Real-time simulation of DEVS models in CD++

Gabriel A. Wainer

Department of Systems and Computer Engineering,
Carleton University Centre for Visualization and Simulation (V-Sim),
Carleton University,
3216 VSim, 1125 Colonel By Drive,
Ottawa, ON, K1S 5B6, Canada
Email: gwainer@dc.uba.ar

Abstract: The CD++ toolkit was developed in order to implement the theoretical concepts specified by the DEVS formalism. The tool allows the execution of both DEVS and cell-DEVS models. In this work, we present the definition and implementation of a real-time simulator. In such simulations, events must be handled timely and time constraints can be stated and validated accordingly. The new simulation technique allows the interaction between the model and its surrounding environment. Additionally, a non-hierarchical simulation approach is presented and introduced to CD++ in order to reduce the communication overhead.

Keywords: discrete event systems specification; DEVS; real-time DEVS; real-time systems; CD++ toolkit.

Reference to this paper should be made as follows: Wainer, G.A. (2016) 'Real-time simulation of DEVS models in CD++', *Int. J. Simulation and Process Modelling*, Vol. 11, No. 2, pp.138–153.

Biographical notes: Gabriel A. Wainer received his PhD degree (1998, with highest honours) at the University of Marseille, France. He is a Full Professor and Associate Chair in the Department of Systems and Computer Engineering at Carleton University. He authored three books and over 320 research articles; he edited four other books and helped organising a large number of conferences, being one of the founders of SimAUD, SIMUTools and TMS-DEVS. He is a Special Issues Editor of *SIMULATION*, member of the editorial board of *IEEE Computational Science and Engineering*, *Wireless Networks* (Elsevier), and *Journal of Defense Modeling and Simulation* (SCS). He received the IBM Eclipse Innovation Award, SCS Leadership Award, and various Best Paper awards; also, the First Bernard P. Zeigler DEVS Award (2010), the SCS Outstanding Professional Award (2011), Carleton University's Mentorship Award, SCS Distinguished Professional Achievement Award (2013) and Carleton University's Research Achievement Award (2005 and 2014).

1 Introduction

Simulation is a powerful tool for analysing, understanding and developing a wide variety of complex systems. The discrete event systems specification (DEVS) formalism (Zeigler et al., 2000) is a framework for the construction of discrete-event hierarchical modular models that allows for model reuse and reduced development time. In DEVS, basic models (called atomic) are specified as black boxes, and several DEVS models can be coupled forming a hierarchical structural model (called coupled). This formalism provides precision and speedups in the simulations, and a formal approach that can be used to prove properties about the models. CD++ (Wainer, 2002, 2009) is a tool that implements the DEVS theory, and has been widely used to model a variety of applications with success. As a consequence of the modular nature of DEVS, these models can be easily reused to build new systems saving development time. In the last years, CD++ was revised and extended several times (Wainer, 2009; Liu and Wainer, 2007; and Wainer and Wainer, 2013).

Real-time (RT) systems are defined as those whose correctness depends not only on the logical results of computation, but also on the time at which the results are produced (Liu, 2000). If a system delivers the correct answer after a certain deadline, it could be regarded as an unsuccessful response. Therefore, a RT simulator must handle events in a timeliness fashion where time constraints can be stated and validated. CD++ enables important advantages in relation to the performance of execution of RT simulation. A DEVS model advances in continuous time and receives instantaneous asynchronous events that can cause the model to change its state. Potentially, this characteristic allows better performance execution of models using the DEVS formalism and, in particular, using the CD++ toolkit. We provided a thorough study to analyse the performance of a few DEVS simulators in Wainer et al. (2011), and Moallemi and Wainer (2013), focusing on the performance of simulation techniques available in CD++

and showing the feasibility of implementing a new RT simulator based on the CD++ toolkit.

We present the implementation of a RT simulation engine in the CD++ toolkit, which allows interaction between a simulated model and its surrounding environment. In a RT simulation, inputs can be received by ports connected to real input devices such as sensors, timers, thermometers or even data collected from human interaction. Similarly, outputs can be sent through output ports connected to devices such as motors, transducers, gears, valves or any other component. In addition, all the available models developed for previous versions of CD++ may also be executed under the new RT simulator without any modification. The simulation tool allows running similar DEVS models as others built in CD++, and executes them in RT checking the RT constraints. The tool provides a mechanism for checking if deadlines were met or lost. The tool supports all kinds of discrete-event models that can be built with DEVS (which is the most generic discrete-event system specification formalism). The tool executes effectively and with low overhead. We present different results showing the performance of the new tool.

2 Background

The development of embedded RT controller's software has usually posed interesting challenges to the developers 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 modelling 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. We propose to use M&S based on the DEVS formalism with this goal.

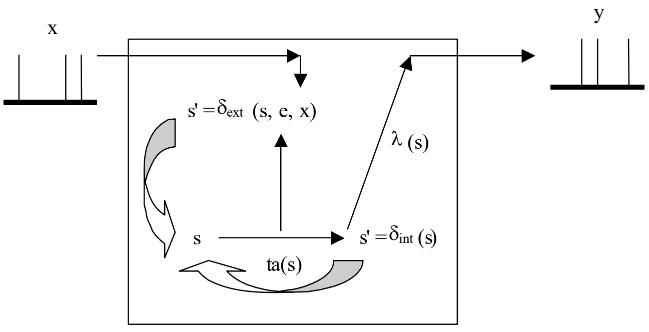
DEVS (Zeigler et al., 2000) is a formalism that specifies systems whose states change either upon the reception of an input event or due to the expiration of a time delay. It allows hierarchical decomposition of the model defining a way to couple existing DEVS models. A real system modelled using DEVS can be described as a composition of atomic (behavioural) and coupled (structural) components. An atomic model is defined by:

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

where X is the input events set, S is the state set, Y is the output events set, δ_{int} is the internal transition function, δ_{ext} is the external transition function, λ is the output function, and *ta* is the time advance function. Figure 1 shows states and variables in DEVS models.

As we can see in Figure 1, the semantics of DEVS models is as follows. Each atomic model can be seen as having an interface consisting of input (X) and output (Y) ports to communicate with other models. Every state (S) in the model is associated with a time advance (*ta*) function, which determines the duration of the state. Once the time assigned to the state is consumed, an internal transition is triggered. At that moment, the model execution results are spread through the model's output ports by activating an output function (λ). Then, an internal transition function (δ_{INT}) is fired, producing a local state change. Input external events are collected in the input ports. An external transition function (δ_{EXT}) specifies how to react to those inputs.

Figure 1 DEVS semantics



Source: Zeigler et al. (2000)

A coupled model groups several DEVS components into a compound model that can be regarded, owing to the closure property, as a new DEVS model. This allows hierarchical model construction. 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 another component, or as an output of the coupled model itself.

CD++ (Wainer, 2009) implements DEVS, allowing the definition and simulation of models using the concepts described previously. The tool provides a specification language that allows describing model coupling, setting initial values, and indicating external input events. Additionally, atomic models are developed under C++. Two basic abstract classes exist: *model* and *processor*. The former is used to represent the behaviour of the atomic and coupled models, while the latter implements the simulation mechanisms. The *atomic* class implements the behaviour of an atomic component, whereas the *coupled* class implements the mechanisms of a coupled model. A *simulator* object manages an associated atomic object, handling the execution of its δ_{int}, δ_{ext} and λ(s) functions. A *coordinator* object manages an associated coupled object. Only one *root coordinator* exists in a simulation, which manages global aspects of the simulation environment, maintains the global time, and it starts and stops the simulation. Lastly, it receives the output results that must be sent to the environment. The simulation process is message driven; it is based on the message exchange among processors.

DEVS was extended in several other ways to adapt the formalism for various goals. These extensions include: fuzzy DEVS (Zeigler et al., 2000) providing a fuzzy logic to the state definition, SymbolicDEVS (Zeigler and Chi, 1992) using symbolic algebra to represent time, and rational time advance discrete event systems specification (RTA-DEVS) (Saadawi and Wainer, 2012, 2013a, 2013b; Saadawi et al., 2012) using intervals arithmetic to operate timelines. Symbolic DEVS was developed in early '90s; it extends the DEVS formalism to define time as linear polynomials in place of real numbers. This formalism can be used to study the fault conditions and other properties when doing model verification. Its abstract simulator examines all possible strict choices of imminent forking execution when needed. RTA-DEVS (Saadawi and Wainer, 2012, 2013a) is another proposed extension to DEVS formalism. In this case, time is defined as intervals with rational borders. The goal of this formalism is to allow only the specification of models that can be automatically verified using model checking methods. To achieve this goal, they reduce the set of specifiable models to those that never have irrational time advances. The RT and embedded extensions of the DEVS methodology (Saadawi and Wainer, 2012, 2013a; Cho et al., 2000; Furfaro and Nigro, 2009; Song and Kim, 2005) provide a model-driven approach towards developing embedded controllers. The formal and intrinsic advantages of DEVS are combined with RT features to propose a design scheme for such applications. Issues such as hardware-in-the-loop simulation (HILS) or human-in-the-loop simulation are addressed in this framework which allows for interfacing the DEVS simulation models with the RT environment. The benefits of simulation-based verification can be employed, allowing for pervasive verification of the system under development in a risk-free setting, exploring varying test scenarios. A high-level abstract hardware-software modelling scheme can be provided, where different components of the target system can be modelled together.

3 RT extension to the CD++ toolkit

In Wainer et al. (2011), an analysis of simulation performance showed that a relatively small overhead is incurred when executing mid to large-scale models. The analysis showed that a RT extension to the toolkit, introduced with further details in this paper, was feasible. Firstly, we describe the virtual-time simulation approach that was previously available in CD++. Secondly, we introduce the new RT extension developed in the toolkit. In addition, some examples are presented to show both simulation approaches.

3.1 Virtual-time simulation

In order to execute a simulation using the virtual-time approach, CD++ maintains a variable in which the current simulation time is stored and updated. This value is not linked to any physical clock. The update of that time variable is performed by the simulator as follows. When the simulation starts, this simulation time is initialised to zero. Then, the imminent event (i.e., the event with the earliest time of occurrence) is computed and the simulation time is advanced accordingly in order to process that event. Once it has been processed completely, the new imminent event is computed, the simulation time is advanced and the next event is processed. This cycle of advancing the simulation time and processing the imminent event is repeated. The execution ends when the simulation time reaches the stop time indicated by the user, or else when there are no more pending events.

The execution of a model using the virtual-time approach with the CD++ toolkit is illustrated in the following example. This simple DEVS model is formed by two inner-coupled components, four input ports, and four output ports. Each inner-coupled component has two atomic models that execute time-consuming code in their internal and external transition functions. The workload of each component varies from one component to another.

Figure 2 Sample model

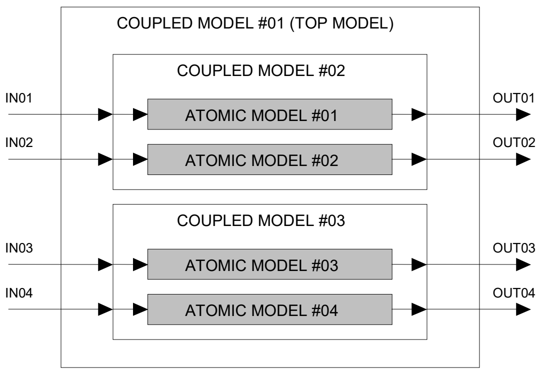


Table 1 Sample event file for the given model using the virtual time approach

Event time	*Input port*	*Value*
00:00:05:000	in01	1
00:01:28:100	in02	1
00:18:21:000	in03	1
00:31:15:500	in04	1
00:45:30:200	in02	1
01:05:00:500	in01	1
02:15:00:900	in04	1
05:50:30:200	in03	1

The model in Figure 2 can receive external events through its input ports. This information is supplied by the user in the event file where times are written in the *hours:minutes:seconds:milliseconds* format as seen in Table 1. This table shows that the first event arrives at time 00:00:05:000 through the port in01 with a value of 1, a second event arrives at time 00:01:28:100 through the port in02 also with a value of 1, and so on. When CD++

executes the previous model with the event file previously shown, results can be found in an output file.

Table 2 shows the output file obtained, illustrating how time evolves when the virtual-time approach is used. The first column indicates the physical time (i.e., wall-clock time) at which the output has been sent, representing the time elapsed since the beginning of the simulation. The simulation time associated to each message is shown in the second column. The output port and the associated value that was sent are shown in the third and fourth columns respectively. For instance, the first line describes an output sent through port out01 with a value of 1, which occurred at the simulation time 05:000 and at the corresponding physical time 00:060.

Table 2 Output file using the virtual time approach

Wall-clock time	*Simulation time*	*Output port*	*Value*
00:00:00:060	00:00:05:000	out01	1
00:00:00:130	00:01:28:100	out02	1
00:00:00:230	00:18:21:000	out03	1
00:00:00:320	00:31:15:500	out04	1
00:00:00:390	00:45:30:200	out02	1
00:00:00:430	01:05:00:500	out01	1
00:00:00:520	02:15:00:900	out04	1
00:00:00:620	05:50:30:200	out03	1

As we can see, all periods of inactivity are skipped, which leads to the particularity shown above when executing a model. Thus, events are not processed at their actual scheduled times.

3.2 RT simulation

Varied modifications were made to allow RT simulation in DEVS (Cho and Kim, 1998; Hong et al., 1997; Hu and Zeigler, 2005) and in the CD++ toolkit. A RT simulator must handle events in a timely fashion where time constraints can be stated and validated. The new features allow interaction between the simulator and the surrounding environment. Therefore, inputs can be received by ports connected to real input devices such as sensors, timers, thermometers or human interaction. Similarly, outputs can be sent through output ports connected to devices such as motors, transducers, gears, valves or any other component.

To implement the RT extension to the toolkit, the advance of the simulation-clock is now tied to the wall-clock (i.e., physical time). For that reason, the root coordinator now manages the time advance along the execution of a simulation. In addition, it is responsible for starting each new simulation cycle by issuing the corresponding message. A coordinator must wait until the physical time reaches the next event time to initiate the new cycle. A new simulation cycle can be started either by:

1 the reception of an external event

2 the consumption of time indicated by *ta*(*s*).

Periods of inactivity cannot be skipped and the simulation remains quiescent during these periods. Instead of forcing a time advance up to the next programmed event and thus anticipating the execution of a programmed task, the root coordinator expects the scheduled time to be reached and only then starts the new simulation cycle. Hence, messages interchanged between processors are sent, ideally, at their actual scheduled time. However, this ideal timely processing of events might not be achieved if the incurred overhead degrades performance significantly, which is an important concern addressed later in this article and also in Wainer et al. (2011).

Timeliness along a simulation is crucial in the RT approach. When a model is being executed using this technique, it is important to check different time constraints throughout the simulation. Particularly, the time at which an event has been completely processed is a meaningful measure of success.

Typically, a model must react to an external event within a given time and produce an output to respond to the problem. For instance, if a sensor indicates dangerous overheat, an energy plant needs to shut down a part of its system within a given period of time or severe consequences might occur.

Consequently, we provided a means to indicate the deadline for each external event in the RT extension of CD++. The new extended format of the event file is illustrated in the next Table.

Table 3 Format of the event file in the RT extension

Event time	*Deadline*	*Input port*	*Output port*	*Value*
hh:mm:ss:ms	hh:mm:ss:ms	Port name	Port name	Numeric value

As we can see, a deadline and an output port must be stated, and the simulator can check whether the physical time meets the associated deadline when sending an output through the associated port. Once the execution is over, both successful and unsuccessful responses are stored for further analysis. A RT simulation based on the model in Figure 2 is illustrated in Table 4.

Table 4 Sample event file for the given model using the RT approach

Event time	*Deadline*	*Input port*	*Output port*	*Value*
00:00:05:000	00:00:05:020	in01	out01	1
00:01:28:100	00:01:29:000	in02	out02	1
00:18:21:000	00:18:21:050	in03	out03	1
00:31:15:500	00:31:15:540	in04	out04	1
00:45:30:200	00:45:30:270	in02	out02	1

The file shows the event times and their associated deadlines and output ports for each external event. For example, the result for the event arrived at time 05:000 through the input port in01 should be completed and an output generated before 05:020 through the output port out01 (a deadline of 20 ms.). In order to verify these constraints, the toolkit informs:

1 the actual output time
2 the simulation time
3 the associated deadline
4 the output port
5 the issued value, for each event in the output file.

Additionally, a result column is included with one of these two values:

success	if **actual output time** = **associated deadline**
not met	if **actual output time** > **associated deadline**

Table 5 shows the corresponding output file for the model executed using RT engine.

Table 5 Output file using the RT approach

Wall-clock time	*Simulation time*	*Deadline*	*Result*	*Output port*	*Value*
00:00:05:060	00:00:05:000	00:00:05:020	Not met	out01	1
00:01:28:170	00:01:28:100	00:01:29:000	Success	out02	1
00:18:21:090	00:18:21:000	00:18:21:050	Not met	out03	1
00:31:15:580	00:31:15:500	00:31:15:540	Not met	out04	1
00:45:30:270	00:45:30:200	00:45:30:270	Success	out02	1

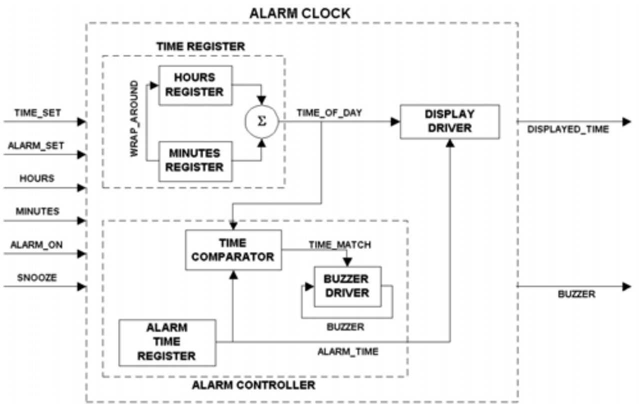
The first column shows the actual time at which the output has been sent (the wall-clock value since the beginning of the simulation). The second column shows the simulation time at which this output has been scheduled, whereas the third column shows the associated deadline time for the given event. It is possible to check whether the deadline has been met (i.e., the actual output time ≤ the associated deadline) simply by looking at the fourth column. Finally, the output port and the obtained value are shown in the remaining columns.

For instance, the first line shows a deadline that has not been met in the execution (the deadline was set at 05:020, while the actual output was at 05:060). Consequently, not met is printed in that line, along with the output port out01 and the value that has been issued. On the other hand, the second line shows an event whose deadline has been met successfully.

Different types of models can be executed with the new CD++ RT extension. For instance, an alarm clock model (Wainer, 2009) was developed to analyse the RT constraints under the new simulation approach. This model can be considered as a part of a more complex system. The model, which clearly has an important component of time, is depicted in Figure 3.

The entire model has three levels. The top level is the *ALARM CLOCK*, which has six input signals representing the push buttons and switch positions in the real system. The input port *TIME_SET* is used in combination with *HOURS* and *MINUTES* to set the time of day. Similarly, the input port *ALARM_SET* is used in conjunction with *HOURS* and *MINUTES* to set the alarm time. The buzzer sounds if *ALARM_ON* is set and the actual time (i.e., time of day) is equal to the alarm time. *SNOOZE* stops the buzzer for a period of 10 minutes after which the buzzer will automatically sound again if *ALARM_ON* is set. The model has two output ports: *DISPLAY_TIME* represents a four-digit display, while *BUZZER_ON* represents the output of the buzzer speaker.

Figure 3 Alarm clock conceptual model



Source: Wainer (2009)

The top model is subsequently decomposed into sublevels. The first level consists of three components: the *TIME_REGISTER* which holds and automatically increments the time of day, and the *ALARM_CONTROLLER* which holds the alarm time and decides whether the buzzer should be turned on or off. The third component is an atomic model named *DISPLAY_DRIVER*, which determines if *time of day* or *alarm time* must be displayed. The second level consists of five different atomic components. The *HOURS_REGISTER* and *MINUTES_REGISTER* respectively hold the information about the time of day. The *TIME_COMPARATOR* compares the current time with the alarm time to detect a match and potentially sound the buzzer. The *ALARM_TIME_REGISTER* holds the alarm time. Finally, the *BUZZER_DRIVER* decides when the buzzer needs to be activated or deactivated.

Table 6 shows results obtained after the execution of the alarm clock. Generally, we can see that as time passes, the actual time is obtained through the DISPLAY_TIME port, which resembles the usual digital display of an alarm clock. In addition, information about the buzzer alarm is obtained in the output file. The buzzer is turned on at 30:00:000 and this is notified through the output port BUZZER_ON at that time. The time still evolves normally and the actual time is obtained through the DISPLAY_TIME port. The user turns off the buzzer at 32:45:500, where the BUZZER_ON issues a 0. Recall that the buzzer can be deactivated using the SNOOZE button, but the alarm will buzz again after an idle period of ten minutes. Hence, at time 42:45:500 the buzzer is turned on again, when the output port BUZZER_ON issues a 1. Note that actual output times are equal to their corresponding simulation times. This fact shows that delays are imperceptibly small all along the simulation of this alarm clock. Therefore, such simulation could meet easily the deadlines imposed by the user.

Table 6 Output file excerpt – execution of the alarm clock

Wall-clock time	*Simulation time*	*Output port*	*Value*
00:01:00:000	00:01:00:000	DISPLAY_TIME	00:01
00:02:00:000	00:02:00:000	DISPLAY_TIME	00:02
00:03:00:000	00:03:00:000	DISPLAY_TIME	00:03
(…)	(…)	(…)	(…)
00:30:00:000	00:30:00:000	DISPLAY_TIME	00:30
00:30:00:000	00:30:00:000	BUZZER_ON	1
00:31:00:000	00:31:00:000	DISPLAY_TIME	00:31
00:32:00:000	00:32:00:000	DISPLAY_TIME	00:32
00:32:45:500	00:32:45:500	BUZZER_ON	0
00:33:00:000	00:33:00:000	DISPLAY_TIME	00:33
. . .			
00:42:00:000	00:42:00:000	DISPLAY_TIME	00:42
00:42:45:500	00:42:45:500	BUZZER_ON	1
00:43:00:000	00:43:00:000	DISPLAY_TIME	00:43

3.3 *Flat simulation technique*

Recall from Section 2 that CD++ builds a hierarchy of model objects (atomic and coupled) and a corresponding hierarchy of processor objects (simulator and coordinator objects) to perform the simulation. A simulator is created for each atomic component, whereas one coordinator is created for each coupled one. Thus, as size and complexity of the simulated model are increased, the associated processor structure is increased accordingly.

The simulation evolves based on the exchange of messages among simulators and coordinators. The number of intermediate coordinators in the hierarchy can be arbitrarily high depending on the model. Hence, the number of messages needed for a single simulation cycle can be considerable for large-scale models, because of the size of the corresponding hierarchy. Therefore, a main problem to be resolved is the overhead incurred by message passing among processors (Wainer et al., 2011). To alleviate this message passing overhead, a new flat simulation technique was implemented (Glinsky and Wainer, 2002). The approach is called flat in contrast to the hierarchical one that was explained previously.

When the flat simulation technique is used, the associated processor hierarchy is greatly simplified. Messages are exchanged only between the root coordinator and the flat coordinator, the only two processors that are required in the new hierarchy. More information about the flat technique is studied in Glinsky and Wainer (2002), and is out of the scope of this work. Additionally, a similar development for a different DEVS simulator can be found in Kim et al. (2000).

4 Performance analysis of the RT simulator

Testing the performance of a simulator tool is a very complex task. To make the analysis easier and more accurate, a synthetic experimental frame has been developed. To perform a thorough and accurate study of the overhead, the synthetic model generator must be able produce a wide variety of models. The produced models must be similar to the ones that are studied in the real world. To characterise a model, one should consider different aspects of its shape and behaviour. Some of the most important characteristics are: number of levels in the model hierarchy, number of atomic components, number of coupled components, total size, number of interconnections between components, and workload in internal and external transitions.

We used Dhrystone benchmark code (Weicker, 1984) to generate different workload in atomic transition functions. Dhrystone code is a synthetic benchmark intended to be representative for system (integer) programming. Thus, it is possible to make atomic models execute time-consuming code inside their internal and external transition functions in order to simulate real tasks that could be performed by them. We used a Dell workstation with quad-Core Xeon processor and 4 GB of RAM to conduct these experiments (and the ones in Section 5).

The tool can generate three different types of models: Type-1 (low complexity), Type-2 (medium complexity) and Type-3 (high complexity). See Weicker et al. (2011), and Moallemi and Wainer (2013) for a more detailed description of the synthetic model generator.

Representative models can be created with the synthetic generator. Tests results show both the percentage of success and the worst-case response time for each case. The former is obtained as follows:

$$\begin{aligned}&\textit{Percentage of success} =\\ &\frac{(\textit{number of events} - \textit{number of missed deadlines}) * 100}{\textit{number of events}}\end{aligned}$$

On the other hand, the worst-case response time is obtained as follows:

$$\textit{Worst-case response time} = \max(r_1, r_2, ..., r_N)$$

where r_i is the response time for the i^{th} event, and N is the *number of events* for the given simulation.

Figure 4 shows the corresponding charts for the models that were created using our synthetic model generator. The experiments have been grouped in categories:

1 Type-1 models with variable depth

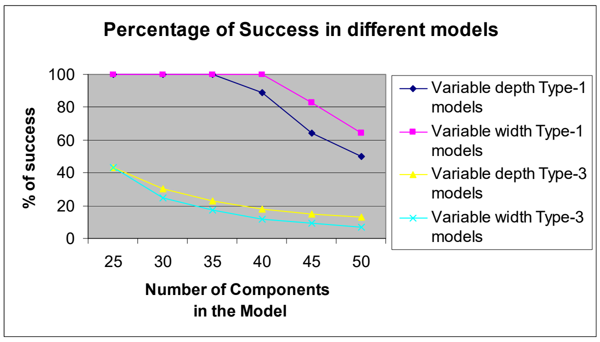
2 Type-1 models with variable width

3 Type-3 models with variable depth

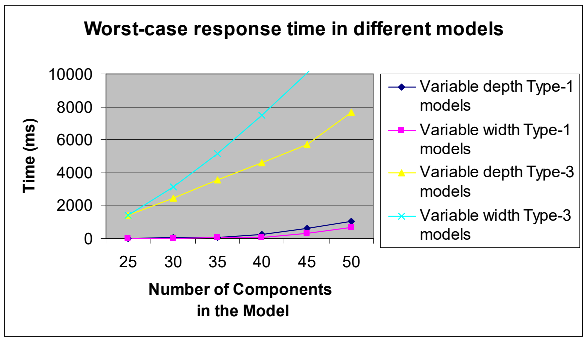
4 Type-3 models with variable width.

These results show the performance of the RT simulator under very stressful scenarios, with external events arriving at high frequencies with very strict deadlines. Under these circumstances, the difference between the time to be spent executing on the transition functions and the associated deadline for the events was minimum. In some cases, that difference was as less as 10 milliseconds, which might represent less than 1% of the theoretical execution time.

Figure 4 Model execution with variable depth/width, (a) % of success (b) worst-case response time (see online version for colours)



(a)



(b)

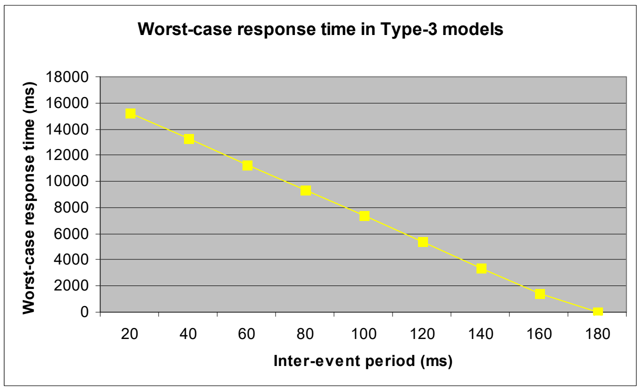
Figure 4(a) shows the percentage of success for Type-1 and Type-3 models when depth is variable and the width is fixed, and also when the width is variable and the depth is fixed, whereas Figure 4(b) illustrates the worst-case response time for the same models. As we have explained before, this is a very stressful scenario and the conditions are very demanding. In particular, the previous charts show that it is harder to simulate Type-3 models when the number of components increases due to their complex structure, in comparison with the equivalent (and simpler) Type-1 models. Consequently, the worst-case response times are remarkably increased for Type-3 models.

Under these conditions, when a Type-1 model has 40 active components in its structure, more than 90% of success can be achieved. Alternatively, less than 20% of the deadlines are met for Type-3 models with 40 components in their structures.

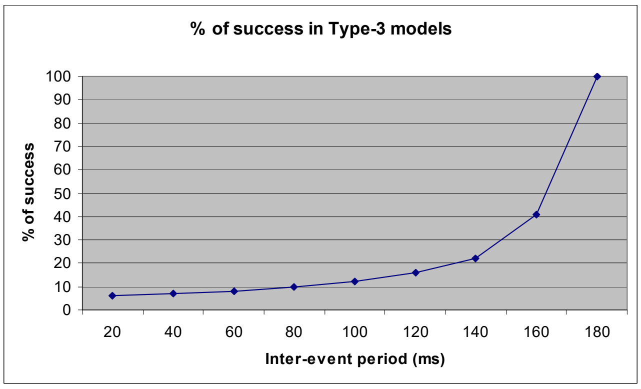
In general, the previous charts illustrate that if the number of components is increased, deadlines are more likely to be missed and therefore the percentage of success is reduced. This is because the number messages needed to perform the simulation grows in relation to the size and complexity of the model.

The previous cases studied variations to the models themselves, taking into consideration their depth and width. A different approach can also be considered, where the shape of the models remain unchanged but the scenario in which they execute is modified. Hence, different inter-event periods (i.e., the frequency of event arrivals) are used in these experiments. Consequently, we simulate external events that arrive at a different pace to the model, and analyse the behaviour of the simulator under such circumstances. In addition, the impact of varying the amount of time-consuming code in transition functions is also tested. The following charts show the results for Type-3 models with five components per level (i.e., four atomic components and one coupled component in each level) and five levels in their hierarchy that were also generated using our benchmark.

Figure 5 RT execution of models with variable inter-event period (see online version for colours)



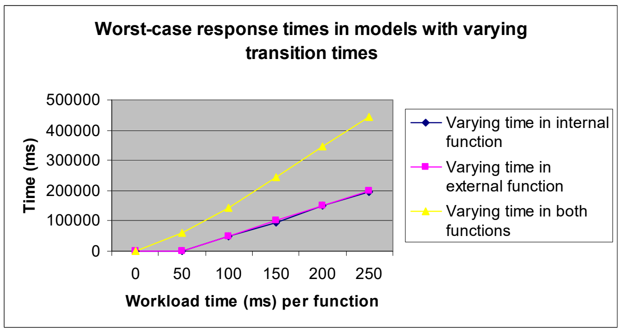
(a)



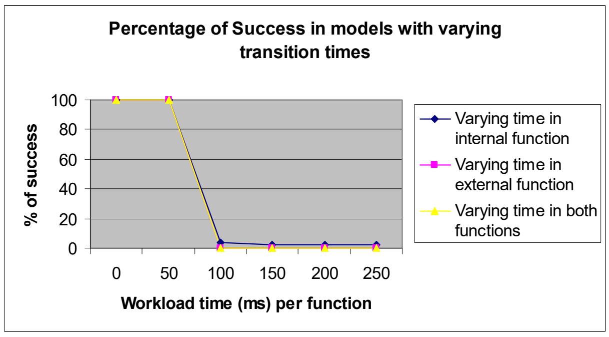
(b)

Figures 5(a) and 5(b) illustrate the influence of inter-event periods on a simulation. When the frequency of event arrival is extremely high, the worst-case response times are notoriously increased. This situation occurs because excessively small inter-event times do not allow the simulator to process all the messages involved with the event e_k before the arrival of the next event, e_{k+1}. When this situation arises, queued unprocessed messages are accumulated, and therefore the simulation presents an evident degradation of performance. The degradation of performance can be noticed by observing the worst-case response time for a given simulation. Here, a simulation with an inter-event period of 20 milliseconds results in a worst-case response time of 15,260 milliseconds. On the other hand, Figure 5(b) shows that larger inter-event periods result on greater percentages of success. When the intervals between events become greater than 180 milliseconds, the simulator meets all the associated deadlines for the execution.

Figure 6 RT execution of models with variable transition time (see online version for colours)



(a)



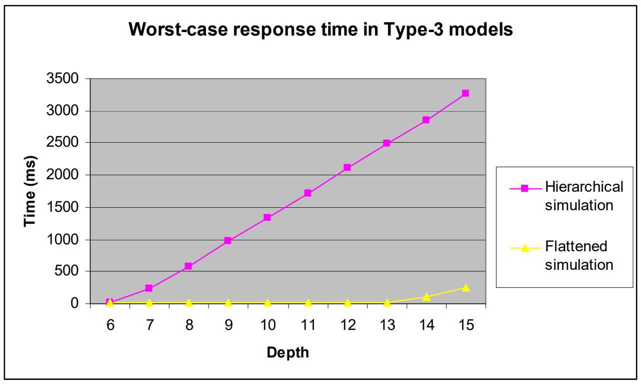
(b)

Figure 6 shows the results when the models spent different amounts of time in the internal and external transition functions. As expected, Figure 6(a) shows that the worst-case response times are remarkably increased when the time-consuming code is increased, because of the time that has to be spent on executing each atomic transition function. When the workload is executed in both transition functions, the worst-case response time is doubled. We can see that proper percentages of success when the workload time per function is 0 or 50 milliseconds, in spite of the place where the time-consuming code is being executed. In contrast, a noticeable reduction in the percentage of success is observed in all cases when the time in the transition functions is increased to 100 milliseconds. In general and as it was expected, if the workload in the transition functions increases, the percentage of success is reduced consequently. On the other hand, Figure 6(b) shows that the model cannot meet its deadlines if a large amount of time is spent in the transition functions.

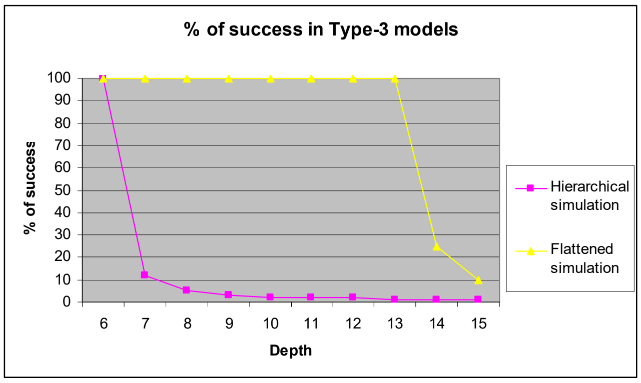
Finally, the RT performance of the flat simulator is analysed. Its execution is compared to that obtained using the flat simulator to determine its efficiency.

In these experiments, both simulators execute Type-3 models with variable depth and width, and results are shown in Figures 7 and 8, respectively. The depth of each model ranges from 6 to 15, whereas the width of each model ranges from 5 to 11. The size of the resulting models, which is a very important parameter of the simulation, lies between 40 and 120 components per model.

Figure 7 Comparison of hierarchical and flat approaches (variable depth) (see online version for colours)



(a)

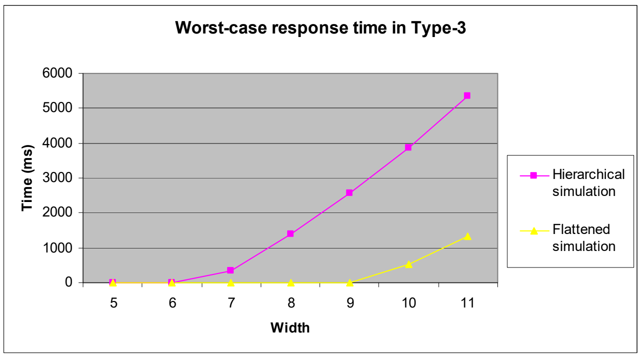


(b)

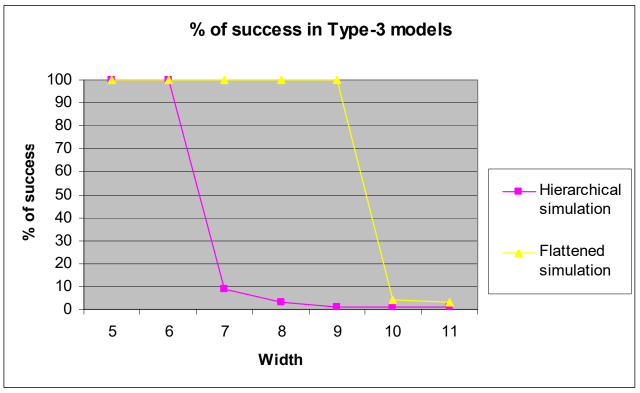
Figure 8(a) shows a comparison of worst-case execution times for both simulation approaches when models with different sizes and shapes are executed. Notice that in deeper models, the difference between the hierarchical and flat simulators becomes more noticeable. Hence, Figure 8(b) illustrates that when the model's depth is equal or larger than 7, it is not possible to meet all the deadlines, and the percentage of success is remarkably reduced if the hierarchical approach is used. However, if the flat technique is employed, the percentage of success remains optimal for models whose depth is 13.

Figures 9(a) and 9(b) show similar results for models with variable width. Figure 9(b) shows the percentages of success for models with different width using both simulation techniques. Wider models are more complex, and therefore the overhead incurred by the hierarchical simulator is increased. However, the flat coordinator can also manage wider models more effectively, as it is shown in both figures, where the flat approach outperforms the hierarchical approach.

Figure 8 Comparison of hierarchical and flat approaches (variable width) (see online version for colours)



(a)



(b)

In general, we can see that the flat technique outperforms the hierarchical one, usually achieving better response time and better percentage of success in the execution. Thus, the use of the non-hierarchical approach allows the simulation of larger models with better performance results. These results are a consequence of the reduced number of messages exchanged in the simulation that exist in the flat simulation mechanism.

5 RT model execution

We showed the possibility of executing models under RT CD++ with relatively small overhead. In this section, two sample models are studied. The first is a vending machine model; the second is a more complex model that interacts with real hardware.

5.1 Case study 1: a vending machine model

A vending machine model (Wainer, 2009) was used for further analysis of the RT extension in CD++. This simulated vending machine is similar to those existing in many cafeterias. Different items can be purchased by inserting the sufficient amount of money and then selecting the appropriate button to dispense the desired product. The machine returns the correct amount of change, keeps track of the number of items that were dispensed and informs out-of-stock products to the customer when necessary.

The system is composed of several atomic components (a *coin collector*, an *item selector*, a *change maker*, a *balance display*, an *item processor* and others) and coupled components (a *service controller,* which is composed by a *vending controller*). The model has three input ports. Coins are inserted via the *COIN_IN* port, items are selected via the *ITEM_IN* port and change is requested via the *REQUEST_IN* port. The output ports are used as follows: *ITEM_OUT* is used to dispense the products, *OUT* resembles the balance display of the machine and *CHANGE_OUT* is used for the returned coins.

Figure 9 Vending machine conceptual model

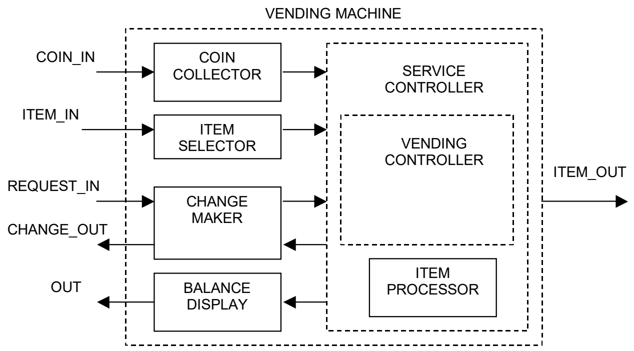


Table 7 shows a sample event file for the vending machine model. Here, a customer inserts different amounts of money and requests a particular item. Deadlines are imposed to each incoming event.

Table 7 External event file – vending machine

Event time	*Associated deadline*	*Input port*	*Associated output port*	*Value*
00:00:10:000	00:00:12:500	COIN_IN	OUT	0.25
00:00:15:000	00:00:17:500	COIN_IN	OUT	1.00
00:00:20:000	00:00:25:500	COIN_IN	OUT	0.25
00:00:25:000	00:00:30:000	ITEM_IN	ITEM_OUT	28
(…)	(…)	(…)	(…)	(…)

Table 8 Output file excerpt – execution of the vending machine in RT

Output time	*Output port*	*Value*
00:00:12:010	OUT	0.25
00:00:17:010	OUT	1.25
00:00:22:010	OUT	1.50
00:00:28:020	ITEM_OUT	28
00:00:30:010	OUT	0.00
(…)	(…)	(…)

For instance, the first quarter is received through the COIN_IN port at time 00: 00:10:000, and the associated output is expected through the port OUT before 10:250. Then, a dollar (1.00) is received at time 15:000, and so on. Finally, the item 28 is selected at time 25:000.

Table 8 shows an excerpt of the vending machine output file. The balance display corresponds to the OUT port, and its value is updated two seconds after each coin is inserted. The item 28 is dispensed through the ITEM_OUT port at time 00:28:000, and the table shows that events are processed on time.

5.2 Case study 2: HILS

A more complex example is shown here, which incorporates hardware-in-the-loop as a part of the model. A more detailed study of this example model can be found in Li et al. (2003).

The development approach consists in building a model in a simulated environment, and incrementally replaces some of the simulated components with real hardware. Thus, we will first use a model that was entirely developed in RT CD++, which will be referred to as the experimental frame *Test B-1*. Then, some of the components are replaced by the real hardware surrogate in the experimental frame for *Test B-2*. The following figure shows a block diagram of the 2189M EZ-KITLITE DSP Board, which was used in the experiment.

Figure 10 Scheme of the analogue devices 2189M EZ-KITLITE evaluation board

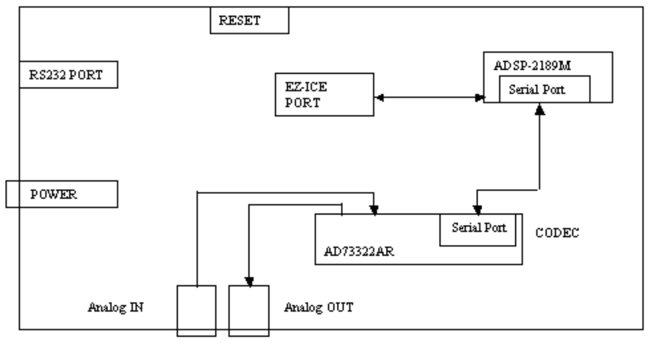
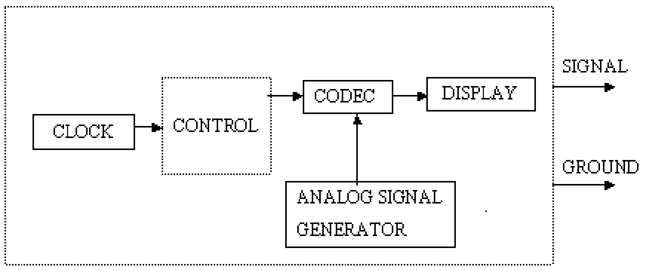


Figure 11 Experimental frame for Test B-1



The sampling rate of the CODEC is programmable with four separate settings, offering a 64 kHz, 32 kHz, 16 kHz or 8 kHz sampling rate (from a master clock of 16.384 MHz). In order to control the CODEC and make use of the ADC (A/D converter) and DAC (D/A converter), we need to program the control registers. This prototype includes hardware and software components and is implemented in two stages. In the first stage, a set of models was built using RT CD++. Figure 11 shows the experimental frame for Test B-1, which was used to test the behaviour of the CODEC model.

The experimental frame for Test B-1 has five atomic models:

- *Clock:* this model generates control signals with a predefined period.
- *Control:* this model distinguishes the incoming signal. If a command signal is received, it will invoke the CODEC atomic model.
- *CODEC:* this model simulates the behaviour of the CODEC.
- *Analogue signal generator:* this model generates an analogue signal.
- *Display:* this model updates and displays the results obtained from the CODEC model.

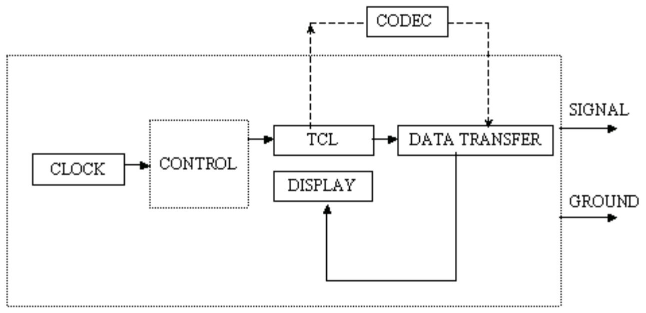
The coupled model shown in Figure 11 was built to test the behaviour of the software version of the CODEC. Table 9 shows the simulation output for Test B-1.

Table 9 Output file excerpt for Test B-1

Output time	*Port*	*Value*
00:00:40:708	Signal	0001
00:00:41:008	Signal	0110
00:00:41:308	Signal	0011
00:00:41:608	Ground	0000
00:00:41:900	Signal	1010
00:00:42:200	Ground	0000
(…)	(…)	(…)

Table 9 shows the output time, the associated port and the received value. Hence, the first digital signal is obtained at time *40:708* through the *signal* output port. Its associated value is *0001*, a binary string that represents the digital value of the signal. At the time *41:608*, another digital signal arrives through *ground* output port and its value is *0000*. All digital signals are processed successfully through the software CODEC model.

Figure 12 Experimental frame for Test B-2 (includes hardware-in-the-loop)



The application model described previously was reused, replacing the CODEC by the actual hardware. The experimental frame for Test B-2 was modified to support interaction with the real CODEC on the board, and existing atomic models were reused. This experimental framework is depicted in the next figure.

In this test, a CLOCK model generates periodic signals to awake the control model. Then the CONTROL model invokes the TCL model, which is in charge of initialising the CODEC and start the conversion. When the CODEC finishes the conversion, the DATATRANSFER model will acquire these data and send them to the control model. Finally, the DATATRANSFER model will feed the data to the DISPLAY model.

The new atomic models are:

- *TCL:* this model invokes and opens the VisualDSP debugger system. Once the debugger system is opened, the different TCL files needed for A/D and D/A access can be invoked to obtain samples and regenerate the analogue signal.
- *DATATRANSFER:* this model reads the digital samples and sends them back to the control model. In addition, the data will be written back to the board for display if the DAC is working.

To study the correctness and performance of the whole simulation, different tests were conducted to evaluate the software and hardware components, as well as the program as a whole. A test is presented below.

At the initialisation phase, the CLOCK model generates a control signal to start the CONTROL model. The CONTROL model then invokes the TCL model to start an IDE application and to load the TCL script. After pressing the reset button on the DSP board, the IDE debugger is opened. It automatically loads the talk-through program into the DSP board. At the same time, the DATATRANSFER model is in sleep state and waits for the data to be generated by the CODEC on the DSP board. Upon reception of the converted digital data, the DATATRANSFER model takes the data and sends them back to the CONTROL model. Finally, through the DISPLAY model, these samples will be updated and stored in a bin. After one simulation cycle, the clock will generate the next command. The simulation can be terminated by a preset simulation time or manually.

The following figure shows the output file obtained after one simulation cycle:

According to the above execution results in Table 10, all the signal conversions are successful and the values are obtained properly. If we compare these results with those corresponding to Test B-1, we can see that they match. Differences in the value column are a result of the representation format and the analogue waveform used. In this final stage, D/A functionality of the CODEC is added into the simulation cycle. The purpose of this test is to verify the correctness of the digital samples obtained from the previous testing case. Ideally, when these digital samples are converted back into the analogue form, the oscilloscope should display exactly the same waveform used for the previous testing.

Table 10 Output file excerpt for an experiment using Test B-2 experimental frame

Output time	*Port*	*Value*
00:00:40:708	Signal	0.50391
00:00:41:008	Signal	–1.00000
00:00:41:308	Signal	0.97546
00:00:41:608	Ground	0.03967
00:00:41:900	Signal	–1.00000
00:00:42:200	Ground	0.03857
00:00:42:500	Signal	–1.00000
(…)	(…)	(…)

5.3 Case study 3: a robotic arm controller

The Lego Mindstorms NXT is a programmable robotics kit. The main component of the kit is a computer, called the NXT, which can take inputs from up to four sensors and control up to three motors via RJ12 cables. The kit includes three identical servomotors that have built-in reduction gear assemblies and can sense their rotations within one degree of accuracy. The kit also includes four sensors, each with a different capability. The touch sensor, the light sensor, the sound sensor, the ultrasonic sensor can measure distances and detect movement. Using NXT++ (a C++ open source library that provides functions to control the sensors and motors via an USB or Bluetooth connection from a computer), users can program different applications.

Using the toolkit, we built a robotic arm that can lift, pivot, and grab objects with its claws. The robot uses sound, touch and colour sensors, and two motors: one for moving the arm up and down and one for the claw to grab and release. In this case, we built a DEVS model to control the behaviour of the robot (using ECD++ on a Linux system), and we installed the code needed in the NXT microcontroller. After compiling the model ECD++ provides an interface to open a Telnet connection to the target board and then the executable simulation file, the model file and the event files were copied to the target board. The connection between the robot arm and the board was via USB port.

Figure 13 Robotic arm architecture (includes hardware-in-the-loop) (see online version for colours)

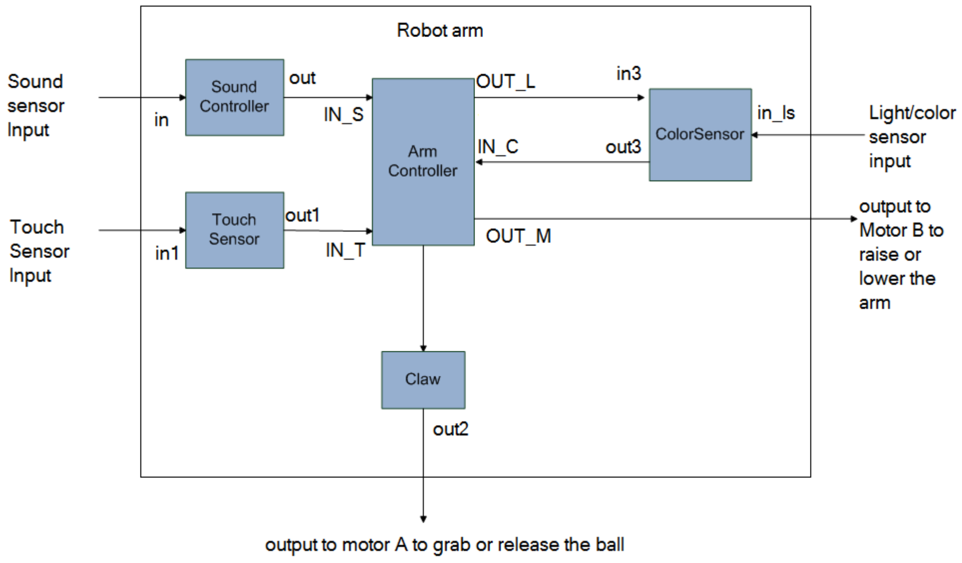
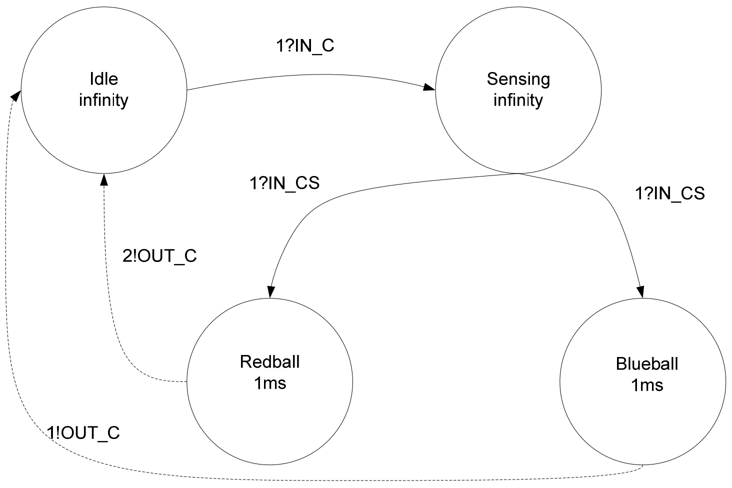


Figure 14 ColorSensor DEVS graph



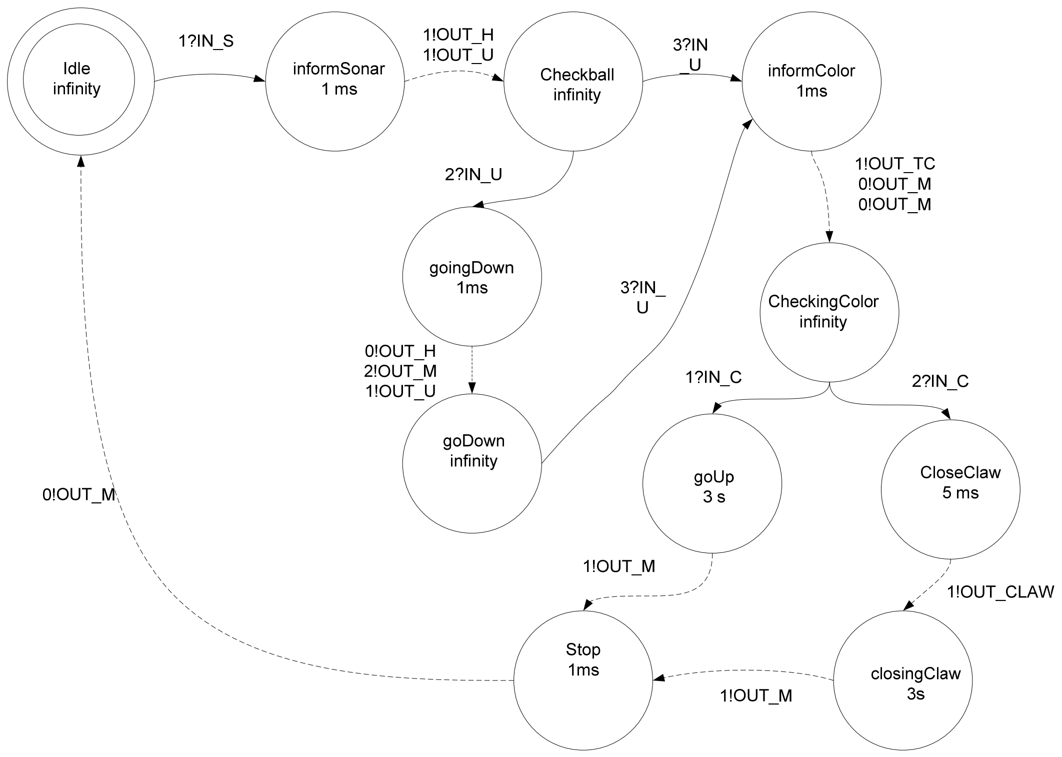
The arm controller waits for a sound input from the user. On getting a sound input, the arm moves down for a pre-specified amount of time or until the touch sensor hits a ball. Figure 13 shows the design hierarchy of atomic models in the top coupled model for RoboArm model. The following figure shows a model of the robotic arm, a coupled model composed of five atomic models: *ArmController*, *Sound Controller*, *UltrasonicSensorC*, *ColorSensor* and *Claw*.

There is a top coupled model that contains five atomic models. Sound, touch and colour sensor atomic models are responsible for sound, touch and colour sensor controlling. These models receive the input of the sensors and forward it to the arm controller model. The arm controller model is responsible for controlling the arm. It receives inputs from sensors models and sends outputs to colour sensor model, claw model and arm motor. The claw model is only responsible for the claw motor. The sound controller waits until it receives an input from a sound sensor, and then triggers the arm controller. The controller uses the information from all the sensors, and decides how to react according to the values received. The ultrasonic sensor model is responsible for determining when the robot has found a ball. When a ball is found, a colour sensor is activated. The *ArmController* uses this value, and tells the claw to grab the ball or not based on the colour.

When the touch sensor detects a ball, the arm stops. Then the light sensor allows deciding what to do depending on the colour of the ball. The sound and touch sensor only receive an input when there is an input available for them, but the colour sensor receives input in a periodic manner, during a certain time interval which the arm controller asks it to detect the colour.

The idea is to simulate the model first, and then to port the simulated software to control the actual robotic arm. Figure 14 describes the behaviour of the colour sensor controller described above, defined as an atomic model using a DEVS graph.

Figure 15 ArmController DEVS graph



The external transitions receive inputs and initiates appropriate state transitions. Once the right state is chosen (*redball, blueball*), the value is transmitted through the corresponding output port. Then, the model waits for the next input. The sensor takes 1 ms to detect the colour (simulated or RT), after which the output and internal transition are triggered. The other models are similar to this one. For instance, the *UltraSonicSensorC* model waits for inputs, and depending on the distance, it sends an output to start the motor. The Claw atomic model receives different inputs and according to their values, it gives the order to grab the ball, release it, or finish. The *ArmController* model is described in Figure 15.

The *ArmController* waits for inputs and when received, the arm changes to the *informsonar* state for 1 ms. After this time, an output is sent to ultrasonic sensor requesting horizontal movement. The controller then changes to state *checkball* and waits for an input (the idea is to connect this model to the ultrasonic sensor, but the input could come from any source of inputs). When a new input is received, the controller goes into the *informColor* or *goingDown* states (depending on the input received). The colour must be checked, or the horizontal and vertical motors must be moved. When in the *informColor* state, an output is sent to request the colour values. The *goUp* state is used when a blue ball is detected (making the arm to move up). Otherwise, the order to close the claw is sent.

As explained above, the idea is to simulate the model first, and then to port the simulated software to control the actual robotic arm. For instance, when we want to test the ultrasonic sensor model, we obtain the following simulation results:

```
INPUTS            OUTPUTS
05:00 IN_U1 1     06:001 out_u 1
06:00 IN_U 19     09:001 out_u 2
07:00 IN_U 2      11:001 out_u 3
08:00 IN_U1 1
09:00 IN_U 9
10:00 IN_U1 1
11:00 IN_U 2
```

As we can see, we can test the atomic model using various inputs and obtain the desired outputs within tight deadlines, but in simulation time. After, each model was ported to execute in E-CD++. The models are not modified, and they still control the system when running in RT. Following, we show the inputs and outputs of the ultrasonic sensor atomic model executing in RT using E-CD++ in RT mode.

```
INPUTS
05:00 06:00 IN_U1 OUT_U 1
06:00 07:00 IN_U OUT_U 19
07:00 08:00 IN_U OUT_U 2
08:00 09:00 IN_U1 OUT_U 1
09:00 10:00 IN_U OUT_U 9
10:00 11:00 IN_U1 OUT_U 1
11:00 12:00 IN_U OUT_U 2


OUTPUTS
Cur Time: 06:001 Deadline: 06:000
(NOT succeeded) OutPort: out_u PortValue:
```

```
1


Cur Time: 09:001 Deadline: 07:000
(NOT succeeded) OutPort: out_u PortValue:
2


Cur Time: 11:001 Deadline: 08:000
(NOT succeeded) OutPort: out_u PortValue:
3
```

As we can see, the tool helps the designer in finding problems with the RT constraints, and fixing the model accordingly. The following test shows the overall results for the robot arm controller, and Figure 16 shows a picture of the actual robotic controller in action.

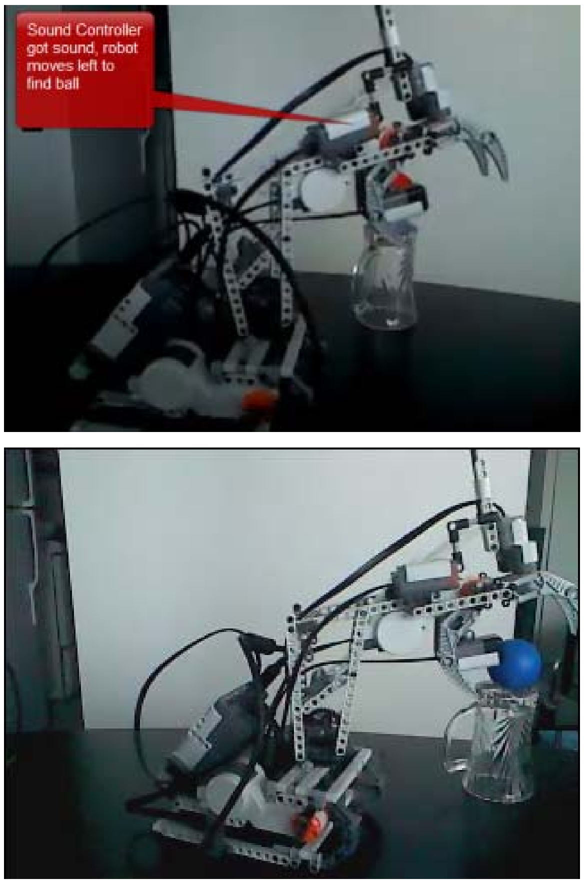
```
INPUTS
05:00 06:00 IN_S OUT_M 30
07:00 06:00 IN_U OUT_M 4
09:00 06:00 IN_U OUT_M 2
10:00 06:00 IN_CS OUT_M 9

OUTPUTS
Cur Time: 05:002 (no dealine specified)
OutPort: out_h PortValue: 1
Cur Time: 07:002 Deadline: 06:000(NOT
SUCCEEDED) OutPort: out_m PortValue: 0
Cur Time: 07:002 (no dealine specified)
OutPort: out_h PortValue: 0
Cur Time: 11:025 (no dealine specified)
OutPort: out_claw PortValue: 1
Cur Time: 12:025 Deadline: 08:000 (NOT
succeeded) OutPort: out_m PortValue: 1
Cur Time: 17:025 Deadline: 10:000 (NOT
succeeded) OutPort: out_m PortValue: 0
```

5.4 Discussion

Testing the performance of a simulator is usually a very complicated task. We have developed a synthetic model generator to facilitate the testing phase. To emulate several degrees of complexity in their structures, different types of models have been defined. In addition, it is possible to determine a given workload to be executed in the atomic transition functions. A thorough testing has been carried out on different simulation techniques provided in the CD++ toolkit (Moallemi and Wainer, 2010; Rafsanjani-Sadeghi et al., 2010; Shang and Wainer, 2008). The performance of each simulator has been characterised. The overhead incurred by the different simulators is bounded and the performance is appropriate in most cases. The obtained results have shown the possibility of developing a RT extension to the toolkit.

The benchmark experiments have shown good results on RT executions. We have studied the percentage of success and worst-case response times under different scenarios. Several properties of the model and its environment have been analysed. Some weaknesses have been pointed out in the analysis of the tool, specifically on the execution of extremely large models. The message-passing process may affect the execution performance, mainly if the model structure is too large or complex. Even though the performance degradation was small, it was desirable to provide more efficiency not only in RT but also in virtual time simulations. Thus, a new flattened simulator has been presented to overcome the described problems.

Figure 16 Robotic arm with red and blue ball (see online version for colours)



Source: http://youtube.com/arslab

The flattened simulator transforms the hierarchical structure of a model to a flattened structure in order to reduce the overhead incurred by the message passing among simulators and coordinators. The resulting non-hierarchical structure is simpler and more effective. The non-hierarchical approach can be applied not only for DEVS but also for Cell-DEVS simulations.

A thorough testing has been performed to the flattened simulator. In most cases, the flattened technique outperforms the hierarchical technique. We have conducted a thorough testing of the new flattened simulator, comparing the results with those obtained using the hierarchical simulator. Both synthetically generated models and existing models from the CD++ library were executed. The experiments included virtual time and RT model execution.

When the virtual time approach is used, in most cases the flattened simulator is more efficient and reduces the simulation time. On the other hand, when the RT approach is used, the flattened simulator provides better response times and greater percentages of success.

Not only DEVS but also cell-DEVS models have been executed employing the new simulation technique. When the flattened simulator is used, the processor structure is more simple and, usually, more effective.

The use of the non-hierarchical simulator reduces the number of messages exchanged in the simulation process. This reduction of overhead leads to better performance results. In general, we have shown that the new flattened simulator outperforms the hierarchical one.

6 Conclusions

We have provided a means to execute models in RT using the CD++ toolkit. When the new RT extension is employed, events must be handled timely and time constraints can be stated and validated accordingly. The RT simulator ties the advance of the simulation time to the wall-clock time. Consequently, these new features allow interaction between the simulator and the surrounding environment. The new RT simulator has been tested and analysed.

The RT performance of the CD++ toolkit was analysed. The benchmark experiments showed good results on RT executions. We studied the success rate and worst-case response times under different scenarios. Several properties of models and their environment were analysed. Some weaknesses were pointed out in the analysis of the tool, specifically about the execution of extremely large models. The message-passing process may influence the execution performance, mainly if the model structure is too large or complex. The flat simulator transforms the hierarchical structure of a model into a flat structure to reduce the overhead incurred by message passing among simulators and coordinators. The resulting non-hierarchical structure is simpler, and its performance was analysed in this work. In general, we showed that the flat simulator outperforms the hierarchical, alleviating the overhead incurred by message passing when large or complex models are executed.

Finally, we presented some examples to illustrate how to execute DEVS models in RT CD++. In the last example, we showed a way to execute HILS of discrete event models using RT CD++. Firstly, a DEVS model was built to simulate the behaviour of a CODEC together with a test program using that device. Secondly, the actual CODEC was deployed as a hardware prototype to replace the CD++ model, integrating the prototype into the original DEVS component. Hence, we demonstrated the use of RT CD++ to analyse models in a simulated environment and to execute these models in a hardware surrogate.

Acknowledgements

This work was partially funded by NSERC. We want to acknowledge Lidan Li, Mohammad Moallemi, Ezequiel Glinsky, Faezeh Rafsanjani and Christian Jacques who developed some of the software and models presented here.

References

Al-Zoubi, K. and Wainer, G. (2013) 'RISE: a general simulation interoperability middleware container', *Journal of Parallel and Distributed Computing*, Vol. 73, No. 5, pp.580–594, Elsevier.

Cho, K., Zeigler, B.P., Cho, H.J. et al. (2000) 'Design considerations for distributed real-time DEVS', *AIS 2000*.

Cho, S.M. and Kim, T.G. (1998) 'Real-time DEVS simulation: concurrent time-selective execution of combined RT-DEVS and interactive environment', *1998 Summer Computer Simulation Conference*, Reno, Nevada, USA, pp.410–415.

Furfaro, A. and Nigro, L. (2009) 'A development methodology for embedded systems based on RT-DEVS', *Innovations in Systems and Software Engineering*, June, Vol. 5, No. 2, pp.117–127.

Glinsky, E. and Wainer, G. (2002) 'Performance analysis of real-time DEVS models', *Proceedings of SCS Winter Computer Simulation Conference*, San Diego, CA.

Hong, J.S., Song, H.H., Kim, T.G. and Park, K.H. (1997) *A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development*, Springer, Netherlands.

Hu, X. and Zeigler, B.P. (2005) 'Model continuity in the design of dynamic distributed real-time systems', *IEEE Transactions on Systems, Man and Cybernetics, Part A*, Vol. 35, No. 6, pp.867–878.

Kim, K., Kang, W., Sagong, B. and Seo, H. (2000) 'Efficient distributed simulation of hierarchical DEVS models: transforming model structure into a non-hierarchical one', *Proceedings of the 33rd Annual Simulation Symposium*, Washington DC, USA.

Li, L., Pearce, T. and Wainer, G. (2003) 'Interfacing real-time DEVS models with a DSP platform', in *Proceedings of the EuroSim Industrial Simulation Symposium 2003*, Valencia, Spain.

Liu, J. (2000) *Real-time Systems*, Prentice Hall, Upper Saddle River, NJ, USA.

Liu, Q. and Wainer, G. (2007) 'Parallel environment for DEVS and cell-DEVS models', *Simulation, Transactions of the SCS*, Vol. 83, No. 6, pp.449–471.

Moallemi, M. and Wainer, G. (2010) 'Designing an interface for real-time and embedded DEVS', *Proceedings of 2010 Spring Simulation Conference (SpringSim10), DEVS Symposium*, Ottawa, Canada.

Moallemi, M. and Wainer, G. (2013) 'Modeling and simulation-driven development of embedded real-time systems', *Simulation Modelling Practice and Theory*, Vol. 38, No. 11, pp.115–131.

Rafsanjani-Sadeghi, F., Moallemi, M. and Wainer, G. (2010) 'Modeling and controlling a robotic arm with E-CD++', *Proceedings of the 2010 ACM/SCS Summer Computer Simulation Conference (Poster Session)*.

Saadawi, H. and Wainer, G. (2012) 'Hybrid DEVS models verification', *Proceedings of 2012 SCS/ACM/IEEE Symposium on Theory of Modeling and Simulation, TMS/DEVS'12*, Orlando, FL.

Saadawi, H. and Wainer, G. (2013a) 'On the verification of hybrid DEVS models', *Proceedings of 2013 SCS/ACM/IEEE Symposium on Theory of Modeling and Simulation, TMS/DEVS'13*, San Diego, CA.

Saadawi, H. and Wainer, G. (2013b) 'Principles of DEVS models verification', *SIMULATION: Transactions of the Society for Modeling and Simulation International*, January, Vol. 89, No. 1, pp.41–67.

Saadawi, H., Wainer, G. and Moallemi, M. (2012) 'Principles of DEVS models verification for real-time embedded applications', in Popovici, K. and Mosterman, P. (Eds.): *Real-time Simulation Technologies: Principles, Methodologies, and Applications*, CRC Press, Taylor and Francis.

Shang, H. and Wainer, G.A. (2008) 'Dynamic structure DEVS: improving the real-time embedded systems simulation and design', *41st Annual Simulation Symposium*, Ottawa, Canada.

Song, H.S. and Kim, T.G. (2005) 'Application of real-time DEVS to analysis of safety-critical embedded control systems: railroad crossing control example', *SIMULATION*, February, Vol. 81, No. 2, pp.119–136.

Wainer, G. (2002) 'CD++: a toolkit to develop DEVS models', *Software – Practice and Experience*, Vol. 32, No. 3, pp.1261–1306.

Wainer, G. (2009) *Discrete-Event Modeling and Simulation: A Practitioner's Approach*, CRC Press, Boca Raton, FL.

Wainer, G., Glinsky, E. and Gutiérrez-Alcaraz, M. (2011) 'Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark', *SIMULATION: Transactions of the Society for Modeling and Simulation International*, July, Vol. 87, No. 7, pp.555–580.

Weicker, R.P. (1984) 'Dhrystone: a synthetic systems programming benchmark', *Communications of the ACM*, Vol. 27, No. 10, pp.1013–1030.

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

Zeigler, B.P. and Chi, S. (1992) 'Symbolic discrete event system specification', *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 22, No. 6, pp.1428–1443.