Model-Based Development of Embedded Systems with RT-CD++

Gabriel Wainer **Ezequiel Glinsky**

Dept. of Systems and Computer Engineering
Carleton University
4456 Mackenzie Building
1125 Colonel By Drive
Ottawa, ON. K1S 5B6. CANADA.
gwainer@sce.carleton.ca

Abstract

The Modeling and Simulation-Driven Engineering (MSDE) approach relies on simulation-based modeling for developing components of real-time embedded systems. We propose the use of the DEVS formalism for MSDE activities. We present the MSDE of an application using incremental development, seamlessly integrating hardware components with models simulated in CD++, a DEVS modeling simulation tool. We show how to create different experimental frameworks, allowing the analysis of model execution in a risk-free environment. The approach allows secure, reliable testing, analysis of different levels of abstraction in the system, and model reuse.

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 [SER00, DP02]. M&S is widely used for the early development stages; however, when the development tasks switch towards the target environment, the early models and simulation artifacts are often abandoned [Pea03]. The Modeling and Simulation-Driven Engineering (MSDE) initiative aims to integrate the use of M&S as a fundamental cornerstone in all aspects of real-time embedded system engineering. MSDE proposes a discrete-event simulation architecture to be used as the final target architecture for products. The approach supports rapid prototyping, encourages reuse, and lends itself to the development and maintenance of multiple consistent artifacts [Pea03]. We present a MSDE framework based on the DEVS (Discrete EVents Systems specification) formalism [ZKP00]. DEVS provides a formal foundation to M&S that 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.

CD++ [Wai02] is a software implementation of DEVS with extensions to support real-time model execution [GW02a]. We will explain how to use the MSDE frame to seamlessly integrate simulation models with hardware components. The method proposes to start by developing models entirely built in CD++ and to replace them incrementally with hardware surrogates at later stages of the process. The transition can be done in incremental steps, incorporating models in the target environment after thorough testing in the simulated platform. 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.

DEVS AND CD++

DEVS [ZKP00] is a formal M&S framework based on systems theory. DEVS has well-defined concepts for coupling of components and hierarchical, modular model composition. DEVS defines a complex model as a composite of basic components (called **atomic**), which can be hierarchically integrated into **coupled** models. A DEVS atomic model is described as:

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

Every state **S** is associated with a lifetime **ta**, which is defined by the time advance function. When an event receives an input event ***X***, the external transition function $\boldsymbol{\delta_{ext}}$ is triggered. This function uses the input event, the current state and the time elapsed since the last event in order to determine which is the next model's state. If no events occur before the time specified by the time advance function for

that state, the model activates the output function $\boldsymbol{\lambda}$ (providing outputs ***Y***), and changes to a new state determined by the internal transition function $\boldsymbol{\delta_{int}}$.

A DEVS coupled model is defined as:

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

Coupled models are defined as a set of basic components $\mathbf{M_i}$ ($i \in \mathbf{D}$) interconnected through their interfaces (***X, Y***). The translation function $\mathbf{Z_{ij}}$ converts the outputs of a model into inputs for others using I/O ports. To do so, an index of influencees is created for each model ($\mathbf{I_i}$). This index is used to connect outputs in model M_i are connected with inputs in the model M_j ($j \in I_i$). The formalism is closed under coupling, therefore, coupled and atomic models are semantically equivalent, which enables model reuse.

The execution of a DEVS model is defined by an abstract mechanism that is independent from the model itself. DEVS also permits defining independent experimental frames for the model, that is, a set of conditions under which the system is observed or experimented with. The **CD++** toolkit [Wai02] implements DEVS theory. Atomic models can be defined using C++. Coupled models are defined using a built-in language that follows DEVS formal specifications. Figure 1, shows parts of *ButtonInputModule*, a button controller model that is one of the components of a cruise control system (CCS) modeled with CD++ [TMG03].

```
ButtonInputModule::ButtonInputModule
        ( const string &name ) : Atomic( name ),
in_BUTTON( addInputPort( "in_BUTTON") ),
out_ON( addOutputPort( "out_ON") ), (...),
out_RESUME( addOutputPort( "out_RESUME") )
{   reactionTime = VTime( 0, 0, 0, 15 );  }

Model &ButtonInputModule::externalFunction
        ( const ExternalMessage &msg ) {
  if( msg.port() == in_BUTTON )   {
    inType=(int)msg.value();
    holdIn( active, reactionTime );
  } }

Model &ButtonInputModule::outputFunction
        ( const InternalMessage &msg ) {
  switch(inType) {
    case ON: //take action {
      sendOutput( msg.time(), out_ON, HIGH) ;
      break; }
    case OFF: //take action {
      sendOutput( msg.time(), out_OFF, HIGH) ;
      break; }
  ...} }

Model &ButtonInputModule::internalFunction
        ( const InternalMessage & ) {
  passivate();}
```

Figure 1. Specification of *ButtonInputModule* in CD++.

RT-CD++ [GW02a] uses the real-time clock to trigger the processing of discrete events in the system. Figure 2 outlines the processor's hierarchy generated by RT-CD++ in order to execute the CCS model. The root coordinator created at the top level manages the interaction with an experimental frame in charge of testing the model, and returns outputs. A coordinator is created to handle the coupled model *ProcModule*, whereas simulators objects are created to handle the atomic *ButtonInputModule* and *outputModule*. Timing constraints (deadlines) can be associated to each external event. When the processing of an event is completed, the root coordinator checks the deadline. In this way, we can obtain performance metrics (number of missed deadlines, worst-case response time).

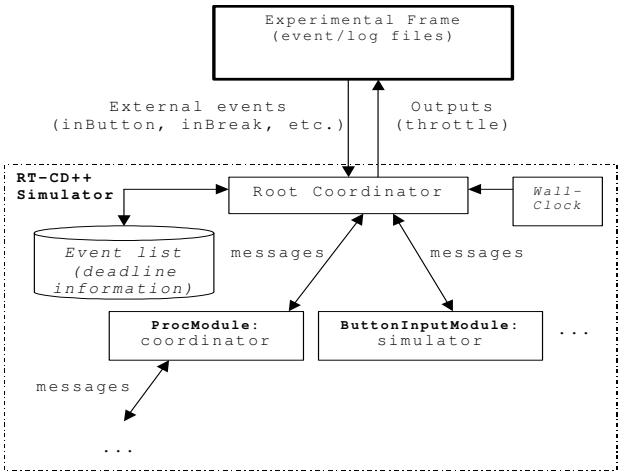

Figure 2. RT-CD++ simulation scheme

We thoroughly tested the execution performance of RT-CD++, and obtained constrained overhead results (2% to 3% for fairly large models). We then explored hardware-in-the-loop simulations [LPW03]: we built a model of the CODEC of Analog Devices 2189M EZ-KITLITE [AD00]. Different tests showed the feasibility of the approach, as we were able to reproduce simulated results in the HARDWARE SURROGATE. Nevertheless, when building components on the board, some of the existing models needed rework (due to the use of an IDE in charge of the communications between CD++ and the hardware). These problems were solved by incorporating communication facilities into CD++.

APPLYING CD++ FOR MSDE

The MSDE approach was used to build the software for an elevator servicing a four-floor building. We started by modeling the entire system in CD++, using the model structure presented in Figure 3. The system is conformed by an elevator control unit (ECU), one coupled component (an elevator box), and three atomic components. The elevator box is formed by two atomic models (the engine and a sensor controller). The button and sensor controllers were defined as atomic components (reusing parts of the atomic model *ButtonInputModule* presented before).

These models, which are defined as previously showed in Figure 1, receive events from the environment, and forward them to the ECU, resembling the real components of the system. The display controller activates LEDs (indicating that a button has been pressed), and a digital display (showing the direction of the elevator). The ECU

receives input signals from sensors and buttons, determining the current location, the floor selected by the user, and dispatching the elevator in the required direction. It also sends outputs to the display controller informing the status of the elevator.

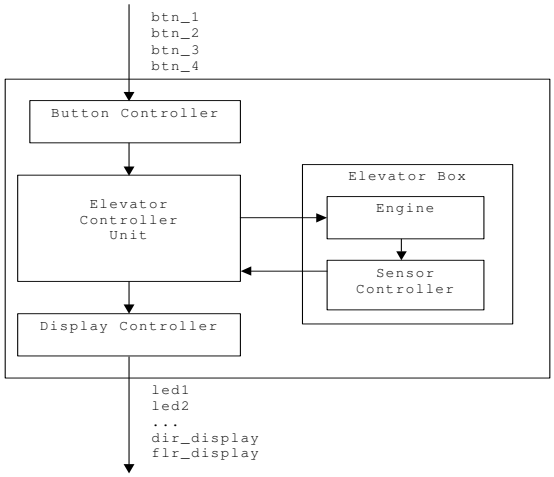

Figure 3. Scheme of the elevator system (entirely in CD++)

Most of the logic of the ECU is located in the external transition function, which handles the incoming events (a new floor is requested) and schedules the next internal transition function to activate or deactivate the engine or to display a new value (e.g., the elevator starts moving or a new floor is reached). Users can define the activation time for the engine, customizing its timing behavior.

Different experimental frames were applied to this model, allowing the analysis of different scenarios. We started by analyzing the behavior of each submodel independently (using the specifications for their physical counterparts) and then, we conducted integration tests. Figure 4 shows a sample event file for one of such experiments.

```
Time        Deadline    In-port   Out-Port      Value
00:11:500   00:11:700   btn_3     led3           1
00:14:600   00:14:800   sensor_2  flr_display    1
00:19:500   00:19:700   sensor_3  flr_display    1
00:25:100   00:25:300   btn_4     led4           1
```

Figure 4. Experimental frame for the elevator system

Initially, the elevator is on the first floor and there are no pending events. The first event represents a user willing to go to the third floor. The event occurs at time 00:11:500, and the simulator receives it via input port *btn_3*. As a result, we expect to turn on the button LED in less than 200 ms. The second event in the list represents the activation of *sensor_2* (i.e., the elevator has reached the second floor). In this case, we expect an output via port *flr_display* before 00:14:800. The value of 1 represents the activation of buttons and sensors. Figure 5 shows the outputs generated by the real-time simulator for this experiment.

```
Time        Deadline    Out-port       Value
00:11:510   00:11:700   led3           1
00:11:510               dir_display    1
00:14:610   00:14:800   flr_display    2
00:19:510   00:19:700   led3           0
00:19:510               flr_display    3
00:19:510               dir_display    0
...
```

Figure 5. Outputs generated by the elevator system

As we can see, the deadlines were met in every case. We used different experimental frames to thoroughly test this model, and once satisfied with its behavior, we progressively started to replace simulated components with their hardware counterparts. The first step was to replace the button controller model. User requests are now received on the button pad and sent to the simulated model. The rest of the components remain unchanged from the architecture described in Figure 3.

```
components:  elevBox   ec@ECU   dis@Display
in   : button_1  button_2  button_3  button_4
out  : flr_display
link : button_1 button_1@ec
link : button_2 button_2@ec
...
link : sensor_1@elevBox sensor_1@ec
link : sensor_2@elevBox sensor_2@ec
...
link : floor_disp@ec  flr_display@dis
link : floor_disp@ec  floor_disp
...
[elevBox]
components:  sb@SensorController   eng@Engine
in   : activate  direction
out  : sensor_1 sensor_2 sensor_3 sensor_4
link : activate    activate@eng
link : direction   direction@eng
link : sensor_1@sb sensor_1
...
```

Figure 6. Coupled model: button controller in hardware

Here, *elevBox* is a coupled component, whereas *ec* and *dis* are atomic. The top model input ports are used to receive events from the button controller now running in the external board. Replacing a CD++ component with its counterpart running in the external devices is straightforward, since the modeler only has to remove the original model from the coupled model definition. Likewise, testing this model only requires reusing the previously defined experimental frames. As the button controller model was built using the hardware specifications for the actual buttons, and the interfaces of the submodels do not change, the transition is transparent. The results obtained are equivalent to Figure 5, regardless the changes.

After conducting extensive tests, we also moved the display controller to the microcontroller. By simply removing the display controller from the coupled model specification in Figure 6, we were able to execute the new application without any modifications.

The final step was to implement the complete elevator system on the microcontroller. Figure 7 shows the scheme for this experimental frame, in which only the engine component is still simulated in CD++.

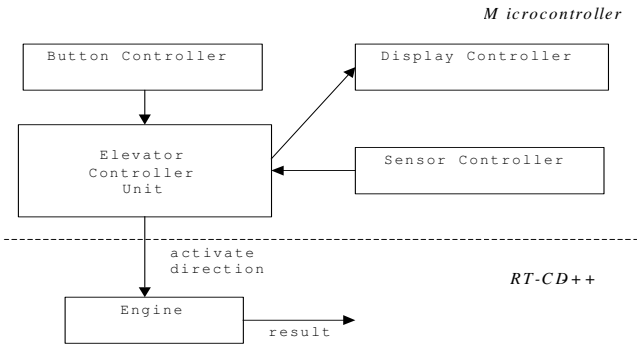

Figure 7. Elevator controller implemented in hardware

Figure 8 shows the events generated by the model running in the microcontroller, which represents users requesting service from. Figure 9 shows the activation and deactivation of the engine when the requests are received, which is the result of the activity in the microcontroller (up=1, down=2, stop= 0).

Time	Port	Value
00:06:120	direction	1
00:06:130	activate	1
00:15:930	activate	0
00:56:800	direction	2
00:56:810	activate	1
...		

Figure 8. Event log generated by the engine model

Time	Out-port	Value
00:06:130	result	1
00:15:930	result	0
00:56:810	result	2
01:01:130	result	0
01:22:720	result	2
...		

Figure 9. Outputs for the model in Figure 7.

CONCLUSION

M&S techniques offer significant support for the design and test of complex embedded real-time applications. We showed the use of DEVS as the basis for MSDE, which allowed us to develop incrementally a sample application including hardware components and DEVS simulated 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 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. DEVS can be applied to improve the development of real-time embedded applications.

The concept of experimental framework eases the testing tasks, as one can build independent testing frames for each submodel. Thanks to the closure under coupling property, models can be decomposed in simpler versions, always obtaining equivalent behavior. Likewise, model's functions can be reused by just associating them with new models as needed. For instance, we are now building an extension to the examples presented here that will handle four different elevators in a 20-floor building. Extending the model here presented requires modifying only the external transition function in the ECU, and defining a new coupled model including the four elevators, while keeping the remaining methods unchanged.

Currently we are developing a new version of RT-CD++ to run in an embedded platform (one running on the bare hardware, and the second version on top of RT-Linux). We are also developing a verification toolkit to use the timing properties of the DEVS models under development. In this way, we will have an environment for MSDE in which the user creates models, test them in the simulated environment, uses verification tools to analyze timing properties, and downloads the resulting application to the target platform, being able to provide rapid prototyping and enhanced development capabilities.

REFERENCES

[AD00] Analog Devices. "ADSP-218x DSP Hardware Reference". 2000

[GW02a] Glinsky, E.; Wainer, G. "Definition of Real-Time simulation in the CD++ toolkit". Proc. of SCS Summer Comp. Simulation Conference. San Diego, USA. 2002.

[LDNA03] Ledeczi, A.; Davis, J.; Neema, S.; Agrawal, A. "Modeling methodology for integrated simulation of embedded systems". ACM TOMACS 13(1), 82-103. 2003.

[LPW03] Li, L.; Pearce, T.; Wainer, G. "Interfacing Real-Time DEVS models with a DSP platform". Proc. of Industrial Simulation Symposium. Valencia, Spain. 2003.

[Mot02] Motorola Inc. MC68HC812A4 Data Sheet. 2002.

[Pea03] Pearce, T. "Simulation-Driven Architecture in the Engineering of Real-Time Embedded Systems". Proc. of RTSS-WIP. Cancun, Mexico. 2003.

[SER00] Schulz, S.; Ewing, T.C.; Rozenblit, J.W. "Discrete Event System Specification (DEVS) and StateMate StateCharts Equivalence for Embedded Systems Modeling". Proc. of 7th IEEE Intl. Conf on Eng. of Comp. Based Systems. 2000.

[TMG03] Thurairasa, S.; Mahendran, V.; Gnanapragasam, C.; "Simulating a Control Cruise System with CD++". Technical report. SCE, Carleton University. 2003.

[Wai02] Wainer, G. "CD++: a toolkit to develop DEVS models". Software-Practice and Exp. 32, 1261-1306. 2002.

[ZKP00] Zeigler, B.; Kim, T.; Praehofer, H. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. 2nd Edition. Academic Press. 2000.