eCD++: an engine for executing DEVS models in embedded platforms

Yinfeng Henry Yu Gabriel Wainer

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

ABSTRACT: We introduce Embedded CD++ (eCD++), an engine that can execute DEVS models in embedded environments. It deploys a Flat Coordinator and the GGAD Graphical Modeling tool, which supports the Parallel DEVS formalism. Its real-time extensions allow users to develop hardware-in-the-loop applications with ease, being able to integrate them in DEVS-based environments.

1. Introduction

Modeling and simulation (M&S) has gained popularity in a wide variety of fields ranging from biotechnology to digital circuits design, from aerospace engineering to environmental studies, from economics to national defense. Scientists and engineers use M&S mythologies and tools to understand and analyze complex problems. Different M&S techniques have been introduced with success. Among these, the DEVS formalism [ZPK00] provides a framework for the construction of hierarchical models in a modular manner, allowing for model reuse and reducing development time and testing. It defines a way to specify systems whose states change either upon the reception of an input event or due to the expiration of a time delay. Furthermore, it allows hierarchical decomposition of the model by defining a way to couple existing DEVS models. The CD++ toolkit [Wai02] is an object-oriented software that implements the DEVS simulation mechanism.

Simulation tools such as CD++, are only used in the simulated world. In other words, it is very difficult for the modeller to validate his or her simulated solution in the real world. Moreover, this separation makes solutions developed in the simulated world unable to precisely solve real world problems, because for many complex problems, such as real-time systems, the simulated models cannot describe their real world counterparts in 100% accuracy.

Here, we present a new mechanism to overcome this limitation. We start the problem analysis by developing solutions entirely in the simulated world, and we can execute the same model in the real world by an embedded version of the CD++ (named **eCD++**) running on a Single Board Computer (SBC). The simulated results are compared with that from the real world. If the two results fail to agree, the simulated solution can be revised in the simulated world for retest. The DEVS models are modular. Thus, subcomponents of larger models can be validated individually. If a subcomponent is validated in the real world, it can be replaced by real hardware. This technique enables incremental transition from the simulated models to the actual hardware counterparts. This is a low-risk and cost-effective approach to develop hardware-in-the-loop applications.

DEVS technology has been usually applied to large-scale dynamic systems, with implementations, such as CD++, running on workstations and servers. As these systems focus on the high level modeling and simulation, another branch of DEVS application is on real-time event-based control [HZC01]. These low level applications exist largely on embedded systems, which are usually characterized as "intelligent devices" consisting of computer hardware and real-time software. This work mainly studies how to use DEVS technology to design real-time embedded systems.
In the following sections, we will present the design and development of eCD++. eCD++ can run on embedded systems and can execute DEVS models that interact with real world events. This capability makes eCD++ become a useful tool to develop real-time applications with hardware-in-the-loop [LTW03]. This work represents a hardware-in-the-loop simulation methodology that uses eCD++ to develop hybrid hardware and software systems. The technique enables incremental transition from simulated models to the actual hardware counterparts and supports experimental frameworks to facilitate testing in a risk-free environment. To demonstrate this methodology, this work used eCD++ to build a hybrid automated manufacturing system (AMS) with both real hardware and simulated DEVS models.

2. Background

DEVS (Discrete EVents Systems Specification) [ZPK00] is a formalism originally created for modeling and simulating discrete event dynamic systems. In DEVS, a model is specified as a black box with a state and a duration for that state. DEVS models can be put together by linking the outputs of a model to inputs of other models to form **coupled models**. Models made out of only one component are called **atomic models**.

A **Parallel DEVS (P-DEVS)** model [Cho94a] is described as a set of basic and coupled models. Atomic models are still the most basic constructions, which can be combined with other models into coupled models. The **P-DEVS atomic model** has the following structure:

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

where

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

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

S is the set of *sequential* states;

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

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

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

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

$ta : S \rightarrow R_0^+ \cup \infty$ is the *time advance function*; with

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

The semantics of the P-DEVS definition are as follows. At any given time, a basic model is in a state *s*. And in the absence of external events, it will remain in that state for a period of time as defined by ta(*s*). When an internal transition takes place, the system outputs the value $\lambda(s)$, and changes to state $\delta_{int}(s)$. If one or more external events $E = \{ x_1 .. x_n / x \in X_M \}$ occurs before ta(*s*) expires, i.e., when the system is in the state (*s*, e) with $e \leq ta(s)$, the new state will be given by $\delta_{ext}(s, e, E)$. Suppose that an external and an internal transition collide, i.e., an external event *E* arrives when $e = ta(s)$, the new system's state could either be given by $\delta_{ext}(\delta_{int}(s), e, E)$ or $\delta_{int}(\delta_{ext}(s, e, E))$. The modeler can define the most appropriate behavior with the δ_{conf} function. As a result, the new system's state will be the one defined by $\delta_{conf}(s, E)$.

A **P-DEVS coupled model (CM)** is defined by:

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

where

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

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

M_d is a set of atomic models, and D is a set of the atomic models' names, where for each $d \in D$, $M_d = (X_d , Y_d , S, \delta_{ext} , \delta_{int}, \delta_{con}, \lambda, ta)$ *is a DEVS basic structure* with $X_d = \{(p,v) \mid p \in IPorts, v \in X_p \}$;

$Y_d = \{(p,v) \mid p \in OPorts, v \in Y_p \}$;

The couplings are subject to the following conditions:

- *external input couplings (EIC)* connect external inputs to component inputs: $EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d \}$
- *external output couplings (EOC)* connect component outputs to external outputs: $EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d \}$
- *internal couplings (IC)* connect component outputs to component inputs: $IC \subseteq \{((a, op_a), (b, ip_b)) \mid a, b \in D, op_a \in OPorts_a , ip_b \in IPorts_b \}$

DEVS also defines an abstract simulation mechanism that is independent of the model itself and that can be easily implemented by computer software. This mechanism provides a high level description of how the simulation of DEVS models should be executed by a **simulator**. Two kinds of **simulators** are defined, one for atomic and the other one for coupled models, this latter known as a **coordinator**. These simulators progress through the simulation by exchanging messages as described by the abstract simulation mechanism. **CD++** [Wai02] is a simulation software which implements the DEVS simulation formalism. In CD++, simulators and coordinators progress through the simulation by exchanging messages as described by the abstract simulation mechanism.

While P-DEVS provides sound modeling principles to characterize structural and behavior aspects of real-time systems, recent research suggests that transforming (or mapping) DEVS models to actual designs of real-time embedded systems is non-trivial [HS04]. Recent research, therefore, has been focusing on developing schemes to support the transformation from simulation modeling to designs of real systems. One attempt was the DEVS-on-a-chip approach, which implements DEVS on a microprocessor that has limited memory and processing ability [HZC01]. It creates a just-as-needed real time environment. This effort, however, did not implement a full scale of RT-DEVS specifications on the chip. As a result, it only demonstrates the capability of creating real-time embedded systems that have relatively simple compositions. Another research effort in this area focused on how to use RT-DEVS as a framework to develop hardware-in-the-loop applications [LPW03]. These applications are complex as a result of the high degree of interaction between software and hardware components. Therefore, the development of these applications is a challenging process in which M&S can become essential. The technique of applying RT-DEVS to develop hardware-in-the-loop applications seamlessly integrates simulation models with hardware components and also enables incremental transition from the simulated models to the actual hardware counterparts.

3. eCD++ functionality

eCD++ provides four main functionalities: a Flat Coordinator, Parallel DEVS (P-DEVS) simulation, a GGAD (Generic Graphical advanced environment for DEVS modeling and simulation) interpreter, and a real-time

extension. The Flat Coordinator technique simplifies the simulation hierarchy by eliminating the coordinators in the hierarchy and by making direct messaging communications between the Flat Coordinator and the simulators [Kim00], as shown in the following figure.

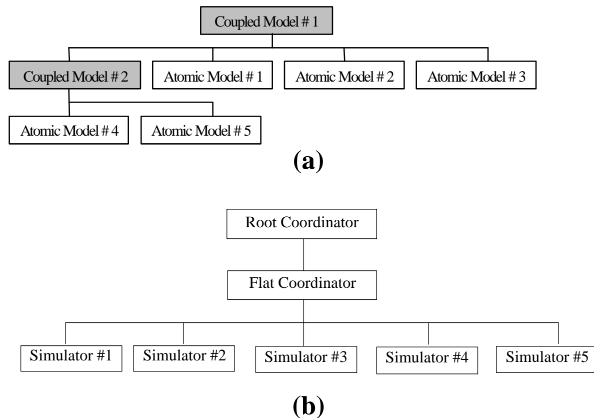

Figure 1. Flat Coordinator (a) Example of a model hierarchy, (b) Associated processor hierarchy

The following are the implementation details of the Flat Coordinator technique:

- The Flat Coordinator must transform the hierarchical structure of the model to a flattened structure in order to reduce the overhead incurred by message passing. The resulting non-hierarchical structure is used by the Flat Coordinator.
- Due to the absence of the usual coordinators, any port links that link to coordinators' ports must be re-wired to reach the far-end atomic ports. Then, the component links are handled by the Flat Coordinator, which forwards the events as needed.
- The Flat Coordinator must receive and send messages directly with the Root Coordinator in order to carry out the simulation process.

The CD++ toolkit contains a tool that provides a graphical user interface (GUI) for modellers to specify atomic models graphically, enabling non-expert users to define atomic models in a easier and more intuitive way. The tool generates textual specifications of the models represented graphically in the GUI. According to the specifications of the graphical notation [HSKP97], an atomic model is placed inside a box. An external state transition is represented by a dotted line above which the input event is represented by "?". Similarly, an internal state transition is represented by a solid line above which the output event is represented by "!". For example, an input event *in?m* means that a message *m* is input at the input port *in*, and an output event *out!m* means that a message *m* is output at port *out*. Input and output ports are denoted by black triangles.

The following figure includes a graphical representation of an atomic model called *coin*. This model simulates the behaviour of the coins displayer in a vending machine. It has one input port, namely *coin_in*, representing the input coins slot and one output port called *coin_out* which represents the display of the inserted coins' amount. When coins are inserted into the coins slot, it takes one second for the coins displayer to display the coins' amount.

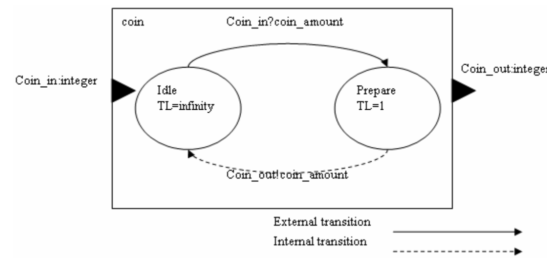

Figure 2. Graphical definition of an atomic model

4. eCD++ Software Architecture

eCD++ is modularized in the way that systems' objects (written in C++) run as separate software modules with well-defined behaviours and independent functionality. The following are the major components:

- Main Simulator
- DEVS Modeling Subsystem
- Simulation Subsystem
- Messaging Subsystem

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

- Registers Atomic model objects, which are C++ objects derived from the Atomic class;
- Reads in the external events and builds an external events table (if one is created);
- Reads in the model file and builds the model hierarchy;
- Creates the Root Coordinator and trigger it start to run

The class diagram of the *MainSimulator* is shown in the following figure.

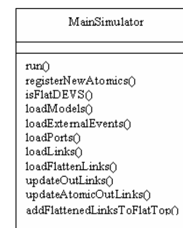

Figure 3. Main Simulator Class

The *DEVS Modeling Subsystem* provides a logical representation of the DEVS models defined by the modeller. The subsystem is composed by the *Models Manager* and the *DEVS Models Hierarchy Tree*. The *Models Manager* manages the models hierarchy. More precisely, it does the following 2 tasks:

- *Main Simulator* registers Atomic model objects, and *Models Manager* creates and manages the Atomic models objects database (a dictionary data structure that stores Atomic model string name-Atomic object pairs). It also creates the *Models Hierarchy Tree* which is composed by atomic and coupled models;
- Employs the *Processor Manager* to create Processor class objects when the *Main Simulator* loads atomic models.

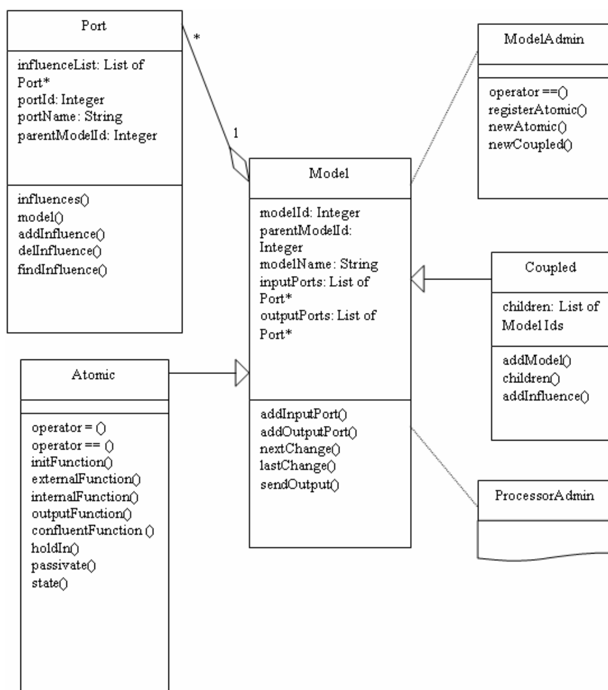

Figure 4. DEVS Modeling Subsystem Class Diagram

The *Models Manager* is implemented by the *ModelAdmin* class, while the implementations for the atomic and coupled models are encapsulated by the Atomic and Coupled class respectively.

The Simulation Subsystem consists of Simulators, coordinators, and the Processors Manager. The *Processors Manager*, which is implemented by the ProcessorAdmin class, manages the Processor class objects. It maintains a hashing table of pointers to Processor class objects, such that actions, such as searching, can be performed upon those objects.

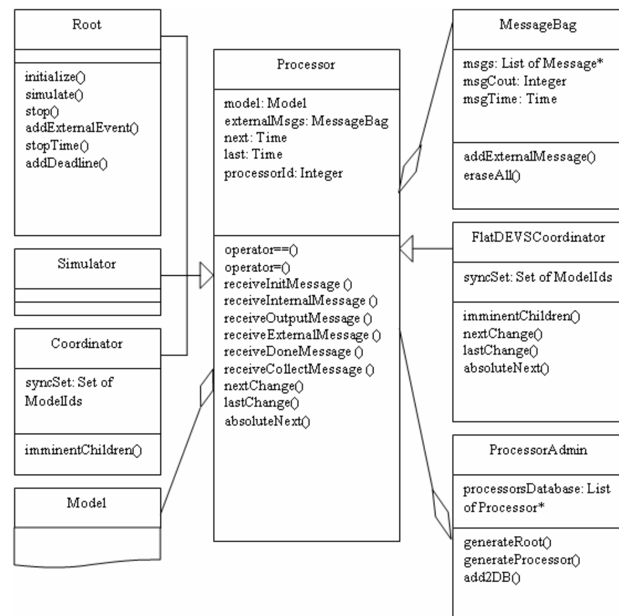

Figure 5. Simulation Subsystem Class Diagram

The *Processor* class implements the DEVS simulation framework in which Atomic and Coupled models run. It defines message handlers that respond to various DEVS messages, such as internal and external state transition messages. The Simulator and Coordinator class are subclasses of Processor. The Root Coordinator is a special Coordinator that manages and controls the simulation cycles. It receives the incoming external events and sends the corresponding External Messages to the Top Coordinator. It also advances the Global Simulation Time. The following figure is the class diagram for the Simulation Subsystem.

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

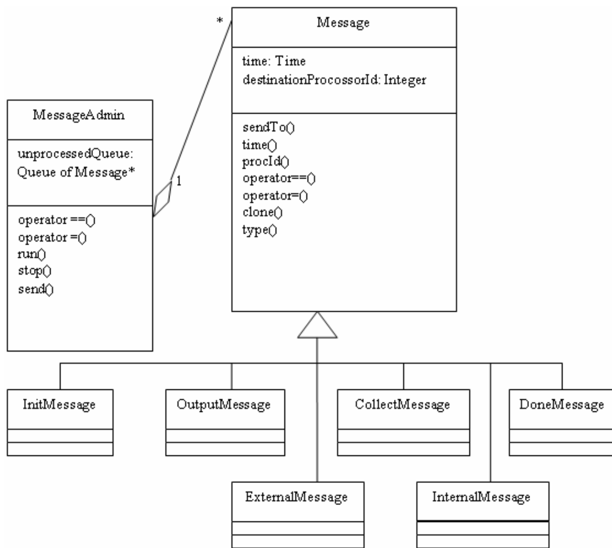

Figure 6. Messaging Subsystem Class Diagram

eCD++ also incorporates a GGAD model loader that parses GGAD files and builds equivalent atomic models. The GGAD model loader is a software module that consists of a front-end and back-end. The front-end is a GGAD parser, written in lex and yacc, that parses the input GGAD files and builds the syntax tree and the symbol table in a similar way as what a typical compiler's front-end would do. The parse() method in the GgadParser class calls GGADparse() which is the main function generated by lex and yacc.

The back-end builds the GGAD atomic models based on the syntax tree and the symbol table. The GGAD models behave exactly the same as if they were written C++. This is achieved by GGAD models providing the same API as that provided by the Atomic class. Providing a consistent API makes the integration of the GGAD model loader with the rest of the eCD++ code become easy.

The *GGAD Parser* is written in Lex and Yacc, which are the tools used to define the context-free grammar of GGAD model files. The GGAD Parser parses the GGAD model file and builds the syntax tree and the Symbol Table. The implementation of the GGAD Parser is encapsulated in the GgadParser class. The *Syntax Tree*, built by the GGAD Parser based on the GGAD file, is a tree structure of GgadSyntaxNode class objects. The GgadSyntaxNode class has 6 subclasses: *GgadFunctionNode*, *GgadConstantNode*, *GgadInputNode*, *GgadPortInNode*, *GgadVariableNode*, and *GgadActionNode*. These classes form a C++ presentation of the GGAD language definition. In other words, the Syntax Tree, which is the composition of the GGAD syntax nodes, provides the behaviours of the atomic model which is described by the input GGAD model file in terms of C++ objects that can be executed during run time. The compositions of an atomic model, such as inputs and outputs ports, variables, and transition functions, are represented by various subclasses of *GgadSyntaxNode*. The Syntax Tree is used by the Ggad Transitions Execution Engine.

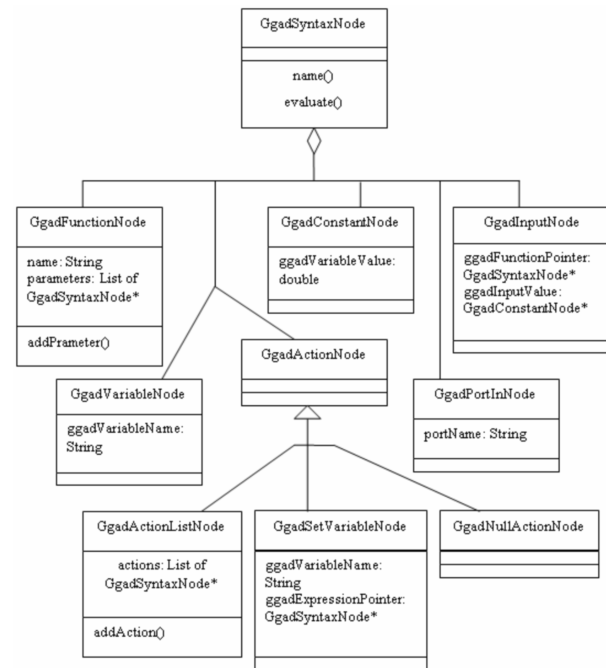

Figure 7. GGAD Syntax Tree Class Diagram

The *GGAD Transitions Execution Engine* is the main body of the GGAD Model Loader. It is responsible for the executions of external, internal, output, and initialization transition functions of the GGAD models. It interacts with the GGAD syntax nodes objects and carries out the calculation results of those atomic model functions. The implementation of the execution engine is encapsulated in the GgadImpl class, of which the class diagram is shown in the Figure 8.

A *real-time solution* is defined as a solution whose correctness depends not only on the computational results, but also on the time at which the results are produced [Sta96]. If a system delivers the correct answer after a certain deadline, it could be regarded as an unsuccessful response. Consequently, a real-time simulator must handle events in a timeliness fashion where time constraints can be stated and validated. eCD++ allows real-time simulation. It also allows interaction between the simulator and the surrounding environment. The inputs of the eCD++ can be received by ports connected to real input devices such as sensors, timers, thermometers, or data collected from human interaction. The outputs can be sent through output ports connected to devices such as motors, transducers, gears, valves, or any other component.

In order to implement the real-time extension, advance of the simulation-clock must be tied to the wall-clock

(*i.e.* physical time). To do so, the *root coordinator* has been modified to provide this functionality. The *root coordinator,* inherited from the *coordinator* class, manages the time advance along the execution of a simulation. It is also responsible for starting each new simulation cycle by issuing the corresponding message. When the *virtual time* approach is used, the messages are immediately generated and sent by the *root coordinator* to initiate the new cycle. For the *real-time* simulation, however, the coordinator must wait until the physical time reaches the next event time to initiate the new cycle. This implies that the periods of inactivity are not skipped in the real-time extension. The simulation process remains quiescent while these periods are being experienced. Instead of logically advancing the virtual time up to the next programmed event and thus anticipating the execution of a programmed task, as what's done by the virtual time approach, the *root coordinator* expects the scheduled wall-clock time to be reached and only then starts the new simulation cycle. In other words, a new simulation cycle can be started either due to the reception of an *external event*, or due to the consumption of the time indicated by *ta(s)* of the root coordinator. Hence, messages interchanged between processors are sent at their actual scheduled time.

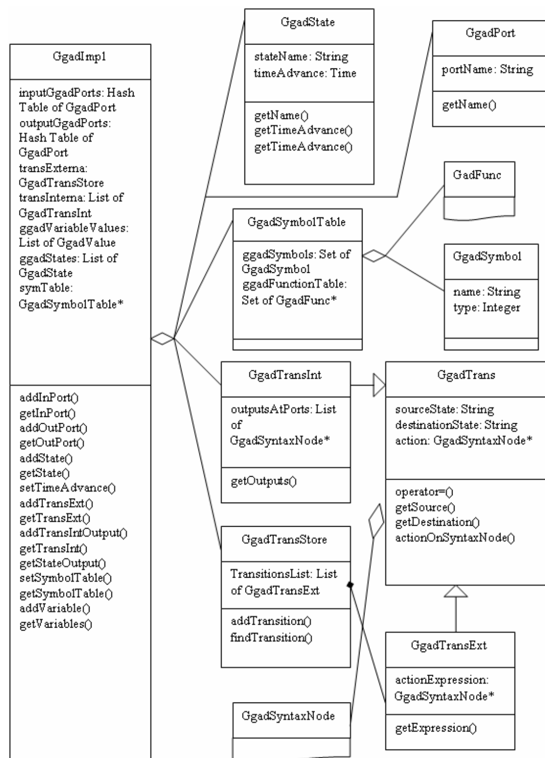

Figure 8. GGAD Transitions Execution Engine Class Diagram

To achieve this in implementation, the eCD++ creates a state machine and uses standard UNIX timer facilities provided by the <linux/time.h> library. The starting state is the "inactive state" in which the simulator is completed passive. When the simulator reads in external events file, it calls add_timer() to add timers with the associate expiry timestamps for all the external events, and then it calls interruptible_sleep_on() to put the simulator into the sleep state in which the simulator is also passive except that the timers start to count down. As the wall-clock time advances, those timers will expire at the exact moments when the external events arrive, their timeout functions will be invoked. The timeout functions call wake_up_interruptible() to wake up the simulator to start a new simulation cycle. When the simulator finishes processing an external event, it calls del_timer_sync() to delete the timer associated with that event and then calls interruptible_sleep_on() to go to the sleep state. It will be waken up upon the arrival of the next external event.

The other situation where the UNIX timers are used is the consumption of the time indicated by *ta(s)* of the root coordinator. If *ta(s)* is set to infinity, then interruptible_sleep_on() is called to deactivate the simulator. Otherwise, a new timer with the expiry timestamp set to *ta(s)* is created, so that the simulator will be waken up after *ta(s)* time, and the timer is deleted upon expiry.

5. A sample application: building an Automated Manufacturing System

eCD++ has been used to implement an Automated Manufacturing System (AMS) defined earlier in [WGM04]. The AMS was built with both hardware and simulated components. The system is composed by dedicated workstations, such as painting or baking machines, that perform tasks on products being assembled and conveyors that transport the products to or from those workstations. The production cycle is organized by a scheduler, which depends on the type of piece being assembled. The scheduler determines which station should receive and work on the product.

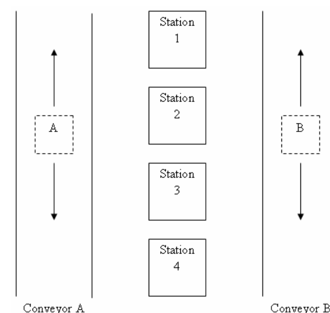

Figure 9. Layout of the AMS

The DEVS model for the AMS system is composed by two coupled components (conveyors), and five atomic components (a controller system, a scheduler, a display controller, and 2 notification bells). Each conveyor is formed by two atomic models (an engine and a senor controller). We have made four atomic components in

hardware: the scheduler, the display controller, and the 2 bells. The rest of the AMS are still in simulation. The resulting configuration is shown in the following figure. The scheduler, the display controller and the bell interact with the simulated Controller Unit through the real I/O ports on the development board (i.e., SBC). And the Controller Unit interacts with the three hardware components the same way as if they were simulated atomic components.

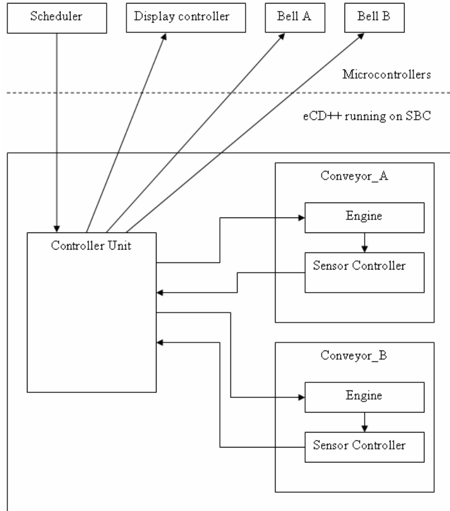

Figure 10. Scheme of AMS (scheduler and display in hardware)

Initially, a product is placed in station 1 of each conveyor belt, and there are no pending events. The *scheduler* hardware is a microcontroller that generates events to the simulated model, indicating that a product has to be sent to a destination station. The following Figure is a sample event file sent to the controller unit by the scheduler.

Event time	***Associated deadline***	***input port***	***associated output port***	***value***
00:02:100	00:05:300	Btn3A	bellA	1
00:06:130	00:10:300	Btn4B	bellB	1

Figure 11. An experimental event schedule

The first event in the figure represents a job scheduled for product A in station 3. The event occurs at time 00:02:100, of which is the format is read as hh:mm:ss:msec, and the simulator receives it via input port *btn3A*, which denotes for station 3A. Assuming the AMS is at its initial state, the first event requires conveyor A transport its product from station 1 to station 3 before the deadline 00:05:300. The second event in Figure 11 shows that conveyor B will start to move its product at time 00:06:130 to station 4, and the given deadline is 00:10:300.

There are two *notification bells*, one for each conveyor. Once the conveyor finishes transporting the product to the destination station, the bell associated with that conveyor will ring indicating the completion. The actual completion time is then checked against the specified deadline of the event.

The *Control Unit (CU)* receives input signals from sensors and the scheduler and determines where to dispatch each piece activating the engines of the conveyor belts. The *Display Controller* handles the digital display based on the signals from the controller unit. It displays the moving directions of the 2 conveyers and the position statuses of the moving products. The moving directions are displayed by the output ports *dirn_disp_a* and *dirn_disp_b*, with the output value 0, 1, or 2 indicating stopping, moving forward, or moving backward, respectively. The position status of a moving piece is shown on the display via the output port *stn_disp*, which outputs the value *ij* to indicate that the product in conveyor *j* has reached station *i*. The display controller also has four LEDs output ports, namely *Led1*, *Led2*, *Led3*, and *Led4*. These LEDs are destination indicators, and each LED port is associated with one station. If, for instance, Led3 is on (with value being 1), that means a product needs to be transported to station 3. And the LED will be turned off (with value being 0) when the product reaches its destination.

eCD++ running on the SBC interacts with the scheduler chip via I/O ports and prints the *actual output time* and the *associated deadline* for each event in the **output file**. The following figure shows the corresponding output file generated by the eCD++ based on the event file shown earlier.

output time	***Deadline***	***result***	***output port***	
00:02:300		**No deadline**	Led3	1
00:02:300		**No deadline**	dirn_disp_a	1
00:03:350		**No deadline**	Stn_disp_a	21
00:04:350		**No deadline**	Stn_disp_a	31
00:04:350		**No deadline**	dirn_disp_a	0
00:04:350		**No deadline**	Led3	0
00:04:360	00:05:300	**Succeeded**	Bell_A	1
00:06:330		**No deadline**	Led4	1
00:06:330		**No deadline**	dirn_disp_b	1
00:07:380		**No deadline**	Stn_disp_b	22
00:08:380		**No deadline**	Stn_disp_b	32
00:09:380		**No deadline**	Stn_disp_b	42
00:09:380		**No deadline**	dirn_disp_b	0
00:09:380		**No deadline**	Led4	0
00:09:380	00:100:300	**Succeeded**	Bell_B	1

Figure 12. Output of the Display Controller

The result in the first column shows the *actual time* at which the output has been sent, which is the wall-clock value at that time (the time elapsed since the beginning of the simulation execution). The second column shows

the *associated deadline time* for the given event. The third column indicates whether the deadline has been met (*i.e.* the actual output time = the associated deadline). Finally, the *output port* and the obtained *value* are shown in the remaining columns.

The following figure shows an execution example of the previous model. As we can see, the different components are activated according to the model specification, and executed in real-time (these results log the execution results on an AMPRO LB700 board).

```
MSG:I/00:000/Root TO flattop
MSG:D/00:000/flattop/... TO Root
MSG:X/00:02:100/Root/btn3a/1.0 TO flattop
MSG:*/00:02:100/Root TO flattop
MSG:X/00:02:100/flattop/b3a/1.0 TO cu
MSG:*/00:02:100/flattop TO cu
MSG:D/00:02:100/cu/00:00:200 TO flattop
MSG:D/00:02:100/flattop/00:00:200 TO Root
MSG:@/00:02:300/Root TO flattop
MSG:@/00:02:300/flattop TO cu
MSG:Y/00:02:300/flattop/led3/1.0 TO Root
MSG:Y/00:02:300/flattop/dirn_disp_a/1 TO Root
MSG:X/00:02:300/flattop/engdirection/1 TO enga
MSG:X/00:02:300/flattop/startstop/1.0 TO enga
MSG:D/00:02:300/cu/... TO flattop
MSG:D/00:02:300/flattop/00:000 TO Root
MSG:*/00:02:300/Root TO flattop
MSG:*/00:02:300/flattop TO enga
MSG:*/00:02:300/flattop TO cu
MSG:D/00:02:300/enga/00:00:050 TO flattop
MSG:D/00:02:300/cu/... TO flattop
MSG:D/00:02:300/flattop/00:00:050 TO Root
MSG:@/00:02:350/Root TO flattop
MSG:@/00:02:350/flattop TO enga
MSG:D/00:02:350/enga/... TO flattop
MSG:D/00:02:350/flattop/00:000 TO Root
MSG:*/00:02:350/Root TO flattop
MSG:*/00:02:350/flattop TO enga
MSG:D/00:02:350/enga/00:01:000 TO flattop
MSG:D/00:02:350/flattop/00:01:000 TO Root
...
```

Figure 13. Simulation Results

6. Conclusion

We presented the design and implementation of eCD++, a modeling and simulation tool that can run on embedded systems and can execute DEVS models that interact with real world events. This capability makes eCD++ become a useful tool to develop real-time applications with hardware-in-the-loop. The technique enables incremental transition from simulated models to the actual hardware counterparts and supports experimental frameworks to facilitate testing in a risk-free environment.

This new development cycle proved to be both useful and effective to develop real-time applications. The use of the P-DEVS formalism, provided a better simulation framework for real-time systems modeling and was able to handle conflicts arising by the execution of simultaneous events.

References

[Cho94] Chow, A.; Kim D.; Zeigler, B. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism". In *Proceedings of Winter Simulation Conference*, Orlando, Florida, 1994. SCS.

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

[HS04] Huang, D., H.S. Sarjoughian. "Software and Simulation Modeling for Real-time Software-intensive System". *The 8th IEEE International Symposium on Distributed Simulation and Real Time Applications*, pp. 196-203, Oct., Budapest, Hungary. 2004

[HSKP97] Hong, J. S.; Song, H. S.; Kim, T. G.; Park, K. H. "A real-time Discrete Event System Specification formalism for seamless real-time software development". *Discrete Event Dynamic Systems* 7 (4): 355-75. 1997

[HZC01] Hu, X.; Zeigler, B.P.; Couretas, J. "DEVS-on-a-Chip: Implementing DEVS in Real-time Java on a Tiny Internet Interface For Scalable Factory Automation". In *Proceedings of the 2001 IEEE Systems, Man, and Cybernetics Conference*. 2001.

[Kim00] Kim, K.; Kang W.; Sagong, B.; Seo, H. "Efficient Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-Hierarchical One". In *Proceedings of the 33rd Annual Simulation Symposium.* Washington DC, USA. 2000.

[Sta96] Stankovic J.; "Strategic Directions in Real-Time and Embedded Systems". *ACM Computing Surveys*, 50th Anniversary Issue, Vol. 28, No. 4, pp. 751-763, December, 1996.

[Wai02] Wainer, G. "CD++: a toolkit to develop DEVS models". Software – practice and Experience. Vol. 32, pp. 1261 – 1306. 2002.

[WGM05] G. Wainer, E. Glinsky, P. MacSween. "Model-Driven Architecture of Real-Time Systems". In *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*. S. Beydeda and V. Gruhn eds., Springer-Verlag. 2005.

[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.