Advances in Engineering Software 79 (2015) 111–126

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Using a Discrete-Event System Specifications (DEVS) for designing a Modelica compiler

Gabriel A. Wainer [a,*], Mariana C. D'Abreu [b]

[a] Department of Systems and Computer Engineering, Centre for Visualization and Simulation (V-Sim) Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
[b] Computer Science Department, Universidad de Buenos Aires, Pabellón I, Ciudad Universitaria, 1428 Buenos Aires, Argentina

ARTICLE INFO

Article history:
Received 30 January 2014
Received in revised form 25 September 2014
Accepted 28 September 2014
Available online 28 October 2014

Keywords:
Simulation
Object Oriented Modeling
Discrete event systems
Hybrid systems
Modelica
DEVS

ABSTRACT

We introduce a new architecture for the design of a tool for modeling and simulation of continuous and hybrid systems. The environment includes a compiler based on Modelica, a modular and a causal standard specification language for physical systems modeling (the tool supports models composed using certain component classes defined in the Modelica Standard Library, and the instantiation, parameterization and connection of these MSL components are described using a subset of Modelica). Models are defined in Modelica and are translated into DEVS models. DEVS theory (originally defined for modeling and simulation of discrete event systems) was extended in order to permit defining these of models. The different steps in the compiling process are show, including how to model these dynamic systems under the discrete event abstraction, including examples of model simulation with their execution results.

1. Introduction

In recent years, there has been a number of software tools built for Modeling and Simulation (M&S) of Continuous Variable Dynamic Systems [1–4]. These systems are usually represented by continuous variables on a continuous time basis (i.e., mechanical, electrical, electro-magnetic, hydraulic physical systems, belong to this category). Although the analysis of these complex systems had been traditionally tackled with different mathematical formalisms, including Differential Algebraic Equations (DAEs), Ordinary Differential Equations (ODEs), or Partial Differential Equations (PDEs) [5] for most complex systems, the solutions to these equations are very difficult or impossible to find. Instead, M&S tools have been successful in finding approximate solutions to these equations, allowing studying many different phenomena where analytical results are unfeasible.

Most of these tools implement numerical methods that discretize time in order to find approximate solutions to the equations [6]. Instead, in the last few years, a radically different approach based on the DEVS (Discrete Event System Specification) formalism [7] has been explored. This method presents some advantages over time-stepped simulation, including the reduction of the number of calculations for a given accuracy [8] and the seamless integration of complex systems modeled by both continuous time and discrete events. 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 [9]. The state variables of the system are converted into a piecewise constant function via a quantization function. The Quantized State System (QSS) method [10] is an extension to Q-DEVS in which the trajectory of each state variable uses a quantization function equipped with hysteresis, which constitutes a general method for ODEs integration using discrete event theory.

Based on this theoretical framework, the design of an environment for M&S of continuous and hybrid systems using such a discrete-event modeling approach is presented here. This resulted in the design of an open-source tool, called M/CD++, based on Modelica [11] and CD++ [12]. Modelica is an object-oriented language for modeling physical systems, designed to support library development and model exchange. Models in Modelica are mathematically described by differential, algebraic and discrete equations. CD++ is an M&S tool implementing DEVS theory. M/CD++ is a proof-of-concept implementation, which allows building dynamic systems belonging to the electrical domain, and it shows how this language can be converted into a discrete event approximation. M/CD++ includes Modelica v2.1 language support for electrical circuits construction [11].

* Corresponding author. Fax: +1 613 520 5727.
E-mail addresses: gwainer@sce.carleton.ca (G.A. Wainer), mdabreu@dc.uba.ar (M.C. D'Abreu).

http://dx.doi.org/10.1016/j.advengsoft.2014.09.009
0965-9978/

One of the original contributions is that, internally, the models are represented using Bond Graphs (BG) [13], which allows domain-independent description of the dynamic behavior in the physical systems being modeled. BG representation provides a sound mathematical foundation for model verification, as they can be manipulated and converted, checking for algebraic loops and singularities (elements that have discontinuities), which will be discussed further in Section 2. The compiler builds an optimized BG corresponding to the original circuit in Modelica. This is a major difference between M/CD++ and other compilers like Adevs (http://web.ornl.gov/~1qn/adevs). The Bond Graph intermediate representation can be composed with other models as it is a generic modeling language.

Another original contribution is that this optimized BG, in turn, is used to generate a DEVS model specification (an executable DEVS model according to the rules of CD++). The resulting CD++ model represents a discrete-event version of the equations associated to the original electrical circuit. Based on the QSS and QBG theories, CD++ is used to approximate the solution numerically using a discrete event approach. The main objective of this work is to show how to integrate these methodologies and to discuss the results obtained in such process.

2. Modeling continuous systems

The behavior of continuous dynamic systems is usually described in terms of DAEs, ODEs or PDEs. Simulations based on these formalisms are mainly accomplished by describing the set of equations, and finding a numerical approximation to them based on consistent initial conditions [5,6,14,15]. Recently, a different approach has been proposed, which focuses on decomposing models of the physical systems into smaller submodels interfaced by distinct connections. Some of them have used an Object-Oriented approach to promote model specification in a more natural way, decreasing the abstraction gap between the real system and its representation, while improving development and reusability of the models [16–18]. Many of these concepts were adopted for the design of a new family of languages [13,16] that were standardized by Modelica [11].

2.1. Modelica

Modelica [11] is an object-oriented language intended for modeling continuous systems within many application domains (electrical, hydraulics, mechanics, thermo-dynamical, etc.) that supports several formalisms (e.g. ODEs, DAEs, etc.). It is a non-causal language that includes mathematical equations and object-oriented constructs, allows library development and model exchange. The model semantics in Modelica is specified by a set of rules used to translate the models to a corresponding flat hybrid DAE. A model is represented using classes that can be developed hierarchically, allowing component and knowledge reuse. Fig. 1 presents an example of an electrical circuit specified using Modelica *electrical* library. Fig. 1(a) shows the Modelica specification of the circuit presented in Fig. 1(b).

2.2. Bond graphs

As discussed earlier, M/CD++ can simulate electrical circuit models defined with Modelica, and the compiler uses Bond Graphs as intermediate model representation. BG were used because they provide multiple advantages:

- They allow domain-independent description of the dynamic behavior of the physical systems.
- They can be applied to multiple physical domains, thus, intermediate models could be reused in a different context.
- They use modular and hierarchical models (BG submodels connect via well-defined interfaces), improving model reuse.
- No algebraic manipulation is needed.
- They can be easily translated to equivalent block diagrams.

A BG is a directed graph where the bonds represent the ideal exchange of energy between physical processes represented by the graph vertices, as seen in Fig. 2 [19]. BGs represent continuous systems as a set of elements interacting with each other by *energy* interchange that determines the dynamics of the system. *Power* (the time derivative of energy) is the product of two factors: *effort* and *flow*. In electrical systems, for instance, power is the product of voltage and current; in hydraulic systems, power is the product of pressure and volume flow rate. If generalized effort and flow variables are defined, their product gives the power exchanged by the elements of the system. Besides effort and flow, *momentum* (p) and *displacement* (q) are the result of an accumulation (integration) process. These quantities represent the state variables of the system.

BG modeling is based on two main assumptions: the energy conservation law and the use of a lumped approach. These characteristics imply that model properties can be separated from each other using components that can be coupled using ideal connections. As seen in Fig. 2, these abstract entities are called energy *ports*. Ports connections represent the energy flow, guaranteeing the power continuity quality (i.e. no energy is generated or dissipated by the ports). The power direction on a bond determines the flow sign (inward pointing bonds represent positive flow; outward pointing bonds have negative flow). Elements can use more than one port. One-port elements use one energy port represented with a bond. Two-port elements are represented with two ports (and bonds) in which the exchange of power between the elements occurs.

BGs are *a causal*, meaning that components can determine the two power variables – effort and flow – at the same time. In order to simulate BGs, causal analysis is essential to describe a BG model in computational terms and to derive its set of differential equations. Given a pair of elements connected through a bond, their *causality* determines which component causes the flow and which one the effort, as seen in Fig. 3.

BG with causality conflicts describe implicit models whose representation generates a set of DAEs. In the absence of differential causality (non-dependent storage elements) and algebraic loops, the set of equations derived from the causal BG corresponds to first-order ODEs.

The BG elements are the following: Capacitor (C), Inductor (I), Resistor (R), Effort source (Se), Flow source (Sf), Transformer (TF), Gyrator (GY), 1-junction and 0-junction. As an example, Fig. 4 shows the specification an Inductor (I elements), in which a conserved quantity p is stored by accumulating effort e. Examples of I elements include inductors and mass (in the electrical and mechanical domains respectively). The constitutive equations are defined as $p = e$ and $f = f(p)$. The equations for a linear inductor are $p = \int_0^t edt$ and $i = \frac{1}{L}p$ where L is the inductor's constant.

Junctions represent the constrained interactions between elements and couple the components in a power-continuous way, with no energy dissipation or storage. Since there are only two ways in which components can exchange power (serial or in parallel), only two types of junctions are needed (see Fig. 5).

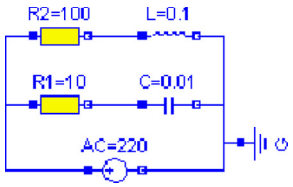
The 0-*junction* represents a node where all the efforts of the connecting bonds are equal (i.e. a parallel connection in an electrical circuit). In terms of power continuity, the sum of all the flows (considering power direction) is zero. This corresponds to Kirchhoff's law of current in electrical networks (all currents on a node,

```
model SampleCircuit

  Modelica.Electrical.Analog.Basic.Resistor R1(R=10);
  Modelica.Electrical.Analog.Basic.Inductor L(L=0.1);
  Modelica.Electrical.Analog.Basic.Capacitor C(C=0.01);
  Modelica.Electrical.Analog.Basic.Ground G;
  Modelica.Electrical.Analog.Basic.Resistor R2(R=100);
  Modelica.Electrical.Analog.Basic.VsourceAC AC;

equation
   connect (AC.p, R1.p);
   connect (R1.n, C.p);
   connect (C.n, AC.n);
   connect (AC.n, G.p);
   connect (R1.p, R2.p);
   connect (R2.n, L.p);
   connect (L.n, C.n);
end SampleCircuit;
```

(a)



(b)

Fig. 1. Modelica specification of a simple electrical circuit (a) Modelica Specification and (b) graphical representation.

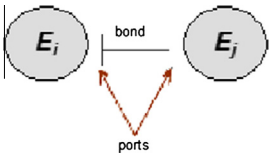


Fig. 2. BG representation of energy flow from E_i to E_j.

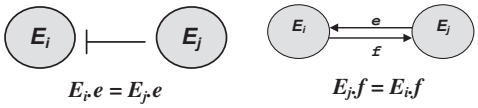


Fig. 3. Causal bond, equivalent graph, equations.

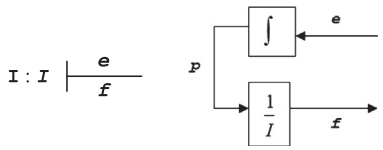


Fig. 4. *I* element with preferred causality and block diagram representation.

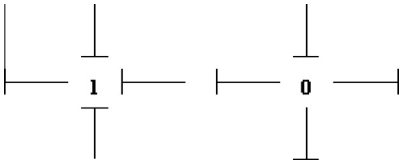


Fig. 5. (a) 1-Junction – serial junction. (b) 0-Junction – parallel junction.

considering their signs, sum to zero). The 1-*junction* represents a node where all the flows of the connecting bonds are equal (i.e. a serial connection in an electrical circuit). Due to power continuity, all the efforts must sum to zero (which corresponds to Kirchhoff's voltage law in electrical networks; in the mechanical domain, 1-junctions represent the principle of d'Alembert for force balance).

2.3. The DEVS formalism

In M/CD++, the intermediate BG models are subsequently mapped into DEVS (Discrete Event System specifications) [7]. DEVS, a universal formalism for M&S of discrete-event dynamic systems, 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. DEVS models can be described using a mix of modular components called atomic or coupled. Atomic models are formally defined by:

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

A DEVS model is in state $s \in S$, waiting for the next internal transition determined by $ta(s)$ (i.e., the time advance function applied to the current state). When this time passes, the model can generate an output event $y \in Y$ using the output function $\lambda(s)$. The model also changes to a new state given by $\delta_{\text{int}}(s)$ (the internal transition function applied to s). Whenever the model receives an external event $x \in X$, the external transition function δ_{ext} is executed to compute the new model's state.

These behavioral models can be composed hierarchically. These structural models (called coupled) are defined as:

$$\text{CM} = \langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{\text{ij}}\}, \text{ select} \rangle$$

For each $j \in I_i$, Z_{ij} is a function that translates the outputs from model i to j. These models M_i are indexed by D. When an internal event occurs on i, a signal is sent to component j at the same time. External inputs and outputs X and Y define the model's interface. Two or more components might have their internal transitions scheduled at the same time, generating ambiguity during the simulation. The select function solves this problem by defining the rules needed to determine which one of the imminent components will execute next.

DEVS has been extended in order to be able to model and simulate continuous and hybrid systems. Different research teams [8–10,20,21,23–27] have 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, which improves verification and validation, also enhancing reusability.
- Seamless integration with models defined with other modeling techniques (Petri Nets, Automata, Statecharts, etc.).
- Simulation of time-stepped (discrete time) models, which are a particular case of discrete event methods.
- Hybrid systems modeling: the discrete event paradigm provides a unified theory to model and simulate systems with continuous and discrete components.

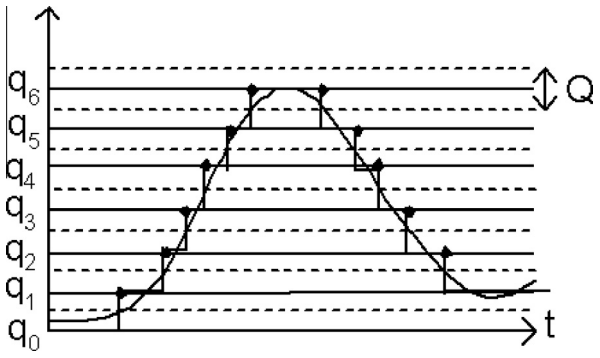


Fig. 6. Signal quantization.

Most of these techniques are based on Q-DEVS [9], whose main idea is to represent continuous signals by the crossing of an equal spaced set of boundaries (as shown in Fig. 6). This approach requires determining at what time a dependent variable will enter a given state. QSS (Quantized State Systems) [10,23] is a method based on this idea that includes hysteresis, which, as proved in [10], can be used to approximate differential equation systems by legitimate DEVS models.

We also defined some models based on the GDEVS formalism [25]. GDEVS [36] uses polynomials of arbitrary degree (as opposed to constant values), to represent the piecewise input–output trajectories of a DEVS model. In GDEVS, the system is modeled through piecewise polynomial coefficients. As these have piecewise constant trajectories, we can build a discrete event abstraction in the coefficient space using the concept of coefficient event (an instantaneous change of, at least, one of the value of the coefficients defining the polynomial trajectory). An event is a list of coefficient values defining a polynomial describing the trajectory. QSS is an approximation method to model and simulate continuous systems, which are usually modeled with Ordinary Differential Equations (ODEs) and Algebraic Equations. Traditional method to obtain a detailed description of a system behavior entails solving these equations simultaneously. To do this, a technique of numerical integration is used to solve ODEs such as Euler, and Runge–Kutta. All these methods rely on discrete-time integration of ODEs. In this way, time progresses in small steps, and at each step, an approximation is computed for the ODEs solution. When a system modeled by ODEs has a discontinuity (i.e. sudden jumps in its variables values with regard to time), the numerical integration method may produce unacceptable errors. However, these kinds of discontinuity are normal properties in hybrid systems, which can be seen as operating in different modes, each described with a specific ODE, for example, a heating system with an on–off thermostat controller. Quantized State Systems QSS is a different method for approximation. This is a quantization-based method that models hybrid systems as discrete-event systems and not as discrete-time. This solves the above problem around discontinuities while solving hybrid system. Let $D = \{d_0, \ldots, d_n / d_i \in \Re, d_{i-1} < d_i \text{ with } 1 \leqslant i \leqslant r\}$ and $x \in \Omega$ a continuous trajectory with x: $\Re \to \Re$. Let b: $\Omega \to \Omega$ be a mapping and q a trajectory defined by $q = b(x)$. A fundamental property of function b is given by $d_0 \leqslant x(t) \leqslant d_r \Rightarrow |q(t)-x(t)| = |b(x(t))-x(t)| \leqslant \max_{1 \leqslant i \leqslant r} (d_i - d_{i-1}, \varepsilon)$ where ε is the size of the hysteresis window. The addition of hysteresis removes the problem of a possible infinite number of transitions performed by a model in a finite time interval greater than zero [23]. The existence of a minimum time interval between events constitutes a sufficient condition to obtain legitimate models [10]. A detailed discussion on how to choose the quantum size for QSS simulations can be found in [35].

These methods are the basis of M/CD++, which was implemented on CD++, an environment for DEVS-based M&S [12]. In CD++, atomic models are implemented using C++, and coupled models are described using a built-in specification language, which includes information about the components, the coupling and the input and output ports. The following sections will explain how these methods were used in building this M&S environment and its implementation using CD++.

3. M/CD++

M/CD++ allows the definition and simulation of dynamic systems in the electrical domain, providing support for electrical circuits using a subset of Modelica Standard Library v2.1 [11]. The compiler builds an object-oriented representation of an electrical circuit that is translated into a BG to verify model properties. Then, the BG is converted into a DEVS model (where atomic models are based on QSS and Quantized BG). These models, added to CD++, can numerically approximate the solution using a discrete event approach. Although this work focuses on the electrical library, extending the compiler to other elements based on the techniques presented here is straightforward.

The objects defined in the Modelica Electrical library and 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, Step Voltage, SineVoltage, PulseVoltage, Constant Current, Step Current, SineCurrent and PulseCurrent. These components are described according to Modelica specifications [11], as seen in Fig. 7. The pulse voltage source uses various parameters: *I* (default = 1V) defines the pulse amplitude; and *width* defines the duty cycle (50% by default). The remaining parameters include the *period* (1 Hz by default), *voltage* and *time offsets*.

M/CD++ connects the different theories and languages presented in Section 2. M/CD++ is defined by various core components shown in Appendix I. The compiler starts with an electrical circuit model specified using Modelica, and finishes with a log file including the simulation results. The following sections describe the generalities of each of these components.

3.1. Modelica parser & checker

The first component executed is in charge of parsing the Modelica input file, and of checking the syntax and grammar of the electrical circuit model. The parser was built using GNU Bison [28], a general-purpose parser generator that uses the description of an LALR context-free grammar. Bison allows the specification of actions that accompany the syntactic rules. These actions are used to build the input file syntax tree, which is in turn used to perform semantic validation and construction of the electrical circuit. This includes the following checks:

- Specification of valid and supported packages.
- Specification of valid and supported types/classes.
- Checking for undeclared symbols.
- Specification of valid references to component attributes.

Only if a complete syntax tree is validated successfully, the electrical circuit model can be built; otherwise, the process is aborted (and the syntax/grammar error is reported). Several verifications were implemented to preserve the model properties, which are checked during the parsing phase (when the electrical circuit is built). This includes:

```
model PulseCurrent "Pulse Current source"
   parameter SI.Current I=1 "Amplitude of pulse";
   parameter Real width(
         final min = Modelica.Constants.small,
         final max=100) = 50 "Width of pulse in %
                                  of period";
   parameter SI.Time period(final min=Modelica.
       Constants.small)= 1 "Time for one period";
   extends Interfaces.CurrentSource(
     redeclare Modelica.Blocks.Sources.Pulse
        signalSource(amplitude={I}, width={width},
                  period={period}));
end PulseCurrent;
```

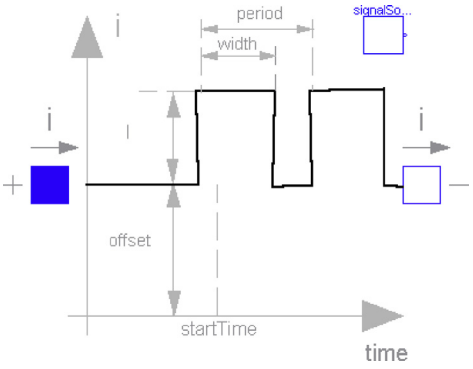


Fig. 7. Definition: sources.pulsecurrent.

- Valid specification of pin references.
- Definition of connections between existing elements.
- Connections from a component to itself.
- At least one source component defined.

A class hierarchy was implemented in CD++ 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 OO model associated to these concepts [21].

Fig. 8 shows an example of a circuit, its corresponding Modelica specification file, and the objects model constructed by the parser. The *EC* model shown in Fig. 8(a) and (b) uses various Modelica objects:

- *V*: instance of the pulse Voltage class (one-port element)
- *C*: instance of the Capacitor class (one-port element)
- *R*: a resistor component (one-port element)
- *I*: and Inductor component (one-port element)
- Gnd: instance of Ground class (one positive pin).

As discussed earlier, the electrical circuit is translated into an internal representation based on a graph notation that we use to build the Bond Graph, showed in Fig. 8(c). This is used to build the corresponding DEVS models in CD++ (showed in Fig. 8(d)). This intermediate notation represents each component using as many nodes as the number of pins of the element (for instance, V uses 2 nodes – 1 per pin – and GND only needs one). In addition, each element is represented by two nodes corresponding to the positive and the negative pins. Generalizing, *k*-port elements are represented by 2.k nodes. The graph distinguishes two types of connections: physical (solid lines) and logical (dotted lines). The former corresponds to the physical coupling between the elements of the circuit (for instance, Vp → Rp is a physical connection). Logical connections correspond to the associations between pins and ports: the pins of a given port connector are linked by dashed lines, while the port connectors are linked by dotted lines.

This graph representation considers the BG generation algorithm used in the mapping phase. The idea is to have a suitable data structure and model representation to optimize the generation process of the BG associated to the circuit, discussed next (see Fig. 9).

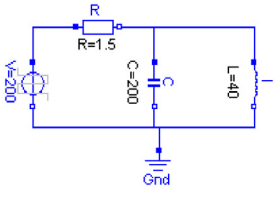
3.2. Mapping electrical circuits to bond graphs

After the compilation process finishes with no errors, we have a syntactically correct Modelica electrical circuit, the structure of the model in a graph notation, and the list of objects needed. Then, these data structures are used to generate a BG model. The BG

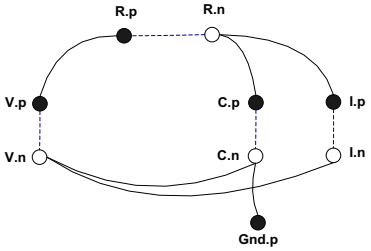
```
model EC
 Modelica.Electrical.Analog.Sources.PulseVoltage
    V(V=200, period=1, width=10);
  Modelica.Electrical.Analog.Basic.Capacitor C(C=200);
  Modelica.Electrical.Analog.Basic.Resistor R(R=1.5);
  Modelica.Electrical.Analog.Basic.Inductor I(L=40);
  Modelica.Electrical.Analog.Basic.Ground Gnd;

 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;
```

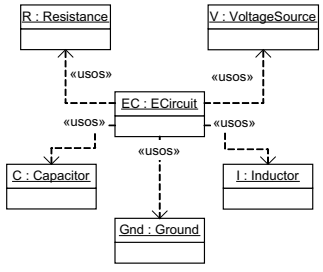
(a)



(b)



(c)



(d)

Fig. 8. (a) Modelica input file (b) electrical circuit model (c) graph intermediate representation on M/CD++ and (d) objects generated by the M/CD++ parser.

generation algorithm is based on the Karnopp's circuit construction method [29], which creates a BG that resembles the circuit graphically. Then, a simplified BG is built, based on various circuit properties, using the following steps:

1. For each node with a distinct potential, insert a 0-junction.
2. Add a 1-junction to each 1-port element. Add 1-junctions between the appropriate pair of 0-junctions (elements C, I, R, Se, Sf).
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.

These steps are discussed in detail below.

- Step 1: 0-junction insertion

The goal of this step is to insert a 0-*junction* at each node with distinct potential. The idea is to apply the transitive closure to every node on the graph (given only adjacency information, the transitive closure tells if there is a path between arbitrary nodes *x* and *y* in a graph). Calculating the transitive closure for every node guarantees that the 0-*junction* elements will be inserted correctly, no matter how connections between coupled elements were described on the Modelica input file (moreover considering that the user has a number of different possibilities to specify the parallel coupling between the elements of a circuit). This process is illustrated in Fig. 10, which shows two ways of defining the nodes in Fig. 8. Starting in V_n, a 0-*junction* is inserted on each serial connection.

- Step 2: 1-junction insertion

A 1-*junction* component is added to each 1-*port* element, inserting it between each corresponding pair of 0-junctions (the method supports *k-port* elements, adding *k* 1-*junction* elements to each *k-port* component). In this step, the nodes describing logical relations (dashed lines) are also merged, as the representation of these relations is no longer needed for the simulation process. This kind of edge is deleted, and the corresponding nodes joined, reducing the cardinality of the graph nodes and edge sets, as seen in Fig. 11.

Fig. 12 shows the graph obtained when this method is applied to graph in Fig. 10(c).

- Step 3: Assigning power direction

The power direction specifies the direction in which the power flow is assumed positive. A standard convention that assumes a positive direction when it flows out of the sources *Se* and *Sf* and into elements *C*, *I* and *R*. For two-port elements *TF* and *GY*, a power-in convention is used.

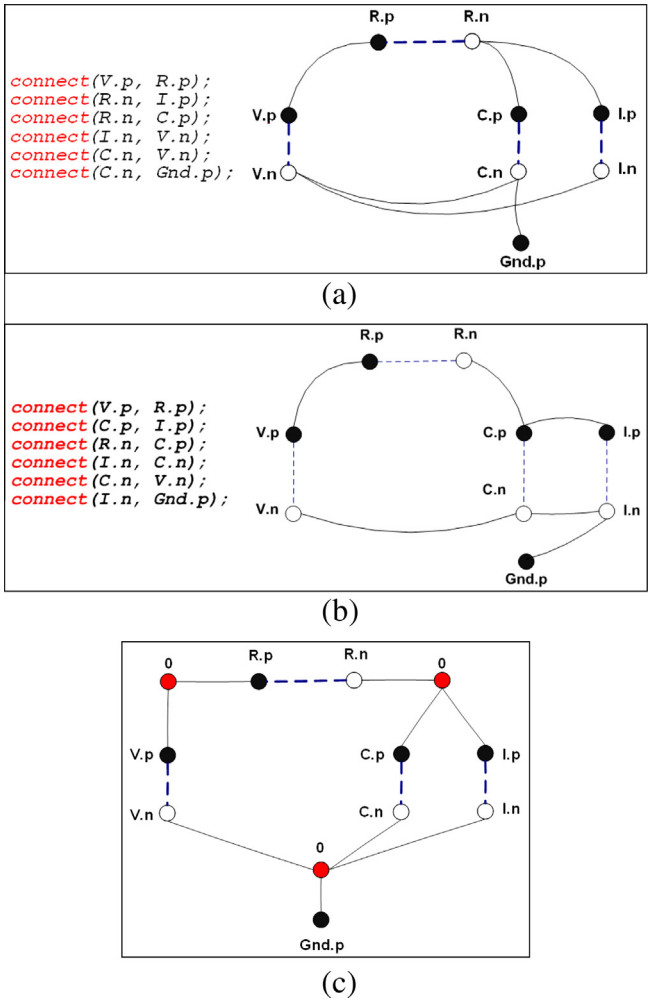


Fig. 10. 0-Junction insertion process (a) and (b): alternatives for Fig. 8. (c) Graph representation with 0-junctions inserted.

To assign power direction to all bonds, a propagation algorithm was used, which "extends" the power through the graph using the standard conventions and the information compiled from the Modelica model. Depending on the connections and port types (positive/negative), direction is inferred for bonds where no conventions apply. Once direction is assigned to all bonds, a directed BG is obtained (Figs. 13 and 14).

- Step 4: Deleting ground potential.

All the explicit ground potentials are deleted from the graph (as shown in Fig. 15); if no explicit ground potential is found, the 0-junction nearest to each source element is erased. The 0-junctions

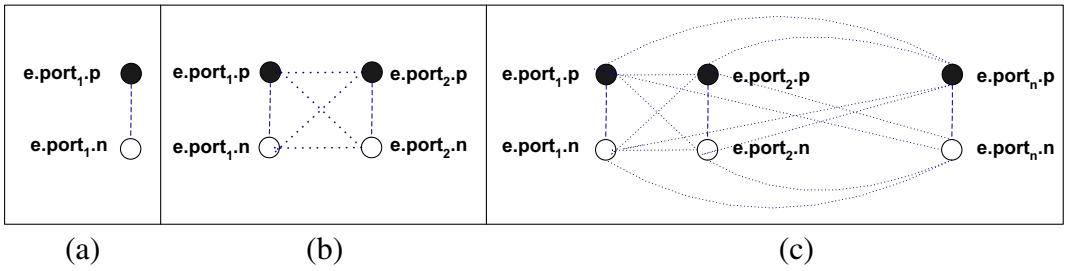


Fig. 9. Node representation of ports (a) one-port (b) two-port and (c) *k*-port.

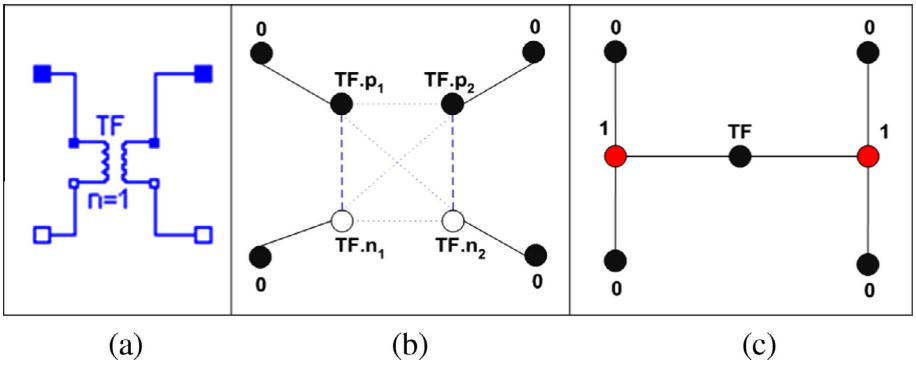


Fig. 11. (a) Transformer element (b) graph representation with 0-junctions insertion and (c) graph with 1-junctions insertion and no logical-linked nodes.

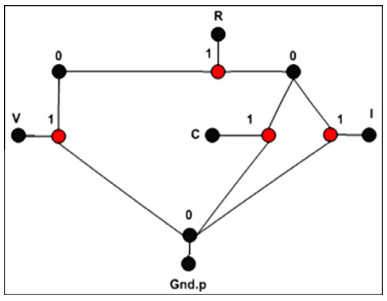


Fig. 12. 1-Junctions insertion and logical-linked nodes merging.

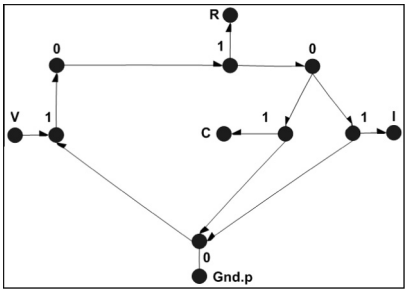


Fig. 14. Power direction assigned to the graph in Fig. 12.

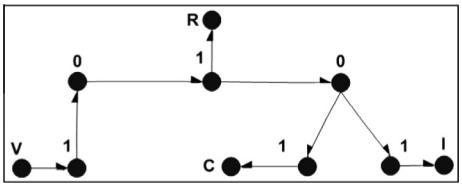


Fig. 15. Ground potential elimination for the graph in Fig. 14.

selected are only those associated with the negative pin of every source port.

- `Step 5: BG simplification.`

The BG is simplified applying the following rules:

- A junction between two bonds with through power direction can be left out from the graph.
- A bond connecting two junctions of the same type can be deleted and the junctions joined.

When these simplification rules are applied to the example graph in Fig. 15, the result in Fig. 17 is obtained (see Fig. 16).

3.3. Bond graph validation

After the BG is constructed and simplified, different error detection techniques are applied to the resulting BG. The first one, causalization, is the process where the signal direction of the bonds is determined.

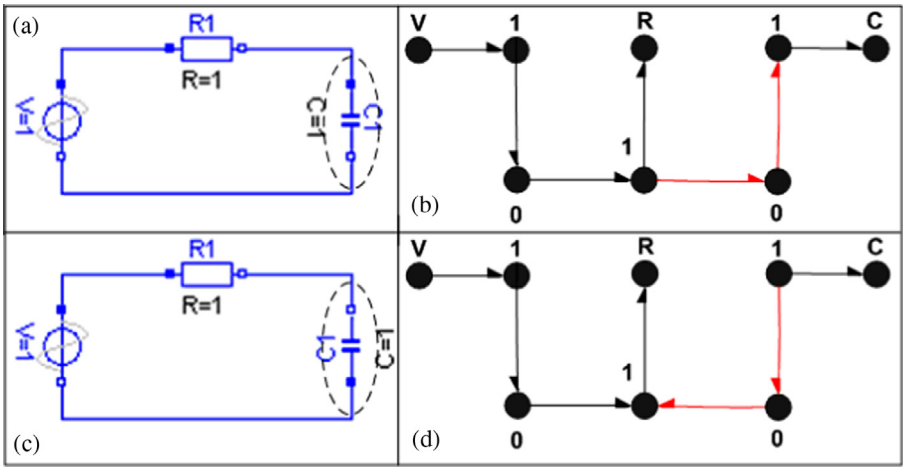


Fig. 13. (a) and (b) Electrical circuit and its directed BG representation (c) and (d) capacitor change reflected on the bonds power direction.

Once this process finishes, each bond can be interpreted as a bi-directional signal flow and the causal BG can then be seen as a compact block diagram. A port can impose four different causal constraints to its connected bonds, depending on its base 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: this is a relation between the causalities of the different ports within the components defining the 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 TF elements, one bond has *effort causality* and the other *flow causality*; at GY elements, both bonds have either *effort* or *flow causality*.
- Preferred causality: on storage elements, it determines whether integration or differentiation (with respect to time) will be used. Integration has preference, representing the *preferred causality*. Likewise, the preferred causality at *C* elements 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 by Karnopp and Rosenberg assigns causality to the bonds of a given BG [29]. This method starts by choosing a fixed causality element (i.e., a source), and then it propagates the assignment through structural components (junctions, transformer and gyrator) whenever it is possible, according to the causality restrictions. Once all sources have been processed, a storage element (*C* or *I*) is selected, and the preferred causality is applied. The process is repeated until all storages have their causalities assigned. If the graph is not completely causal, the iteration is repeated with a resistor (*R*). If the graph is still not causal, the model contains algebraic loops.

The causality assignment process in M/CD++ was almost totally based on the SCAP method. The algorithm above was developed with some restrictions, in order to check and inform structural conflicts within the model. Given the problems of differentiation algorithms (i.e. infinite outputs for a step input function) only the preferred causality was implemented for storage elements. *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 (see Fig. 18).

Whenever structural singularities or algebraic loops are discovered, the processing stops and the exception is informed to the modeler. As mentioned, only integral causalities are assigned to storage components. Then, for structural singularities (i.e. coupled storages), one of the elements should be assigned the derivative causality. The component causing the preferred causality violation can be used to detect *dependent storages*, which do not represent a state variable of the system. An example of such a model with a structural singularity is shown in Fig. 19. For this error, the circuit can thus be modified by adding a dissipater with a very low resistance value.

Algebraic loops are found by inspection of closed causal paths (i.e., paths defined by bonds with the same causal orientation that not include storages or sources). If at least one algebraic loop exists in the closed causal path, an inspection algorithm implemented within M/CD++ allows the listing of the resistance elements defining the loop (as seen in Fig. 20).

Fig. 21 shows a model with an algebraic loop and the modified circuit with a storage element to break the loop.

4. Bond-Graph library

After the compiler finishes successfully with steps discussed in Section 3, a model representing the original Modelica code converted into its causal, error free BG representation is obtained. Now, this model needs to be converted into an executable. To do so, a library of Bond-Graphs was developed using DEVS [21]. This library was defined based on the concepts of GDEVS, QDEVS and QSS. The library provides modular means for building and simulating BG using DEVS, allowing code reuse and the various advantages discussed in Section 1.

The first step is the transformation of the BG into an equivalent Quantized Bond Graph model (QBG). A QBG represents an approximation of the standard BG that is thus suitable for discrete-event simulation where all the storages and sources are quantized elements [30]. For instance, a quantized capacitor (or inertia) needs its displacement related to the effort (or flow) via a quantization

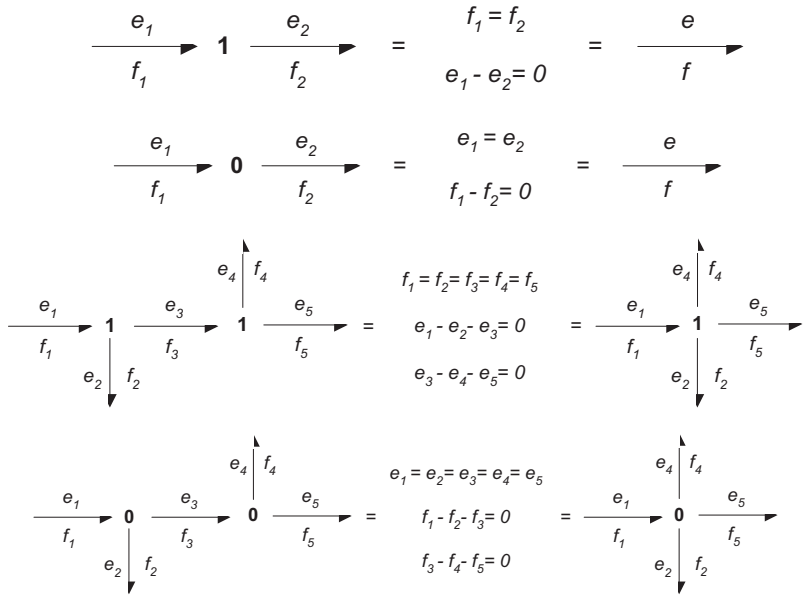


Fig. 16. Simplification examples: Rule I and Rule II.

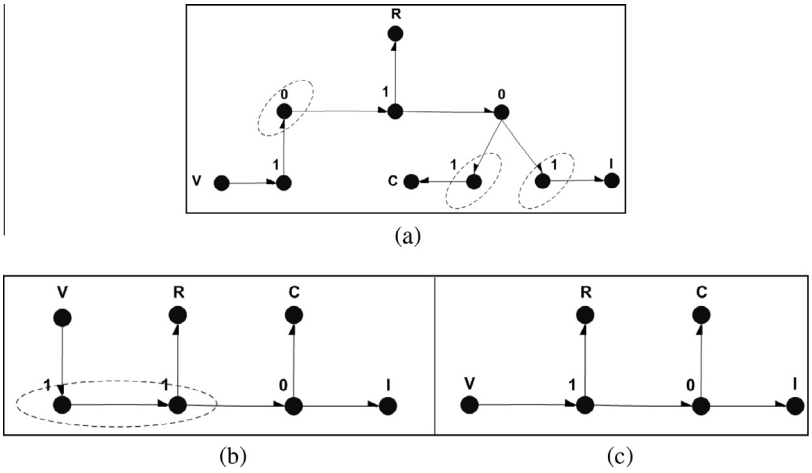


Fig. 17. Simplification rules applied to the BG in Fig. 15 (a) Simplification by rule I, (b) simplification by rule II, and (c) resulting BG.

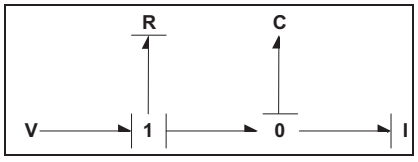


Fig. 18. Causality assignment to the BG in Fig. 17.

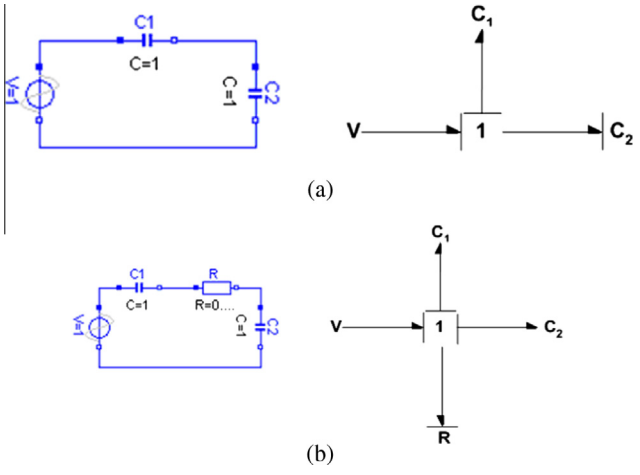


Fig. 19. (a) Circuit with coupled capacitors; and (b) problem elimination.

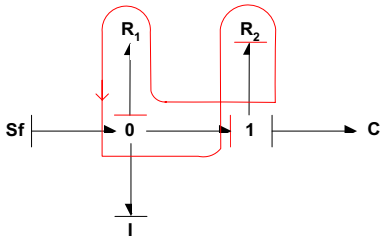


Fig. 20. Loop in closed casual path with resistance elements.

function with hysteresis. Likewise, a quantized effort (or flow) source has a piecewise constant trajectory and a bounded function. QBG can be modeled as a DEVS coupled model in which each component of the QBG is an atomic DEVS. As proved in [30], QBG can be

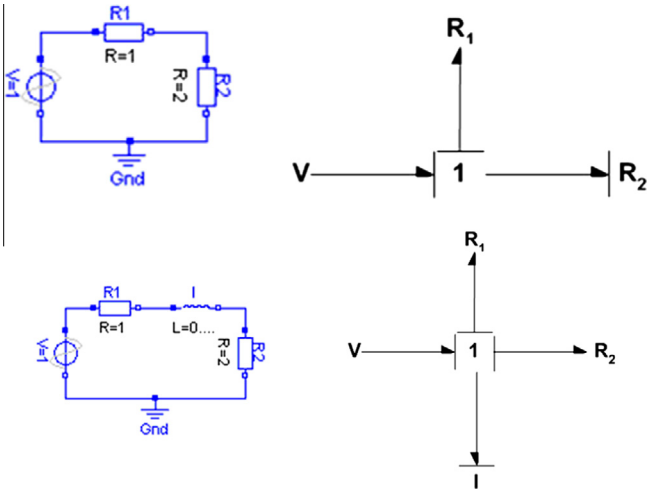


Fig. 21. (a) Electrical circuit with an algebraic loop between *R*1 and *R*2 and BG representation and (b) storage insertion to break the algebraic loop.

represented exactly by DEVS, which can exactly model systems whose input and output trajectories are piecewise constant. QSS error, convergence and stability properties are valid on QBG models.

Based on these ideas, all BG primitive elements were built as a library of atomic DEVS models whose elements (some of which are shown in Appendix II) are the following:

- BG: this abstract class, the base for primitive BG elements, introduces basic functionality to add bonds to components.
- Bond: they connect different components, modeling the exchange of power between them. Every bond is associated with an input port and an output port, which must transport the effort and flow values between components. Different attributes specify the power direction and the causality restrictions.
- *Quantizer:* it represents the output trajectories as piecewise constant segments through a quantization function. Two types of quantizers were implemented: *uniform* (which specifies a fixed quantum size for all intervals) and *non-uniform* (or *intervals*, which allows different interval lengths). A quantizer with hysteresis was also included.
- *Integrator:* it outputs the integrated value of its input. It implements the Euler integration algorithm, a first order method that provides numerical solution to integral calculus.

- *Resistance*: it calculates the effort value according to the resistance equation: *effort* = *R.flow*, with *R* a resistance constant, and a *flow* received from its adjacency. The model's internal transition implements a GDEVS polynomial approximation of the continuous *effort* function. To do so, the instants of input arrivals $t_1,\ldots,t_n$ are associated to tuples (a_i,b_i) whose values correspond to the coefficients used to approximate the effort curve by a polynomial function: $effort(t) = a_i.t + b_i \ \forall\ t \in \langle t_i,t_j\rangle$.
- *Capacitor*: it models the static relation between effort and displacement. All storage elements, including Capacitors, have a preferred causality. The equation defined by the Capacitor element can be expressed in two forms: $effort = \frac{1}{C}\int_{-\infty}^{t} flow\,dt$ or $flow = C\frac{deffort}{dt}$, where *C* is the Capacitor constant. A capacitor defines the following relation between the power variable *e* (*effort*) and the energy variable *q* (*displacement*): $e(t)-g(q(t)) = 0$; $q(t)-f(t) = 0$. The integral causality assignment makes the 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, the function is transformed in $e(t) = g(q_q(t))$, with $q_q(t)$ the quantized version of $q(t)$ [30]. This equation can be rewritten 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 and a continuous function). The same reasoning can also be applied to the inertias, which can be replaced by their quantized version.
- *EffortSource* and *FlowSource*: these components generate signals according to a given frequency. *EffortSource* sends the effort through the output port, while the *FlowSource* sends the flow value. Several types of output signals were implemented, providing varied functions to use in different contexts: *Constant, Step, Ramp, Sine, ExpSine, Exponential, and Pulse*.
- *Inductor*: it defines the static relation existing between flow and momentum. The following preferred equation was used: $flow = \frac{1}{L}\int_{-\infty}^{t} effort\,dt$ where *L* corresponds to the Inertial constant. The model transition functions are similar to those used for the Capacitor, but in this case, the Inductor load (flow) is calculated as the integral of effort value.
- *Transformer*: this model conserves power and it transmits the factors with the proper scaling defined by the transformer modulus. As this element has two bonds connected to it, both output effort and flow values must be calculated. The modulus equation defines the following relations: $f_j = r{\cdot}f_i$, and $e_j = (1/r){\cdot}e_i$, with *r* the transformer modulus and (e_i,f_i), (e_j,f_j) the (*effort, flow*) values transported by $bond_i$ and $bond_j$ attached to the component. New input effort data arriving at the component is processed in the external transition function and it is used to compute the outgoing effort according to the modulus relation.
- *Gyrator*: this model establishes the relationship between flow to effort and effort to flow, keeping the power unchanged. The relations are defined by: $e_j = \mu{\cdot}f_i$, $e_i = \mu{\cdot}f_j$ where μ is the gyrator modulus, and where (e_i,f_i) and (e_j,f_j) are the (*effort, flow*) values transported by $bond_i$ and $bond_j$ attached to the component.
- 0-*junction* (1-*junction*): it processes the arrival of new effort (flow) data in the model's external function, sending the value received through all the output effort (flow) ports. On the other hand, the arrival of new flow (effort) in one of the bonds generates the recalculation of the equation and flow (effort) is sent out through the output port.

For each of these models, a DEVS formal specification (which was used for verification) was defined, and a model was built in CD++ based on such specification. The simulation of the atomic DEVS models is based on the detection of effort value changes, as the input flow and the output effort are piecewise constant and the displacement trajectories are piecewise linear. This time is calculated as the distance of the displacement value to the next quantum crossing and the flow value, and it is associated to the time advance function. In the case of input flow variations, the time to the next effort change must be recalculated. The current displacement is computed according to the elapsed time and the previous flow value. Input flow changes are associated to *external events* on the DEVS model.

The following QBG components were implemented: QBGCapacitorFlowIn, QBGInductorEffortIn, QBGResistance FlowIn, QBGResistanceEffortIn, QBGSourceEffort_Constant, QBGSourceEffort_Step, QBGSourceEffort_Sine, QBGSource Effort_Pulse, QBGSourceFlow_Constant, QBGSourceFlow_ Step, QBGSourceFlow_Sine, QBGSourceFlow_Pulse, QBG Transformer, QBGGyratorFlowIn, QBGGyratorEffortIn, QBG SerialJunction and QBGParallelJunction. In order to show how this was done, the definition of the model QBGSourceEffort_Sine is described below (the remaining components were built using a similar approach, and details can be found in [21]). First, the formal definition of the model is presented, and then the implementation in CD++. QBGSourceEffort_Sine is an atomic DEVS component that models the BG source element generating the sine signal. Like the rest of the source DEVS models, it is simulated using QSS (quantization function with hysteresis) applied to signal function generator. The simulation calculates the time to the next quantum crossing, and uses this value as the time advance function ta (implemented by the holdIn function in CD++). The model generates a sine signal with angular frequency Ω, amplitude A and quantum value equal to $A{\cdot}d_u$.

$$SourceSine = \langle X, S, Y, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta\rangle$$

where $X = \varnothing;\ Y = \Re;\ S = \Re \times \Re^{+}$

$\delta_{\text{int}}(s) = \delta_{\text{int}}(\tau,\sigma) = (\tau',\sigma')$ where

$$\sigma' = \begin{cases} \frac{\arcsin[\sin(\omega\tau)+du]}{\omega} - \tau & if(\sin(\omega\tau)+du \leqslant 1 \wedge \cos(\omega\tau) > 0) \\ & \vee(\sin(\omega\tau)-du > -1) \\ \frac{\text{sgn}(\tau)\pi-\arcsin[\sin(\omega\tau)+du]}{\omega} - \tau & otherwise \end{cases}$$

$$\text{and}\quad \tau' = \begin{cases} \tau+\sigma' & if\ \omega(\tau+\sigma') < \pi \\ \tau+\sigma'-\frac{2\pi}{\omega} & otherwise \end{cases}$$

$$\lambda(s) = \lambda(\tau,\sigma) = (A\cdot\sin(\omega\tau), 1)$$

$$ta(s) = ta(\tau,\sigma) = \sigma$$

The model shown in Fig. 22 presents the implementation of the model in CD++.

As we can see, the initialization function `&QBGSource_Sine::initFunction` starts by computing function *s* (which implements the σ function defined in the formal specification *SourceSine* above). It then computes its quantized value, and triggers an internal transition immediately (*ta* with time = 0). As a consequence, the output function `&QBGSource_Sine::outputFunction` is activated, and this function it checks if the next quantum threshold has been passed. In that case, it transmits the results. Then, the internal transition function `&QBGSource_Sine::internalFunction` is activated, which computes function *s* again, following the formulas above. Then, the value obtained quantized, the time to cross the next threshold is computed and used as time advance. If the quantized state value does not change, a timeout is used (TICK_VALUE) to trigger a change if needed (for instance when the slope of the function equals 0).

Coupled components are built as DEVS models representing model's interconnections as follows:

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

- $X_{self} = Y_{self} = \{\varnothing\}$ no external inputs or outputs are used,

```
  qvalue = quantizer->quantize(value);
  holdIn(active, Time::Zero); // ta()=0
}

Model &QBGSource_Sine::internalFunction() {
    RealValue time = msg.time().asMsecs();
    RealValue v1, v2, t1, t2, t;

    RealValue s = signal->s(time); // compute sigma
    qvalue = quantizer->quantize(s);

    if(qvalue == qvaluePrev) // threshold not passed
      holdIn( active, Time(TICK_VALUE) ); //timeout
    else {        // compute thresholds
      v1 = s - quantum;  v2 = s + quantum;
      t1 = ceil(v1);     t2 = ceil(v2);          }

    value = signal->s(t);
    qvalue= quantizer->quantize(value);
    RealValue waitTime = (t - time) * TICK_VALUE;

    holdIn( active, Time(waitTime) );  //ta(s)
}


Model &QBGSource_Sine::outputFunction() {
  if(qvalue != qvaluePrev) { // threshold passed
    sendOutput(msg.time(), out, qvalue) ;
    qvaluePrev = qvalue;            }
}
```

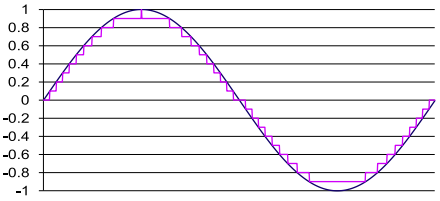


Fig. 22. *QBGSource_Sine* DEVS model and simulation execution.

- D is the set of all the BG elements components, and $\forall\ i \in D$,
- M_i is a DEVS atomic model representing a QBG component,
- IC is the internal coupling set defined as $IC = \{ice_{ui,vj}\} \cup \{icf_{vj,ui}\}$ 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).

$$ice_{ui,vj} = \begin{cases} ((u,out_{ei}),(v,in_{ej})) & \text{if } v \text{ isnotasource}(flowsource) \\ \varphi & otherwise \end{cases}$$
$$\text{if } u \text{ isa} 1-junction \text{then } i = 1$$
$$(only\ one\ effort-out\ port)$$

$$icf_{vj,ui} = \begin{cases} ((v,out_{fj}),(u,in_{fi})) & \text{if } u \text{ is not a source } (effortsource) \\ \varphi & otherwise \end{cases}$$
$$\text{if } v \text{ is a } 0-junction \text{ then } j = 1$$
$$(only\ one\ flow-out\ port)$$

- *select*: the *tie-breaking function,* gives priority to the structural components (junctions, transformer and gyrator).

Given the definition of the coupled DEVS model associated to a QBG, M/CD++ generates a coupled model using the specification language supported by CD++. To generate this specification file, the compiler executes the following steps:

1. For each component u of the QBG, add u to the *component* section of the model file. Considering the assigned causalization, select the valid implementation class for the component (as discussed earlier, two implementations are available for components with both causality types)
2. For each bond $b = (u,v)$ of the QBG, generate the coupling information between u and v on the *links* section in the model file. Follow the coupling definitions formalized above.
3. For each component u of the QBG, generate the component's configuration information in the *parameterization* section on the model file.

The process is illustrated in Figs. 23 and 24.

Fig. 23 shows a simple circuit in Modelica, and Fig. 24 the CD++ coupled model file generated by the compiler, using the model structure generated based on the BG specification.

The compiler uses the models and generates a coupled model structure following the procedures presented in Sections 3 and 4. In this case, it generates three atomic models: `$SJ1`, an instance of `@QBGSerialJunction R1`, an instance of `QBGResistanceEffortIn`, and `V`, an instance of `QBGSourceEffort_Sine` (these three models are in the library). Then, the models are connected through their input/output ports (defined by the *link* clauses). The model's parameters are extracted from the Modelica specification and converted into CD++ arguments for the coupled model. For instance, we can see that the Source generates a Sine signal with amplitude of 15 V, and a frequency of 20 ms (based on the original specification of 50 Hz). Finally, the discrete event simulator CD++ is internally invoked from M/CD++ and the simulation results generated.

5. M/CD++ execution examples

In this section, the simulation results of executing different electrical circuits using M/CD++ on an Intel workstation is presented (numerous tests were carried out; further details can be found in [31–33,26]). The goal of this section is to show that the results obtained by M/CD++ are consistent with the real behavior of the circuits studied.

Our first example considers the simulation of the electrical circuit with sine voltage presented in Figs. 23 and 24 (where V generates a sine voltage). After the model is compiled and executed, the following results were obtained. Fig. 25(a) shows the voltage on source V (a sine wave with amplitude 15 V) and Fig. 25(b) shows the voltage and current on resistor R. The resistor is a passive element in this oscillating circuit, so the current and the voltage on the resistor are in phase; the amplitude of the current is $I = V/R1$.

The example in Fig. 26 shows a model of an electrical circuit with pulse voltage. The structure of the model is similar to the one presented in Fig. 23 using a different voltage generator and including a capacitor.

After the complete model is generated and executed, we obtain the results shown in Fig. 27. The resistor is a passive circuit element and, as in previous example, its current is $I = V/R1$. The voltage on the capacitor increases until the maximum value (10 V), and when it is fully charged, the current becomes zero. The voltage source provides only positive pulses, so the capacitor will not be discharged.

The following example includes an inductor added to the original circuit, using a pulse voltage (see Fig. 28).

The inductor is an active circuit element. When the current varies from zero to its nominal value (transient regime of the circuit), there is an induction phenomenon which generates voltage in this transition stage. Therefore, the voltage on the inductor varies in the (−6 V, 6 V) interval while the source voltage V varies in the interval [0 V, 10 V].

In order to validate the simulation results, exhaustive tests were carried out, comparing the results to those obtained with

```
model circuit
  Modelica.Electrical.Analog.Sources.SineVoltage
        V(V=15,freqHz=50);
  Modelica.Electrical.Analog.Basic.Resistor R1(R=10);
  Modelica.Electrical.Analog.Basic.Ground Gnd;

equation
  connect(V.p, R1.p);
  connect(R1.n, V.n);
  connect(R1.n, Gnd.p);
end circuit;
```

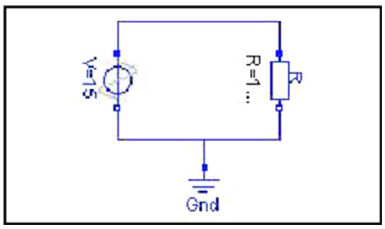
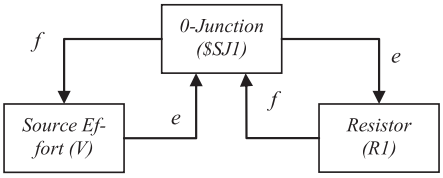


Fig. 23. Electrical circuit model and representation in Modelica.




```
components : $SJ1@QBGSerialJunction R1@QBGResistanceEffortIn V@QBGSourceEffort_Sine
link : f2p@$SJ1  f1n@V    link : e1n@V  e2p@$SJ1
link : e1n@$SJ1  e1p@R1   link : f1p@R1 f1n@$SJ1

[R1] R :     10000

[V]
quantum : 0.1       hystWindow : 0.01
signal : Sine       startTime : 000
amplitude : 015     wavefreq: 20
```

Fig. 24. Coupled model representation in CD++ notation.

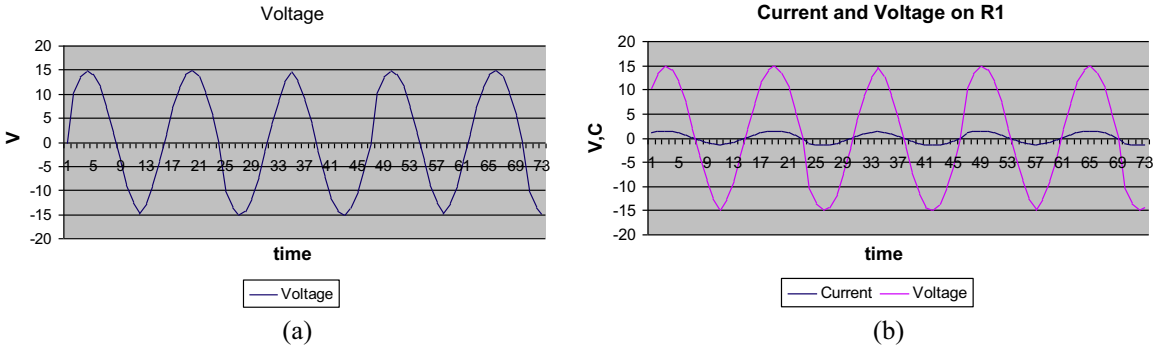


Fig. 25. Current/voltage on *V*/*R*1.

```
model circuit
    Modelica.Electrical.Analog.Sources.PulseVoltage
          V(V=10,width=50,period=2);
    Modelica.Electrical.Analog.Basic.Resistor R1(R=10);
    Modelica.Electrical.Analog.Basic.Capacitor C(C=15);
    Modelica.Electrical.Analog.Basic.Ground Gnd;

  equation
    connect(V.p, R1.p);
    connect(R1.n, C.p);
    connect(C.n, V.n);
    connect(V.n, Gnd.p);
  end circuit;
```

Fig. 26. Electrical model in Modelica: pulse voltage and capacitor.

Dymola [34], a commercial toolkit for complex physical system M&S with Modelica support. The idea was to compare the results of M/CD++ (which took approximately nine person-months to be developed) with those obtained by a fully featured commercial tool. The results obtained show that, besides the various advantages discussed in Section 1, a DEVS-based approach can also improve the development of a complex tool while having minimum error.

The test cases presented here were executed using both M/CD++ and Dymola, varying simulation parameters in order to compare their results and calculate the error between them. The objective of this comparison is not to discuss performance or precision of the results, neither the qualitative or quantitative results, but to show the final results obtained when executing all the steps in the tool chain, and to show the validity of the tool results when compared with a commercial tool with thousands of person-hours of development. Numerous examples were presented in [31–33,26] (only one example is introduced due to space limitations) (see Fig. 29).

The first test presented here shows the execution of the circuit using the DASSL integration method on Dymola for 15 s of

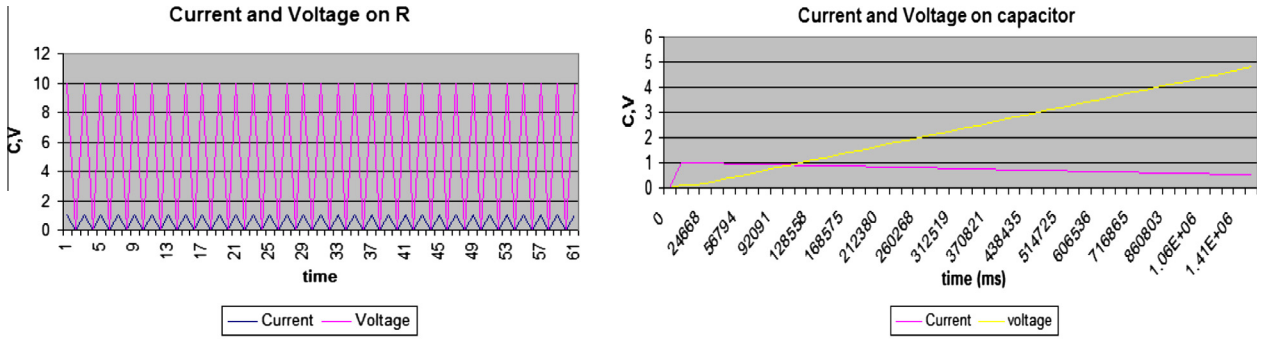


Fig. 27. Electrical model with pulse signal: (a) current and voltage on R1; and (b) current and voltage on the capacitor.

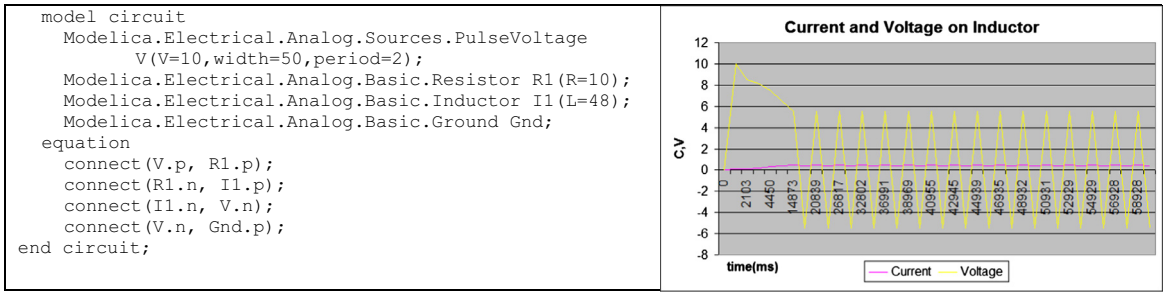


Fig. 28. Pulse signal and inductor: current and voltage on the inductor.

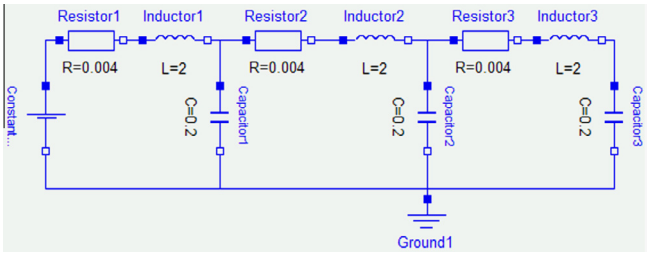


Fig. 29. An oscillating electrical circuit.

simulation time (using intervals of 500 time units, and a tolerance of 0.0001). The results were compared with those obtained by M/CD++ using a quantum size $q = 1.6$ for Capacitors (with a hysteresis window of 0.5) and $q = 0.5$ for Inductors (with a hysteresis window of 0.1). Fig. 30 shows the state trajectories on capacitor C1 and inductor I1 for the given simulation parameters.

As seen in the figure, the error produced is minimum (the highest error was obtained when values are close to zero, because, as the quantum size used during the simulation is a fixed value for all the points on the curve, smaller values will produce greater relative errors; this can easily be improved with QSS of higher order). The error curve decreases when time advances. The results highly improve when decreasing the quantum and hysteresis window size on M/CD++.

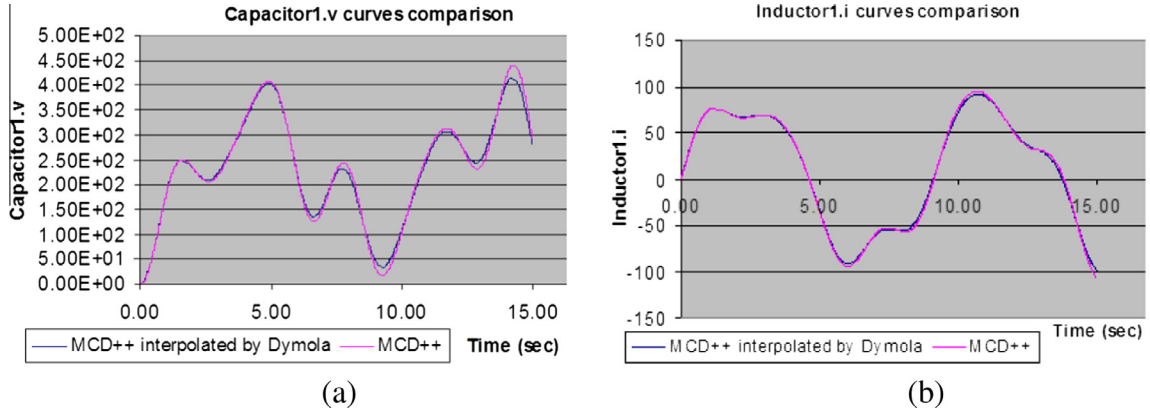


Fig. 30. Voltage curve on capacitor 1 and current curve on inductor 1.

6. Conclusion

The DEVS formalism is a methodology introduced for M&S of discrete event systems. During the last decade the DEVS theory has evolved, and new methods allow DEVS simulation of continuous and hybrid systems. The design of a tool for M&S of continuous systems was introduced. Models are described using Modelica, a modular and causal standard specification language for physical system modeling and then translated into DEVS. Examples of model simulation with their execution results were included. The simulation results generated by M/CD++ were compared with those produced by the commercial simulation environment *Dymola*. Several test cases were executed using both toolkits, varying the quantization parameters used on M/CD++ and the integration methods utilized by Dymola. It was shown that a higher relative error is obtained for slopes near to zero on a trajectory. This is related with the fixed quantum size used by the quantization function over a state trajectory, and it can be solved with higher order methods like QSS2 or QSS3, as M/CD++ approximates the system solution based on the QSS method, which uses a simple first order integration approach. Most of the results produced by M/CD++ 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 produces more accurate solutions and decrease error.

In the long term, new libraries will be built in order to make it easy to use the 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 the discrete event paradigm, thus allowing the development of a simulation environment for complex systems, modeled as hybrid systems, where all paradigms merge (continuous time, discrete time, and discrete event).

Appendix I. M/CD++ components UML class diagram

See Fig. A1.

Appendix II. Bond graph class hierarchy

See Fig. A2.

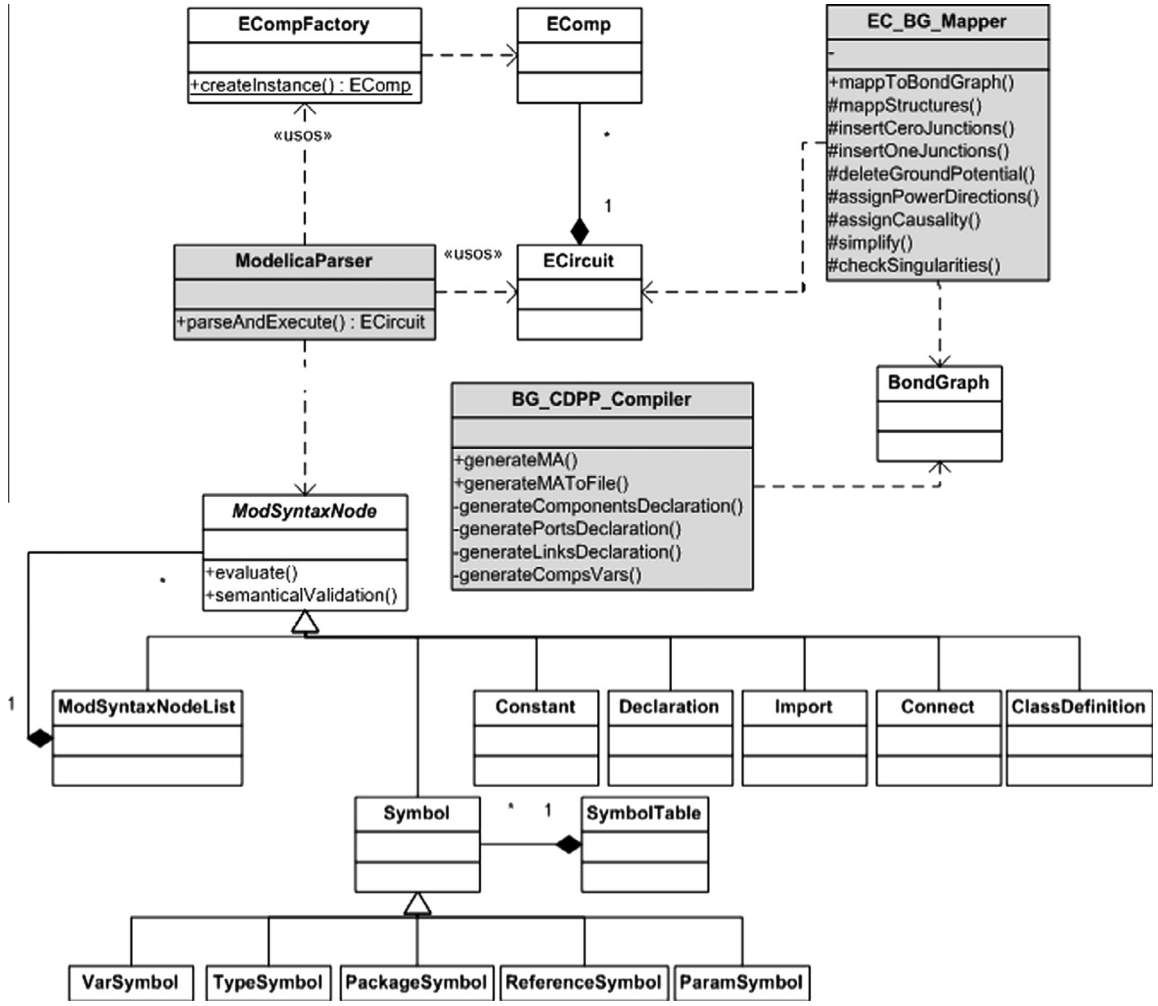


Fig. A1.

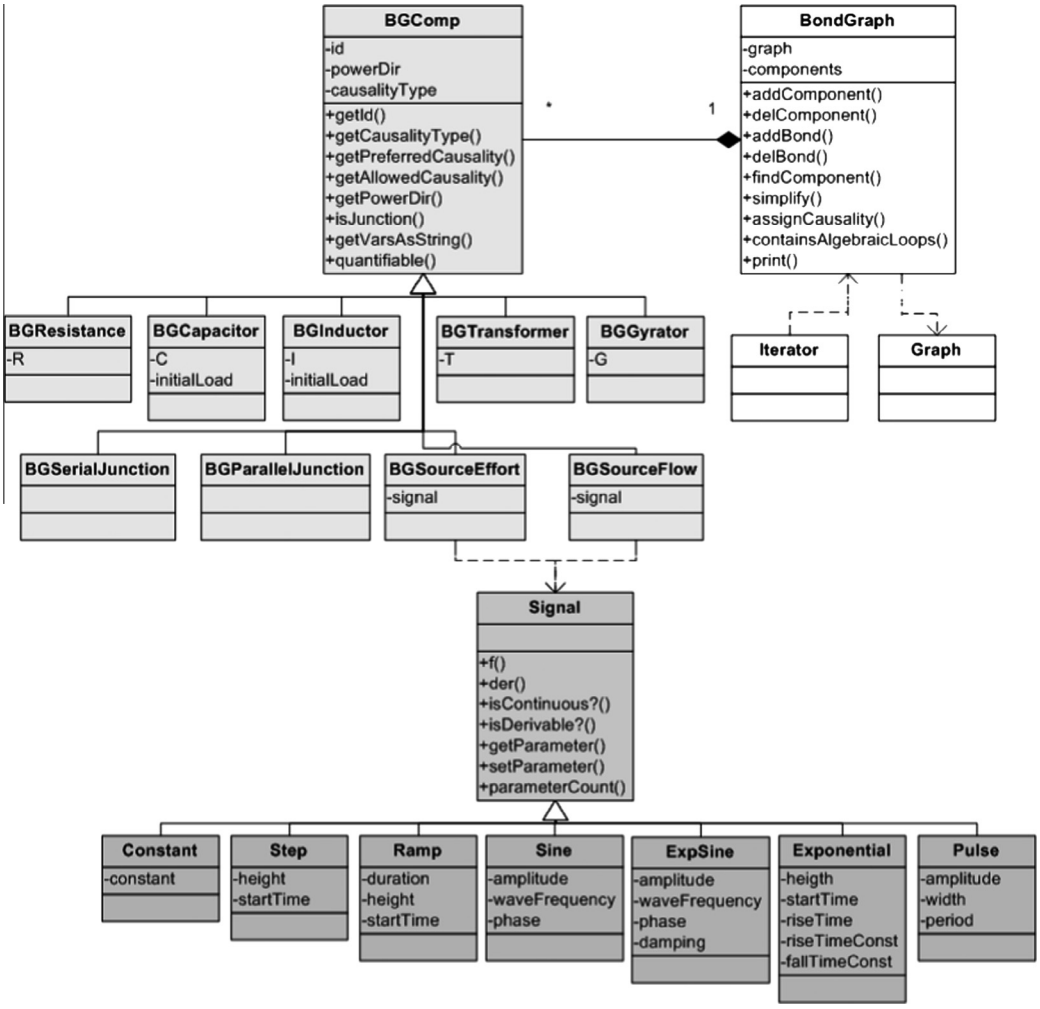


Fig. A2.

References

[1] Gilat A. MATLAB: an introduction with applications. 2nd ed. John Wiley & Sons; 2004.
[2] Wolfram S. Mathematica: a system for doing mathematics by computer. 2nd ed. Redwood City (CA, USA): Addison Wesley; 1991.
[3] Zimmerman W. Multiphysics modeling with finite element methods. River Edge (NJ): World Scientific Publishing Co., Inc.; 2006.
[4] Heck A. Introduction to maple. Springer; 1996.
[5] Taylor M. Partial differential equations: basic theory. NY: Springer; 1996.
[6] Press WH, Flannery BP, Teukolsky SA, Vetterling WT. Numerical recipes. Cambridge (MA): Cambridge University Press; 1986.
[7] Zeigler B, Kim T, Praehofer H. Theory of modeling and simulation. 2nd ed. New York: Academic Press; 2000.
[8] Zeigler BP. Continuity and change (activity) are fundamentally related in DEVS simulation of continuous systems. LNCDS, vol. 3397. NY: Springer-Verlag; 2005. p. 1–17.
[9] Zeigler B. DEVS theory of quantization. In: DARPA Contract N6133997K-007. Tucson (AZ): ECE Dept., University of Arizona; 1998.
[10] Kofman E. Discrete event based simulation and control of continuous systems. Simulation 2003;79(7):363–76.
[11] Modelica language specification. Version 2.1. <http://www.modelica.org>. [accessed: March 2010].
[12] Wainer G (. Discrete-event modeling and simulation: a practitioner's approach. Boca Raton (Florida): CRC press, Taylor & Francis; 2009.
[13] Åström KJ, Elmqvist H, Mattsson SE. Evolution of continuous-time modeling and simulation. In: 12th European simulation multiconference, Manchester, UK; 1998.
[14] Pantelides C. The consistent initialization of differential algebraic systems. SIAM J Sci Stat Comp 1988;9:213–31.
[15] Fábián GD, van Beek DA, Rooda JE. Substitute equations for index reduction and discontinuity handling. In: Proc. of the third IMACS symposium on mathematical modeling, Vienna, Austria; 2000.
[16] Cellier FE, Elmqvist H. Automated formula manipulation supports object-oriented continuous-system modeling. IEEE Control Syst 1993;13(2):28–38.
[17] Geuder DF. Object oriented modeling with SIMPLE++. In: Proceedings of winter simulation conference, Arlington, VA; 1995.
[18] Roberts C, Dessouky Y. An overview of object-oriented simulation. Simulation 1998;70:359–68.
[19] Samantaray A. About bond graph-the system modeling world [online]. <http://www.bondgraph.info/about.html.> [accessed April 2010].
[20] Kofman E, Junco S. Quantized state systems. A DEVS approach for continuous system simulation. Trans SCS 2001;18(3):123–32.
[21] D'Abreu M, Wainer G. Defining hybrid system models using DEVS quantization techniques. In: Proceedings of the winter simulation conference. New Orleans, LA, USA; 2003.
[23] Nutaro James J, Zeigler Bernard P, Jammalamadaka R, Akerkar S. Discrete event solution of gas dynamics within the DEVS framework. In: International Conference on Computational Science; 2003. p. 319–28
[24] Bolduc Jean-Sébastien, Vangheluwe Hans. Mapping ODEs to DEVS: adaptive quantization. In: Summer computer simulation conference, Montréal, Canada; 2003. p. 401–7
[25] Giambiasi N, Escude B, Ghosh S. GDEVS: a generalized discrete event specification for accurate modeling of dynamic systems. Trans SCS 2000;17(3):120–34.
[26] D'Abreu M, Wainer G. M/CD++: modeling continuous systems using Modelica and DEVS. In: Procedings of IEEE/ACM MASCOTS 2005, Atlanta, GA; 2005.
[27] Sanz V, Urquia A, Cellier FE, Dormido S. Modeling of hybrid control systems using the DEVSLib Modelica library. Control Eng Pract 2012;20(1):24–34. 2011.
[28] GNU. Introduction to Bison [online]. <http://www.gnu.org/software/bison/bison.html.> [accessed 26.04.10].

[29] Karnopp D, Margolis D, Rosenber R. System dynamics: a unified approach. Wiley; 1990.
[30] Kofman E, Junco S. Quantized bond graphs: an approach for discrete event simulation of physical systems. In: Proceedings of ICBGM'01. Phoenix, Arizona, Jan. 2001; p. 369–74.
[31] D'Abreu M, Wainer G. Experimental results on the implementation of Modelica using DEVS modeling and simulation. In: Proceedings of SpringSim 2006 (DEVS Symposium). Huntsville (AL) USA; 2006.
[32] Chechiu L, Wainer G. Experimental results on the use of Modelica/CD++. In: Proceedings of the 2005 SCS summer computer simulation conference (student workshop). Philadelphia, PA; 2005.
[33] Sanz V, Jafer S, Wainer G, Nicolescu G. Hybrid modeling of opto-electrical interfaces using DEVS and Modelica. In: Proceedings of SCS/ACM Springsim 2009 (DEVS symposium). San Diego, CA. USA; 2009.
[34] Dymola. <http://www.3ds.com/products/catia/portfolio/dymola>. accessed July 2010.
[35] Kofman E, Cellier FE, Migoni G. Continuous system simulation and control. In: Wainer GA, Mosterman PJ, editors. Discrete event simulation and modeling: theory and applications. Boca Raton (FL): CRC Press; 2010. p. 75–107.
[36] Giambiasi N, Escude B, Ghosh S. Generalized discrete event simulation of dynamic systems. Trans. Soc. Comput. Simul. Int. 2001;18(4):216–29.