Designing an Interface for Real-Time and Embedded DEVS

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.
{moallemi,gwainer}@sce.carleton.ca

Keywords: Discrete event simulation, DEVS, Embedded Systems, Real-Time Simulation and Control, Model Based Approach

Abstract

In this work, we are proposing a hardware-in-the-loop model-driven method to develop real-time and embedded applications based on DEVS (Discrete Event Systems Specification) formalism. This approach combines the advantages of a simulation-based approach with the rigor of a formal methodology. This framework can be used to develop embedded applications incrementally, and integrate simulation models with hardware components seamlessly. We have defined structural modifications to the current DEVS abstract simulator, allowing for integration with hardware devices, using external ports of the model and adding hardware control mechanisms. The use of this methodology provides model continuity from the early stages of model design to embedding it on the target. We have discussed the details of implementation of the proposed technique on E-CD++ (a DEVS based toolkit).

1. INTRODUCTION

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. 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.

M&S methodologies and tools have provided means for cost-effective validity analysis for real-time embedded systems[1], [2]. M&S-based testing is a popular technique, which is widely used for the early stages of a project; however, when the development tasks switch towards the target environment, the early models and simulation artifacts are often abandoned. We propose a Model-driven framework to develop embedded systems based on DEVS formalism [3]. DEVS provides a formal foundation to M&S which proved to be successful in different complex systems. This approach combines the advantages of a simulation-based approach with the rigor of a formal methodology. The approach supports rapid prototyping, and encourages reuse. Many existing techniques that have been widely used for the development of embedded and Real-Time systems, also mapped into DEVS models.

The use of DEVS improves reliability (in terms of logical correctness and timing), enables model reuse, and permits reducing development and testing times for the overall process. Consequently, the development cycle is shortened, its cost reduced, and quality and reliability of the final product is improved.

2. RELATED WORK AND MOTIVATION

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).

There has been few works on mapping DEVS models to real-time and embedded environment. A consistent model-based approach using DEVS would have a progressive effect on embedded application development. The main motivation for using DEVS (a mathematical based formalism) for embedded application development is the reliability and portability of this approach. Model continuity from early simulation models to final embedded

implementation, increases portability and reliability in terms of ease of verification for this kind of application.

A Parallel DEVS (P-DEVS) model**Error! Reference source not found.** [4] is described as a set of basic atomic 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:

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

$Y_M = \{(p, v)| p \in OPorts, v \in Y_p \}$ 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

$Q: = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ 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 δ_{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_{con} function. As a result, the new system's state will be the one defined by $\delta_{con}(s, E)$.

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_{con} function.

The Real-Time DEVS (RT-DEVS) formalism [5] is an extension of the DEVS formalism for real-time systems simulation. An atomic model in RT-DEVS formalism (RTAM), is defined as:

$RTAM=<X, S, Y, \delta_{ext}, \delta_{int}, \lambda, ta, ti, \psi, A >$, Where:

X, S, Y, δ_{int}, λ and ta are the same as original DEVS.

$\delta_{ext}: Q \times X \rightarrow S$, an external transition function, where Q is the total state set of $M= \{(s, e)|s \in S \text{ and } 0 \leq e \leq ti(s)|_{max}\}$

ti: a time interval function,

ψ: an activity mapping function,

A: a set of activities, with constraints:

$ta: S \rightarrow A$, $ti: S \rightarrow R^+_{0,\infty} \times R^+_{0,\infty}$,

Where $ti(s)|_{min} \leq t(a) \leq ti(s)|_{max}$, $ti(s)|_{min} \leq ta(s) \leq ti(s)|_{max}$, $s \in S$, $a = ti(s) \in A$ 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 ψ and an activity set A are defined to advance time with an executable activity associated with an event. The regular **ta** time advance function only verifies the correctness of activity mapping time constraints and compensates 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 will not occur 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 [6] a software development methodology for dynamic distributed real-time systems was presented. The methodology is based on DEVSJAVA modeling and simulation environment. It supports model continuity so that a dynamic distributed real-time system can be designed, analyzed and tested by simulation methods, and then migrated to be executed in a distributed network while preserving its control models. To handle the dynamic properties of a distributed real-time system, the variable structure modeling capability is integrated into the proposed methodology. Stepwise simulation methods such as central simulation, distributed simulation, and hardware-in-the-loop (HIL) simulation are developed to incrementally test the control models in a virtual environment. A distributed robotic "team formation" example was developed and presented in the paper to demonstrate how this dynamic system can be developed by applying the proposed methodology in different stages.

In [7], RTDEVS/CORBA, is presented as a modeling and simulation framework, to support the development of distributed real-time systems. The framework supports model continuity for real-time software development from model design to performance evaluation and even to final real-time control. This approach is based on RT-DEVS formalism and maps activities to each state. The authors do not mention details about real-time control part and the focus is on real-time simulation and a case study is also presented.

3. PROPOSED REAL-TIME DEVS APPROACH

In this paper, a more efficient real-time extension to P-DEVS formalism is proposed, which does not change the main formalism and defines driver model 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 mention details of implementation of hardware interaction for a model for embedded control applications and its main application is real-time simulation.

3.1. 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 by real-time. The actual advance of simulation time is the real execution time of δ_{ext} and δ_{int} functions.

In the proposed approach, P-DEVS formalism is used with the following modifications:

1) The time advance function (**ta**) counts the wall clock time, hence the simulation/execution proceeds with real-time clock and events are processed at the wall clock time ticks that they are supposed to be injected to the model. While real execution on hardware, the model also listens to the hardware and accepts hardware inputs
2) The concept of state reflection to the hardware using the output function (λ) is introduced here. In this approach the output function is responsible of reflecting the state change to the actual hardware. Therefore, each hardware device 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 changes the state, based on the current state. All the hardware control signals are produced in the output function.

Thus, the new atomic model is formally defined by:

RTAM = < X, S, Y, δ_{ext}, δ_{int}, δ_{con}, λ, ta >, where:

X, S, Y, δ_{ext}, δ_{int}, δ_{con} 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 function.

This Approach does not modify P-DEVS model definition. Thus, makes model reuse from DEVS and P-DEVS to real-time and embedded simulation/execution possible and provides a verification mechanism, for real-time and embedded application development.

3.2. Hardware Interface and Deadline

The most critical attribute 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, to make sure the deadline of each state is met. Here the concept of deadline is embedded in the driver object of the **Top** model (The top most coupled model containing the entire model hierarchy). Assuming any input that comes to the system can finally produce an output to the hardware, the deadline is defined for each input event from the hardware.

In [8] RT-DEVS has been used with slight modification and addition of the concept of driver for hardware interaction. The main function of the driver model is to translate the inputs from external world to the RT-DEVS model and from RT-DEVS model to the external world. The inputs can come from hardware device, network interface or software interface and outputs can be directed to any of them. All the interactions are through DEVS input and output ports. Every **Top** coupled model port which is connected to an external device has a driver object associated with it which provides the required user defined interface for that specific port. Driver object model provides flexibility in terms of different possible external communications and extensibility in terms of interaction with outside world to RT-DEVS model. The definition for a real-time driver model, 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.

In this approach, all the input and output ports of the **Top** coupled model own a driver object. This lets the model to be portable on any environment platform (the only part that changes is the driver object.) The definition of driver model is limited to the **Top** coupled model, therefore the P-DEVS notation of **Top** coupled model is redefined as follows:

TOPCM = <X, Y, OS, IS, DX, DY, D, {M_d | 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** ∈ Output Signals to Hardware, **oy**∈ Y output port that the signal will be submitted to, pt∈ $R^{+}_{0,\infty}$ processing time from when the associated **is** signal has been received} is the set of hardware output signals.
DX: IS →Xv: converts external hardware inputs signals to input port value (Xv)
DY: Yv →OS: converts output port value to external hardware outputs signals (Yv) with constraint (∀ iy = oy → pt ≤ dl)

3.3. Internal Time Management

The proposed approach does not define activity mapping time constraint. Instead, the deadline of output is used to check the time constraint of each activity. Therefore, activity is not limited to one state of a model and it can be spread over a sequence of. In the other words, a deadline can be placed for a sequence of activities. The time stamp of the messages transferred for triggering state changes and events do not get updated by real-time clock and are the same during the lifetime of the message keeping the need for δ_{con} tie breaking function for atomic models. As a result, the whole model can be considered 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.

3.4. DESIGN CONSIDERATIONS

There are some considerations that must be addressed while designing a real-time model using the proposed approach. One advantage of this approach is that the designer is not forced to map a model state to an activity in the hardware device. Thus, some atomic models or states can only be dedicated to processing, using state changes. To keep track of hardware behavior, an atomic model can be defined for each hardware device (sensors, motors, actuators …).

3.4.1. 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 and rapid input sequence that locks the model because the time interval between two consequent inputs is too small compared with the **ta**(s) of the input state to finish and produce output. To overcome this problem, the model must ignore any input while it is in **ta**(s) of such critical input states to complete the state and produce output. The output will signal an activity in the hardware device and resumes the sensor to its normal condition.

3.4.2. Periodic Inputs

Periodic inputs are those that happen at certain periods of time (e.g. distance sensor). A model that receives this type of input must avoid deadlock that happens because of period < **ta**(s) of the input state. There are two strategies to avoid deadlock with these inputs:

1) If the model is sensitive to certain ranges of input 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). Usually close ranges of data are received close to each in time. Though, the model gets flooded with inputs while it is in **ta**(s) and starts a new external transition each time it receives an input which prevents the model from finishing the state to producing output. (e.g. the robotic car has become close to an obstacle and the distance sensor is sending small ranges which the model is sensitive to them and must react)
2) If the model must react to each input value 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 device to react to the input that receives from the model.

4. IMPLEMENTATION ON EMBEDDED CD++

CD++ [9] is an open-source simulation software which implements the DEVS simulation formalism. In CD++, simulators and coordinators progress through the simulation by exchanging messages as described by the abstract simulation mechanism. CD++ benefits from object-oriented design which allows the developer to make use of powerful object-oriented tools to integrate simulation code with modeling code that will be added by user.

E-CD++ (Embedded CD++) [10] is an extension of CD++ toolkit that has been developed based on P-DEVS formalism which has converted the virtual time function of CD++ into a real-time function (using a time advance function tied to the real-time clock).

Working on E-CD++ can be done writing C++ code in a text-based Linux environment with open source tools. In order to improve the development and simulation experience, an IDE is provided for the E-CD++ simulator as an Eclipse plug-in that contains E-CD++ functionalities. It also has a graphical model designer that supports GGAD (Generic Graphical Advanced environment for DEVS modeling and simulation) diagram [11].

4.1. E-CD++ Software Structure

E-CD++ is modularized in the way that systems' objects (written in C++) run as separate software modules

with well-defined behaviors and independent functionalities. Four main components of E-CD++ are: **Main Runtime System**, **Modeling Subsystem**, **Runtime Subsystem** and **Messaging Subsystem**.

Main Runtime System manages the overall aspects of the runtime system. It is the first object that is created when the Runtime System starts. In general, it does the following tasks in sequence:

Registers Atomic model objects, which are C++ objects derived from the Atomic class;

Reads in the external events (from event file) and builds an external events table;

Reads in the model file and builds the model hierarchy;

Creates the *Root* Coordinator and triggers it to run

The *Runtime Subsystem* consists of Runtime Systems, coordinators, and the Processors Manager. The Processors Manager maintains a hashing table of pointers to *Processor* class objects, such that actions, such as searching, can be performed upon those objects.

The *Root* Coordinator is a special Coordinator that manages and controls the Runtime cycles. It receives the incoming external events and sends the corresponding External Messages to the underlying coupled and atomic models in the hierarchy of model objects. The *Root* coordinator advances the Global Runtime System Time.

The *Messaging Subsystem* consists of the *Message Manager* and various *Messages* class objects. Processors and coordinators send messages via the *Messages Manager* which is responsible for delivering messages. The incoming messages are first buffered into the *Message Queue* and are processed by the *Messages Manager* in FIFO order. Each *Message* object contains information to identify the *sender* and the *receiver*. A time-stamp for the message and an associated *value* and *port* are also included in the packet.

The *Modeling Subsystem* provides a logical representation of the DEVS models defined by the modeler. The subsystem is composed by the *Models Manager* and the *DEVS Models Hierarchy Tree*. The *Models Manager* manages the models hierarchy. More precisely, it does the following two tasks:

Main Runtime System registers Atomic model objects, and *Models Manager* creates and manages the Atomic models objects database (a dictionary data structure that stores Atomic model string names). It also creates the *Models Hierarchy Tree* which is composed by atomic and coupled models.

4.2. Proposed Approach on E-CD++

E-CD++ class structure has been modified to implement the proposed model.

Port Admin object has been added to the software architecture which maintains a hashing table of pointers to the **Top** model ports that are connected to hardware. The hardware driver for each port will be programmed by user for any specific hardware.

Driver object has been added which provides hardware initialization and termination functions and catches incoming real-time events from hardware devices and sends output commands to hardware by calling user implemented driver objects. The *Driver* class is also responsible for providing interruptive and periodic behavior for input hardware.

The followings are modifications and additions to existing objects to implement hardware interface feature on E-CD++:

Main Simulator initializes and terminates hardware connections using *Root* coordinator functions and registers **Top** model ports that are connected to hardware.

Port object has been modified to provide input/output driver for input/output **Top** model ports. Each **Top** model port which is connected to the hardware will be implemented by user as a child of *Port* class object.

Figure 1 shows the modified E-CD++ software structure in which yellow objects are the main hardware related objects.

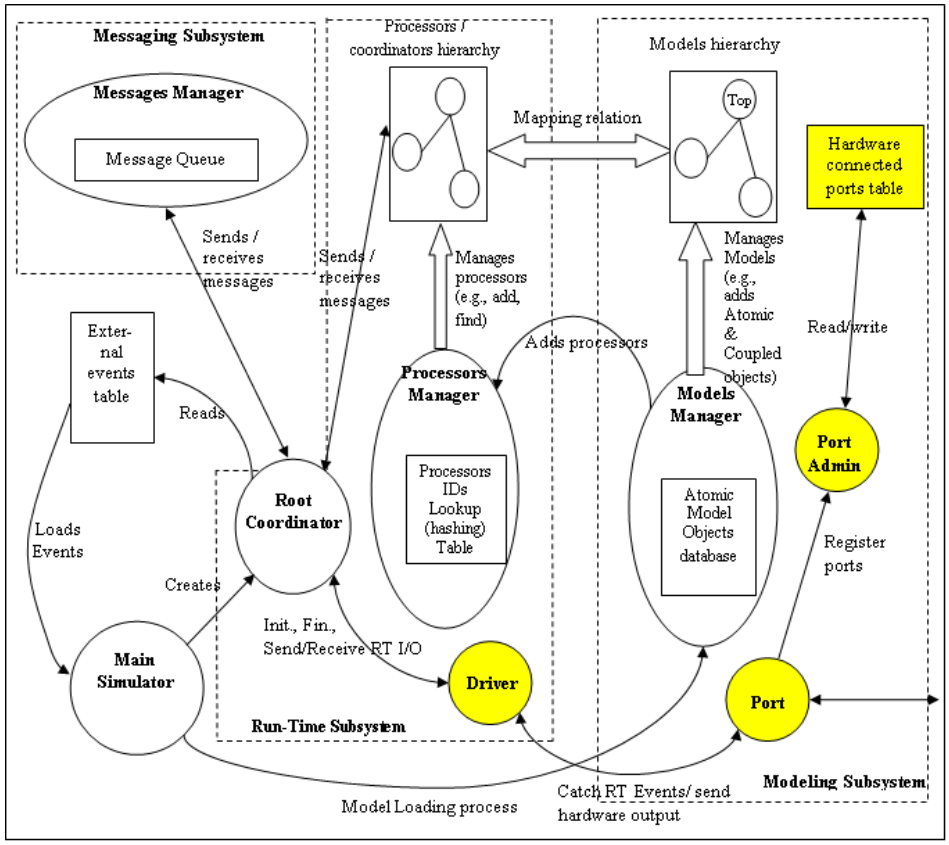


Figure 1. E-CD++ modified software structure

5. ROBOTARM MODEL

RobotArm is a sophisticated robotic arm that can lift, pivot, and grab objects by its claw. It is consisted of Sound, Touch and color sensors and two motors: one for moving the arm up and down and one for the claw to grab and release.

A DEVS model specification has been defined for RobotArm model which is shown in Figure 2. There is a Top coupled model that contains five atomic models. Sound, touch and color sensor atomic models control the functionality of sound, touch and color sensors. These models receive inputs of the sensors and forward them to the arm controller model and provide interruptive or periodic behavior for sensors. The arm controller model is responsible for controlling the arm motor. It receives inputs from sensor models and sends outputs to color sensor model, claw model and arm motor. The claw model is only responsible for the claw motor.

At the start of the execution the arm motor starts spinning and brings the arm down until it touches the ball. As soon as the touch sensor detects a ball, the arm stops and the color sensor provides the intelligence to the robot arm to decide what to do depending on the color of the ball. If it is a red ball, the claw grabs the ball and the arm goes back up, taking the ball with it. If it is a blue ball the claw does not grab and the arm just goes back up. The touch and sound sensor models define interruptive inputs, but the color sensor uses periodic inputs.

Figure 2 illustrates the DEVS model hierarchy of the RobotArm model.

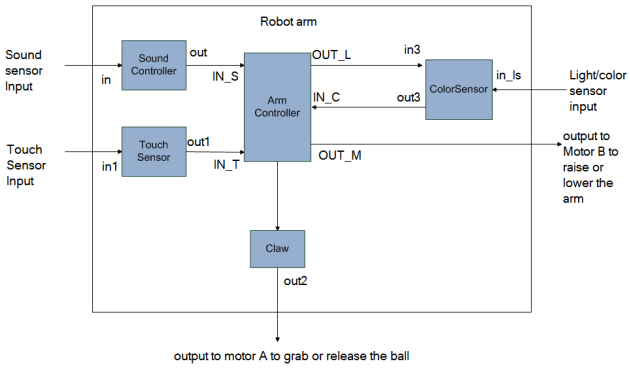


Figure 2. DEVS hierarchical model for RobotArm

The DEVS formal specifications for Arm controller model is as follows:

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

X: IN_S, IN_T, IN_C.

S: Idle, Prepare_Going_Down, Gowing_Down, Prepare_Stop, Get_Action, Prepare_Grab, Grab, Prepare_Going_Up, Going_Up.
Y: OUT_L, OUT_M, OUT_C.
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 (arm motor): 1 for going down, 2 for Stop, 5 for going up.
ta: real-time advance function for each state.

Figure 3 illustrates the GGAD diagram of the arm controller atomic model. Note that the continuous lines show external transitions and dashed lines show internal transitions between states. The labels on external transitions show the input ports and input values and the labels on internal transitions show output ports and output values.

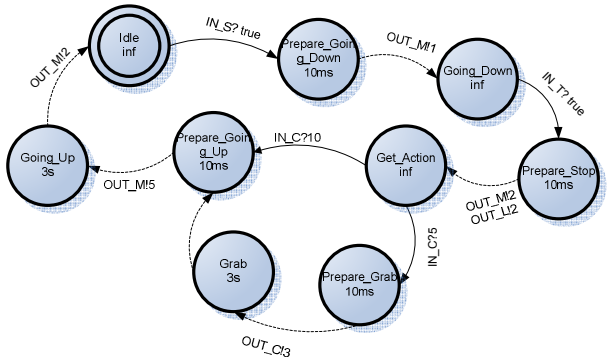


Figure 3. GGAD diagram of the Arm Controller atomic model

5.1. Simulation and Execution Results

Variety of tests for different scenarios has been carried out. The proposed implementation enabled E-CD++ to execute the simulation model on the target device with hardware inputs and outputs. A robot has been built and the execution model has been run on the hardware both with eventfile inputs and hardware inputs.

RobotArm model outputs of E-CD++ is shown in Table 1 for blue ball scenario.

Note that the output file only shows the outputs of the Top model ports. The first row shows that at the time 4 seconds and 293 milliseconds from the start of the simulation the arm controller sent the value of 1 to the output port "out_m" and no deadline has been specified for inputs. Value 1 means going down for the arm motor, which shows that, the sound sensor model detected a sound command and forwarded it to the arm controller model. The second row shows the value 2 which means stop, line 3 shows value 5 meaning "going up" and row 4 shows stop.

Table 1. ECD++ outputs for blue ball

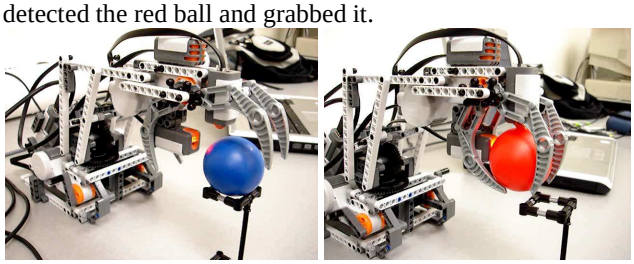
Time	Deadline	Output Port	Output Value
00:00:04:293	No Deadline	out_m	1
00:00:11:274	No Deadline	out_m	2
00:00:11:235	No Deadline	out_m	5
00:00:14:437	No Deadline	out_m	2

Table 2 shows the output file for red ball scenario in which, first row shows the start of going down by arm motor, second row: stop, third row: grab by claw motor (port out2 of Top model), forth row: stop by claw motor, fifth row: go up by arm motor and sixth row: stop by arm motor.

Table 2. ECD++ outputs for red ball

Time	Deadline	Output Port	Output Value
00:00:01:053	No Deadline	out_m	1
00:00:03:364	No Deadline	out_m	2
00:00:03:472	No Deadline	out2	3
00:00:05:467	No Deadline	out2	6
00:00:06:473	No Deadline	out_m	5
00:00:09:475	No Deadline	out_m	2

Figure 4.a shows a shot of RobotArm in the lab when it detected the blue ball while it discarded it and is moving back up and Figure 4.b shows the RobotArm while it detected the red ball and grabbed it.



a) *b)*

Figure 4. a) RobotArm discarding blue ball. b) RobotArm grabbing red ball

6. CONCLUSIONS

M&S techniques offer significant support for the design and test of complex embedded real-time applications. In this paper the use of DEVS as the basis for developing model-based embedded systems has been showed, which allowed the incremental development of the sample case study application including hardware components and DEVS simulated models. The use of different experimental frameworks permitted analyzing the model execution in a simulated environment, checking the model's behavior and timing constraints within a risk-free environment. The simulation results were then used in the development of the actual application. The integration of hardware components

into the system was straightforward. The transition from simulated models to the actual hardware counterparts can be incremental, incorporating deployed models into the framework when they are ready. Testing and maintenance phases are highly improved due to the use of a formal approach like DEVS for modeling.

The proposed approach has been implemented on CD++, an open-source DEVS tool that has been built following DEVS formal definitions and implementation details are presented. A simple robotic case study model is developed and presented.

The proposed approach offers following advantages over existing DEVS based real-time approaches:

The same models that are defined for DEVS and P-DEVS formalisms can be reused.

Hardware interface definition is clear and accompanied by the model (each input and output port that is connected to hardware is specified and the output values are also predefined with the model unlike RT-DEVS in which each state contains a hardware activity).

The proposed hardware driver can be easily implemented on the existing DEVS tools.

DEVS output function has been used to reflect state changes to the hardware which makes it more formal and robust.

Design considerations with different types (interruptive and periodic) of inputs are discussed.

References

[1] B. Alpern; F. Schneider, “Verifying Temporal Properties without Temporal Logic,” ACM Trans. Programming Lang. and Systems, Vol. 11, No. 1, 1989, pp.147-167.

[2] A. Alfonso., V. Braberman, D. Garbervetsky, N. Kicillof, A. Olivero, F. Schapachnik. “VInTiMe: Combining High-Level Finesse with Low-Level Muscle to Verify Real-Time Systems”. In Proc. of the First International Conference on Principles of Software Engineering, PRISE 2004.

[3] B. Zeigler, T. Kim, H. Praehofer. “Theory of Modeling and Simulation”. Academic Press 2000, ISBN-10: 0127784551.

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

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

[6] Hu, X.; Zeigler, B.P. “Model Continuity in the Design of Dynamic Distributed Real-Time Systems”, IEEE Transactions on Systems, Man And Cybernetics— Part A: Systems And Humans, 35: 6, pp. 867- 878, November, 2005.

[7] Cho, Y. K.; Hu, X.; Zeigler, B.P. “The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems”, SIMULATION: Transactions of the Society for Modeling and Simulation International, Volume 79, Number 4, 2003.

[8] Cho S. M. and Kim T. G. “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.

[9] Wainer, G. "CD++: a toolkit to define discrete-event models". Software, Practice and Experience. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002.

[10] YU, J.; WAINER, G. “E-CD++: a tool for modeling embedded applications”. In Proceedings of the 2007 SCS Summer Computer Simulation Conference. San Diego, CA. 2007.

[11] G. Christen, A. Dobniewski and G. Wainer, "Modeling State-Based DEVS Models in CD++". In Proceedings of MGA, Advanced Simulation Technologies Conference 2004 (ASTC'04). Arlington, VA. U.S.A.