M/CD++: modeling continuous systems using Modelica and DEVS

Mariana C. D'Abreu

Computer Science Department
Universidad de Buenos Aires
Pabellón I, Ciudad Universitaria
(1428) Buenos Aires, ARGENTINA.
mdabreu@ciudad.com.ar

Gabriel A. Wainer

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

Abstract

DEVS theory (originally defined for modeling and simulation of discrete event systems) was extended for modeling and simulation of continuous/hybrid systems. We present the M/CD++, a tool to build such models using a subset of Modelica, a modular and acausal specification language for physical systems. Models are created using Modelica standard notation, and a translator converts them into DEVS models. We show how to model these dynamic systems under the discrete event abstraction, including different execution results.

1. Introduction

Continuous Variable Dynamic Systems are represented by continuous variables on a continuous time basis. Analysis of these complex systems has usually been tackled using different mathematical formalisms, including Differential Algebraic Equations (DAEs), Ordinary Differential Equations (ODEs), or Partial Differential Equations (PDEs). Most existing simulation tools implement numerical methods based on the discretization of time to find approximate solutions to these equations, which are based on discretization of time. In the last few years, different approaches developed tried to simulate continuous systems under the discrete event paradigm. This presents some advantages over discrete time simulation, including reduction of the number of calculations for a given accuracy [1] and seamless integration of complex systems composed by both continuous time and discrete event paradigms. These solutions are based on the DEVS (Discrete Event Specification) formalism [2], originally created for specifying discrete event models. The idea of this method, called *Quantized Systems* theory (Q-DEVS), is to provide quantization of the state variables obtaining a discrete event approximation of the continuous system [3]. The state variables of the system are converted into a piecewise constant function via a quantization function [3]. The *Quantized State System (QSS)* method [4] is an extension to Q-DEVS in which the trajectory of each state variable is converted into a piecewise constant function via a quantization function equipped with hysteresis.

We will present a toolkit to simulate physical systems based on these methods. The tool, called *M/CD++*, is based on *Modelica* [5] and *CD++* [6], a modeling and simulation (M&S) tool implementing DEVS theory. Modelica is an object-oriented language created for modeling physical systems, which support library development and model exchange. Models in Modelica are described by differential, algebraic and discrete equations. Modelica provides libraries of standard components in different ODEs, block diagrams, and electrical and mechanical models. M/CD++ allows the creation of dynamic systems belonging to the electrical domain. Internally, the models are represented using the Bond Graph (BG) formalism [7], which permit to reuse models created by the toolkit to support simulation of physical systems in different domains. These models are used to check for algebraic loops and singularities, and to generate an optimized BG corresponding to the electrical circuit, which, in turn, is used to generate a DEVS model specification according to the rules used by CD++.

2. Background

Continuous systems are usually described using DAEs, ODEs and PDEs, and simulators based on these formalisms usually solve numerically the set of differential equations. Recently, some authors have tried to model these systems via system decomposition, dividing the system into a number of smaller subsystems, using Object-oriented modeling to promote model specification in a more natural way [7][8]. Modelica [5] is one of such languages, intended for modeling within many application domains such as electrical circuits, hydraulics, mechanics, etc. Modelica is built on non-causal modeling with mathematical equations and object-oriented constructs, allowing library development and model exchange. The semantic of these models is specified by a set of rules used to translate the model to its corresponding flat hybrid DAE. A model is represented using classes that can be developed hierarchically, allowing components reuse. An example of an electrical circuit specified using the electrical library provided by Modelica is presented below:

```
encapsulated model ChuaCircuit
       "Chua's circuit, ns, V, A"
  import Modelica.Electrical.Analog.Basic;
  import Modelica.Electrical.Analog.
                     Examples.Utilities;
  Basic.Inductor L(L=18);
  Basic.Resistor Ro(R=12.5e-3);
  Basic.Conductor G(G=0.565);
  Basic.Capacitor C1(C=10, v(start=4));
  Basic.Capacitor C2(C=100);
  Utilities.NonlinearResistor   Nr(Ga(min=-
1)            =-0.757576,Gb(min=-1)=-
0.409091,Ve=1);
  Basic.Ground Gnd;

 equation
  connect(L.p, G.p);
  connect(G.n, Nr.p);
  connect(Nr.n, Gnd.p);
  connect(C1.p, G.n);
  connect(L.n, Ro.p);
  connect(G.p, C2.p);
  connect(C1.n, Gnd.p);
  connect(C2.n, Gnd.p);
  connect(Ro.n, Gnd.p);
 end ChuaCircuit;
```

Figure 1. Chua's circuit [9] in Modelica.

The tool we implemented is able to simulate electrical circuit models defined using a subset of the Modelica's language specification (a complete description of the grammar supported, check [10]). M/CD++ models are converted into Bond Graphs (BG), a modeling formalism that allows domain-independent description of the dynamic behavior of the physical systems we model. Systems can be described in a hierarchical way, using BG submodels connected via ports. BG represent continuous systems as a set of elements that can interact with each other by exchanging energy and information, and this exchange determines the dynamics of the system. Power (energy time derivative) is the product of effort and flow. In electrical systems, power is the product of voltage and current; in hydraulics, it is the product of pressure and volume flow rate, etc. A generalized effort variable and a generalized flow variable can be defined whose product gives the power exchanged by the elements of a system. In BG, physical processes are represented as vertices in a directed graph, and the edges represent the ideal exchange of energy between the vertices [8]. The energy flow is represented via bonds with direction and the elements exchange effort and flow through them. The exchange of power is assumed to occur through abstract entities called *energy ports*. A fundamental concept to understand how information flows between components is causality: no component can determine the two power variables, effort and flow, at the same time. Given a pair of elements connected through a bond, causality determines which component causes the flow information and which causes the effort.

In M/CD++, we generate intermediate BG that are subsequently translated into DEVS, a formalism for M&S of discrete-event dynamic systems. DEVS can be seen as a mechanism to specify systems whose inputs, states and outputs are piecewise constant, and whose transitions are identified as discrete events [1]. DEVS models can be described using **atomic** and **coupled** components. Atomic models are independent modular objects that specify behavior; these can be composed in order to form coupled components. A **coupled** model is composed by several atomic or coupled submodels.

DEVS has been used recently for continuous systems simulation by different research teams [1][3][4][11][12][13][14][15][16]. In these articles, it has been shown that discrete event methods in general and DEVS in particular, present several advantages:

- Computational time reduction: for a given accuracy, the number of calculations can decrease

- Hierarchical modular modeling

- Seamless integration with models defined with other modeling techniques mapped to DEVS

- Simulation of discrete time models: can be seen as particular cases of discrete event methods

- Hybrid systems modeling: the discrete event paradigm provides the theory to develop a uniform approach to model and simulate systems with continuous and discrete components.

Most of these techniques are based on Q-DEVS [3], whose main idea is to represent continuous signals by the crossing of an equal spaced set of boundaries. This approach requires a fundamental shift in thinking about the system: instead of determining what value a dependant variable will have at a given time, we must determine at what time a dependant variable will enter a given state. *QSS* (*Quantized State Systems*) [4][11] is an extension to Q-DEVS, which allows continuous systems simulation based on quantization and hysteresis. In [4] it was proved that differential equation systems can be approximated by legitimate DEVS models with QSS, and the existence of a minimum time interval between events constitutes a sufficient condition to obtain legitimate models [4].

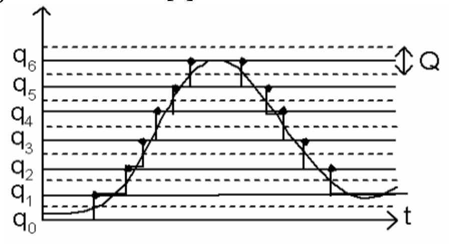


Figure 2. Signal Quantization.

CD++ is a toolkit that implements DEVS theories [6]. Atomic models are implemented using a built-in specification language or C++. New atomic models extend the behavior of the basic atomic model and they must inherit from the *Atomic* class, provided by the tool. Coupled models are described in a configuration file using a specification language provided by the tool.

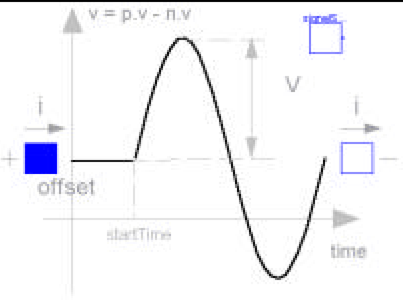
3. M/CD++

M/CD++ allows simulating dynamic systems in the electrical domain using CD++. The electrical circuits can be modeled using an Object-Oriented methodology, and then simulated through a discrete event simulator. Three main features were added to extend CD++:

Modelica v2.1 language support for electrical circuits construction: a subset of *Modelica v2.1* [5] was implemented, and the components needed to allow electrical circuits construction included the basic components provided by the Modelica Electrical library.

Electrical circuit modeling and compilation utilities: several techniques were developed to translate the object-oriented representation of the electrical circuit into a valid DEVS model, capable to be simulated with CD++. Electrical circuit simulation capabilities based on the implementation of QSS concepts: the resulting CD++ model represents the equations system associated to the electrical circuit that has to be solved. Based on QSS and QBG theory, atomic models were constructed and added to CD++, approximating the solution using a discrete event approach.

The objects defined in the Modelica Electrical library supported on M/CD++ are: ***Modelica.Electrical.Analog.Basic:*** *Ground, .Resistor, .Conductor, .Capacitor, .Inductor;* ***Modelica. Electrical.Analog.Ideal:*** *.IdealTransformer, .IdealGyrator;* ***Modelica.Electrical.Analog.Sources:*** *ConstantVoltage, .StepVoltage, .SineVoltage, .PulseVoltage, .ConstantCurrent, .StepCurrent, .SineCurrent and .PulseCurrent.* These components are described according Modelica's specifications [5]. The following figure shows one of these components (further details can be found in [10]).



```
model SineVoltage "Sine voltage source"
parameter SI.Voltage V=1 "Amplitude";
parameter SI.Angle phase=0 "Phase ";
parameter SI.Frequency freqHz=1"Frequency
";
extends                              Inter-
faces.VoltageSource(redeclare
    Modelica.Blocks.Sources.Sine
     signalSource(amplitude={V},
       freqHz={freqHz}, phase={phase}));
end SineVoltage;
```

Figure 3. Definitions: Modelica.Electrical. Analog.Sources.SineVoltage

The sine voltage defines the amplitude, phase and frequency (in Hz) of the sine wave (defaults: 1, 0, 1), and generates a voltage/time. M/CD++ is defined by several core components related to file parsing, model generation, compilation and CD++ simulator invocation. The compiling process starts with an electrical circuit model specified using Modelica, and finishes with a log file including the simulation results [10]. The following sections describe each step.

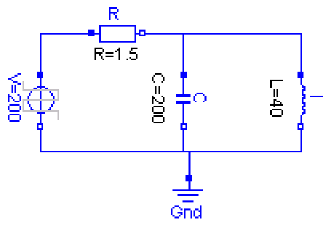
4. Modelica Parser & Checker

This component checks and parses the input file, building and validating the electrical circuit model. We built a parser using a general-purpose parser generator that takes an LALR context-free grammar, and describe the actions that accompany the syntactic rules. These actions were implemented to build the input file syntax tree; which is in turn used to perform semantic validation and electrical circuit construction, checking:

- *Specification of valid and supported packages*
- *Specification of valid and supported types/classes*
- *Undeclared symbols checking*
- *Specification of valid component attribute*

Only if the complete syntax tree is successfully validated the electrical circuit model is built. Several verifications preserve the model properties; these restrictions are checked during the electrical circuit building phase accomplished by the parser, e.g.:

- *Valid specification of pin references*
- *connections between existing elements*
- *No connections from a component to itself*
- *specification of at least one source component*

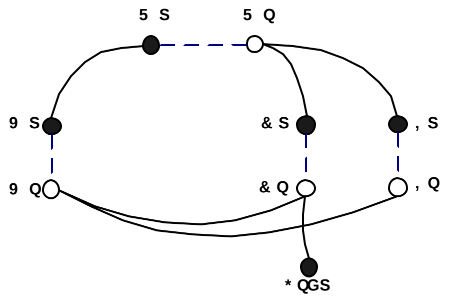


(a)

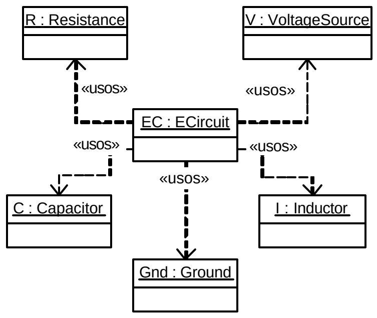
```
model          circuit                Mode-
lica.Electrical.Analog.
   Sources.PulseVoltage    V(V=200,    pe-
riod=1,
         width=10);
Mode-
lica.Electrical.Analog.Basic.Capacitor
      C(C=200);
Mode-
lica.Electrical.Analog.Basic.Resistor
      R(R=1.5);
Mode-
lica.Electrical.Analog.Basic.Inductor
      I(L=40);
Modelica.Electrical.Analog.Basic.Ground
       Gnd;

equation
   connect(V.p, R.p);
   connect(R.n, I.p);
   connect(R.n, C.p);
   connect(I.n, V.n);
   connect(C.n, V.n);
   connect(C.n, Gnd.p);
end circuit;
```

(b)



(c)



(d)

Figure 4. (a) Electrical circuit model (b) Electrical circuit input file (b) Electrical circuit graph representation on M/CD++ (d) Objects model generated by the M/CD++ parser

A class hierarchy was implemented to model the electrical circuit objects, its components and attributes. The definitions of *pin* (*positive* and *negative*), *port*, *one-port element*, *two-port element*, *electrical component* (*resistance*, *capacitor*, *source*, etc) and *circuit* generate the model associated to these concepts [10][12].

Figure 4 shows the electrical circuit objects constructed by the parser given the corresponding Modelica specification file. The electrical circuit object, *EC*, is modeled as the composition of the following objects:

- *R*: instance of *Resistance* (resistor component; *one-port* element)
- *V*: instance of *VoltageSource* (voltage source component with signal *s*; *one-port* element.
- *C*: instance of *Capacitor* (capacitor component; *one-port* element).
- *I*: instance of *Inductor* (inductor component; *one-port* element).
- *Gnd*: instance of *Ground* (ground component; one positive pin).

The electrical circuit is modeled using an OO abstraction, which uses an internal representation based on a graph notation, as showed in Figure 4 (c). Every electrical component on the circuit is represented in the graph using *n* nodes, where *n* corresponds to the number of pins of the element. *One-port* elements are represented by two nodes on the graph, *element.port$_1$.p* (positive pin) and *element.port$_1$.n* (negative pin). Generalizing, *k-port* elements will be represented by *2.k* nodes as: *element.port$_1$.p*, *element.port$_1$.n*, *element.port$_k$.p* and *element.port$_k$.n*. There are two types of connections between nodes: physical and logical. The former type corresponds to the physical coupling between the elements of the circuit. Logical connections correspond to the associations between pins and ports of an element.

These implementation decisions were made having in mind the BG generation algorithm, developed for the mapping phase. The idea was using a suitable data structure and model representation to optimize the generation process of the BG associated to the electrical circuit, as discussed next.

5. Mapping electrical circuits to BG

Our BG generation algorithm is based on the Karnopp's circuit construction method [17]. The basic approach is to construct a BG resembling the circuit structurally, and to simplify the BG based on selected circuit properties. The construction method is:

-For each node in the circuit with a distinct potential write a 0-junction.

-Insert each one-port circuit element by adjoining it to a 1-junction and inserting the 1-junction between the appropriate 0-junctions (C, I, R, Se, Sf elements)

-Assign power directions to all bonds

-If the circuit has an explicit ground potential, delete those 0-junctions and their bonds from the graph. If no explicit ground potential is shown, choose any 0-junction and delete it.

-Simplify the resulting BG

After, a causalization process is applied, as described in [10]. After this process, the signal direction of the bonds is determined. Once this process is applied to the graph, each bond can be interpreted as a bidirectional signal flow. The causal BG can then be seen as a compact block diagram. Structural singularities and algebraic loops within a model are detected following the procedure defined in [10], but not corrected. In these cases, processing is aborted and an exception is raised. Only integral causalities are assigned to the storage components. In presence of structural singularity, i.e. coupled storages, one of the elements should be assigned the derivative causality, causing the toolkit to generate an exception. The error message informed will contain the name of the component causing the preferred causality violation. That can be used to detect *dependent storages*, which are storages that do not represent a state variable of the system. Figure 5 shows a graphical of an electrical circuit, and its transformation.

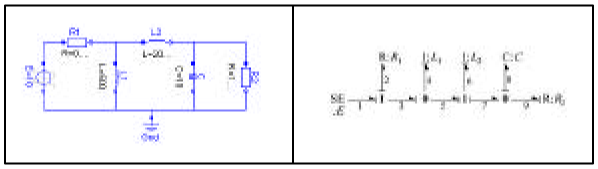


Figure 5. (a) circuit, (b) generated BG.

6. Quantized BG implementation on CD++

A Quantized BG (QBG) is a BG where all the storages and sources are quantized elements. Given that QSS modifies the original system, only the storage elements (capacitor and inertia) need to change in order to use the QSS method on QBG [4]. Several atomic DEVS models where developed on CD++, implementing the QSS and QBG concepts on the toolkit. Each component of the QBG was implemented as an atomic DEVS model, using BG and QSS definitions. For each model, we defined a DEVS formal specification, and built a model in CD++ based on such specification. Following, we present the formal definition of the *QBGCapacitorFlowIn* model (the remaining constructs were built using a similar approach, and details can be found in [10]). This is an atomic DEVS model corresponding to a QBG capacitor, which was developed following QSS definitions, implementing the capacitor as a quantized function applied to the displacement, *q(t)*, being the integrator's output.

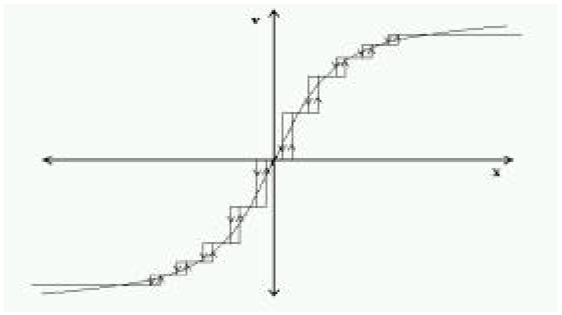


Figure 6. Constitutive relation of a storage element; quantization with hysteresis [K03]

The *QBGCapacitor* is defined as in [4]:

$$M = < X, S, Y, d_{int}, d_{ext}, l, ta >$$

$X = \hat{A}$; $Y = \hat{A} \times N$; $S = \hat{A}^2 \times Z \times \hat{A}^+$
$d_{int}(s) = d_{int}(x, d_x, j, s) = (x + s. d_x , d_x , j + sgn(d_x), s_1)$
$d_{ext}(s,e,x_u) = d_{ext}((x, d_x, j, s), e, x_v) =$
$(x+e.d_x, ,x_v , j, s_2)$
$l(s) = l(x, d_x, j, s) = (Q_{j+sgn(dx)},1)$
$ta(s) = ta(x, d_x, j, s) = s$

$$s_1 = \begin{cases} \dfrac{Q_{j+2} - (x + s.d_x)}{d_x} & if\ d_x > 0 \\ \dfrac{(x + s.d_x) - (Q_{j-1} - e)}{|d_x|} & if\ d_x < 0 \\ \infty & if\ d_x = 0 \end{cases}$$

$$s_2 = \begin{cases} \dfrac{Q_{i+1} - (x + e.d_x)}{x_v} & if\ x_v > 0 \\ \dfrac{(x + e.d_x) - (Q_j - e)}{|x_v|} & if\ x_v < 0 \\ \infty & if\ x_v = 0 \end{cases}$$

The simulation of this atomic DEVS model is based on the detection of effort value changes, regarding that the input flow and the output effort are piecewise constant, and the displacement trajectory is piecewise linear. This time is calculated dividing the distance of the displacement value to the next quantum crossing and the flow value. In case of input flow variations, the time to the next effort change must be recalculated. The current displacement is then computed according the elapsed time and the previous flow value and then used as the new initial value. Therefore, the new *ta* function value is calculated as it was already described. Input flow changes are associated to *external events*.

7. BG Compiler for CD++

Once the BG model has been generated and completely causalized, it compiled into CD++ notation. As the input to CD++ must be a valid DEVS model, the BG generated must be transformed into its corresponding DEVS representation. Thus, we built a BG Compiler for CD++. The first step done during the BG compilation is the transformation to its equivalent Quantized BG model explained in the previous section.

For each component **u** of the QBG, add **u** to the declaration section within the CD++ coupled model file. Select a valid implementation class for the component. For each bond **b = (u,v)** of the QBG, generate the coupling information between u and v on the links section within the CD++ coupled model, using DEVS formal coupling definitions. For each component **u** of the QBG, generate the component's configuration information within the parameterization section on the CD++ coupled model file.

```
components   :   $PJ2@QBGParallelJunction
$PJ3@QBGParallelJunction
C@QBGCapacitorFlowIn
$SJ2@QBGSerialJunction
$SJ3@QBGSerialJunction ...

link : e2n@$PJ2 e2p@$SJ2
link : f2p@$SJ2 f2n@$PJ2
...
link : e1n@V    e3p@$SJ3
link : e3n@$PJ2 e1p@I1
[C]  quantum : 0.0002   hystWindow : 0.01
C : 10.000  initialLoad : 0.000
[L1] quantum : 0.0002   hystWindow : 0.01
I : 500.000  initialLoad : 0.000
[L2] quantum : 0.0002   hystWindow : 0.01
I : 2000.000  initialLoad : 0.000
[R1]  R :      0.001
[R2]  R :   1000.000
[V]  quantum : 0.0002   hystWindow : 0.01
signal : Pulse  offset : 000  startTime :
000
amplitude : 010    period : 2.5  width :
050
```

Figure 7. Coupled DEVS model representation and CD++ notation

8. M/CD++ execution examples

In this section, we present the simulation results of a simple electrical circuit using the tools. Different studies were carried out, using different examples (details can be found in [10] and [18]). The goal of this section is to show the results obtained when compiling a M/CD++ model, and comparing it with the actual model behavior. We present the simulation results of the circuit introduced in Figure 5, whose Modelica definition can be seen in Figure 8.

```
model circuit
 Modelica.Electrical.Analog.Sources.
    PulseVoltage   V(V=10,width=50,   pe-
riod=2.5);
Modelica.Electrical.Analog.Basic.Resistor
    R1(R=0.001);
Modelica.Electrical.Analog.Basic.Inductor
    I1(L=500);
Modelica.Electrical.Analog.Basic.Inductor
    I2(L=2000);
Modelica.Electrical.Analog.Basic.Capacitor
    C(C=10);
Modelica.Electrical.Analog.Basic.Resistor
    R2(R=1000);
```

```
Modelica.Electrical.Analog.Basic.Ground
Gnd;

equation
  connect(V.p,  R1.p);  connect(R1.n,  I1.p);

 connect(R1.n, I2.p); connect(I2.n, C.p);
 connect(I2.n, R2.p); connect(C.n, I1.n);
 connect(R2.n, C.n); connect(I1.n, V.n);
 connect(V.n, Gnd.p);
end circuit;
```

Figure 8. Modelica specification of the circuit

The simulation of the example circuit was run for 1 minute of simulated time, using a quantum value equal to 0.0002 and an hysteresis window size of 0.01, applied to all of the quantizable components within the circuit (I_1, I_2, C_1).

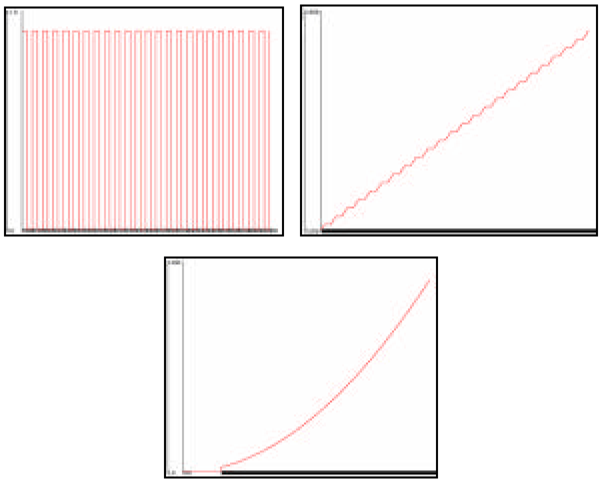


Figure 9. (a) Pulse Voltage Source (b) Current on inductor I_1 (c) Voltage on Capacitor C

Figure 9 shows the simulation results for the model executed. For the given pulse voltage source, we obtained the desired voltage on capacitor *C*, and the expected current on the inductor *Ii*. In order to validate our simulation results, we compared them with results obtained using *Dymola* [19], a commercial toolkit with full support for Modelica. The idea about using Dymola was comparing the results given by M/CD++ with those calculated by an advanced commercial tool with Modelica support. The test cases here presented were executed using both M/CD++ and Dymola simulators, varying simulation parameters in order to compare the results and calculate the error between them.

We simulated the previous example circuit using the DASSL integration method on Dymola during 10 seconds of simulation time (using intervals of 500 time units, and a tolerance of 0.0001). We compared the results with those obtained by M/CD++ using a quantum size q=0.0001 and a hysteresis window of a/2.

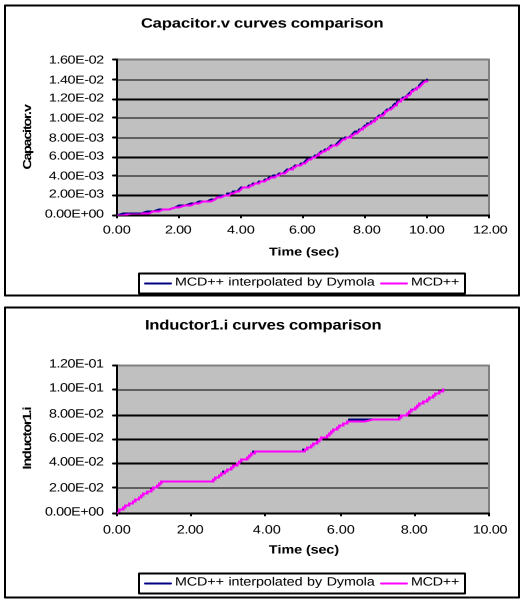
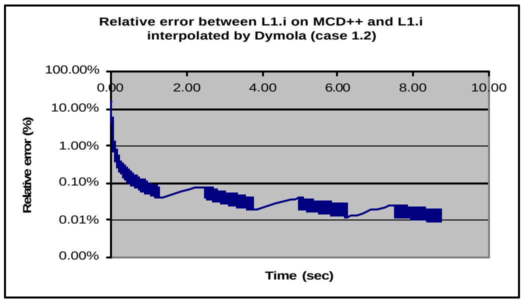


Figure 10. Comparison for voltage on Capacitor and current on Inductor1

The following figures show the error between the capacitor (*C*) and inductor (*I1*) state trajectories on M/CD++ and Dymola for the given simulation parameters. In Figure 10 we can see that the relative error is minimum (the highest error was obtained when values are close to zero, because, as the quantum size used during the simulation is fixed value for all the points on the curve, smaller values will produce greater relative errors). The error curve decreases when time advances. The results were even better when we decreased the quantum and hysteresis window size on M/CD++ (using q(C)=0.000005, and q (L1)=0.00005, and hysteresis = q/2).The following figures show the relative error between the capacitor (*C*) and inductor (*L1*) state trajectories on M/CD++ and Dymola for the given simulation parameters. This test case was simulated with the DASSL method on Dymola, as the previous case, but now decreasing the quantum and hysteresis window size used on M/CD++ simulation.



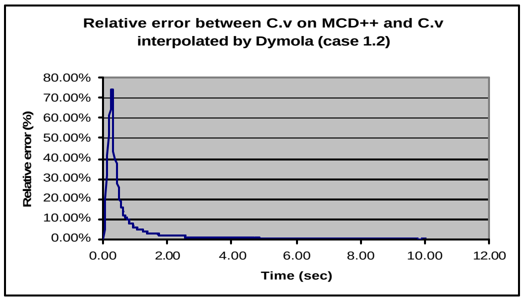
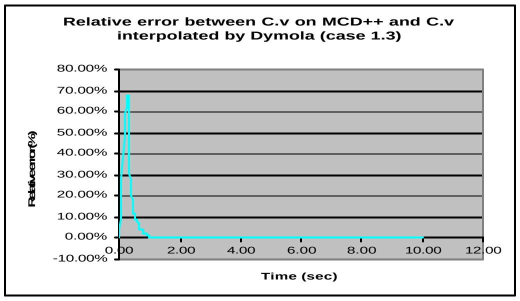


Figure 11. Relative error: (a) current on Inductor1 (b) voltage on Capacitor

The figure shows how the relative error is decreased for values near to zero. Finally, we repeated the tests using the same quantum size, but changing the integration method on Dymola (Euler, integration step 0.05). The following figures show the error between the capacitor (*C*) and inductor (*L1*) state trajectories on M/CD++ and Dymola for the given simulation parameters.



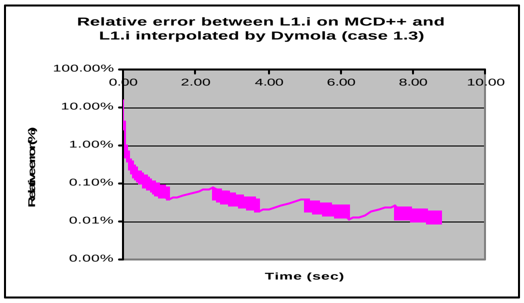


Figure 12. Relative error: (a) voltage on Capacitor (b) current on Inductor1

The first two test cases were used to compare the results given by M/CD++ with those generated by a higher order and variable step-size method, DASSL. Given that M/CD++ uses a first order method to integrate the state trajectories (QSS), a first order method was also used on Dymola simulation in order to make comparisons.

	Relative error average		
Curve	***Case 1***	***Case 2***	***Case 3***
C.v	2.75 %	0.87 %	0.42 %
L_1.i	0.28 %	0.11 %	0.11 %

9. Conclusion

DEVS is a formalism defined for M&S of discrete event systems, which has been recently used for simulation of continuous and hybrid systems. We introduced a tool for M&S of continuous systems, in which models are described using Modelica. The simulation results generated by MCD++ were compared with those produced by a complex physical system simulation environment with Modelica support called *Dymola*. It was shown that a higher relative error is obtained for values near to zero on a trajectory. This is related with the fixed quantum size used by the quantization function over a state trajectory. Then, for smaller values, greater differences are given. An approach to improve the simulation results could be developed using an adaptive quantization function, making the quantum vary according to the trajectory evolution. It is important to have in mind that MCD++ approximates the system solution based on the QSS method, which uses a simple first order integration approach. Most of the results produced by MCD++ were contrasted with results generated using a higher order and variable step-size integration method, DASSL. It was shown that, in general, choosing adequate quantization parameters produce accurate solutions and decrease error.

In the long term, we want to attack the development of hybrid systems based on the DEVS formalism and its extensions, building libraries to make easy to use components developed on top of DEVS modeling tools. One of the benefits is that for a given accuracy, the number of transitions can be reduced, decreasing the execution time of simulations. Discrete time models can be simulated under discrete event paradigm, thus allowing the development of a simulation environment for complex systems, modeled as hybrid systems, where all paradigms merge together (continuous time, discrete time, discrete event).

References

[1] Zeigler, B. "Continuity and Change (Activity) are Fundamentally Related in DEVS Simulation of Continuous Systems", LNCS, Vol. 3397, Springer-Verlag, 2005.

[2] Zeigler, B; Kim, T; Praehofer, H. "Theory of M&S". New York, 2000.

[3] Zeigler, B. DEVS. "Theory of Quantization". DARPA Contract N6133997K-007. University of Arizona, 1998.

[4] Kofman, E. "Discrete Event Based Simulation and Control of Continuous Systems". Ph.D. thesis. Universidad Nacional de Rosario, Argentina. August 2003.

[5] Modelica Language Specification, version 2.1, http://www.modelica.org. March 2004.

[6] Wainer, G. CD++: a toolkit to define discrete-event models. *Software, Practice and Experience*. Wiley. Vol. 32, No. 3. pp. 1261-1306. 2002.

[7] Åström, K. J; Elmqvist, H.; Mattsson, S. E. "Evolution of continuous-time M&S". European Simulation Multiconference, Manchester, UK, 1998.

[8] Cellier, F.E.; Elmqvist, H. "Automated formula manipulation supports object-oriented continuous-system modeling". IEEE Control Systems, 13(2), pp. 28-38, 1993.

[9] L.O. Chua. *"The Genesis of Chua's Circuit"*, AEU **46**, 250. 1992.

[10] D'Abreu, M. "Defining a compiler for discrete-event simulation of continous systems". M. Sc. Thesis. Computer Science Dept. Universidad de Buenos Aires, Argentina. 2004.

[11] Kofman, E.; Junco, S. "Quantized State Systems. A DEVS Approach for Continuous System simulation". *Transactions of the SCS*, 18(3), pp. 123-132, 2001.

[12] D'Abreu, M.; Wainer G. "Defining hybrid system models using DEVS quantization techniques". Winter Simulation Conference. New Orleans, LA. U.S.A., 2003.

[13] Wainer, G., B.P. Zeigler, "Experimental Results of Timed Cell-DEVS Quantization, AI and Simulation," AIS 2000, pp. 203-208, Tucson, AZ, March 2000.

[14] James J. Nutaro, Bernard P. Zeigler, R. Jammalamadaka, S. Akerkar. "Discrete Event Solution of Gas Dynamics within the DEVS Framework". International Conference on Computational Science, pp. 319-328, 2003.

[15] Jean-Sébastien Bolduc and Hans Vangheluwe. Mapping ODEs to DEVS: Adaptive Quantization. *Summer Computer Simulation Conference*, pp. 401-407. Montréal, Canada. 2003.

[16] Giambiasi, N.; Escude, B.; Ghosh, S. "GDEVS: A Generalized Discrete Event Specification for Accurate Modeling of Dynamic systems". Transactions of the SCS, 17(3) pp. 120-134, 2000.

[17] Karnopp, D.; Margolis, D.; Rosenber R. "System Dynamics: A Unified Approach". Wiley, 1990.

[18] L. Chechiu, G. Wainer. "Experimental results on the use of M/CD++". Proceedings of SummerSim. Philadelphia, PA. 2005.

[19] Dynasim Laboratories. "Dymola". [online]. Available online via: http://www.dynasim.com/dymola.htm [Accessed June 13 2004]