Proceedings of the 2004 Winter Simulation Conference
R .G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds.

MODELING AND SIMULATION OF HARDWARE/SOFTWARE SYSTEMS WITH CD++

Ezequiel Glinsky
Gabriel Wainer

Dept. of Systems and Computer Engineering
Carleton University
4456 Mackenzie Building
1125 Colonel By Drive
Ottawa, Ontario K1S 5B6 CANADA

ABSTRACT

Modeling and simulation (M&S) methodologies can be useful in the development of hardware-in-the-loop applications. CD++ is a toolkit with support for real-time model execution that implements DEVS, a sound, formal M&S framework allowing hierarchical, modular model composition and component reuse. We present a methodology that uses CD++ to develop hybrid hardware/software systems. The technique enables incremental transition from the simulated models to the actual hardware counterparts, supports experimental frameworks to facilitate testing in a risk-free environment, encourages component reuse, and allows developing models with different levels of abstraction. CD++ can reduce cost and time-to-market of hardware-in-the-loop applications, and preserves the benefits of a formal M&S methodology like DEVS.

1 INTRODUCTION

Modeling and simulation (M&S) have gained popularity in a wide variety of fields ranging from biotechnology to game design, from aerospace engineering to economics, from logistics management to fluid dynamics. Scientists and engineers use M&S methodologies and tools to understand and analyze complex phenomena under risk-free environments. Moreover, M&S is used to develop new systems and to improve existing ones in a cost-effective manner. Using a simulated environment, it is possible to verify the correctness of the system under different conditions.

The development of hardware-in-the-loop applications is a challenging process in which M&S can become essential. These applications are inherently complex as a result of the high degree of interaction between software and hardware components. Since different parts of the system are often deployed in parallel and therefore are not available, it is difficult to perform thorough testing in early stages of the development process. Development teams face delays waiting for components to be ready, affecting the time-to-market of the application.

We present a methodology to use M&S with hardware-in-the-loop applications. This approach combines the advantages of a simulation-based approach with the rigour of a formal methodology. DEVS (Discrete EVents Systems specifications) (Zeigler, Kim, and Praehofer 2000) is a formal foundation to M&S, proved to be successful in a wide range of complex systems.

CD++ (Wainer 2002) is a M&S software that implements DEVS theory with extensions to support real-time model execution (Glinsky and Wainer 2002a). CD++ was used as the base for our development, building on previous research focused on real-time applications with hardware-in-the-loop (Li, Pearce, and Wainer 2003).

We will explain how to use this framework to seamlessly integrate simulation models with hardware components. Initially, we develop models entirely in CD++, and we replace them incrementally with hardware surrogates at later stages of the process. Thus, it is possible to make the transition in incremental steps, incorporating models in the target environment with hardware-software components after thorough testing in the simulated platform. The use of this methodology shortens the development cycle and reduces its cost.

Our approach does not impose any order in the deployment of the actual hardware components, providing flexibility to the overall process. 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.

2 BACKGROUND

The **DEVS** (Discrete EVents Systems specifications) formalism (Zeigler, Kim, and Praehofer 2000) is a M&S framework based on systems theory. DEVS has well-defined concepts for coupling of components and hierar-

chical, 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:

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

where ***X*** is the set of input events, **S** is the set of discrete states, ***Y*** is the set of output events, $\boldsymbol{\delta_{int}}$ is the internal transition function, $\boldsymbol{\delta_{ext}}$ is the external transition function, $\boldsymbol{\lambda}$ is the output function, and **ta** is the time advance function. Every state is associated with a lifetime, which is defined by the time advance function. When an event receives an input event, the external transition function 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 (providing outputs), and changes to a new state determined by the internal transition function.

A DEVS coupled model is defined as:

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

where ***X*** is the set of input events, and ***Y*** is the set of output events. **D** is an index of components, and for each $i \in D$, $\mathbf{M_i}$ is a basic DEVS model (atomic or coupled). $\mathbf{I_i}$ is the set of influencees of model i. For each $j \in I_i$, $\mathbf{Z_{ij}}$ is the i to j translation function.

A coupled model is composed by a set of basic models (i.e., atomic or coupled) interconnected through their interfaces. The translation function, Z_{ij}, converts the outputs of a model into inputs for others using I/O ports. An index of influencees is created for each model (I_i), determining the destination models for the outputs. This index is used to connect outputs in model M_i are connected with inputs in the model M_j (for each j in I_i). The formalism is closed under coupling, therefore, coupled and atomic models are semantically equivalent, which enables model reuse.

DEVS models can be executed by an abstract mechanism that is independent from the model itself. As a result of this explicit separation of concerns between modeling and simulation, it is possible to verify each layer independently. 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. Experimental frames formulate the objectives that motivate the project (Zeigler, Kim, and Praehofer 2000). Within the conditions imposed by an experimental frame, the modeler observes the behaviour of the system and determines its correctness.

CD++ (Wainer 2002) is a M&S toolkit that implements DEVS theory. Atomic models can be defined using a state-based approach (coded in C++ or an interpreted graphical notation). The toolkit has been used to model and simulate a wide variety of applications, such as urban traffic, robot path planning, and computer architectures (Wainer 2002).

Figure 1 shows the definition of an atomic model that represents the behaviour of an Ethernet switch using CD++.

```
EthernetSwitch::EthernetSwitch
( const string &name ) : Atomic( name ),
in1( addInputPort( "in1" ) ),
in2( addInputPort( "in2" ) ),
in3( addInputPort( "in3" ) ),
enable( addInputPort( "enable" ) ),
disable( addInputPort( "disable" ) ),
out1( addOutputPort( "out1" ) ), ... , { }

Model &EthernetSwitch::externalFunction
( const ExternalMessage &msg ) {

  if ( (state() == passive) &&
  ((msg.port() == in1) && (enabled1 == 1)) ||
  ((msg.port() == in2) && (enabled2 == 1)) ||
  ((msg.port() == in3) && (enabled3 == 1)) ) {
    request = msg.value();
    request_waiting = 1;
    holdIn (active, delay);
  }

  if ( (state() == passive) &&
  (msg.port() == disable) )   {
    if ( msg.value() == 1 ) enabled1 = 0;
    if ( msg.value() == 2 ) enabled2 = 0;
    if ( msg.value() == 3 ) enabled3 = 0;
    request_waiting = 0;
    holdIn (active, delay);
  }

  if ( (state() == passive) &&
           (msg.port() == enable) )  {
    if ( msg.value() == 1 ) enabled1 = 1;
    ...
    holdIn (active, delay);
  }

}

Model &EthernetSwitch::internalFunction
( const InternalMessage &msg ) {

  request_ready = 0;
  passivate();
  return *this ;

}

Model &EthernetSwitch::outputFunction
( const InternalMessage &msg ) {

  if (request_ready == 1)   {
    if (( request == node_1 ) && ( hab1 == 1 ))
      sendOutput( msg.time(), out1, request ) ;

    if ((request == node_2 ) && ( hab2 == 1 ))
      sendOutput( msg.time(), out2, request ) ;
  ...

}
```

Figure 1: Specification of EthernetSwitch in CD++

Most of the logic of the *EthernetSwitch* is located in the external transition (δ_{ext}). This function determines what to do with the incoming packets. External events arriving via the input ports *in1*, *in2*, and *in3* represent packets received from the network, whereas *enable* and *disable* are used to indicate which ports are working. The next internal event (δ_{int}) is scheduled by the *holdIn* method, which implements the time advance function (*ta*). For example, if an event is received via *in2* and the port is enabled (*enabled2*), the model stores the value received and schedules an internal transition. When the time indicated by the variable *delay* expires, the output function (λ) notices that there is a request ready and directs the output to the corresponding port, according to the value previously received (i.e., request). Enabling and disabling ports do not generate any output. The internal transition function clears the *request_ready* flag and passivates the model (i.e., sets the next internal transition time to infinity).

CD++ also enables the user to define coupled models by using a built-in specification language that follows DEVS formal specifications. Once an atomic model is defined (as in Figure 1), it can be integrated into a coupled model as the one presented in Figure 2.

```
components:  server1   server2
components:  client    eth@EthernetSwitch
in:  eth_enable  eth_disable
in: hss1_start  hss1_stop  hss2_start hss2_stop
...
out: packets     status
link: server_out@serv1     in1@eth
link: out1@eth             server_in@serv1
link: server_out@serv2     in2@eth
link: out2@eth             server_in@serv2
...

[eth]
delay:  00:00:01:000
node_1: 1      node_2: 2      node_3: 3

[client]
components: WSclient   clientNet@Network   com-
ponents: hsclient@HSClient
in:   hs_start    hs_stop   client_in
out:  client_out
link: hs_start    start@hsclient
...
[WSclient]
components: selclient@Selector  display@Display
...
[server1]
components: WSserv1           s1Net@Network
components: hsserv1@HSServer  PDBserv1
components: drvserv1@Driver
...
```

Figure 2: Specification of a Coupled Model in CD++

The top model here is composed of three coupled models (*server1*, *server2*, and *client*) and one atomic component (*eth*, an instance of EthernetSwitch). *client* is composed by two atomic components (*clientNet* and *hsclient*) and one coupled component (*WSclient*). The input and output ports define the model's interface, and the links between components define the model's coupling. The input ports in the top model (e.g., *eth_enable*, *eth_disable*, *hss1_start*) are used to activate and deactivate the Ethernet switch, server nodes, and client. The output ports (e.g., *status, packets*) are used to inform the progress in the system.

Models developed in CD++ are independent from the engine in charge of driving their execution. At present, CD++ is able to execute models in single processor, parallel or real-time mode. The execution engine uses model's specifications, and it builds one object to control each component in the model hierarchy. These objects communicate using message passing, and they are called **processors**. There are different types of processors according to the activity they carry out: **simulators** are specialized in atomic models (executing its associated functions), **coordinators** manage coupled models, and the **root coordinator** controls global execution aspects (time, start/stop, interfacing with the environment, etc.).

RT-CD++ (Glinsky and Wainer 2002a) uses the real-time clock to trigger the processing of discrete events in the system. Figure 3 outlines the processor's hierarchy generated by RT-CD++ to execute the model presented in Figure 2. The root coordinator created at the top level manages the interaction with the experimental frame that tests the model receiving inputs (via *eth_enable*, *eth_disable*, *hss1_start*, etc.), and returns outputs (via *status* and *packets*). The root coordinator exchanges messages with its children. Coordinators are created to handle the coupled models *server1*, *server2*, *client*, etc. Simulators are created to handle the components *eth* (which inherits from the atomic *EthernetSwitch*), *clientNet* (from atomic *Network*), *hsclient* (from atomic *HSClient*), *drvserv1* (from atomic *Driver*), etc.

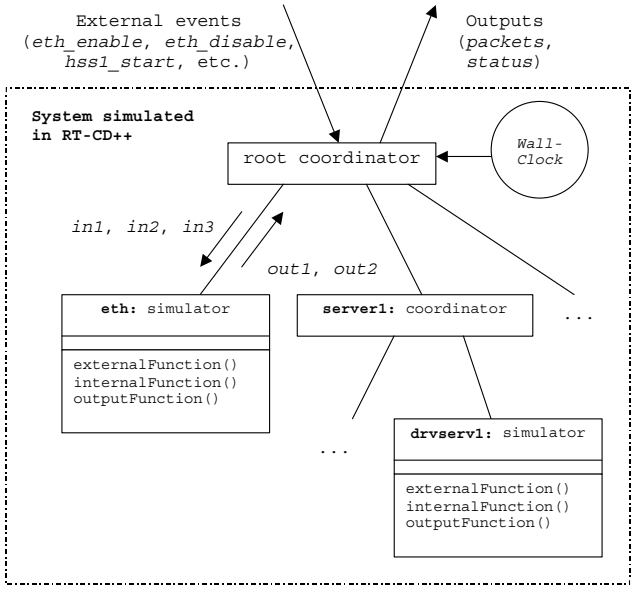

Figure 3: RT-CD++ Simulation Scheme

Model execution is triggered by the real-time clock using the time of the external events. When the root coordinator receives a new event, it forwards the message to the corresponding processor. Timing constraints (deadlines) can be associated to each external event. When the processing of an event is completed, the root coordinator checks if the deadline has been met. In this way, we can obtain performance metrics (number of missed deadlines, worst-case response time).

We thoroughly tested the execution performance of RT-CD++ (Glinsky and Wainer 2002a). These studies showed that models with more than 50 components execute with an overhead below 2%. For larger models (over 200 components), the overhead incurred by the tool is below 3%. We have used RT-CD++ to build simulations hardware-in-the-loop (Li, Pearce, and Wainer 2003), creating a model of the CODEC of an Analog Devices 2189M EZ-KITLITE DSP board. Different tests showed the feasibility of the approach, as we were able to reproduce simulated results in the real-time environment. Nevertheless, when building components on the board, some of the existing models needed some rework (due to the use of Analog Devices' IDE that was in charge of the communications between CD++ and the hardware surrogate). These problems were solved by incorporating communication between facilities into CD++, permitting direct communication with the toolkit and external hardware. In the following section, we will show how to use CD++ to develop a hardware-in-the-loop application. The experiments evolve from a simulated model running in a workstation to a microcontroller-based application. We have used the Motorola 68HC12 board, with a project board (including varied sensors and actuators).

3 AN AUTOMATED FACTORY MODEL

We built an automated manufacturing system (AMS) with both hardware and simulated components. The proposed AMS is composed by dedicated stations that perform tasks on products being assembled, and conveyors that transport the products to/from those workstations.

Figure 4 shows the physical layout of our AMS, which consists of four stations and two conveyor belts to transport the products (A and B). The production cycle is organized by a scheduler, which depends on the type of piece being assembled. The scheduler determines which station (e.g., painting machine, baking machine, storing station) should receive and work on the product.

We started by modeling the entire system in CD++ based on the previous layout. The system is composed by two coupled components (conveyors), and three atomic components (a controller system, a scheduler, and a display controller). Each conveyor is formed by two atomic models (an engine and a sensor controller). Component reuse is an essential aim of our approach. In the development of the

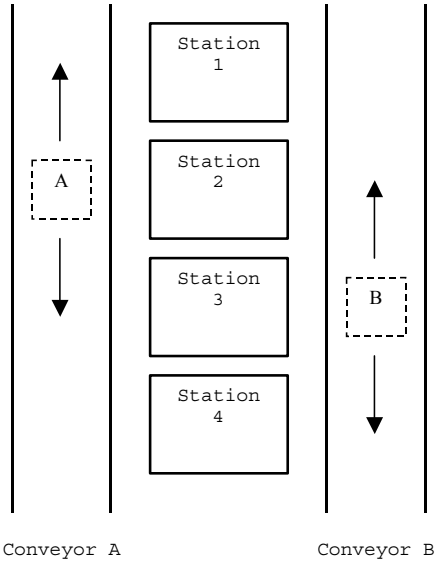

Figure 4: Layout of the AMS

AMS, we reused a controller unit that was implemented for an elevator control system. We also reused a prototype of a painting station, which paints pieces placed on its working area following a predefined sequence (e.g., heat the paint at 80ºC, activate a motor at 50 RPS).

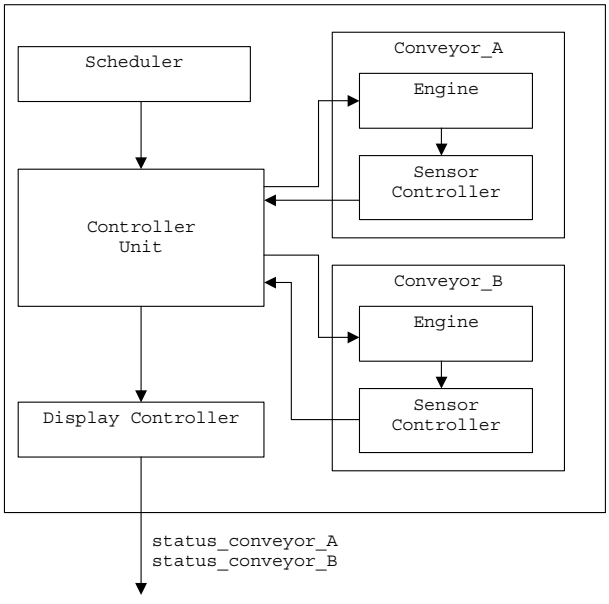

Figure 5: Scheme of the AMS (entirely in CD++)

The sensor controller is an atomic model, defined as shown in Figure 1. It receives events from the environment, and forwards them to the controller unit (CU), resembling the real components of the system. The display controller handles the digital display (showing the location of the piece in each conveyor belt), based on the signals received from the controller unit. The controller receives

input signals from sensors and the scheduler, and determines where to dispatch each piece activating the engines of the conveyor belts. The scheduler stores information about which stations have to work on a specific product. Figure 6 shows the CU of the AMS.

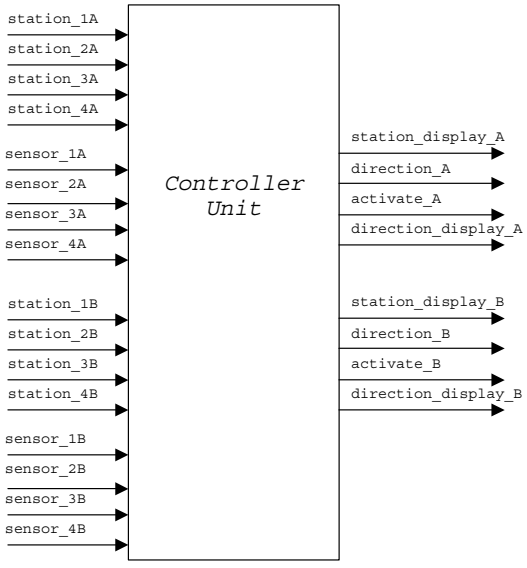

Figure 6: Diagram of the Controller Unit

Most of the logic of the CU is located in the external transition function, which handles the incoming events. Events received via ports *station_ij* are sent by the scheduler, and represent that the product in conveyor belt *j* has to be sent to station *i*. Events received via *sensor_ij* indicate that the product in conveyor *j* has reached station *i*. Thus, the controller can schedule the next internal transition function to activate or deactivate the engine of the corresponding conveyor (via *direction_j* and *activate_j*). It can also signal the display controller when the conveyor belt starts moving or a product reaches a new station (via *direction_display_j* and *station_display_j*). Users can define the activation time for the engine, customizing its timing behaviour.

Different experimental frames were applied to this model, allowing the analysis of different scenarios. We started by analyzing the behaviour of each submodel independently (using the specifications for their physical counterparts) and then, we conducted integration tests. Figure 7 shows a sample event file for one of such experiments.

Time	Deadline	In-port	Out-Port	Value
00:09:100	00:09:300	sta_3A	activate_A	1
00:12:500	00:12:700	sensor_2A	sta_disp_A	1
00:17:500	00:17:700	sensor_3A	sta_disp_A	1
00:35:100	00:35:300	sta_4B	activate_B	1
00:30:000	00:30:200	sensor_2B	sta_disp_B	1
00:34:100	00:34:300	sensor_3B	sta_disp_B	1
...				

Figure 7: Experimental Frame for the AMS Controller Unit

Initially, a piece is placed in station 1 of each conveyor belt and there are no pending events. The first event represents a job scheduled for product A in station 3. The event occurs at time 00:09:100, and the simulator receives it via input port *sta_3A*. As a result, we expect to turn on the conveyor belt in less than 200 ms to transport the product. The second event in the list represents the activation of *sensor_2A* (i.e., the product in belt A has reached the second station). In this case, we expect an output via port *sta_disp_A* before 00:12:700, informing the arrival of the product to that station. The value of 1 represents activation of sensors and scheduling of tasks in stations. Figure 8 shows the outputs generated by the real-time simulator for this experiment.

Time	Deadline	Out-port	Value
00:09:110		direction_A	1
00:09:110	00:09:300	activate_A	1
00:12:510	00:12:700	sta_disp_A	2
00:17:510	00:17:700	sta_disp_A	3
00:17:510		direction_A	0
00:35:110		direction_B	1
00:35:110	00:35:300	activate_B	1
...			

Figure 8: Outputs Generated by the AMS Controller Unit

As we can see, the deadlines were met in every case. For example, the first event met its deadline, activating the engine of conveyor belt A at time 00:09:110 in the correct direction (the value 1 via port direction_A indicates that the belt will move forward). The third output is the result of activating the sensor at the second station in belt A, and the following one represents the product reaching the third station at time 00:17:510. The fifth line shows that the conveyor belt has stopped after product A has reached station 3. The last two lines show the initial activity that generates scheduling a job in station 4 for product B.

We used different experimental frames to thoroughly test this model, and once satisfied with its behaviour, we progressively started to replace simulated components with their hardware counterparts. The first step was to replace the scheduler model, and to execute it on the microcontroller, as shown in Figure 9. The microcontroller generates the events to the simulated model, indicating that a product has to be sent to a station. The rest of the components remain unchanged from the architecture described in Figure 5.

Figure 10 shows the CD++ coupled model specification for this version of the system.

The components for the top model follow the architecture in Figure 9. Here, *conveyor_A* and *conveyor_B* are coupled components, whereas *cu* and *dis* are atomic. Thetop model input ports are used to receive events from the scheduler now running in the external board. Replacing a CD++ component with its counterpart running in the external devices is straightforward, since the model is not changed (an option in the executable engine will establish

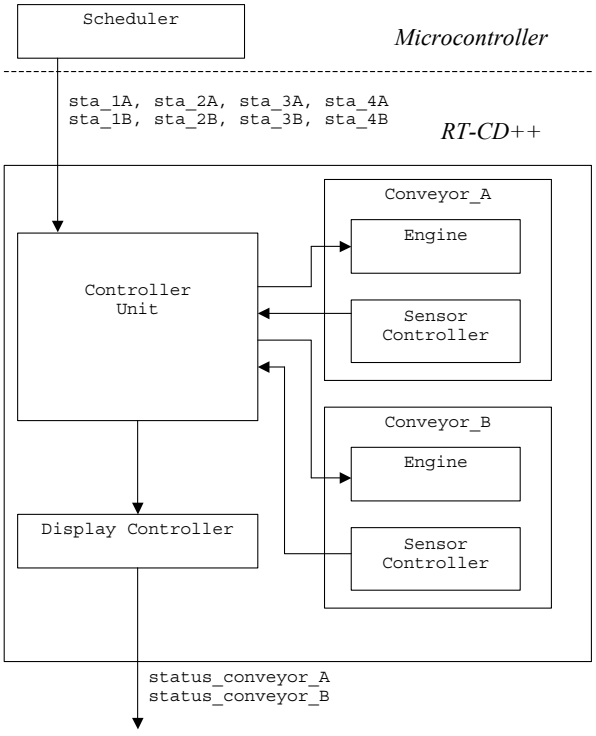

Figure 9: Scheduler in Hardware

```
components:  conveyor_A  conveyor_B scheduler
             cu@CU       dis@Display
in   : sta_1A    sta_2A    sta_3A    sta_4A
in   : sta_1B    sta_2B    sta_3B    sta_4B
out  : status_conv_A
out  : status_conv_B
link : sta_1A    sta_1A@cu
link : sta_2A    sta_2A@cu
...
link : sensor_1@conveyor_A   sensor_1@cu
link : sensor_2@elevBox sensor_2@ec
...
link : dir_display_A@cu    dir_display_A@dis
link : status_conv_A@cu    status_conv_A@dis
link : dir_display_B@cu    dir_display_B@dis
link : status_conv_B@cu    status_conv_B@dis
...
[conveyor_A]
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
...
link : current_pos@eng sensor_triggered@sb
...
[conveyor_B]
components:  sb@SensorController   eng@Engine
...
```

Figure 10: CD++ Model: Scheduler in Hardware

that the scheduler is running in an external device). Likewise, testing this model only requires reusing the previously defined experimental frames. As the scheduler model was built using the hardware specifications for the actual system, and the interfaces of the submodels do not change, the transition is transparent. Figure 11 shows the output of a sample execution of this model. The results obtained are the same as before, regardless of the use of a hardware surrogate.

```
Time          Out-port              Value
00:08:170         status_conv_A          2
00:19:540         status_conv_A          3
00:30:130         status_conv_B          2
00:35:140         status_conv_B          3
00:40:150         status_conv_B          4
...
```

Figure 11: Outputs for Example Shown in Figure 10

In this case, events generated by the scheduler running on the experimental board are sent to CD++. These events trigger the same activities in the model as in the simulated environment (e.g., activating the conveyor engines, displaying the direction of the conveyor belt). In the previous figure, *status_conv_A* and *status_conv_B* show that the products in both belts are transported to the corresponding stations, similarly to what was shown in Figure 10.

After conducting extensive tests, we also moved the display controller to the microcontroller. The value displayed on the digital display (which is informed by the model running in CD++), represents the current station for each product. The display controller and the scheduler were combined in a single application following the previous model specifications. The resulting configuration is shown in Figure 12.

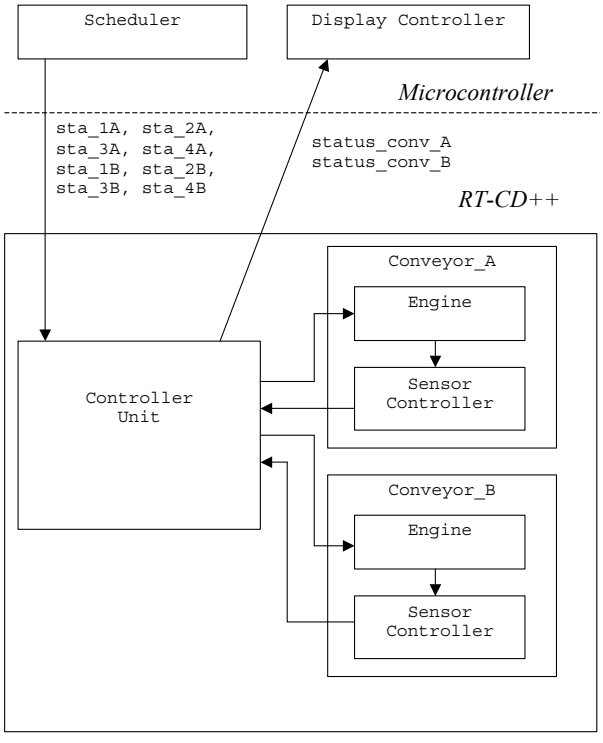

Figure 12: Scheduler and Display Controller in Hardware

By simply activating the simulation engine specifying that the display controller is running in a hardware surrogate, we are able to execute the new application without any modifications.

Every time the models activates the output ports *status_conv_A* and *status_conv_B* in Figure 10, the display controller on the board is activated, showing on the LCD the current location of each product. The following event log was obtained as a result of scheduling jobs in stations (3, 1, 2) for product A and stations (2, 4, 3) for product B, with both pieces located initially on the first station.

The first two lines of the following figure show the product in conveyor A moving from the first to the third station. The third line shows the product in conveyor B moving to station 2 at time *00:34:390*. After station 3 finished its work on product A, the product reaches to station 1 at time *01:15:170*. Product B reaches station 4 at *01:26:170*, which corresponds to the second job scheduled for it.

Time	Out-port	Value
00:27:410	status_conv_A	2
00:33:180	status_conv_A	3
00:34:390	status_conv_B	2
01:10:690	status_conv_A	2
01:15:170	status_conv_A	1
01:21:110	status_conv_B	3
01:26:170	status_conv_B	4
...		

Figure 13: Outputs for Example Shown in Figure 12

When the external display controller receives new data, it displays the value (i.e., the current position of the product in that belt) on the LCD display, and then waits for more data.

The final step was to implement the complete AMS on the microcontroller. Figure 14 shows the scheme for this experimental frame, in which only the engines are still simulated in CD++.

The model does not require any modification, and the model executing in the microcontroller feeds the input ports *activate_in* and *direction_in* in Figure 10.

Figure 15 shows the events generated by the model running in the microcontroller, which represents setting the direction, activation and deactivation of the conveyor belt engines A and B.

Figure 16 shows the activation and deactivation of the belts when the requests are received, which is the result of the activity in the microcontroller. The values issued by the port *result_A* and *result_B* represent that the belt is activated to move forward (1), reverse (2), or deactivated (0).

4 CONCLUSION

The development of M&S applications with hardware-in-the-loop can be significantly beneficiated by using a formal technique like DEVS. In this work, we showed how to use CD++ to develop a sample application in which we incorporated hardware components gradually as the components be came available. Our technique enables a flexible approach to develop embedded applications, which is particularly useful when some of the components are being developed simultaneously and therefore are not available yet.

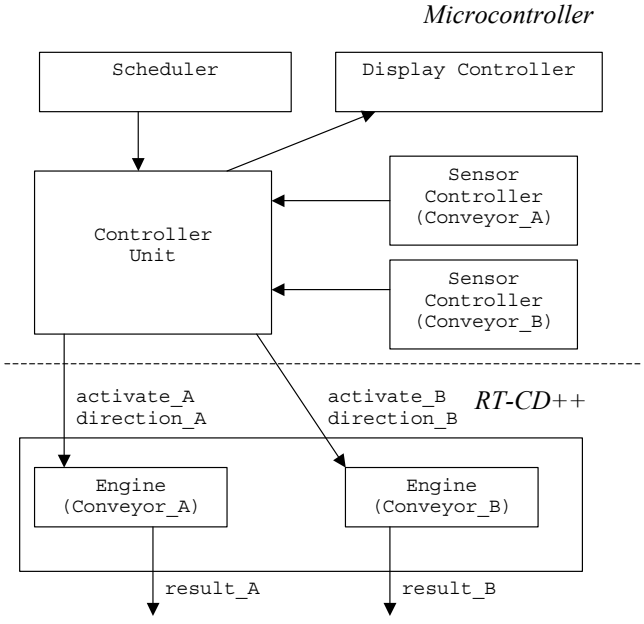

Figure 14: Controller Unit Implemented in Hardware

Time	Port	Value
00:06:120	direction_A	1
00:06:130	activate_A	1
00:15:930	activate_A	0
00:56:800	direction_B	2
00:56:810	activate_B	1
01:01:130	activate_B	0
01:22:710	direction_B	2
01:22:720	activate_B	1
...		

Figure 15: Event Log Generated by the Engines Model

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

Figure 16: Outputs for the Model in Figure 14

Experimental frameworks allowed us to analyze the model execution in a simulated environment, checking the model's behaviour and timing constraints within a risk-free environment. The simulation results were then used in the development of the actual application, permitting developers to validate their systems and subsystems at every stage of the process.

The time required to develop models in RT-CD++ is a major concern, given that time-to-market is generally a crucial factor. The development of an atomic component

with medium complexity like the *EthernetNetwork* required approximately 30 minutes for an experienced CD++ developer (or 2.5 hours to a developer who is new to CD++ but familiar with C++). Developing a coupled component like the conveyor presented in Figure 5 required approximately 2 hours for an advanced CD++ developer (or 5 hours to a new developer). Additionally, 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 provides a sound methodology for developing discrete-event applications, which can be easily applied to improve the development of real-time embedded applications. These advantages include secure, reliable testing, model reuse, and the possibility of analyzing different levels of abstraction in the system. Model execution is automatically verifiable, as the execution processors are built following the formal specifications of DEVS. DEVS bibliography shows how to build execution engines that enable mimicking the model's behaviour in a homomorphic formalism. Hence, the developer only needs to focus on the model under development.

Relying on experimental frameworks facilitates testing in a cost-effective manner, allowing users to build and reuse test frames for each submodel of the system. Since the DEVS formalism is closed under coupling, models can be decomposed in simpler versions, always obtaining equivalent behaviour. Finally, the semantics of models are not tied to particular interpretations, thus existing models can be reused. 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 10 conveyors and 20 stations. Extending the model here presented requires modifying only the external transition function in the CU, and defining a new coupled model including the new stations, while keeping the remaining methods unchanged.

REFERENCES

Glinsky, E. and G. Wainer. 2002a. Definition of Real-Time simulation in the CD++ toolkit. In *Proceedings of the 2002 Summer Computer Simulation Conference*. San Diego, USA.

Glinsky, E. and G. Wainer. 2002b. Performance Analysis of Real-Time DEVS Models. In *Proceedings of the 2002 Winter Simulation Conference*. Eds. E. Yücesan, C. –H. Chen, J. L. Snowdon, and J. M. Charnes. San Diego, USA. 588-594.

Li, L., T. Pearce, and G. Wainer. 2003. Interfacing Real-Time DEVS models with a DSP platform. In *Proceedings of the Industrial Simulation Symposium*. Valencia, Spain.

Wainer, G. 2002. CD++: a toolkit to develop DEVS models. *Software - Practice and Experience*. 32: 1261-1306.

Zeigler, B., T. Kim, and H. Praehofer. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* (2nd Edition.) Academic Press.

AUTHOR BIOGRAPHIES

EZEQUIEL GLINSKY has received a B. Sc. (2000) and M. Sc. in Computer Sciences (2002) from the Universidad de Buenos Aires, Argentina. He is currently a second year Masters student in Electrical Engineering at the Department of Systems and Computer Engineering in Carleton University, Ottawa, ON, Canada. His e-mail address is <eglinsky@sce.carleton.ca>.

GABRIEL WAINER received the M.Sc. (1993) and Ph.D. degrees (1998, with highest honors) of the Universidad de Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. He is Assistant Professor in the Dept. of Systems and Computer Engineering, Carleton University. He was a visiting research scholar at the University of Arizona and LSIS, CNRS, France. He is author of a book on real-time systems and another on Discrete-Event simulation and more than 90 research articles. He is Associate Editor of the Transactions of the SCS. He has been the PI of several research projects, and participated in different international research programs. Prof. Wainer is a member of the Board of Directors and the chair of the SISO DEVS standardization study group. His e-mail and we addresses are <gwainer@sce.carleton.ca> and <www.sce.carleton.ca/faculty/wainer>.