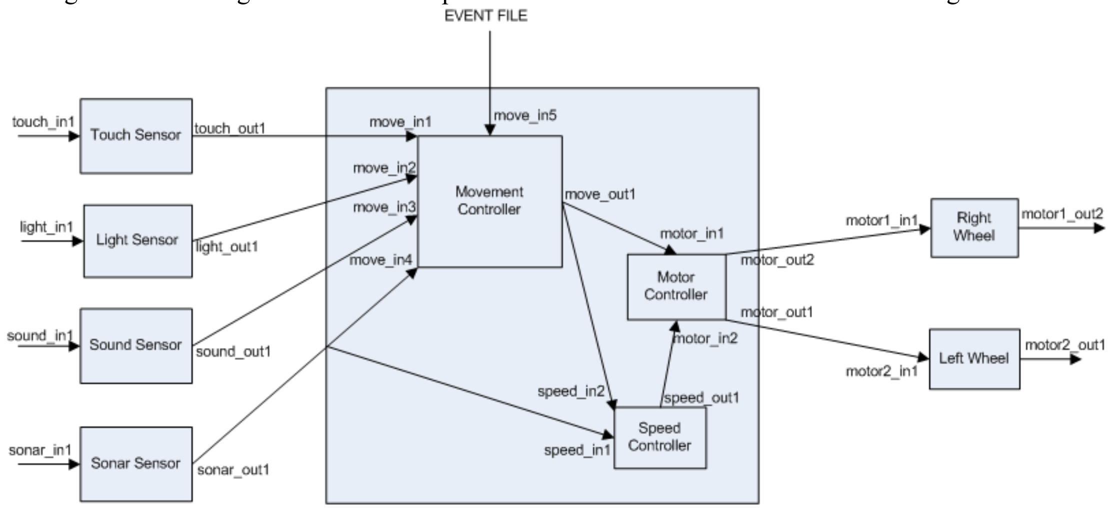

Figure 4: Model hierarchy of the robotic car model

Figure 5 illustrates the GGAD Notation (Christen G et al. 2004) of Movement Controller model. Note that continuous lines are external transitions and dashed-lines are internal transition between states. The label on the external transition connection determines the input port name and input value to that port and the label on internal transition determines the output port name and output signal value of the output function that will be called before that internal transition.

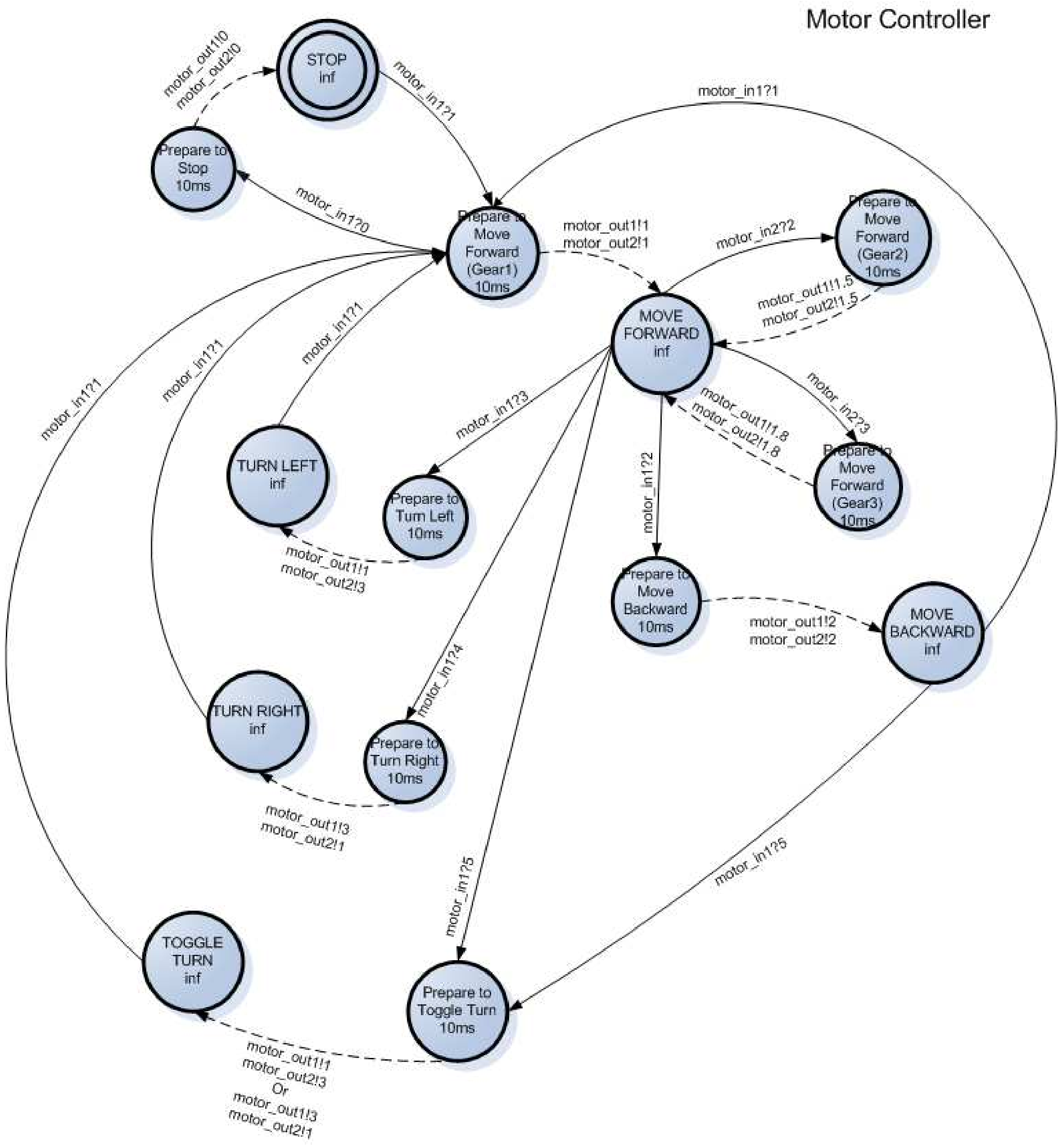
Motor Controller
STOP inf
Prepare to Stop 10ms
Prepare to Move Forward (Gear1) 10ms
MOVE FORWARD inf
Prepare to Move Forward (Gear2) 10ms
Prepare to Move Forward (Gear3) 10ms
TURN LEFT inf
Prepare to Turn Left 10ms
Prepare to Move Backward 10ms
MOVE BACKWARD inf
TURN RIGHT inf
Prepare to Turn Right 10ms
TOGGLE TURN inf
Prepare to Toggle Turn 10ms
motor_out1!0 motor_out2!0
motor_in1?1
motor_in1?0
motor_out1!1 motor_out2!1
motor_in2?2
motor_out1!1.5 motor_out2!1.5
motor_in2?3
motor_out1!1.8 motor_out2!1.8
motor_in1?2
motor_in1?3
motor_out1!1 motor_out2!3
motor_out1!2 motor_out2!2
motor_in1?4
motor_out1!3 motor_out2!1
motor_in1?5
motor_out1!1 motor_out2!3 Or motor_out1!3 motor_out2!1

Figure 5: Movement Controller Model GGAD Diagram

This model has a one to one mapping between the hardware states and the model states. It is responsible for motion control and guiding the robot. However that the outputs of this model do not directly go to hardware and instead go to left and right wheel controllers. The wheel controller receives inputs from motor controller and speed controller and forwards them to motor and signals the motor.

Figure 6 presents the GGAD diagram of a wheel controller atomic model which is part of a robotic vehicle controller. The DEVS formal definition of this model is as follows:

M = <X, S, Y, δ_{ext}, δ_{int}, λ, ta>

X: “rwheel_in1” (Connected to Motor Controller).

S: ”Waiting”, “Sending Brake”, “Sending Spin Clockwise (speed=30)”, “Sending Spin Clockwise (speed=60)”, “Sending Spin Clockwise (speed=90)”, “Sending Spin Counterclockwise (speed=30)”, “Sending Brake for Backward”, “Sending Float”.

Y: “rwheel_out1” (Connected to the wheel).

δ_{ext}: Receives inputs from the input port and initiates appropriate state transitions.

δ_{int}: defines state changes based current state.

λ: based on the input value and the current state sends the following outputs signals to the output port (wheel): 0 for brake, 1 for spinning clockwise in gear 1 (speed=30), 1.5 for spinning clockwise in gear 2 (speed=60), 1.8 for spinning clockwise in gear 3 (speed=90) 2 for spinning counterclockwise (speed=30) and 3 for float (wheels are free, not braking).

ta: real-time advance function.

This model's responsibility is to translate and forward commands from "Motor Controller" model to the wheel of the robotic vehicle. The model is at first in waiting state for infinity and listening to incoming input command from the Motor Controller model. Whenever an input comes it goes to a dummy state for 10 milliseconds and then in the output function produces control signals for wheel and changes back its state to waiting state. Thus whatever command that comes from the Motor Controller model will be reflected to the wheel with 10 ms delay, which in this application is accepted. In this model except the "Waiting" state, other states are dummy states to signal the hardware.

For example when the model is in Waiting mode and an input with value of 2 is received the model changes its state to "Sending Brake for Backward", then after 10ms it generates an output signal with value 0 which tells the motor to brake (the translation of output signals to hardware commands is done in driver model which will be explained in the next section). Right after sending the output, the δ_{int} function is called and changed the state of the model to "Sending Spin Counterclockwise (speed=30)". This state will also last for 10ms. Note that wee need a very small time to make sure that the wheel actually stopped. When the ta(s) of this state is past, the output function produces the moving backward signal and forwards it to the hardware and the internal transition function will change the state of the model back to "Waiting". In this model, while the model is in "Waiting" state the wheel can be in different states. Thus in our approach there is no necessarily one to one mapping between the model states and the hardware states. However that, the model can be designed as a one-to-one mapping between model states and hardware states.

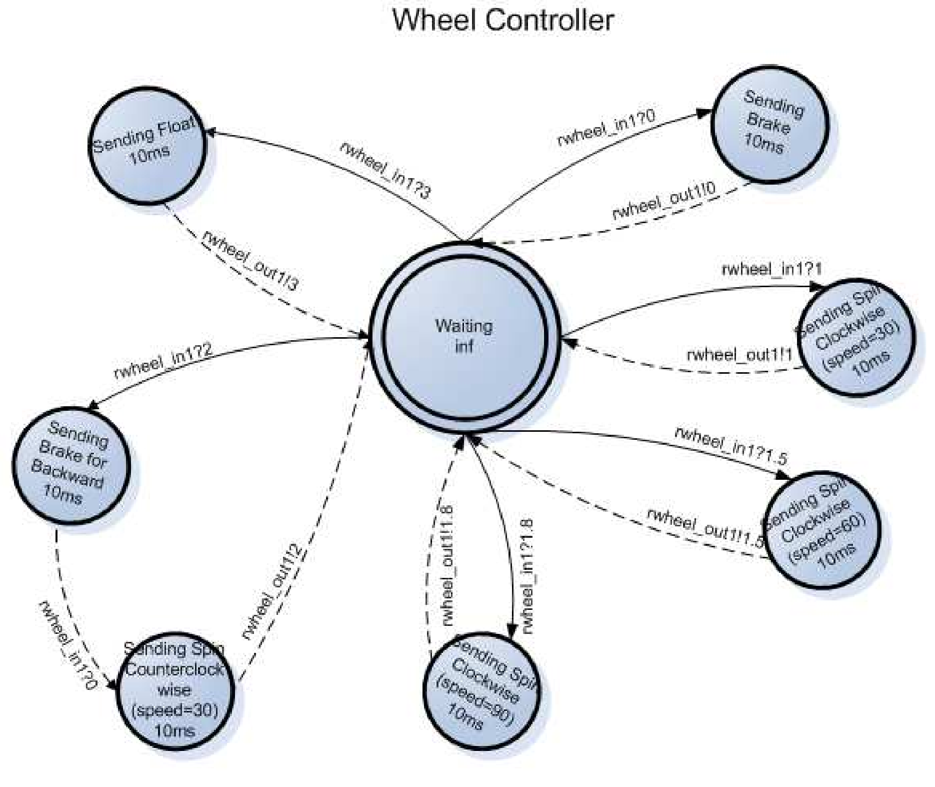

Figure 6: Wheel Controller Model GGAD Diagram

7 CONCLUSIONS

M&S techniques offer significant support for the design and test of complex embedded real-time applications. We showed the use of P-DEVS and RT-DEVS as the basis to present a modified RT-DEVS formalism for developing real-time embedded applications, which allowed us to develop incrementally a sample application including hardware components and RT-DEVS simulated models. We introduced the concept of driver as hardware interface which makes the transition from simulated models to the actual hardware counterparts easier and also makes the design portable on different hardware platforms. Testing and maintenance phases are highly improved due to the use of a formal approach like P-DEVS for modeling. P-DEVS can be applied to improve the development of real-time embedded applications. We introduced a non-changed platform from P-DEVS to RT-DEVS that enables reuse of DEVS and P-DEVS models for real-time applications. The concept of deadline has been added to our RT-DEVS scheme which is a critical issue in real-time and embedded system development. We then analyzed our scheme versus different hardware input generators and elaborated design consideration in different circumstances.

Finally we presented a complex robotic car controller model as a case study, in which we investigated different conditions of a controller model and showed their DEVS models. The use of different experimental frameworks permitted us to analyze the model execution in a simulated environment, checking the model's behavior and timing constraints within a risk-free environment. The simulation results can then be used in the development of the actual application. The integration of hardware components into the system is straightforward by the definition of driver object.

REFERENCES

Zeigler, B. P. 1984. Multifaceted Modeling and Discrete Event Simulation. Orlando: Academic Press.

Zeigler, B. P. 1990. Object-Oriented Simulation with Hierarchical, Modular Models. Orlando: Academic Press.

Thomas D. and Moorby P. 2008. The Verilog hardware description language: Springer Press

Navabi Z. 1997. VHDL: analysis and modeling of digital systems: McGraw-Hill Professional

Bobeanu, C. V, Kerckhoffs, E. J. H and Landeghem, H. V. 2004. Modeling of discrete event systems: A holistic and incremental approach using Petri: ACM Press, New York, NY, USA

Wang J. 1998. Timed Petri Nets: Theory and Application: ACM Press

Raskin J. F. and Thiagarajan P. S. 2007. Formal Modeling and Analysis of Timed Systems: 5th International Conference, Formats, Salzburg, Austria.

Harel D. 1986. Statecharts: A Visual Formalism for Complex Systems: Weizmann Institute of Science, Dept. of Computer Science.

Gill A. 2007. Introduction to the theory of finite-state machines: McGraw-Hill.

Chow A, Kim D, Zeigler B. 1994, Parallel DEVS: A parallel, hierarchical, modular modeling formalism: In Proceedings of Winter Simulation Conference, Orlando, Florida, SCS.

Hong J. S, Song H. H, Kim T. G. and Park K. H, 1997: A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development: Springer Netherlands.

Cho S. M. and Kim T. G. 1998, Real-Time DEVS Simulation: Concurrent, Time-Selective Execution of Combined RT-DEVS Model and Interactive Environment: In Proceeding of 1998 Summer Simulation Conference, Reno, Nevada.

Christen G, Dobniewski A. and Wainer G. 2004. Modeling State-Based DEVS Models CD++: MGA, Advanced Simulation Technologies Conference 2004. Arlington, VA. U.S.A..

AUTHOR BIOGRAPHIES

Mohammad Moallemi

Gabriel Wainer