A Model-Driven Technique for Development of Embedded Systems Based on the DEVS formalism

Gabriel A. Wainer, Ezequiel Glinsky, and Peter MacSween

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

Summary. The development of embedded systems with real-time constraints has received the thorough study of the software engineering community in the last 20 years. Despite these efforts, most existing methods are still hard to scale up for large systems, or they require expensive testing efforts. We propose a model-driven method to develop this kind of applications based on DEVS, a formal technique originally created for modeling and simulation of discrete event systems. This approach combines the advantages of a simulation-based approach with the rigor of a formal methodology. 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, improving quality and reliability of the final product. 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.

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 [SER00, LDNA03]. 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 the DEVS (Discrete Event 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. Another advantage of using DEVS is that different existing techniques (Bond Graphs, Cellular Automata, Partial Differential Equations, Queuing models, etc.) have been successfully transformed into DEVS models. DEVS theory has evolved since the early 1970's, providing a generic framework to model discrete-event systems. Many existing techniques that have been widely used for the development of embedded and Real-Time systems, has been also mapped into DEVS models. Many state-based approaches, such as Verilog [KKK01], VHDL [MW05], Petri Nets [JW02] and Timed Petri Nets, Timed Automata [GPC03], State Charts [BV03] and Finite State Machines [ZW03] have their DEVS equivalents. This permits sharing information at the level of the model, and different submodels can be specified using different techniques, while keeping independence at the level of the execution engine. In this way, we count with a mathematical framework that can be used to describe different modeling techniques and prove properties about general aspects of the system, while having a general method for sharing model information using different approaches, and being able to apply the right technique to each part of the application development process.

CD++ [Wai02] is a M&S software that implements DEVS theory with extensions to support real-time model execution [GW02a]. CD++ was used as the base for our development, building on previous research focused on real-time applications with hardware-in-the-loop [LPW03]. We will discuss how to use this framework to incrementally develop embedded applications, and to seamlessly integrate simulation models with hardware components. Initially, we develop models entirely in CD++, and we replace them with hardware surrogates at later stages of the process. 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. Consequently, the development cycle is shortened, its cost reduced, and quality and reliability of the final product is improved.

2 Background

The DEVS formalism [ZKP00] is a M&S framework based on dynamic systems theory. DEVS 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, discrete event approximation of continuous systems and support for repository reuse. A real system modeled with DEVS is described as a composite of submodels, each of them being behavioral (**atomic**) or structural (**coupled**). A DEVS atomic model is informally described in Figure 1.

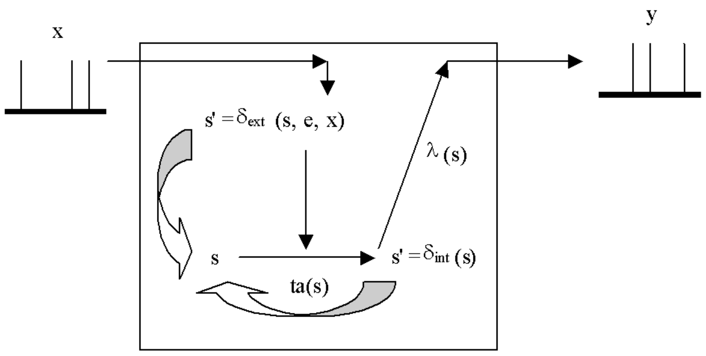


Fig. 1. Informal description of an atomic model.

A DEVS atomic model is formally described as:

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

Each atomic model is seen as having an interface consisting of *input* (X) and *output* ($\boldsymbol{Y}$) *ports*. Every *state* ($\boldsymbol{S}$) in the model is associated with a *time advance* (***ta***) function, which determines the duration of the state. The model will be in the state ***s*** during ***ta*** time units. The time advance is a function in the domain of the Real positive numbers (including zero and infinity). Once this time is consumed, an internal transition is triggered. This involves two actions: first, the model execution results are spread through the model's output ports by activating the *output function* (λ). Then, the *internal transition function* (δ_{int}) is fired, producing a state change. Input external events are collected in the input ports, which have room for only one input, and are cleared after immediately after being processed. The input ports will only receive input events for the current event time, and the external transition function (δ_{ext}) specifies how to react to those inputs.

A DEVS coupled model is composed by several atomic or coupled submodels, as seen in Figure 2.

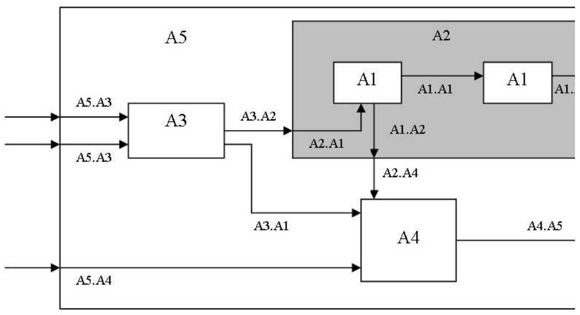


Fig. 2. Informal description of a coupled model.

Coupled models are defined as a set of basic components (atomic or coupled), which are interconnected through the model's interfaces. The model's coupling defines how to convert the outputs of a model into inputs for the others, and to inputs/outputs to the exterior of the model. A DEVS coupled model is formally defined by:

$$\text{CM} = < X,\ Y,\ D,\ \{M\ _d \mid d \in D\},\ EIC,\ EOC,\ IC,\ select >$$

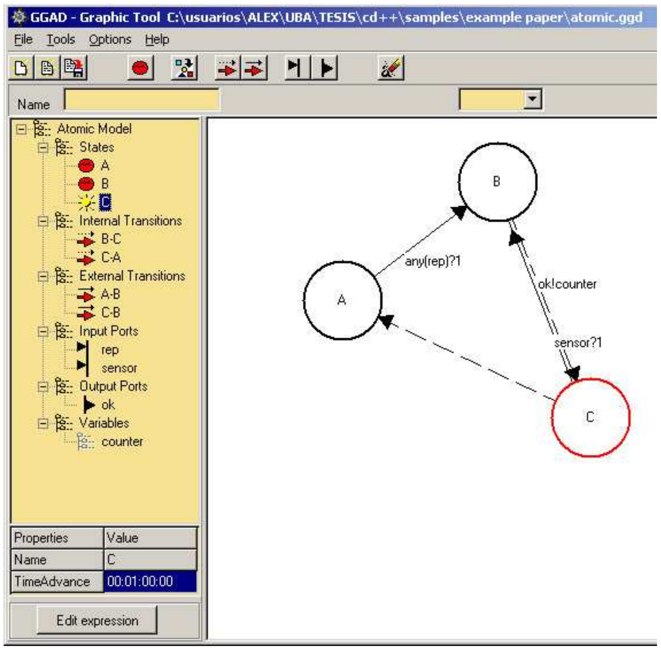
A coupled model groups several DEVS into a compound model that can be regarded, due to the *closure property*, as a new DEVS model. A coupled model is composed by a set (D) of basic models (i.e., atomic or coupled) interconnected through their interfaces (X, Y). When external events are received, the coupled model has to redirect the inputs to one or more components. Similarly, when a component produces an output, it may have to map it as an input to another component, or as an output of the coupled model itself. Mapping between ports is defined by the *EIC, EOC* and *IC* sets, which define how to convert the outputs of a model into inputs for others. *EIC* defines how external inputs are routed to the subcomponents; *EOC* defines how outputs of internal subcomponents are routed outside the coupled model, and *IC* takes care of the internal couplings. *select* is the tiebreaker function, which defines an order over the components.

3 The CD++ Toolkit

CD++ [Wai02] is a modeling tool that was defined using the specifications presented in the previous section, and the basic execution techniques introduced in [ZKP00]. The toolkit includes facilities to build DEVS models. DEVS Atomic models can be programmed and incorporated onto a class hierarchy programmed in C++. Coupled models can be defined using a built-in specification language. CD++ is built as a class hierarchy of models related with processing entities. DEVS Atomic models can be programmed and incorporated onto the *Model* basic class hierarchy using C++. A new atomic model is created as a new class that inherits from the *Atomic* base class. *Atomic* is an abstract class that declares a model's API and defines some service functions the user can use to write the model.

Defining models in C++ provides the users with flexibility to define the model's behavior. Nevertheless, a non-experienced user can have difficulties in defining models using this approach. Graphical specification also improves the interaction with stakeholders and users during system specification, while allowing the modeler to think about the problem in a more abstract way. Therefore, we have used an extended graphical notation to allow defining atomic model's behavior [WCD01, CDW04].

Each model is defined by a unique identifier, and states are represented by vertices (bubbles) in a directed graph. Each bubble includes an identifier and a state lifetime.



Properties	Value
Name	C
TimeAdvance	00:01:00:00

Fig. 3. An atomic model defined as a DEVS graph.

Figure 3 shows a simple atomic model defined in CD++ using this notation. The model includes three states: A, B and C. Dotted lines represent internal transitions, while full lines define external transitions. In this case, if the model is in state A and it receives an external event through the *rep* input port (shown in the left panel), the *any* function is evaluated. If the result of this evaluation is 1, the model changes to the state B. While in B, the model waits its lifetime to be consumed. It then executes the output function, which will send the value of the intermediate state variable *counter* through the output port *ok*. After that, the internal transition function executes, and the model changes to the state C.

Each of the elements in the graphical notation is converted into an analytical representation. This notation can be used both to check validity of the model, and to run these models in CD++ [WCD01].

[modelname] defines the atomic or coupled model name, which will be used subsequently. Model states are declared as: state: state1 state2 ...

States are associated to a time advance value. This attribute are initialized with the name of the object and the list of valid attributes for that object, as follows: state1 : time-expression

One of the states must be declared as the initial state of the model: initial: statename

Then, I/O ports are declared either as follows:

in : inport1 inport2 ...

out : outport1 outport2 ...

Temporary variables are declared by:

var : var1 var2 var3 ...

In addition, they can be optionally initialized as:.

var1 : value1

var2 : value2

The internal transitions use the following syntax:

int : source destination [outport!value]* ({ (action;)* })?

External transitions are defined using the following notation:

ext : source destination EXPRESSION ({ (action;)* })?

Once an atomic model is defined, it can be combined with others into a multicomponent model using a specification language specially defined with this purpose. The user must define the coupling information, and CD++ will generate an analytical specification that can be used for execution. The coupled model at the higher level is always named [top]. Four properties must be configured: components, output ports, input ports and links between models. The following syntax is used:

Components: name1[@atomicClass1] name2 ...

Out: portname1 portname2 ...

enumerates the model's output ports (optional clause).

In: portname1 portname2 ...

enumerates the input ports (optional clause).

Link: source[@model] destination[@model]

describes the internal and external coupling scheme. If the name of the model is not included, the default will be the coupled model currently being defined.

Figure 4 shows a sample coupled model describing an Ethernet switch presented in [WGM05].

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

```
components:  server1   server2 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
...
[eth]
delay:  00:00:01:000
node_1: 1     node_2: 2     node_3: 3

[client]
components: WSclient   clientNet@Network
components: hsclient@HSClient
in:   hs_start    hs_stop   client_in
out:  client_out
link: hs_start    start@hsclient
...
```

Fig. 4. Definition of the Ethernet DEVS coupled model in CD++.

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++ [GW02a] uses the real-time clock to trigger the processing of discrete events in the system. Thus, the same models used for simulation can be later used for execution in real-time. Figure 5 outlines the processor's hierarchy generated by RT-CD++ to execute the model. 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

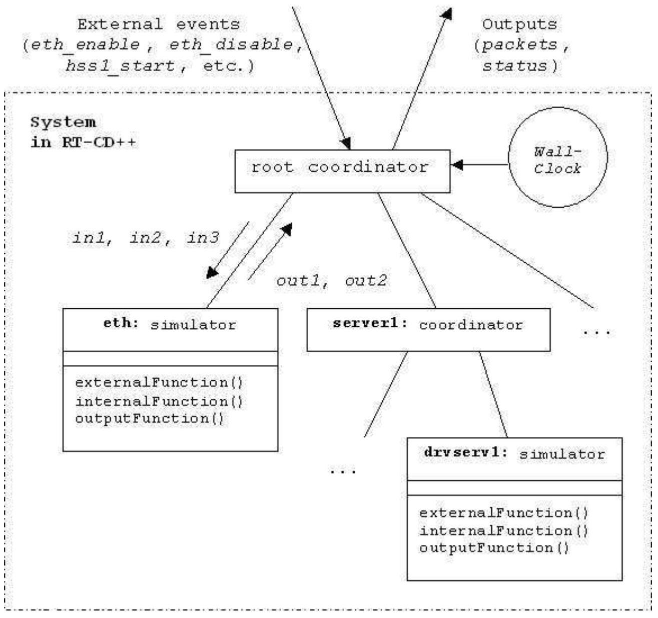


Fig. 5. RT-CD++ execution scheme.

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.

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++ [GW02b], using DEVStone, a synthetic benchmark we created to study the performance of DEVS-based simulators [GW05]. We conducted performance analysis using DEVStone to study the overhead of the real-time engine in CD++. 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%, which is reasonable considering the complexity of the tools.

4 Incremental development of a DEVS simulation model

In this section, we show how to develop incrementally a model based on simple components. The application executes in a simulated environment (i.e., all of the components remain executing in a virtual world). We have built a simulation model integrating components of a Radar system [MW04]. The first stage in the definition of this example consisted on building a model to examine the synchronization effects between radar receivers and transmitters. When using a scanning radar receiver, the interception of radar signals can be severely limited if the scan rate of the receiver becomes synchronized with a radar transmitter. Every effort must be made to generate a receiver scan pattern that limits this effect, as it seriously degrades the Probability of Intercept (POI) for the receiver.

Synchronization occurs when a particular transmitter sends out radar pulses periodically, with the receiver scheduled to scan periodically in such a manner that the receiver is never "listening" when the transmitter is transmitting. This can lead to the transmitter *not* being detected by the receiver, even though it may be transmitting. The sequential operation of the receiver that defines the tuned-frequency, listening-time, azimuth, and beam width are specified by a "scan pattern". Receivers can communicate with each other, with each receiver notifying the other receivers about radar transmitters that have been detected. Each receiver is connected to a simple communications bus, and it maintains a tracking table containing all the information about the currently known transmitters. In order to analyze the behavior or this system, we built a DEVS model, whose structure is the one presented in Figure 6.

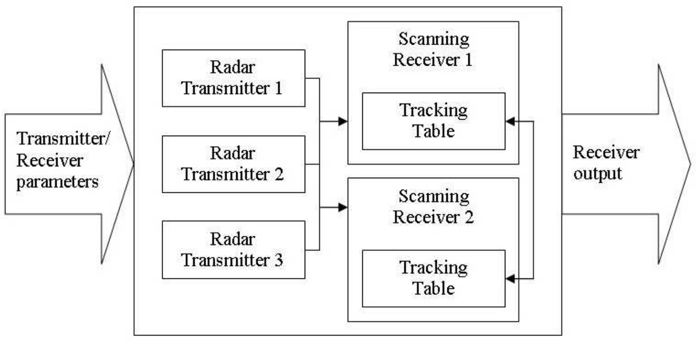


Fig. 6. Structure of the Radar Tx/Rx model.

The first step was to identify and define each one of the model components. Once identified, a DEVS atomic model was built for each subcomponent. Following, we exemplify the definition of one of these models by showing the

Tracking Table atomic model. The Tracking Table model is responsible for maintaining the list of transmitters that are "known" to the local receiver.

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

S = { Receive_Update_From_Bus, Wait, New_Signal_Detected, Send_Update_To_Bus, Notify_New_Freq }
X = { signal_props, bus_receive_freq, bus_receive_id }
Y = { bus_send_freq, bus_send_id, new_freq }
δ_{int} = { δ_{int}(Receive_Update_From_Bus) = Notify_New_Freq,
δ_{int}(Notify_New_Freq) = Wait,
δ_{int}(New_Signal_Detected) = Send_Update_To_Bus,
δ_{int}(Send_Update_To_Bus) = Wait }
δ_{ext}= { δ_{ext}(Wait, signal_props) = New_Signal_Detected,
δ_{ext}(Wait,bus_receive_freq) = Receive_Update_From_Bus }
ta = { ta(Receive_Update_From_Bus) = UPDATE_TIME,
ta(New_Signal_Detected) = PROCESS_TIME,
ta(Send_Update_To_Bus) = BUS_TIME,
ta(Notify_New_Freq) = NOTIFY_TIME }
λ(S) = {λ(New_Signal_Detected) = (bus_send_freq,bus_send_id),
λ(Receive_Update_From_Bus) = new_freq }

This model evolves through different states (S): Receive an update from the bus, wait, detection of a new signal, transmission of an update to the bus, or notification of a new frequency. The model changes from one state to the other by executing the transition functions. As seen in the external transition (δ_{ext}), from a *wait* state, the tracking table receives information from either the local receiver (*signal props,* one of the external input events) or the communication bus (*receive freq*). If the local receiver detects a new signal, the signal is appended to the local tracking table, and an update is sent over the bus for use by any remote tracking tables. If the local tracking table receives an update from the bus, it appends the information to the local tracking table and notifies the local receiver. The Tracking Table then returns to a wait state.

Each of the models were built using CD++ and thoroughly tested, and they performed as described in their conceptual model specifications [MW04]. A problem with the specification of the network receiver was revealed while testing (the tracing of the signals that are received by the network receivers became very difficult when numerous signals are transmitted, and the receivers start to share information).

The use of the formal specification defining the atomic and coupled model behavior was very useful in debugging the models when they were implemented. The iterative procedure of updating the formal specification, then updating the implementation was quite efficient. Following these iterations resulted in the models matching the specifications. Once this stage was completed, a Coupled Model was built, integrating all of the systems' components. The description of this model can be found in Figure 7.

```
[top]
components: tr1@Transmitter tr2@Transmitter
            tr3@Transmitter netrx1 netrx2
out: notify1 notify2 notify3
Link: pulse_out@tr1 ext_signal@netrx1
Link: pulse_out@tr1 ext_signal@netrx2
...
[netrx1]
components: tt1@Tracking_Table rx1@Scanning_Receiver
in: ext_signal brf brid
out: notify bs_id bs_freq
Link: ext_signal ext_signal@rx1
Link: brf bus_receive_freq@tt1
Link: brid bus_receive_id@tt1
...
Link: bus_send_freq@tt1 bs_freq
...
```

Fig. 7. Coupled model definition: Radar Tx/Rx.

The various atomic models contained in the previously defined coupled model were tested using different scenarios. Table 1 shows the result of the testing scenario for the network with a transmitter. In this case, the transmitter sends out pulses at 24kHz, Pulse width of 5 ms, Pulse interval of 40 ms. Bus Message at t=20 ms. Receiver listening between 22kHz and 25kHz. As we can see in the figure, the receiver gets a signal from the transmitter every 40 ms, and a bus message at t=20ms. The bus message is ignored because it is not within the listening range of the receiver (19 kHz and the receiver is listening from 22 to 25 kHz). Note that the model does not queue received pulses or bus messages. For each pulse received by the local transmitter, a bus message is generated after a delay of 15 ms. The bus message stays active for 40ms.

During this phase, we were able to detect a problem with the specification of the network receiver: the signal information received by the bus was sent to the scanning receiver, which treated it like an external signal (thus causing a second bus transmission). The specification was corrected so that signal information is not re-sent over the bus.

Another component of the application describes the behavior of a simple vehicle, which seeks a target. As showed in Figure 8, the seeker acts to steer a vehicle towards a specified position in global space. This behavior adjusts the vehicle so that its velocity is radially aligned towards the target.

Using the hierarchy of motion behaviors defined in [Rey03], the "Action Selection" of the seek behavior is specified by dictating the destination location.

The model components specify the Desired Velocity of the vehicle. The model rules detail the discrete motion that was implemented to simulate the

Table 1. Testing scenario: network with transmitter.

Events	Outputs
00:00:20 brf 19000	00:00:001 notify 1
00:00:20 brid 3	00:00:016 bs_id 1
	00:00:016 bs_freq 24000
	00:00:026 bs_id 0
	00:00:026 bs_freq 0
	00:00:041 notify 0
	00:00:081 notify 1
	00:00:096 bs_id 1
	00:00:096 bs_freq 24000
	00:00:106 bs_id 0
	00:00:106 bs_freq 0
	00:00:121 notify 0

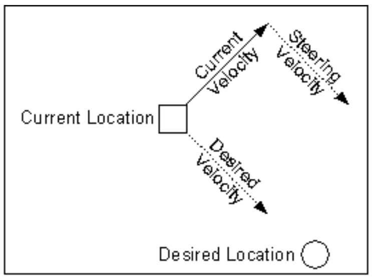


Fig. 8. Informal behavior of the Seek model.

effect of a desired velocity on a vehicle. Multiple combinations of actual and desired velocity could result in the same destination for a vehicle. The model was completely implemented in CD++ following the previous rule specifications, and it and first tested using a single vehicle, with different initial velocities and different desired velocities. After all the rules were implemented, all possible velocities were tested in all possible desired velocities. Following that, collisions were tested using multiple vehicles.

Figure 9 displays the two state variables employed in the definition of the model. The left-hand plane (mostly white) displays the current location and velocity of the three vehicles. The right-hand plane describes the "desired velocity vector field" of the vehicles. The "desired location" for all three vehicles is the center of the plane, and the "desired velocity vectors" steer them to that point. As we can see, the three vehicles enter from the top-right corner of the plane, and they stop when they cannot move any closer to the "desired location".

The final stage of development consisted in showing how to provide interoperation of these models by allowing interaction between the components. This interaction is done at the level of the model, independently of the execution engine chosen (i.e., simulated, real-time or parallel), as the models

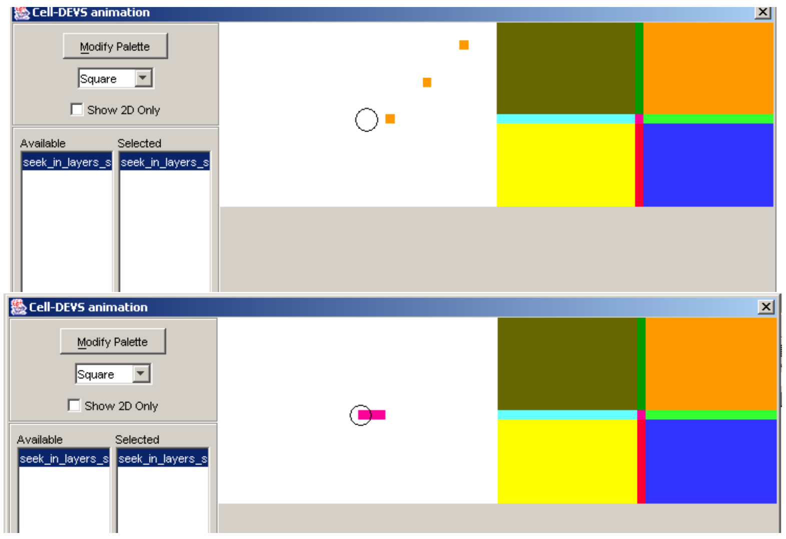


Fig. 9. Three vehicles seeking the desired location.

only communicate at the level of their interfaces. Let us consider, for instance, the existence of a new model, *Radar*. The radar model is prepared to scan a cell space according to a given frequency. Figure 10 shows how to integrate this new model with the two other models defined earlier in this section. These three models were built independently, but they can be easily integrated thanks to the definition of DEVS interfaces.

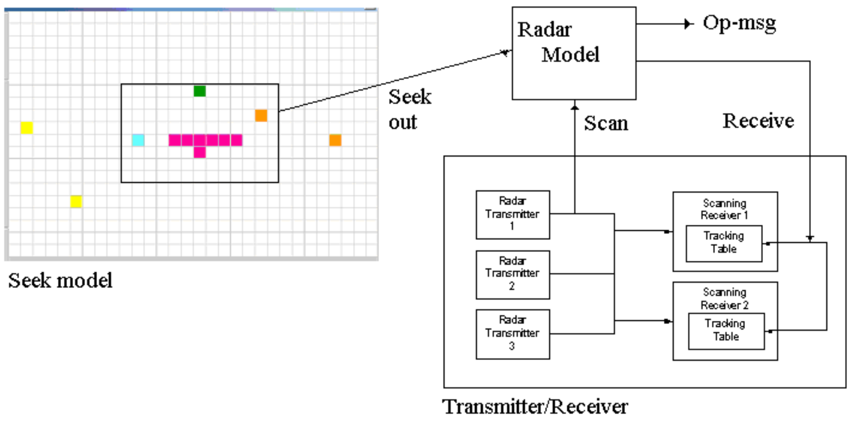


Fig. 10. Multimodel integration.

The Transmitter/Receiver model is used to start radar scanning activities. Upon activation, the Radar will scan the field defined by the Seeker model defined earlier, and will generate two outputs: a reception signal for the Transmitter/Receiver, and a number of Operator Messages according to the values received in the field. The Seeker model advances independently of the execution of the radar, because these models are built as discrete-event specifications, and each subcomponent progresses according its own internal time base. Our *top* model is now integrated by the three original components. The model produces outputs that can be used by the Radar model. We have defined a *zone* in which the cells will generate outputs (by using the *out-rule* definition). Finally, the *Tx-Rx* model, defined earlier in Figure 6, includes two new input/output ports in order to provide interaction with the Radar model. This model is not defined in the file, as it has been defined as a DEVS atomic models, and we just need to define the coupling between this model and the remaining components.

5 Hybrid Applications: an automated factory model

We will now show how to incrementally build an application with components in hardware and simulated modules. The model here represents an automated manufacturing system (AMS) for a factory floor. The AMS is composed by dedicated stations that perform tasks on products being assembled, and conveyor belts transporting the products to/from the workstations. The production cycle is organized by a scheduler, which will define the actions to be carried out according to the type of piece being assembled. The scheduler determines which station (e.g., painting, baking, storage) should receive and work on the product.

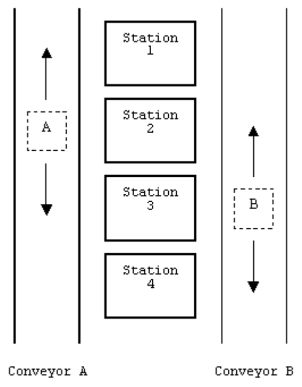


Fig. 11. Layout of the AMS.

Figure 11 shows the physical layout of the AMS, which consists of four stations and two conveyor belts to transport the products (A and B). We started by modeling and simulating the entire system in CD++ based on the layout presented in this figure. The system, shown in Figure 12, is composed by two coupled (conveyors) and three atomic components (a controller, a scheduler, and a display). Each conveyor is formed by two atomic models (an engine and a sensor controller).

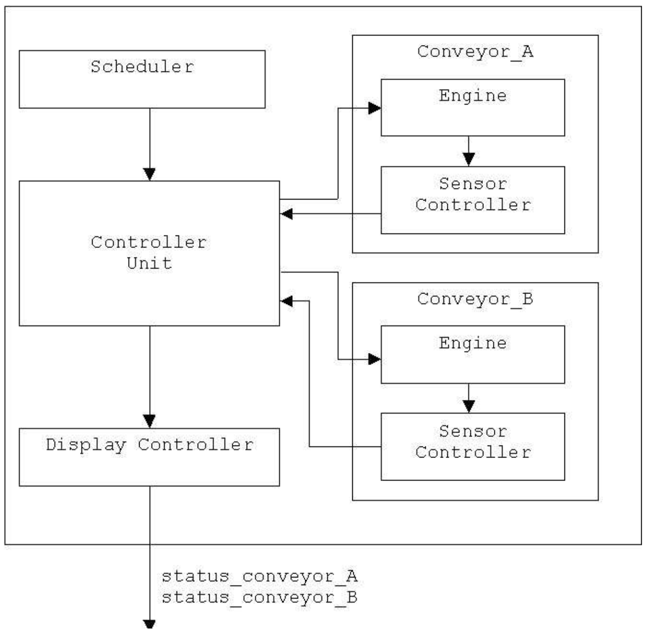


Fig. 12. Scheme of the AMS (entirely in CD++).

The control unit receives events from the environment, and forwards them to the remaining components of the system, using the previously defined coupling scheme. The display controller handles the digital display (showing information about the pieces 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.

Most of the logic of the Controller Unit is located in the external transition function, which handles the incoming events. Events received via ports *station_ij* represent that the product in conveyor belt j has to be sent to station i. Events received on *sensor_ij* indicate that the product in conveyor j has reached station i, thus, we can schedule the next internal transition function to activate/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 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. Initially, we run several experiments using the simulation engine. This permitted to identify some logical errors, which were fixed at this stage. Later, we repeated the tests under the real-time execution engine. This permitted to detect problems with the model's timing constraints in runtime. Once fixed, these models were ready to become the actual software components of the application, running in real-time. Figure 13 shows a sample event file for one of such experiments in the real-time environment.

```
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
...
```

Fig. 13. 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 an activity 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 14 shows the outputs generated by the real-time simulator for this experiment.

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.

```
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
...
```

Fig. 14. Outputs generated by the AMS Controller Unit.

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 behavior, 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. The microcontroller generates the events to the simulated model, indicating that a product has to be sent to a given station. The remaining components are not changed. Figure 15 shows the CD++ coupled model specification for this version of the system.

```
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
...
[conveyor_B]
components:  sb@SensorController   eng@Engine
...
```

Fig. 15. CD++ model: scheduler in hardware

Here, *conveyor_A* and *conveyor_B* are coupled components, whereas *cu* and *dis* are atomic. The top 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 interfaces are not changed (an option in the executable engine will establish that a particular model 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 16 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
...
```

Fig. 16. Outputs for example shown in Figure 15

In this case, events generated by the scheduler running on the board are sent to CD++. These events trigger the same activities in the model as in the simulated environment. In Figure 16, *status_conv_A* and *status_conv_B* show that the products in both belts are transported to the corresponding stations.

After conducting extensive tests, we also moved the display controller to the microcontroller. The value displayed on the digital display (which is updated 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. By simply activating the execution 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 activate the output ports *status_conv_A* and *status_conv_B*, the display controller on the board is activated, showing on the LCD the current location of each product as shown in Figure 17.

```
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
...
```

Fig. 17. Outputs for previous example

The first two lines of the Figure 17 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. 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 18 shows the scheme for this experimental frame, in which only the engines of the conveyor belt are still simulated in CD++.

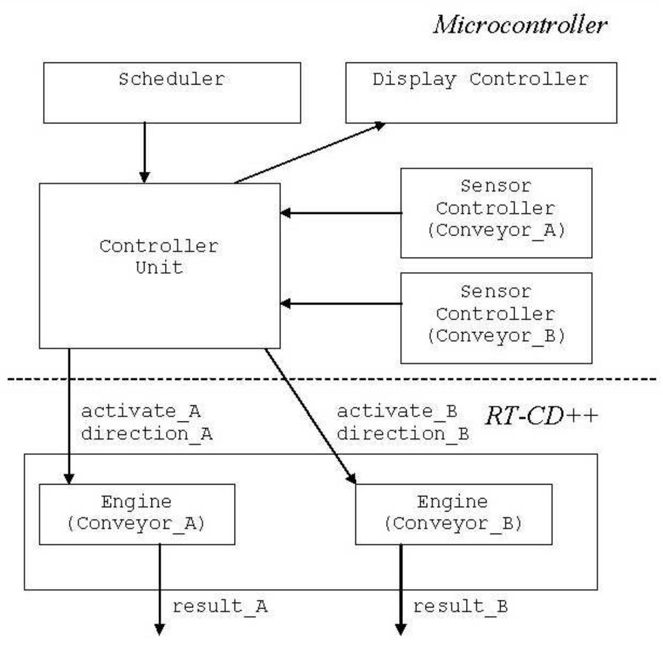


Fig. 18. Controller unit implemented in hardware.

The model does not require any modification, and the model executing in the microcontroller feeds the input ports *activate* and *direction* in Figure 15.

Figure 19 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 20 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).

6 Development improvements

The time required to develop models in RT-CD++ is a major concern, given that time-to-market is generally a crucial factor. Component reuse is an es-

```
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
...
```

Fig. 19. 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
...
```

Fig. 20. Outputs for the model in Figure 18

sential aim of our approach. In the development of the AMS, we reused a controller unit that was implemented for an elevator control system in a previous prototype application. We also reused a prototype of a painting station, which mimics the procedure needed to paint pieces placed on its working area (following a predefined sequence).

We conducted experiments in the classroom, asking students with different experience in the area to build the AMS system using different approaches. Table 2 summarizes the results of this study.

Table 2. Comparing development times for the AMS application.

	Beginners			***Experts***		
	Manual	**CD++**	**Combined**	**Manual**	**CD++**	**Combined**
Prototype	40 m/h	24 m/h	64 m/h	32 m/h	12 m/h	44 m/h
Test	28 m/h	10 m/h	38 m/h	22 m/h	4 m/h	26 m/h
# of bugs	17	2		9	2	
Average Time to Fix	20 min	7 min		18 min	5 min	

The study was conducted with two groups of students: beginners and experts. Different teams were given the same application to develop, both manually and using CD++. The first line shows the average time (in man/hours) taken by the teams to complete the prototype. As we can see, building the application using CD++ always improved when compared building the same application in manually (using a C++ compiler). This is due to the clear separation of concerns of the DEVS models: the students only needed to build the models, not paying attention to any issues related to executing those models. In the worst case (beginners), building the application using CD++ was a 40% faster. The main reason for this is related with the second, third and fourth lines. It is much easier to detect and fix errors using a DEVS-based approach. Likewise, the number of errors found was considerably smaller (mainly due to the reason it is easy to decompose models up to the right level of abstraction, which eases finding and fixing errors).

In the third column, we added the time taken to develop the application using both approaches combined (which result in higher quality software). That is, we use a simulation tool (like CD++) to learn about the system, and then use the knowledge gained by the simulation the experience to build the application manually. If we compare this approach against the use of a tool like CD++ and the application of the DEVS methodology, we obtain higher gains (27% of the original time, in the case of an expert user of CD++). In our case, we are able to move from the simulated world into the real-time application without changing one line of code: the application developed in CD++ and run under the simulation runtime can be later used to run the actual application by just activating the real-time execution engine.

Note that, in this study, we did not take into consideration maintenance costs, which, in any long term project, take a large percentage of the resources spent in the development cycle. Reducing the testing time would improve greatly maintenance, and modifying models is much simpler than focusing on the application from scratch. Simultaneously, it is easy to locate the sources for modification (anything related to reaction to external events should be placed in the external transition function; internal state changes in the internal transitions; and outputs in the output function).

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 Model-Driven development of these systems. 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 integration of hardware components into the system was straightforward. Testing and maintenance phases are highly improved due to the use of a formal approach like DEVS for modeling the system's behavior. The experiments were carried out using CD++, a DEVS tool that has been built following DEVS formal definitions.

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 building models with different resolution at different levels of abstraction. Model execution is automatically verifiable, as the execution, processors are built following the formal specifications of DEVS. Hence, the developer only needs to focus on the model under development. The transition from simulated models to the actual hardware counterparts can be incremental, incorporating deployed models into the framework when they are ready.

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 behavior. 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 Controller Unit, and defining a new coupled model including the new stations, while keeping the remaining methods unchanged.

8 Acknowledgements

This work has been partially supported by NSERC (National Science and Engineering Research Council of Canada), IRIS (Institute for Research on Intelligenet Systems, PRECARN, Canada), and the Intel IXA University program.

References

[BV03] S. Borland and H. Vangheluwe. Transforming statecharts to DEVS. In *Proceedings of the SCS Summer Computer Simulation Conference*, 2003.

[CDW04] 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)*, 2004.

[GPC03] N. Giambiasi, J.L. Paillet, and F. Chane. From timed automata to DEVS models. In *Proceedings of the SCS Winter Simulation Conference*, 2003.

[GW02a] E. Glinsky and G. Wainer. Definition of real-time simulation in the CD++ toolkit. In *Proceedings of the SCS Summer Computer Simulation Conference*, 2002.

[GW02b] E. Glinsky and G. Wainer. Performance analysis of real-time DEVS models. In *Proceedings of the SCS Winter Simulation Conference*, 2002.

[GW05] E. Glinsky and G. Wainer. A benchmarking technique for studying performance of DEVS modeling and simulation environments. Technical Report SCE-05-01, Dept. of Systems and Computer Engineering. Carleton University, 2005. Submitted for publication.

[JW02] C. Jacques and G. Wainer. Using the CD++ DEVS toolkit to develop petri nets. In *Proceedings of the SCS Summer Computer Simulation Conference*, 2002.

[KKK01] J-K. Kim, Y.G. Kim, and T.G. Kim. DHMIF: DEVS-based hardware model interchange format. In *Proceedings of the European Simulation Symposium*, 2001.

[LDNA03] A. Ledeczi, J. Davis, S. Neema, and A. Agrawal. Modeling methodology for integrated simulation of embedded systems. *Proceedings of 7th IEEE International Conference on Engineering of Computer Based Systems*, 13(1):82–103, 2003.

[LPW03] L. Li, T. Pearce, and G. Wainer. Interfacing real-time DEVS models with a DSP platform. In *Proceedings of the Industrial Simulation Symposium*, 2003.

[MW04] P. MacSween and G. Wainer. On the construction of complex models using reusable components. In *Proceedings of SISO Spring Interop-erability Workshop*, 2004.

[MW05] S. Mehta and G. Wainer. Modeling hybrid hardware description languages in DEVS. Technical Report SCE-05-02, Dept. of Systems and Computer Engineering. Carleton University, 2005. Submitted for publication.

[Rey03] C. W. Reynolds. Steering behaviors for autonomous characters, http://www.red.com/cwr/steer/gdc99. Checked on December 2, 2003.

[SER00] S. Schulz, T.C. Ewing, and J.W. Rozenblit. Discrete event system specification (DEVS) and statemate statecharts equivalence for embedded systems modeling. In *Proceedings of 7th IEEE International Conference on Engineering of Computer Based Systems*, 2000.

[Wai02] G. Wainer. CD++: a toolkit to develop DEVS models. *Software - Practice and Experience*, 32:1261–1302, 2002.

[WCD01] G. Wainer, G. Christen, and A. Dobniewski. Defining DEVS models with the CD++ toolkit. In *Proceedings of the European Simulation Symposium*, 2001.

[WGM05] G. Wainer, E. Glinsky, and Peter MacSween. A model-driven technique for development of embedded systems based on the devs formalism. Technical Report SCE-05-03, Dept. of Systems and Computer Engineering. Carleton University, 2005.

[ZKP00] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation.* Academic Press, Inc., 2000.

[ZW03] T. Zheng and G. Wainer. Implementing finite state machines using the CD++ toolkit. In *Proceedings of the SCS Summer Computer Simulation Conference*, 2003.

Index