A BOND-GRAPH MAPPING MECHANISM FOR M/CD++

Mariana C. D'Abreu

Computer Science Department
Universidad de Buenos Aires
Pabellón I, Ciudad Universitaria
(1428) Buenos Aires, ARGENTINA

Gabriel A. Wainer

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

***Abstract**: DEVS theory (originally defined for modeling and simulation of discrete event systems) was extended in order to permit modeling simulation of continuous and hybrid systems. In this work, we present algorithms we presented to construct a compiler of a subset of Modelica, a modular and acausal standard specification language for physical systems modeling. Models are defined in Modelica are translated into Bond Graphs, which are used to analyse correctness of the specifications, prior their translation as DEVS models. We show how to map Modelica into BG, and the algorithms for detection error implemented.*

1. INTRODUCTION

Continuous Systems are those represented by continuous variables on a continuous time basis. Analysis of these complex systems (which include, for instance, mechanical, electrical, hydraulic, etc.) has usually been studied using Differential Equations [1]. A variety of numerical methods were created to find approximate solutions to these equations. Most of these methods (Euler, Runge-Kutta, Adams, etc.) are based on the discretization of time [2]. 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 [3] 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 [4], 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 [5].

Using these ideas, we developed *M/CD++* [6] a modeling and simulation tool for continuous and hybrid systems based on *Modelica* [7] and *CD++* [8], a modeling and simulation tool implementing DEVS theory. Modelica is an object-oriented language created for modeling physical systems, designed to support library development and model exchange. Models in Modelica are mathematically described by differential, algebraic and discrete equations. Modelica has many 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 [6]. The architecture of M/CD++ is defined by several core components related to file parsing, model generation, compilation and CD++ simulator invocation. The steps performed to simulate an electrical circuit model specified in the input file, can be summarised as in Figure 1. The compiling process starts with an electrical circuit model specified using Modelica, and finishes with a log file including the simulation results. As we can see in the figure, the models internally are represented using the Bond Graph (BG) formalism [9], which permit to reuse models created by the toolkit to support simulation of physical systems within different domains. The user must provide a source code file as input to the Modelica compiler. As a result, the model of the circuit is converted into a BG representation. These models are used to check for algebraic loops and singularities (elements that have discontinuities; e.g. diodes). Then, we 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++.

In this work we discuss the techniques employed to create BG for M/CD++, and the different algorithms employed by the compiler to check correctness of the physical system developed.

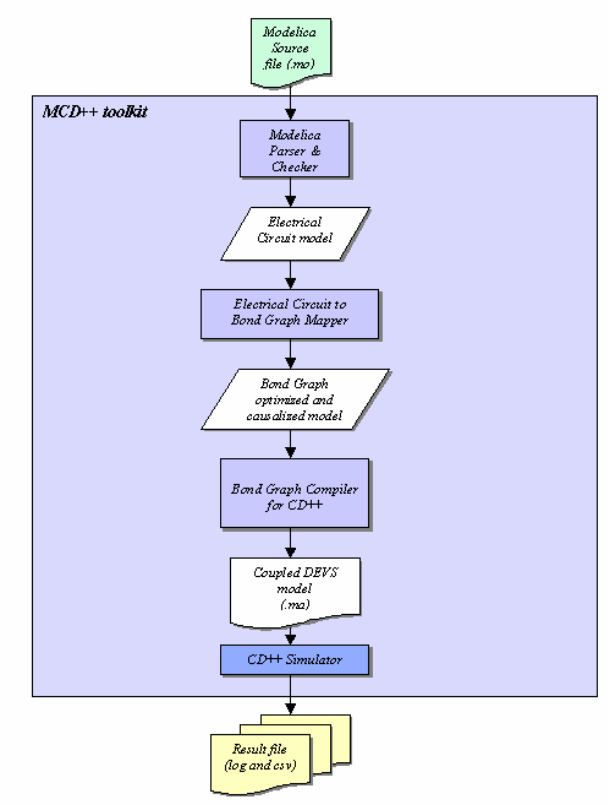


Figure 1. M/CD++ components interaction

2. BACKGROUND

Modelica presents a successful effort to model complex physical system via system decomposition (dividing the system into a number of smaller subsystems interfaced by distinct connections). Modelica uses object-oriented modeling to promote models specification in a more natural way, decreasing the abstraction gap existent between the real system and the representation model and permitting the development and reusability of models within a hierarchical construction process [9]. Numerous of these concepts were adopted and applied to the design of a new family of modeling and simulation tools for continuous systems modeling. Modelica [7], which was created for modeling within many application domains such as electrical circuits, hydraulics, chemical processes, thermodynamical systems, etc. It supports several formalisms, e.g. ODEs, DAEs, 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 description. A model is represented using classes that can be developed hierarchically, allowing components and knowledge reuse.

The tool we implemented is able to simulate electrical circuit models defined using a subset of the Modelica's language specification (for a complete description of the subset of the grammar of Modelica supported, check [6]). M/CD++ models are converted into BG, a modeling formalism that allows domain-independent description of the dynamic behavior of the physical systems we model. It can be used to specify systems within different domains, i.e. electrical, mechanical, hydraulic, thermodynamic, etc. Systems can be described in a hierarchical way, using BG submodels connected via ports through its interface. The election of the BG formalism to model electrical circuits was mainly based on:

- *BG can be applied to multiple physical domains, allowing code reuse (library implementation)*
- *BG allows modular and hierarchical models constructions*
- *BG models can be directly simulated in a simple way*
- *No algebraic manipulation is needed*
- *BG models can be easily translated to an equivalent block diagram*

In M/CD++, we generate intermediate BG that are subsequently translated into DEVS, a formalism for modeling and simulation 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 [4]. 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. Formally, atomic models are defined by:

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

A DEVS model state $s \in S$, is determined by the transition functions. In the absence of external events, the time for the next internal transition is determined by $ta(s)$, the time advance function applied to the current state s. The new state is given by $d_{int}(s)$, the internal transition function applied to s. Before the internal state change, the model can generate an output event, which is defined by the output function $l(s)$. In case that an external event occurs, the new state is determined by the external state transition function $d_{ext}(s, e, x)$, where s is the current state, e is the time elapsed since the last transition and $x \hat{I} X$, is the external event received. In DEVS, basic components can be coupled in order to develop complex models. A **coupled** model is composed by several atomic or coupled submodels. A coupled model is defined by:

$$CM = <X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select>$$

For each j in I_i, $Z_{i,j}$ is a function that translates the output from i to j. The connection between i and j is specified defining j as an *influencee* of i. When an internal event occurs on i, a signal is sent to component j at the same time. The mappings for these sets are also specified by function Z, which determines where the input and output events for the coupled model itself should be redirected to. In a coupled model, it might occur that two or more components had its internal transition scheduled at the same time, generating ambiguity during the simulation process. DEVS solves this problem with the *select* function, which defines the rules needed to determine which one of the imminent components will execute the next event.

Recently, DEVS has been used recently for continuous systems simulation. In most cases, the techniques are based on Q-DEVS [5], 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 as a whole. Instead of determining what value a dependant variable will have (its state) at a given time, we must determine at what time a dependant variable will enter a given state. *QSS* (*Quantized State Systems*) [10] is an extension to Q-DEVS, which allows continuous systems simulation based on quantization and hysteresis.

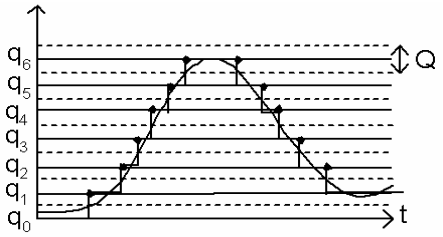


Figure 2. Signal Quantization.

CD++ is a toolkit that implements DEVS and Q-DEVS theories [8]. Atomic models are implemented using a built-in specification language or in C++, adding flexibility and construction power to the developer. 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. The file includes information about the components, the coupling and the input and output ports associated to the model. In [11], we showed how to create BG models using DEVS and the CD++ toolkit. This library was used as a base for M/CD++.

M/CD++ allows simulating dynamic systems in the electrical domain using CD++. M/CD++ includes Modelica v2.1 language support for electrical circuits construction [7]. The resulting CD++ model represents the equations system associated to the electrical circuit that has to be solved. Based on the QSS and QBG theory, atomic models were constructed and added to CD++, in order to numerically approximate the solution using a discrete event approach.

M/CD++ parses and checks input files, building and validating the electrical circuit model. The component takes as input the file with the electrical circuit specification under Modelica language. Once the input file is completely parsed and validated, the corresponding electrical circuit model is generated. Then, the model created is used as the input object for the next phase within the main simulation process. A hierarchy of classes 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* were implemented in order to generate the associated models. Several verifications were implemented to preserve the model inherent properties; these restrictions are checked during the electrical circuit building phase accomplished by the parser, e.g.:

- *Valid specification of pin references*
- *Definition of connections between existing elements*
- *None of the connections are from a component to itself*
- *The specification of at least one source component*

Figure 3 shows an example of the electrical circuit objects model constructed by the parser given the corresponding Modelica specification file:

```
model circuit
  Modelica.Electrical.Analog.Sources.PulseVoltage
                          V(V=200, period=1, width=10);
  Modelica.Electrical.Analog.Basic.Capacitor C(C=200);
  Modelica.Electrical.Analog.Basic.Resistor R(R=1.5);
  Modelica.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;
```

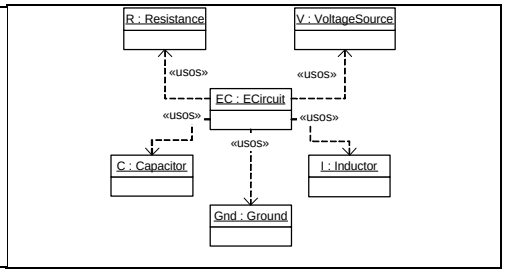


Figure 3. (a) Electrical circuit input file (b) Objects model generated by the MCD++ parser

The electrical circuit object, *EC*, is modeled as the composition of the following objects:

- *R* (instance of the *Resistance*). It models an electrical resistor component as a *one-port* element.
- *V* (instance of *VoltageSource*): it models a voltage source, with signal *s*, as a *one-port* element.
- *C* (instance of *Capacitor*): it models a capacitor component as a *one-port* element.
- *I* (instance of the *Inductor*): it models an inductor component as a *one-port* element.
- *Gnd* (instance of the *Ground*): it models a ground component as an electrical element with 1 positive pin.

The electrical circuit is modeled using an OO abstraction, which has an internal representation based on a graph notation, presented in the following example:

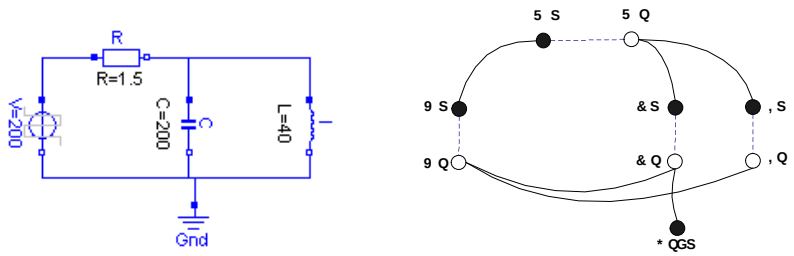


Figure 4 . (a) Electrical circuit model, (b) Electrical circuit graph representation on MCD++

Every electrical component on the circuit is represented in the graph using *n* nodes, where *n* corresponds to the number of pins that the element has defined. *One-port* elements are represented by two nodes, $element.port_1.p$ (positive pin) and $element.port_1.n$ (negative pin). *Two-port* elements are represented by four nodes, $element.port_1.p$, $element.port_1.n$, $element.port_2.p$ and $element.port_2.n$. 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 (solid lines). Logical connections correspond to the associations between pins and ports of an element; the pins of a given port connector are linked by dashed lines, port connectors of a given component are linked by dotted lines.

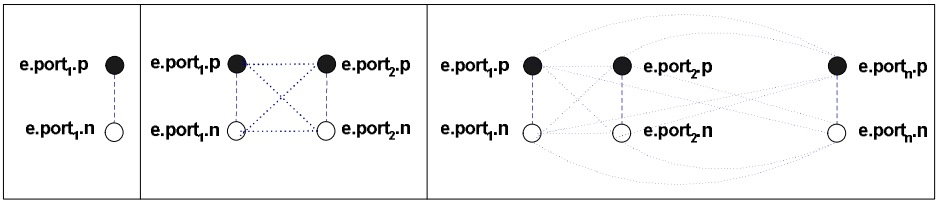


Figure 5 . Node representation of port elements (a) one-port (b) two-port (c) k-Port element

These implementation decisions were made having in mind the BG generation algorithm, developed for the mapping simulation 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.

3. BOND GRAPHS

BG allows graphical representation of any continuous dynamic system, which can be described in a hierarchical way. BG models are non-causal, improving the exchange and reuse of components. BG modelling concepts are based on two main assumptions for dynamic systems representation using network like descriptions: energy conservation law and the use of a lumped approach. These characteristics imply that dynamic system properties can be separated from each other, using submodels, and then connected using ideal connections. These connections represent the energy flow and the ideal property guaranties the power continuity quality. The last property indicates that no energy is generated or dissipated in the ideal connections.

In BGs, physical processes are represented as vertices in a directed graph and the edges represent the ideal exchange of energy between the vertices [12][13]. The energy (or its time derivative, the power), is the fundamental quantity exchanged between elements of the system. Power is the product of flow and effort (in translation mechanics, power is the product of force and velocity; in electrical systems, it is the product of voltage and current; in any system, a generalised flow and effort variable could be defined). 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 port; this way, the concepts of one-port and two-port elements are introduced. One-port elements are the

components that have associated one energy port, represented with a bond. Two-port elements are those that have two energy ports, represented with two bonds. Interactions between components are also restricted and the connectors implement constrained exchanges between elements.

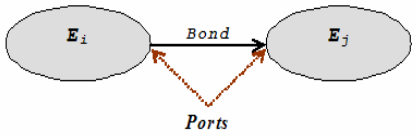
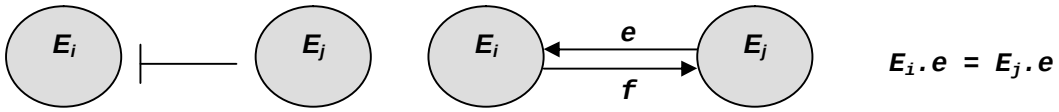


Figure 6. BG representation of energy flow from Ei to Ej.

As we represent the exchange of power between elements, 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 of the components causes the flow information and which causes the effort. Causal analysis is essential to describe a BG model in computational terms and to derive the set of differential equations.



(a) *(b)* *(c)*

Figure 7. Causality form (a) Causalized bond, (b) Equivalent graph, (c) Associated equations

The BG elements are the following:

- *Capacitor* (*C*)
- *Inductor* (*I*).
- *Resistor (R)*
- *Effort source* (Se)
- *Flow source* (*Sf*)
- *Transformer (TF)*
- *Gyrator (GY)*
- *1-junction*
- *0-juction*

Following, we show two examples of specification of these components: Resistors and Gyrators.

- **Resistor:** *R* elements dissipate energy. Examples of the resistor element are resistor, in electrical domain and dampers, in mechanical context. The constitutive equation is defined by an algebraic equation relating *flow* and *effort*: $e = r(f)$ .The electrical resistor is mostly linear and the corresponding equation is: $u = R.i$, where R is the resistance's constant.

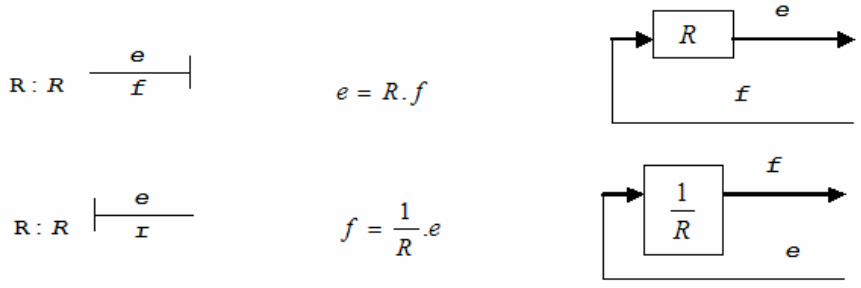


Figure 8 . (a) (b) (c) R element with flow in causality, equations and block diagram representation
(d) (e) (f) R element with effort in causality, equations and block diagram representation

- Gyrator: a gyrator is a two-port element, and, like transformer, it is power continuous (no power is stored or dissipated). An electromotor is an example of the gyrator element. The gyrator establishes the relation between the effort on one side to the flow on the other and vice versa, indicated by $e_2 = m.f_1$; $e_1 = m.f_2$, satisfying the power balance between the both sides.

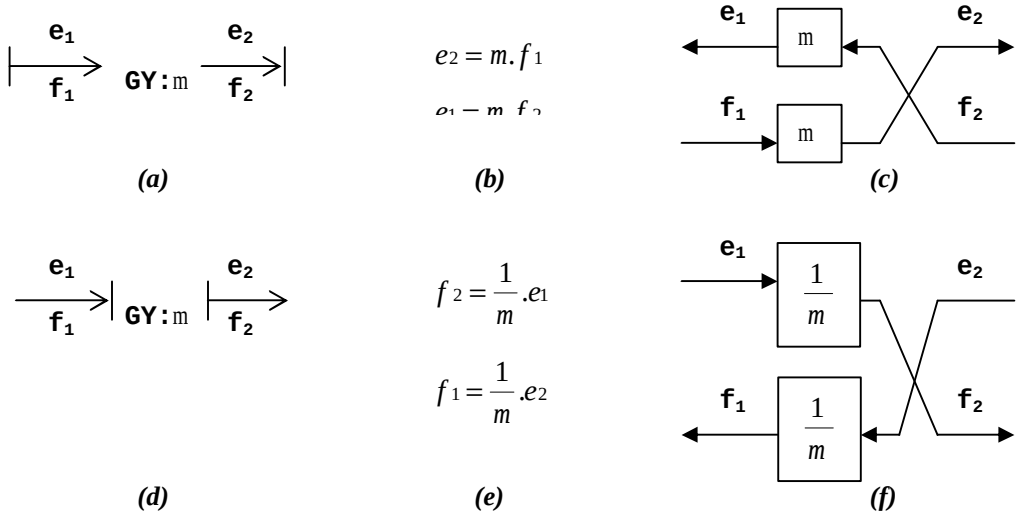


Figure 9. (a)(b)(c)(d)(e)(f) Gyrator element, related equations and block diagram for the two causality types

In the absence of differential causality (non-dependent storage elements) and algebraic loops, the set of state equations derived from the causalized BG corresponds to a set of ordinary first-order differential equations (ODEs). Causal conflicts describe implicit models, whose causalized BG representation generates a set of differential and algebraic equations (DAEs). As it was introduced in chapter 1, different simulation algorithms exist to numerically solve both types of equations systems, most of them based on the discretization of time.

4. MAPPING M/CD++ ELECTRICAL CIRCUITS TO BOND GRAPHS

This component of M/CD++ compiler is responsible for generating the BG model associated to a given electrical circuit. This way, the electrical circuit object, modeled using the class hierarchy mentioned on the previous section, represents the input to the mapper. The generated BG constitutes the output, which is used as the input model on the next simulation phase. The BG generation algorithm is based on the Karnopp's circuit construction method [14]. The basic approach is to construct a bon graph that resembles the circuit structurally and then to simplify the BG based on selected circuit properties. The following steps can be distinguished:

1. *For each node with a distinct potential write a 0-junction.*
2. *Insert each 1-port circuit element by adjoining it to a 1-junction, inserting 1-junctions between the appropriate pair of 0-junctions (C, I, R, Se, Sf elements)*
3. *Assign power directions to all bonds*
4. *If the circuit has explicit ground potential, delete the 0-junction and its bond; if no explicit ground potential is shown, choose any 0-junction and delete it.*
5. *Simplify the resulting BG*

- ***Step 1:*** *0-junction insertion*

The *0-junction* insertion process was implemented using the transitive closure function, applied to every node on the graph. The transitive closure tells if there is a path between arbitrary nodes *x* and *y*, given only the adjacency information. Calculating the transitive closure for every node on the graph guarantees that the *0-junction* elements will be correctly inserted, no matter how connections between coupled elements were described on the Modelica input file. The last is related to the number of different possibilities that the user has to specify parallel coupling between the elements of the circuit. The *0-junction* insertion process is illustrated on Figure 10.

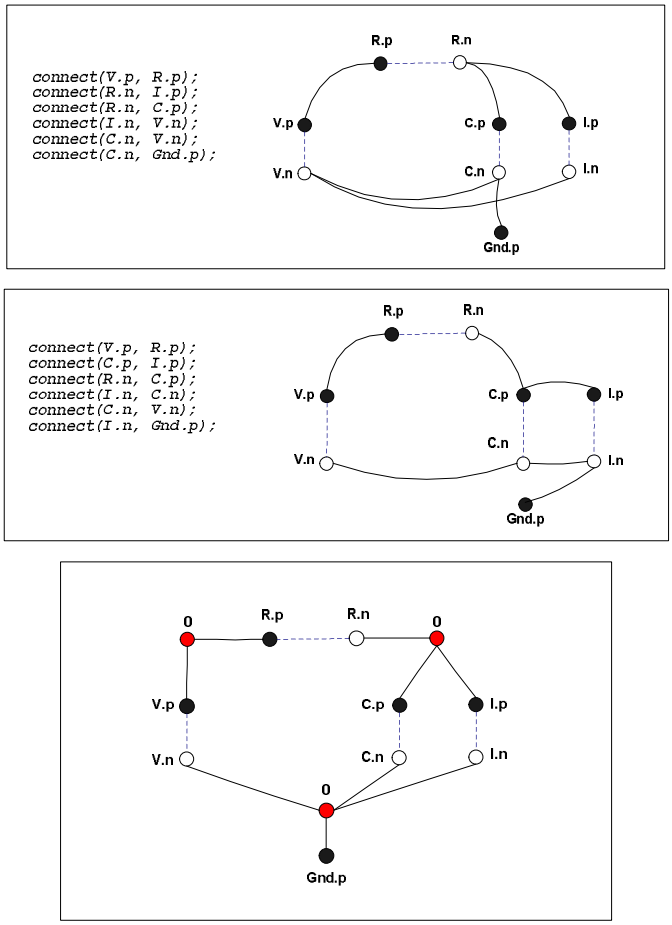


Figure 10. 0-junction insertion process (a) and (b)Alternative specifications and graph representations for the circuit of Figure 4. (c) Graph representation with 0-junctions inserted

- ***Step 2:*** *1-junction insertion:* A *1-junction* component is adjoined to each *1-port* element, inserting it between the corresponding pair of *0-junctions*. In fact, the method was extended to support *k-port* elements, adding *k 1-junction* elements adjoined to each *k-port* component on the circuit. This step also merges the nodes linked by the edges describing logical relations (dashed lines). As the representation of these relations is no longer needed for the simulation process, this kind of edges is deleted, and the corresponding nodes joined, reducing the cardinality of the graph nodes and edges sets.

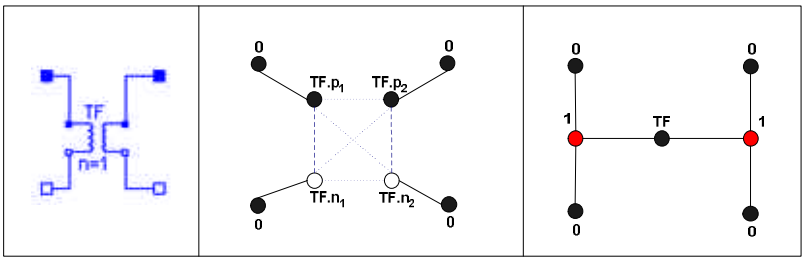


Figure 11. (a) Transformer element (b) Graph representation with 0-junctions insertion (c) Graph with 1-junctions insertion and logical-linked nodes merging

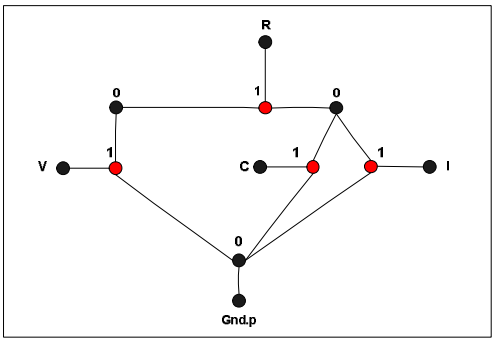


Figure 12. 1-junctions insertion and logical-linked nodes merging for the graph on Figure 10 . c

- **`Step 3:` `Assign power direction:`** The power direction specifies the direction in which the power flow is assumed to be positive. There is a standard convention that assumes possitive direction of power when it flows out of the sources (*Se* and *Sf*) and into *C*, *I* and *R* elements. For two-port elements, *TF* and *GY*, power in convention is used. To assign power direction to all bonds, a power propagation algorithm was developed. This algorithm "extends" the power through the graph using the standard conventions and the information compiled from the Modelica input file. Depending on the connections specified and the elements port type (positive/negative), power direction is inferred for the bonds connecting two junctions, where no conventions apply. Once the algorithm assigns direction to all bonds, a directed BG is obtained.

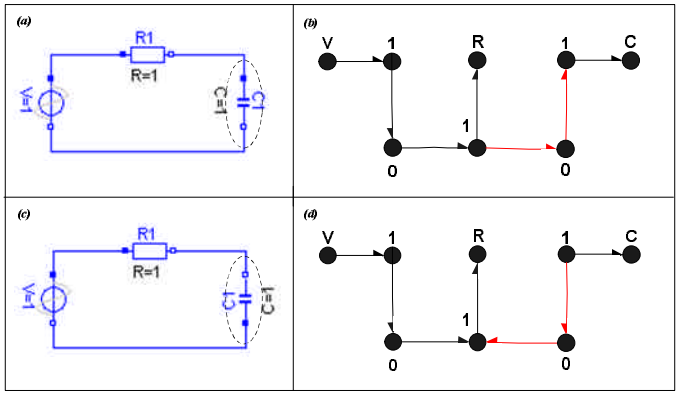


Figure 13. (a),(b) Electrical circuit and its directed BG representation
(c),(d)Capacitor change reflected on the bonds power direction

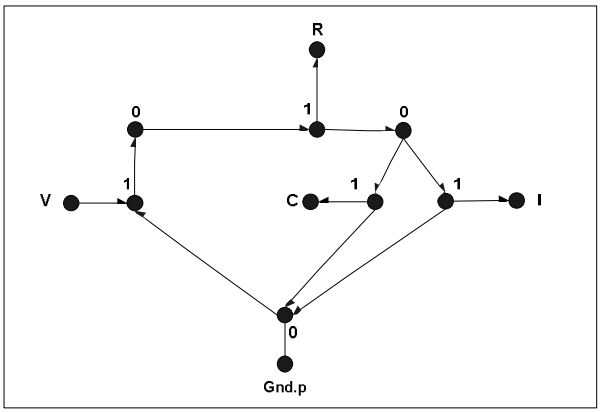


Figure 14. Power direction assigned to the graph on Figure 12

- ***Step 4:*** *Delete ground potential:* All the explicit ground potentials are deleted from the graph; if no explicit ground potential is found, the 0-junction nearest to each source element is erased. The 0-junctions selected are only those associated with the negative pin of every source's port.

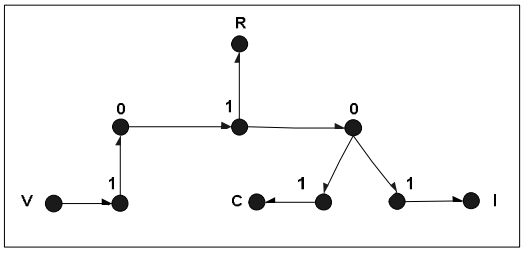


Figure 15. Ground potential elimination for the graph on Figure 14.

- ***Step 5:*** *BG simplification.* The BG is simplified applying the following rules:

  I. *A junction between two bonds having through power direction can be left out from the graph*

  II. *A bond connecting two junctions of the same type can be deleted and the junctions joined*

  Rule I)

$$\xrightarrow[f_1]{e_1}\ 1\ \xrightarrow[f_2]{e_2} \quad = \quad \begin{matrix} f_1 = f_2 \\ e_1 - e_2 = 0 \end{matrix} \quad = \quad \xrightarrow[f]{e}$$

$$\frac{e_1}{f_1}\ 0\ \xrightarrow[f_2]{e_2} \quad = \quad \begin{matrix} e_1 = e_2 \\ f_1 - f_2 = 0 \end{matrix} \quad = \quad \xrightarrow[f]{e}$$

$$\frac{e_1}{f_1}\ \underset{e_2 \mid f_2}{1}\ \xrightarrow[f_3]{e_3}\ \overset{e_4 \mid f_4}{1}\ \xrightarrow[f_5]{e_5} \quad = \quad \begin{matrix} f_1 = f_2 = f_3 = f_4 = f_5 \\ e_1 - e_2 - e_3 = 0 \\ e_3 - e_4 - e_5 = 0 \end{matrix} \quad = \quad \frac{e_1}{f_1}\ \overset{e_4 \mid f_4}{\underset{e_2 \mid f_2}{1}}\ \xrightarrow[f_5]{e_5}$$

$$\frac{e_1}{f_1}\ \underset{e_2 \mid f_2}{0}\ \xrightarrow[f_3]{e_3}\ \overset{e_4 \mid f_4}{0}\ \xrightarrow[f_5]{e_5} \quad = \quad \begin{matrix} e_1 = e_2 = e_3 = e_4 = e_5 \\ f_1 - f_2 - f_3 = 0 \\ f_3 - f_4 - f_5 = 0 \end{matrix} \quad = \quad \frac{e_1}{f_1}\ \overset{e_4 \mid f_4}{\underset{e_2 \mid f_2}{0}}\ \xrightarrow[f_5]{e_5}$$

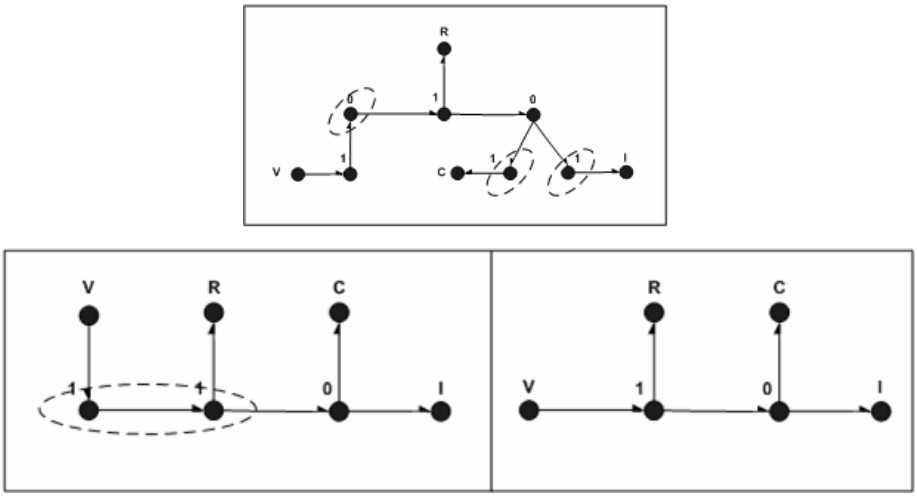


Figure 16. Simplification rules applied to the BG on Figure 15 (a) Simplification by rule I, (b) Simplification by rule II, (c) Resulting BG

5. BG VALIDATION

After the BG is constructed and simplified, we apply different error detection techniques for the resulting BG. **Causalization** is the process where the signal direction of the bonds is determined. Once this process is applied to the graph, each bond can be interpreted as a bi-directional signal flow. The causal BG can then be seen as a compact block diagram. There ARE four different causal constraints that a port element can impose to their connected bonds, depending on its constitutive equations:

- *Fixed causality*: this constraint appears when the equations only allow one of the two port variables to be the outgoing variable, i.e. source effort (*Se*) and source flow (*Sf*) components
- *Constrained causality*: there exist relations between the causalities of the different ports within the component that define causal constraints. At 0-junctions, where all efforts are equal, *exactly one* bond has *flow causality* (*flow-out causality*). The causal constraint at a 1-junction is the dual form of the 0-junction. TF and GY elements also have constrained causality. At a TF element, one bond has *effort causality* (*effort-out causality*) and the other *flow causality*. At GY, both bonds have either *effort causality* or *flow causality*.
- *Preferred causality*: the causality on storage elements determines whether integration or differentiation with respect to time will be used. Integration has preference above differentiation, representing the *preferred causality*. Then, at *C* elements, the preferred causality is the *effort causality*; at *I* elements, the *flow causality*.
- *Arbitrary causality*: arbitrary causality is used when no causal constraints exist, i.e. at *R* elements.

The *Sequential Causality Assignment Procedure (SCAP)* method [14] was developed by Karnopp and Rosenberg in order to assign causality to the bonds of a given BG. The method starts choosing a fixed causality element (source) and then propagates the assignment through the structural components (junctions, transformer and gyrator) whenever it is possible, according causality restrictions. Once that all sources have been processed, a storage element (*C* or *I*) is selected and the preferred causality applied, restarting the propagation step. That is repeated until all storages have their causalities assigned. At last, if the graph is not completely causalized, the iteration is repeated beginning with a dissipator (*R*). If the last step is reached, the model contains algebraic loops. The automatic causality asignment process implemented on *MCD++* is almost totally based on the SCAP

method. The algorithm described above was developed with the addition of some restrictions, in order to check and inform structural conflicts within the model. Only the preferred causality was implemented for the storage elements, given the drawbacks inherent to differentiation algorithms (i.e. infinite output for a step input function). The causalization process was applied to the graph on Figure 16, the resulting BG is shown on Figure 17. *Effort-out* causality is represented with a causal stroke outwards the component. The *flow-out* causality is indicated using a causal stroke inward the component.

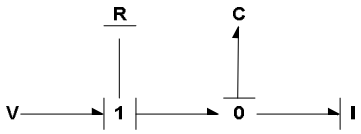


Figure 17. Causality assignment to the BG on Figure 16.

Structural singularities and **algebraic loops** within a model are automatically detected but not corrected. Whatever any of these properties are discovered on the causalized BG, the processing is aborted and the exception informed to the modeler. Several actions can be taken by the user in order to solve the error. Some solutions include adding elements to the circuit. Anyway, the corrections have to be done manually and, when finished, the MCD++ simulator newly invoked. On the implemented causalization procedure, only integral causalities are assigned to the storage components. Then, in presence of a 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.

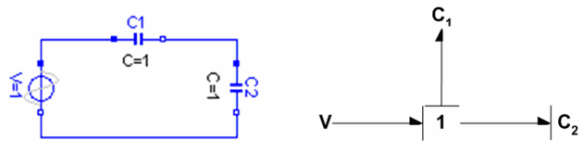


Figure 18. (a) Electrical circuit with coupled capacitors not supported by MCD++ (b) BG representation

An example of a model with a structural singularity and not supported on *MCD++* is shown on Figure 18. The circuit can be modified as shown on Figure 19, adding a dissipator with a very low resistance value, in order to allow simulation within the toolkit.

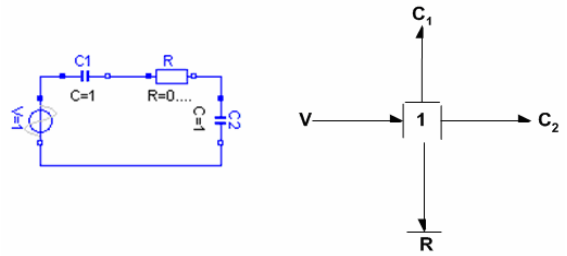


Figure 19. (a) Electrical circuit on Figure 18 with a low resistance dissipator (b) BG representation

Algebraic loops are found by inspection of closed causal paths. A causal path is a path defined by bonds with the same causal orientation, not including storage or source elements. IF the step *9* of the SCAP procedure is reached, that indicates that at least one algebraic loop exists on the model. The closed causal path inspection algorithm implemented within *MCD++* allows the listing of the resistance elements defining the loop.

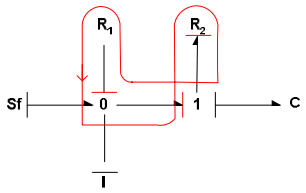


Figure 20. Closed causal path

An example of a model with an algebraic loop is shown on Figure 21. The circuit can be modified as shown on Figure 22, adding a storage element to break the loop.

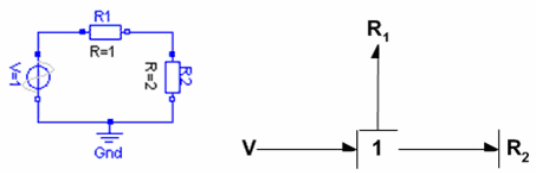


Figure 21. (a) Electrical circuit with an algebraic loop between R_1 and R_2 (b) BG representation

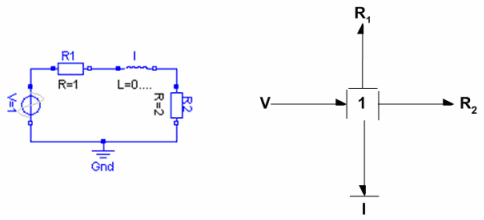


Figure 22. (a) Storage insertion to break the algebraic loop (b) BG representation

6. IMPLEMENTATION ISSUES

Once the BG model from the electrical circuit is generated, and completely causalized, it is ready to be simulated. As it was mentioned, the discrete event approach represents the simulation paradigm used on *MCD++*. In order to perform the BG simulation, the CD++ simulator is internally invoked from *MCD++*. Regarding that the input to CD++ must be a valid DEVS (or Cell-DEVS) model, the BG generated on the previous phase must be transformed into its corresponding DEVS representation. The BG Compiler for CD++, developed within *MCD++*, is the component responsible for this transformation. As a result of the compilation phase, a valid CD++ DEVS model specification is generated. The components in the BG to be constructed are translated into Quantized BG (QBG), that is, a BG where all the storages and sources are quantized elements. Given that QSS modifies the original system adding quantizers equipped with hysteresis to the integrators output, only the storage elements (capacitor and inertia) need to be changed in order to use the QSS method on QBG [15]. A capacitor defines the following relation between the power variable (*e = effort*) and the energy variable (*q = displacement*)

$$e(t) - g(q(t)) = 0 \qquad (4.1a)$$

$$\dot{q}(t) - f(t) = 0 \qquad (4.2a)$$

In a capacitor, the integral causality asignment causes function *f(t)* to represent the input and *e(t)* the corresponding output. The displacement, *q(t)*, is the state variable (integrator's output). Using the QSS method, transforms *4.1a*: $e(t) = g(q_q(t))$ being $q_q(t)$ the quantized version of *q(t)* [15]. This equation can be rewriten using

the composition of the quantization function with function g: $e(t) = g_q(q(t))$ where g_q *is a quantized function (the composition of a quantization function with a continuous function).*

The same reasoning can also be applied to the inertias; then, the QSS method can be used for BG simulation. The capacitors and intertias functions have to be replaced by their quantized version functions, obtaining the QBG representation. The QSS error, convergence and stability properties, described on chapter 3, are valid on QBG models too [15].

Several atomic DEVS models where developed on CD++, implementing the QSS and QBG concepts on the toolkit. That provides the extensions needed to simulate dynamic systems within the CD++ simulator. Each component of the QBG was implemented as an atomic DEVS model on CD++, using the BG and QSS definitions. They are listed bellow:

- *QBGCapacitorFlowIn*
- *QBGInductorEffortIn*
- *QBGResistanceFlowIn*
- *QBGResistanceEffortIn*
- *QBGSourceEffort_Constant*
- *QBGSourceEffort_Step*
- *QBGSourceEffort_Sine*
- *QBGSourceEffort_Pulse*
- *QBGSourceFlow_Constant*
- *QBGSourceFlow_Step*
- *QBGSourceFlow_Sine*
- *QBGSourceFlow_Pulse*
- *QBGTransformer*
- *QBGGyratorFlowIn*
- *QBGGyratorEffortIn*
- *QBGSerialJunction*
- *QBGParallelJunction*

A coupled DEVS representing the resulting model can be formally defined as:

$$CQBG = \langle X_{self}, Y_{self}, D, \{M_i\}, \{IC\}, select \rangle$$

$X_{self} = \{Æ\}$ no external inputs are defined

$Y_{self} = \{Æ\}$ no external outputs are defined

D is the set integrated by all the elements representing BG components, for each i in D,

M_i is a DEVS atomic model representing a QBG component

IC is the internal coupling set defined as: $IC = \{ice_{ui,vj}\} \text{ È } \{icf_{vj,ui}\}$ where $ice_{ui,vj}$and $icf_{vj,ui}$ represent the coupling between effort and flow ports on components u and v, being the effort calculated by element u (the causal stroke outwards u).

$$ice_{ui,vj} = \begin{cases} ((u, out_{ei}), (v, in_{ej})) & \text{if } v \text{ is not a source (flow source)} \\ f & \text{otherwise} \end{cases}$$

if u is a serial junction then i = 1 (only one effort-out port)

$$icf_{vj,ui} = \begin{cases} ((v, out_{fj}), (u, in_{fi})) & \text{if } u \text{ is not a source (effort source)} \\ f & \text{otherwise} \end{cases}$$

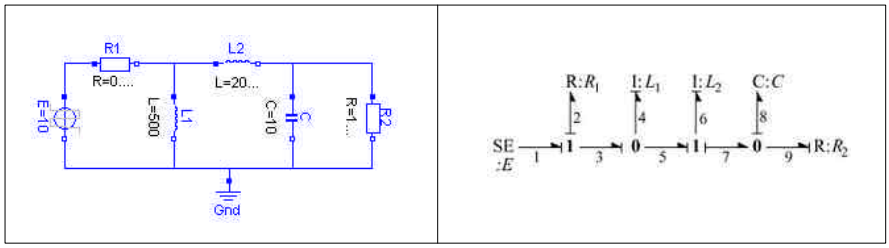
if v is a parallel junction then j = 1 (only one flow-out port)

select the *tie-breaking function* gives priority to the structural components (junctions, transformer and gyrator) in order to avoid loosing any output message.

Given the definition of the coupled DEVS model associated to a QBG, the developed compiler takes the input BG and performs the compilation. That generates the structured DEVS model which is described using the specification language supported by the CD++ simulator (*.ma* file). To generate the CD++ specification file, the compiler executes the following steps:

1. *For each component u of the QBG, add u to the declaration section within the .ma file. Considering the assigned causalization, select the valid implementation class for the component (for components having both causality types two different implementations were developed)*
2. *For each bond b = (u,v) of the QBG, generate the coupling information between u and v on the links section within the .ma. Follow the coupling definitions formalized above.*
3. *For each component u of the QBG, generate the component's configuration information within the parametrization section on the .ma file.*

Figure 23 shows a graphical representation of the BG to the coupled DEVS model transformation, given an input circuit specification:



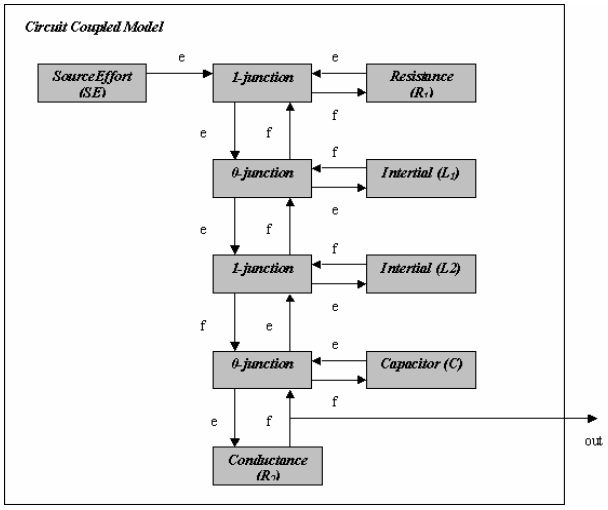


Figure 23. (a) Electrical circuit, (b) BG associated to the circuit, (c) Coupled DEVS model representation

The CD++ coupled model generated by the compiler is shown in the following figure.

```
[top]
components : $PJ2@QBGParallelJunction $PJ3@QBGParallelJunction $SJ2@QBGSerialJunction
$SJ3@QBGSerialJunction C@QBGCapacitorFlowIn  L1@QBGInductorEffortIn L2@QBGInductorEffortIn
R1@QBGResistanceFlowIn R2@QBGResistanceEffortIn V@QBGSourceEffort_Pulse

link : e2n@$PJ2 e2p@$SJ2
link : f2p@$SJ2 f2n@$PJ2
link : f3n@$SJ2 f2p@$PJ3
link : e2p@$PJ3 e3n@$SJ2
link : f1n@$PJ3 f1p@C
link : e1p@C    e1n@$PJ3
link : e3n@$PJ3 e1p@R2
link : f1p@R2   f3n@$PJ3
link : e1n@$SJ2 e1p@I2
link : f1p@I2   f1n@$SJ2
link : f1p@$PJ2 f1n@$SJ3
link : e1n@$SJ3 e1p@$PJ2
link : f2n@$SJ3 f1p@R1
link : e1p@R1   e2n@$SJ3
link : f3p@$SJ3 f1n@V
link : e1n@V    e3p@$SJ3
link : e3n@$PJ2 e1p@I1
link : f1p@I1   f3n@$PJ2

[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          outputFile : out/V.out
```

Figure 24. File generated by the compiler (.ma)

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).

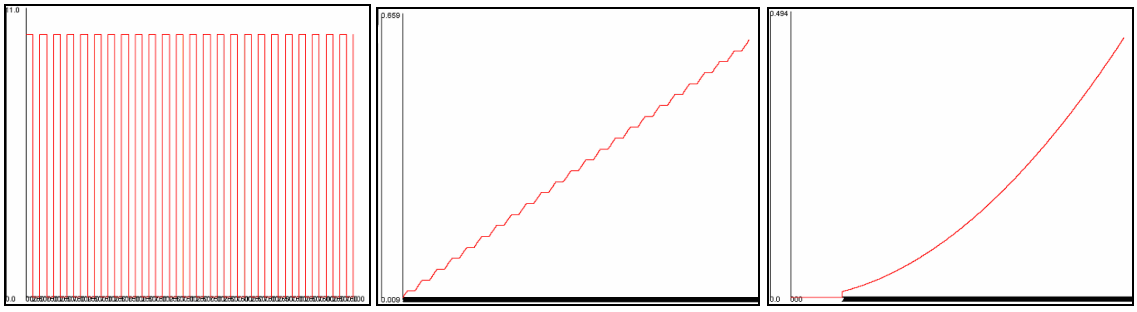


Figure 25. (a) Pulse Voltage Source (b) Current on inductor I_1 (c) Voltage on Capacitor C

7. CONCLUSION

The DEVS formalism is a method defined for modeling and simulation of discrete event systems. During the last years the DEVS theory has evolved, and it was recently upgraded in order to permit simulation of continuous and hybrid systems. We introduced a tool for modeling and simulation of continuous systems based on DEVS. Models are described using Modelica, a modular and acausal standard specification language for physical sys-

tems modeling. Examples of model simulation with their execution results are included. The simulation results generated by MCD++ were compared with those produced by a complex physical system simulation environment with Modelica support called *Dymola*.

We presented the techniques used for transformation of M/CD++ electrical models into Bond Graphs, and the methods to detect causality and algebraic loops in the resulting models. MCD++ approximates the system solution based on the QSS method, which uses a simple first order integration approach.

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] Taylor, M. 1996. *Partial Differential Equations: Basic Theory.* Springer Verlag, NY.

[2] Press, W.H.; Flannery B.P.; Teukolsky, S.A.; Vetterling, W.T. 1986. *Numerical Recipes.* Cambridge University Press, Cambridge

[3] Zeigler, B.P., "Continuity and Change (Activity) are Fundamentally Related in DEVS Simulation of Continuous Systems", LNCDS, Vol. 3397/2005, Springer-Verlag, NY, pp. 1-17.

[4] Zeigler, B; Kim, T; Praehofer, H. “Theory of Modeling and Simulation”. New York, 2000.

[5] Zeigler, B. DEVS. “Theory of Quantization”. DARPA Contract N6133997K-007: ECE Dept., University of Arizona, Tucson, AZ, 1998.

[6] D'Abreu, M.; Wainer G. “Defining a compiler for discrete-event simulation of continous systems”. Technical Report SCE-05-017. Dept. of Systems and Computer Engineering. Carleton University. 2005. Submitted for publication.

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

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

[9] Åström, K. J; Elmqvist, H.; Mattsson, S. E. “Evolution of continuous-time modeling and simulation”. The 12th European Simulation Multiconference, ESM'98, Manchester, UK, 1998.

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

[11] D'Abreu, M.; Wainer G. “Defining hybrid system models using DEVS quantization techniques”. Proceedings of the Winter Simulation Conference. New Orleans, LA. U.S.A., 2003.

[12] Cellier, F. “Continuous System Modeling”. Springer-Verlag, New York, 1991.

[13] Samantaray, A. 2003. About BG.The system modeling world [online]. Available online via http://www.bondgraphs.com/about.html [Accessed April 6 2003].

[14] Karnopp, D.; Margolis, D.; Rosenber R. “System Dynamics: A Unified Approach”. Wiley, 1990.

[15] Kofman, E. “Discrete Event Based Simulation and Control of Continuous Systems”. PhD’s thesis. Facultad de Ciencias Exactas, Ingeniería y Agrimensura - Universidad Nacional de Rosario, August 2003.

APPENDIX: BOND GRAPH CLASS HIERARCHY