Simulation

Simulation: Transactions of the Society of Modeling and Simulation International
87(1-2) 113–132
© The Author(s) 2010
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/0037549710368029
sim.sagepub.com

PowerDEVS: a tool for hybrid system modeling and real-time simulation

Federico Bergero and Ernesto Kofman

Abstract

In this paper we introduce a general-purpose software tool for discrete event system specification (DEVS) modeling and simulation oriented to the simulation of hybrid systems. The environment, called PowerDEVS, allows atomic DEVS models to be defined in C++ language that can then be coupled graphically in hierarchical block diagrams to create more complex systems. The environment automatically translates the graphically coupled models into a C++ code which executes the simulation. A remarkable feature of PowerDEVS is the possibility to perform simulations under a real-time operating system (RTAI) synchronizing with a real-time clock, which permits the design and automatic implementation of synchronous and asynchronous digital controllers. Combined with its continuous system simulation library, PowerDEVS is also an efficient tool for real-time simulation of physical systems. Another feature is the interconnection between PowerDEVS and the numerical package Scilab. PowerDEVS simulations can make use of Scilab workspace variables and functions, and the results can be sent back to Scilab for further processing and data analysis. In addition to describing the main features of the software tool, the article also illustrates its use with some examples which show its simplicity and efficiency.

Keywords

discrete event system specification (DEVS), hybrid systems, real-time simulation, simulation software

1. Introduction

The discrete event system specification (DEVS)[1] is the most general formalism for discrete event system modeling. It allows any system to be represented provided that it performs a finite number of changes in finite intervals of time. Thus, not only Petri nets, state charts, event graphs and other discrete event languages but also all discrete time systems can be seen as particular cases of DEVS.[2,3]

Taking into account that ordinary differential equations can be approximated by discrete time systems, using numerical integration methods, and that these systems are particular cases of DEVS, it results that DEVS can also approximate continuous systems. Moreover, there are numerical methods, such as the quantized state system (QSS) family,[4] which produce simulation models that cannot be represented in discrete time but only as DEVS models.

Thus, simulation tools based on DEVS are potentially more general than tools for different discrete formalisms, including the popular continuous time ones as Simulink[5] (Matlab), Scicos[6] (Scilab), etc.

Among the existing DEVS simulation tools we should mention DEVS–Java,[7] DEVSim++,[8] DEVS–C++,[9] CD++,[10] and JDEVS.[11]

These software tools offer different features which include graphical interfaces and advanced simulation features for general-purpose DEVS models and for some specific domains. However, they were developed before the mentioned discrete event methods for numerical integration of ordinary differential equations (ODEs).

On the one hand, these methods produce DEVS models with some very particular features. In fact, they are usually composed of several atomic DEVS models

Laboratorio de Sistemas Dinámicos, FCEIA–UNR, CIFASIS–CONICET, Riobamba 245 bis, 2000 Rosario, Argentina.

Corresponding author:
Federico Martin Bergero, Laboratorio de Sistemas Dinámicos, FCEIA–UNR, CIFASIS–CONICET, Riobamba 245 bis, 2000 Rosario, Argentina
Email: bergero@cifasis-conicet.gov.ar

which belong to two basic classes: quantized integrators and static functions. The different quantized integrators differ from each other in only two parameters: the quantum and the initial state value. Similarly, different static functions only differ in some parameters as the gain, number of inputs, the calculated expression, etc.

On the other hand, these new numerical methods have many potential users outside the DEVS working community. In strongly discontinuous systems the QSS methods offer solutions which are sensibly better than existing numerical algorithms[12] and they are starting to be used by continuous system simulation people. Unfortunately, most researchers and users of numerical ODE integration methods do not know about DEVS and they would appreciate to use the DEVS-based methods without learning DEVS. Moreover, they would be happier if the software they have to use looks similar to the software they use for conventional numerical methods (Simulink, Dymola, Scicos, etc.).

Taking into account these remarks, a DEVS simulation environment with library handling capabilities and a block-oriented graphical interface such as Simulink where parameters can be changed without modifying the blocks and where the atomic DEVS definitions are *hidden* for non-DEVS users appears as an appropriate solution for hybrid system simulation.

These ideas motivated the development of a general-purpose DEVS simulation software oriented to hybrid system simulation. Here, we call any system involving simultaneous continuous (represented by ODEs) and discrete (represented by DEVS) dynamics a *hybrid system*.

This software package, called PowerDEVS, was conceived to be used by DEVS expert programmers as well as by end users who only want to connect predefined blocks and simulate. The tool, initially developed as a Diploma Work[13] in our university, was then re-written and is currently maintained by the Laboratory for System Dynamics and Signal Processing.

PowerDEVS is composed of various independent programs:

- The *Model Editor*, which contains the graphical interface allowing the hierarchical block-diagram construction, library managing, parameter selection and other high-level definitions as well as providing the linking with the other programs.
- The *Atomic Editor*, which permits editing DEVS atomic models of elementary blocks by defining transition functions, output function, time advance, etc.
- The *Preprocessor*, which translates the model editor files into structure files which contain the coupling structure and the information to build up the simulation code, links the code of the different atomic models according to the corresponding structure file and compiles it to produce a stand-alone executable file which simulates the system.
- The *Simulation Interface*, which runs the stand-alone executable and permits to vary simulation parameters such as final time, number of simulations to perform, and the simulation mode (normal simulation, timed simulation, step-by-step simulation, etc.).
- A running instance of Scilab, which acts as a workspace, where simulation parameters can be read, and results can be exported to. This instance is a modification of Scilab 4.1.2 to support this type of operations.

All of these applications were programmed in C++ with the graphical libraries QT, with the exception of the *Model Editor* that was programmed in Visual Basic. We provide compiled versions of these applications for Linux and Windows, and all of the source code is available for download.

PowerDEVS also runs under a real-time operating system (Real Time Application Interface, RTAI[14]) synchronizing the events with a real-time clock with the capability of capturing interrupts at the atomic level. To this purpose, the DEVS simulation scheme of Zeigler et al.[1] was partially inverted so that simulators of atomic models can send messages to their parents informing about the next event time. In this way, PowerDEVS allows the direct implementation of asynchronous DEVS-based Quantized State Controllers[15] on a PC. In addition to the Linux and Windows distributions, we developed a modified Kubuntu distribution that includes the RTAI kernel and PowerDEVS.

The paper is organized as follows: Section 2 introduces the main concepts used in the rest of the article. Then, Section 3 describes the main features of PowerDEVS and Section 4 analyzes its real-time characteristics. Finally, Section 5 illustrates the usage of PowerDEVS in two examples (including a real-time control application) and Section 6 concludes the article.

2. Background

2.1. Devs

DEVS is a formalism that was first introduced by Bernard Zeigler.[16]

A DEVS model processes an input event trajectory and, according to that trajectory and its own initial conditions, it provokes an output event trajectory.

An *atomic* DEVS model is defined by the following structure:

$$M = (X, Y, S, \delta_{\mathrm{int}}, \delta_{\mathrm{ext}}, \lambda, ta),$$

where:

- X is the set of input event values, i.e. the set of all possible values that an input event can adopt;
- Y is the set of output event values;
- S is the set of state values;
- δ_{int}, δ_{ext}, λ and ta are functions which define the system dynamics.

Each possible state s ($s \in S$) has an associated *Time Advance* computed by the *Time Advance Function* $ta(s)$ ($ta(s) : S \to \Re_0^+$). The *Time Advance* is a non-negative real number saying how long the system remains in a given state in absence of input events.

Thus, if the state adopts the value s_1 at time t_1, after $ta(s_1)$ units of time (i.e. at time $ta(s_1) + t_1$) the system performs an *internal transition* going to a new state s_2. The new state is calculated as $s_2 = \delta_{\text{int}}(s_1)$. The function $\delta_{\text{int}}(\delta_{\text{int}} : S \to S)$ is called the *Internal Transition Function*.

When the state goes from s_1 to s_2 an output event is produced with value $y_1 = \lambda(s_1)$. The function λ ($\lambda : S \to Y$) is called the *Output Function*. In that way, the functions ta, δ_{int} and λ define the autonomous behavior of a DEVS model.

When an input event arrives the state changes instantaneously. The new state value depends not only on the input event value but also on the previous state value and the elapsed time since the last transition. If the system arrived to the state s_2 at time t_2 and then an input event arrives at time t_2+e with value x_1, the new state is calculated as $s_3 = \delta_{\text{ext}}(s_2, e, x_1)$ (note that $ta(s_2) > e$). In this case, we say that the system performs an *external transition*. The function δ_{ext} ($\delta_{\text{ext}} : S \times \Re_0^+ \times X \to S$) is called the *External Transition Function*. No output event is produced during an external transition.

The formalism presented is also called *Classic DEVS* to distinguish it from *Parallel DEVS*,[1] which consists of an extension of the previous version that was conceived to improve the treatment of simultaneous events.

Atomic DEVS models can be coupled. DEVS theory guarantees that the coupling of atomic DEVS models defines new DEVS models (i.e. DEVS is closed under coupling) and then complex systems can be represented by DEVS in a hierarchical way.[1]

Coupling in DEVS is usually represented through the use of input and output ports. With these ports, the coupling of DEVS models becomes a simple block-diagram construction. Figure 1 shows a coupled DEVS model N which is the result of coupling the models M_a and M_b.

According to the closure property, the model N can itself be used as an atomic DEVS and it can be coupled with other atomic or coupled models.

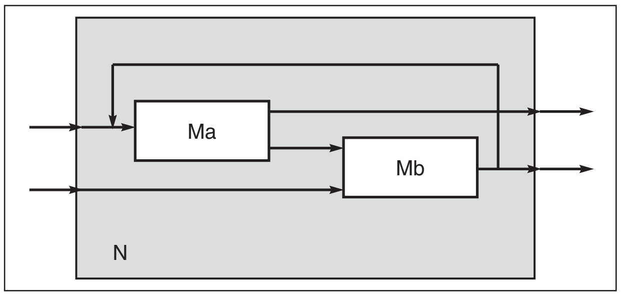


Figure 1. Coupled DEVS model.

2.2. Simulating a DEVS model

One of the most important features of DEVS is that very complex models can be simulated in a very easy and efficient way.

The basic idea for the simulation of a coupled DEVS model can be described by the following steps:

1. Look for the atomic model that, according to its time advance and elapsed time, is the next to perform an internal transition. Call it d^* and let tn be the time of the mentioned transition.
2. Advance the simulation time t to $t = tn$ and execute the internal transition function of d^*.
3. Propagate the output event produced by d^* to all of the atomic models connected to it executing the corresponding external transition functions. Then, go back to step 1.

One of the simplest ways to implement these steps is writing a program with a hierarchical structure equivalent to the hierarchical structure of the model to be simulated. This is the method developed by Zeigler et al.[1] where a routine called *DEVS-simulator* is associated with each *atomic DEVS model* and a different routine called *DEVS-coordinator* is related to each *coupled DEVS model*. At the top of the hierarchy there is a routine called *DEVS-root-coordinator* which manages the global simulation time.

Figure 2 illustrates this idea over a coupled DEVS model.

The simulators and coordinators of consecutive layers communicate with each other with messages. The coordinators send messages to their children so they execute the transition functions. When a simulator executes a transition, it calculates its next state and, when the transition is internal, it sends the output value to its parent coordinator. In all of the cases, the simulator state will coincide with its associated atomic DEVS model state.

When a coordinator executes a transition, it sends messages to some of its children so they execute their corresponding transition functions. When an output

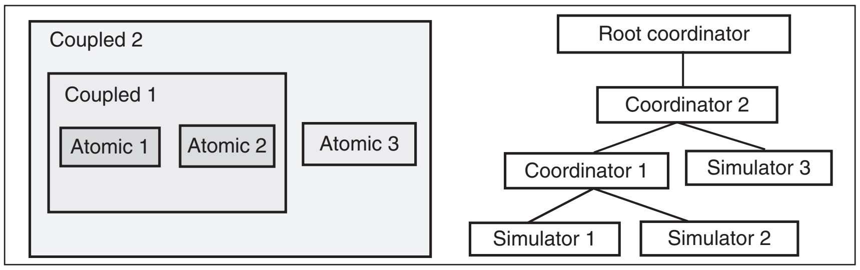


Figure 2. Hierarchical model and simulation scheme.

event produced by one of its children has to be propagated outside the coupled model, the coordinator sends a message to its own parent coordinator carrying the output value.

Each simulator or coordinator has a local variable *tn* which indicates the time when its next internal transition will occur. In the simulators, that variable is calculated using the time advance function of the corresponding atomic model. In the coordinators, it is calculated as the minimum *tn* of their children. Thus, the *tn* of the coordinator at the top is the time at which the next event of the entire system will occur. Then, the root coordinator only looks at this time, advances the global time *t* to this value and then it sends a message to its child so it performs the next transition, and then it repeats this cycle until the end of the simulation.

2.3. DEVS and hybrid systems simulation

Hybrid systems combine discrete and continuous dynamics. While discrete subsystems have a straightforward representation in DEVS, continuous submodels require some type of discretization.

Although DEVS can easily represent the discrete time approximations of continuous systems given by conventional numerical integration methods such as Euler, Runge–Kutta, etc., it can also represent the approximations resulting from state quantization. Quantization-based integration methods are noticeably efficient at simulating hybrid systems due to their ability to handle discontinuities.

2.3.1. Quantization-based integration.

Continuous-time systems can be written as set of ODEs:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)), \qquad (1)$$

where $\mathbf{x} \in \Re^n$ is the state vector and $\mathbf{u} \in \Re^m$ is a vector of known input functions.

The simulation system (1) requires using numerical integration methods. While conventional integration algorithms are based on time discretization, a new family of numerical methods was developed based on state quantization.[4] The new algorithms, called QSS methods, can approximate ODEs such as that of Equation (1) by DEVS models.

Formally, the first-order accurate QSS method approximates Equation (1) by

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{v}(t)), \qquad (2)$$

where each pair of variables q_j and x_j are related by a *hysteretic quantization function*.

The presence of a hysteretic quantization function relating $q_j(t)$ and $x_j(t)$ implies that $q_j(t)$ follows a piecewise constant trajectory that only changes when the difference with $x_j(t)$ becomes equal to a parameter Δq_j called *quantum*. The variables q_j are called *quantized variables*, and can be seen as a piecewise constant approximation of the corresponding state variables x_j.

Similarly, the components of $\mathbf{v}(t)$ are piecewise constant approximations of the corresponding components of $\mathbf{u}(t)$.

Since the components $q_j(t)$ and $v_j(t)$ follow piecewise constant trajectories, it results that the state derivatives $\dot{x}_j(t)$ also follow piecewise constant trajectories. Then, the state variables $x_j(t)$ have piecewise linear evolutions.

Each component of Equation (2) can be thought of as the coupling of two elementary subsystems, a static subsystem

$$\dot{x}_j(t) = f_j(q_1, \ldots, q_n, v_1, \ldots, v_m) \qquad (3)$$

and a dynamical subsystem

$$q_j(t) = Q_j(x_j(\cdot)) = Q_j\left(\int \dot{x}_j(\tau)\,\mathrm{d}\tau\right), \qquad (4)$$

where Q_j is the hysteretic quantization function (it is not a function of the instantaneous value $x_j(t)$, but a function of the trajectory $x_j(\cdot)$).

Since the components $v_j(t)$, $q_j(t)$ and $\dot{x}_j(t)$ are piecewise constant, both subsystems have piecewise constant input and output trajectories that can be represented by sequences of events.

Then, subsystems (3) and (4) define a relation between their input and output sequences of events. Consequently, equivalent DEVS models can be found for these systems, called *static functions* and *quantized integrators*, respectively.[4]

The piecewise constant input trajectories $v_j(t)$ can be also represented by sequences of events, and *source* DEVS models that generate them can be easily obtained.

Then, the QSS approximation Equation (2) can be simulated exactly by a DEVS model consisting of the coupling of n quantized integrators, n static functions and m signal sources. The resulting coupled DEVS model looks identical to the block diagram representation of the original system of Equation (1).

Based on the idea of QSS, a second-order accurate method was developed replacing the piecewise constant approximations by piecewise linear approximations. The method, called QSS2, can be implemented using DEVS in the same way as QSS. However, the trajectories are now piecewise linear instead of piecewise constant. Thus, the events carry two numbers that indicate the initial value and the slope of each segment. Also, the static functions and quantized integrators are modified with respect to those of QSS so they can take into account the slopes.

Following the idea of QSS2, the third-order accurate QSS3 method[17] uses piecewise parabolic trajectories. The family of QSS methods is completed with three methods for stiff systems (Backward QSS and Linearly Implicit QSS of order 1 and 2[18]) and a method for marginally stable systems (Centered QSS[19]).

2.3.2. QSS and hybrid systems.

The interaction of the continuous and discrete dynamics occurring in hybrid systems often implies that the right-hand side of the system of Equation (1), which models the continuous subsystems, contains discontinuities.

If a numerical integration method performs an integration step that crosses through a discontinuity, the result will have an unacceptable error.

Conventional numerical methods solve this problem by finding the instance of time at which the discontinuity occurs (this is usually called *zero crossing*). Then, they advance the simulation up to that point, and they restart the integration from the new condition (after the event).

Although this idea works fine, it adds some computational cost: zero crossing detection implies performing some iterations and the simulation restart can be also quite expensive. Owing to the presence of unbounded iterations, this solution is usually unacceptable in the context of real-time simulation.[4]

In addition to these difficulties, the simulation of a hybrid system also requires the representation and simulation of the remaining discrete subsystems, which in turn calls for the use of a common scheduling algorithm.

The usage of QSS methods on a DEVS simulation engine solves all of the mentioned problems. On the one hand, DEVS provides the unified framework to represent the discrete and the continuous (quantized) dynamics and to couple them on a single model.

Also, according to the order of the QSS method used, the trajectories are piecewise linear, parabolic or cubic. Thus, the zero crossing detection can be analytically solved, without performing iterations at all.

Although the event associated with a discontinuity must occur at the right time, the methods do not need to restart after that. After all, discontinuities in QSS occur all of the time, as the trajectories of $q_j(t)$ are discontinuous. Thus, for the QSS approximations, a discontinuity has the same effect of a normal step.

Moreover, when a component f_j of the system (1) contains a discontinuity, the DEVS static function computing $\dot{x}_j(t)$ will be in charge of detecting the discontinuity and provoking the right event trajectory for the state derivative. In other words, discontinuities are detected and handled locally, without any additional computational cost for the rest of the simulation.

These advantages result in a noticeable simulation speedup with respect to conventional numerical algorithms. In models with rapidly occurring discontinuities such as power electronic systems, the high-order QSS2 and QSS3 and LIQSS2 can perform simulations up to 20 times faster than all existing conventional methods.[12]

2.4. Similar tools

There is a great variety of tools in the field of DEVS simulation. Some are plain DEVS simulators, without real-time support, and others are general continuous system simulation tools. Here we describe a few of these tools.

- ADEVS[20] is a C++ library to simulate DEVS-based models. Unlike PowerDEVS it is based on two DEVS formalism extensions, called *Parallel DEVS* and *Dynamic DEVS* which treat simultaneous events in a different way (*confluent transition*). ADEVS is a library, but it does not have a graphical user interface (GUI) for model coupling and the model has to be described alphanumerically.
- CD++[21] is a DEVS simulation tool developed by Gabriel Wainer's group. It is based on another DEVS extension, called *Cell-DEVS*, which merges DEVS and cellular automata. It has support for real-time simulation.
- DEVSJava[1,22] is a DEVS simulation tool developed by Bernard Zeigler and Hessam Sarjoughian. It has interesting features such as changing the model structure at run time (or simulation time). Also, it offers a

hierarchical view of the model structure. It has support for real-time simulation in a *best shot* way, because it is written in Java and runs under non-real-time operating systems.

- Matlab Real Time Workshop[23] is a toolbox of Simulink. Simulink is a tool for modeling and simulation of continuous systems. It has a GUI in which models can be described in a block-diagram fashion. It supports generating C++ code for running the simulation under a variety of real-time operating systems. The Real Time Workshop (being a part of Matlab) is a proprietary software of *The MathWorks* and its use and distribution is restricted by its license.

3. PowerDEVS

In this section we describe the main components and features of the software developed.

3.1. PowerDEVS components

As we mentioned, PowerDEVS is composed of various independent programs: the *model editor*, the *atomic editor*, the *preprocessor* and the *simulation interface* and a workspace corresponding to a *Scilab* instance.

In this section we describe each of this tools and how they interact which each other.

3.1.1. The Model Editor. The Model Editor is, from a user point of view, the *main* program of PowerDEVS as it provides the graphical interface and the link with the rest of the applications.

In addition to building and managing models and libraries, it permits a simulation to be launched (by invoking the Preprocessor) and editing elementary blocks up to the atomic model definitions (by invoking the Atomic Editor).

The Model Editor main window (Figure 3) allows the user to create and open models and libraries. It also permits the libraries to be explored and blocks to be dragged from the libraries to the models.

There are also some advanced features which can be managed from the main window such as setting which are the *active* libraries (i.e. which libraries are shown when exploring), and configuring the tool bars and menu to invoke new external applications.

Models and libraries can be edited in a model window with the *open* and *new model* commands. Figure 4 shows a model window with a model composed by five submodels.

The Model Windows provide all of the typical graphical edition facilities so that blocks can be

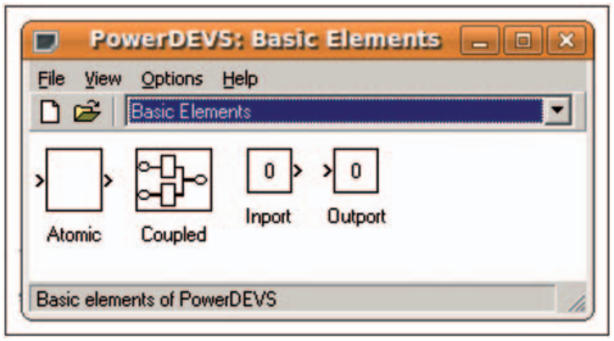


Figure 3. Model Editor main window.

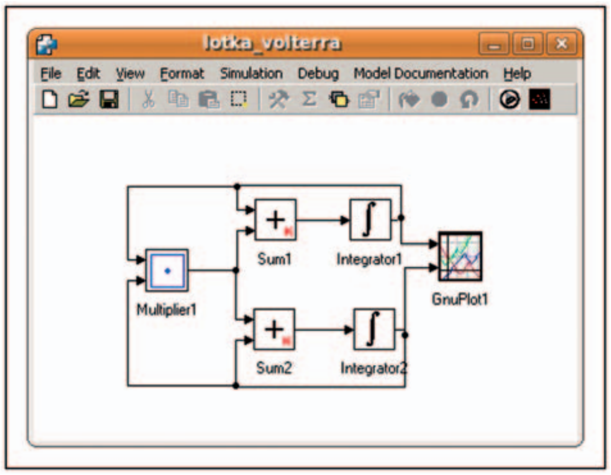

lotka_volterra
File Edit View Format Simulation Debug Model Documentation Help
Sum1
Integrator1
Multiplier1
GnuPlot1
Sum2
Integrator2

Figure 4. Model Window.

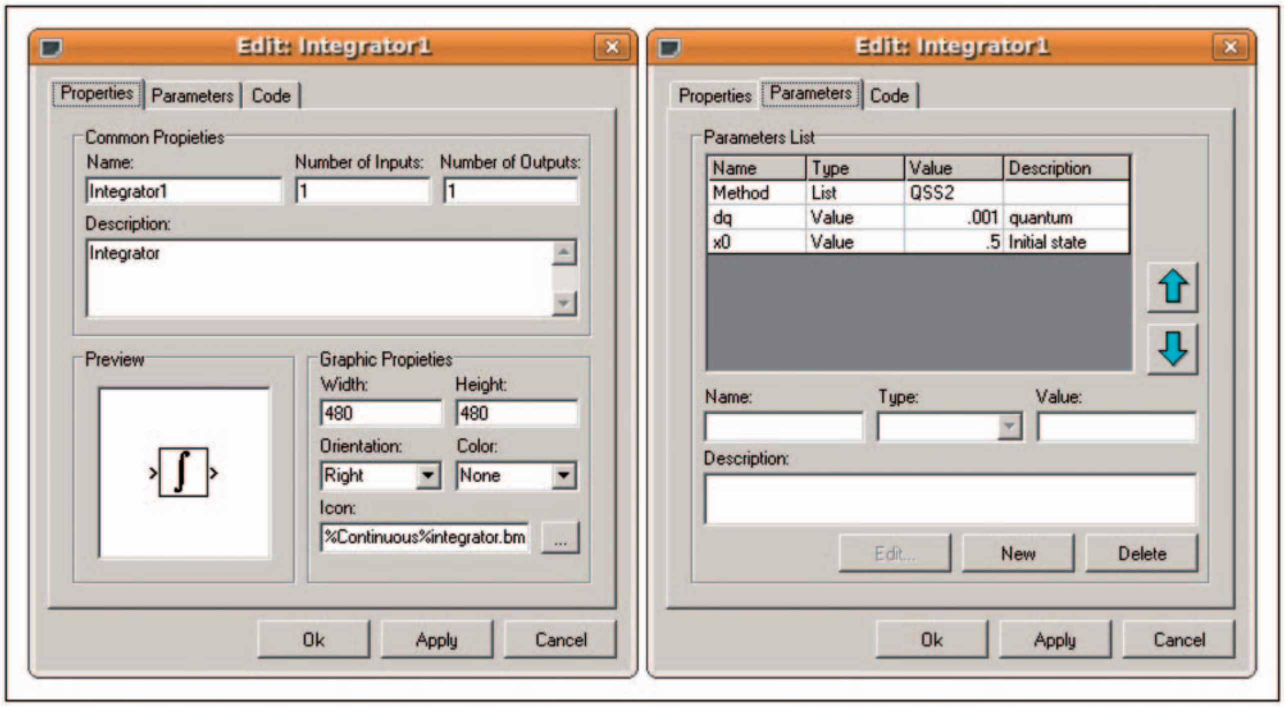
copied, resized, rotated, etc. while the connections can be drawn directly between different ports.

From the edit menu (or with the right button) it is possible to edit the features of each block, no matter whether it corresponds to a coupled or an atomic model.

The Block Edition Window (Figure 5) allows us to configure the graphic appearance of the block, to choose the block parameters and, in the case of atomic models, to select the file which contains the associated code with the DEVS model definitions.

The block parameters are defined and selected in the block edition windows. After being defined, their values can be changed by double-clicking on the block (Figure 6). Thus, when we take predefined blocks from the libraries, we can change the parameter values without editing them. As we shall see in Section 3.1.3, the values of these parameters are passed to the corresponding DEVS atomic or coupled models.

Coupled models do not have an associated code, but they have some extra features which can be modified



Common Propieties
Name: Integrator1
Number of Inputs: 1
Number of Outputs: 1
Description: Integrator

Preview

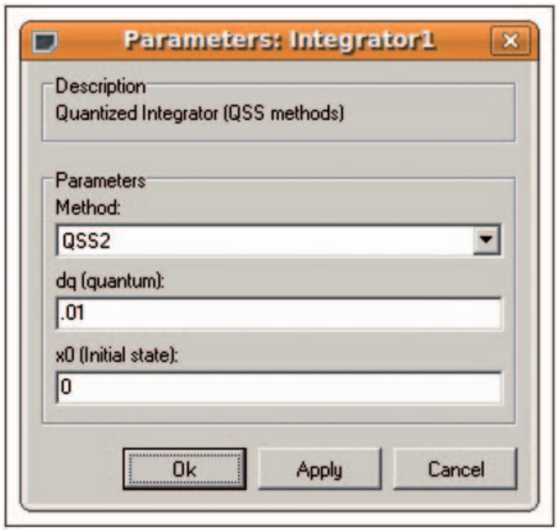
Graphic Propieties
Width: 480
Height: 480
Orientation: Right
Color: None
Icon: %Continuous%integrator.bm

Parameters List

Name	Type	Value	Description
Method	List	QSS2	
dq	Value	.001	quantum
x0	Value	.5	Initial state

Name:
Type:
Value:
Description:

Figure 5. Block Edition Window.



Description
Quantized Integrator (QSS methods)

Parameters
Method: QSS2
dq (quantum): .01
x0 (Initial state): 0

Figure 6. Changing parameter values.

from the block edition window and the edit menu (the internal priorities and the order of the input and output ports, for instance).

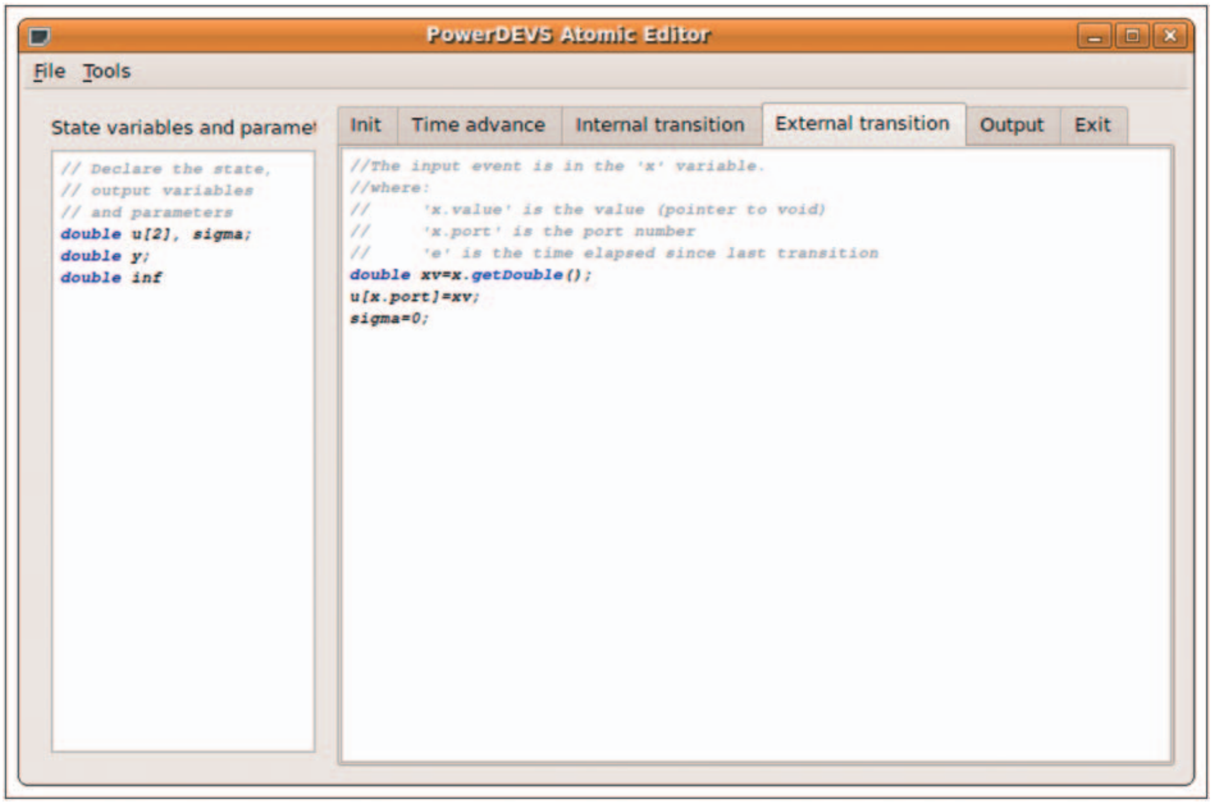
3.1.2. The Atomic Editor. The *Atomic Editor* facilitates the edition of the C++ code corresponding to each atomic DEVS model.

It can be invoked from the Block Edition Window to edit an existing code or to create a new one. It can be also run directly from the OS since it is a stand-alone application. The *Atomic Editor* main window is shown in Figure 7.

Using the atomic editor, the user only has to define the variables which form the state and the output of the DEVS model and the variables which represent the parameters of the system. After that, the C++ code of the time advance, transition and output functions must be placed in the corresponding windows. There are two additional windows (init and exit) where the user can also add a piece of code that is executed before the simulation starts (to set initial states and parameters, for instance) and a piece of code that is executed at the end of the simulation (to close some open files, for instance). When the model is saved, the code is automatically completed and stored in the corresponding .cpp and .h files.

In addition to facilitating the programming, the Atomic Editor was designed to give the user the possibility of writing a code which is very similar to the DEVS model definition. All of the rest of the job, related to simulation and implementation issues, is automatically performed by the program.

Let us illustrate this fact with a simple DEVS model. Consider for instance a system which calculates a static function $f(u_0, u_1) = u_0 - u_1$ with u_0 and u_1 being real-valued piecewise constant trajectories. If we represent those trajectories by sequences of events, as it is



```
// Declare the state,
// output variables
// and parameters
double u[2], sigma;
double y;
double inf
```

```
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
double xv=x.getDouble();
u[x.port]=xv;
sigma=0;
```

Figure 7. Atomic Editor main window.

done in QSS methods, we can build the following atomic DEVS model:

$$
\begin{aligned}
&M = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta) \text{ , where} \\
&S = \Re^2 \times \Re^+ \\
&\delta_{\text{int}}(s) = \delta_{\text{int}}(u_0, u_1, \sigma) = (u_0, u_1, \infty) \\
&\delta_{\text{ext}}(s, e, x) = \delta_{\text{ext}}(u_0, u_1, \sigma, e, x_v, p) = \tilde{s} \\
&\lambda(s) = \lambda(u_0, u_1, \sigma) = (u_0 - u_1, 0) \\
&ta(s) = ta(u_0, u_1, \sigma) = \sigma
\end{aligned}
$$

with

$$\tilde{s} = \begin{cases} (x_v, u_1, 0) & \text{if } p = 0, \\ (u_0, x_v, 0) & \text{otherwise.} \end{cases}$$

We use integer numbers from 0 to $n-1$ to denote the input and output ports (because PowerDEVS does so).

This DEVS model translated into PowerDEVS has the following code (at the Atomic Editor Figure 7):

ATOMIC MODEL STATIC1
State Variables and Parameters:
float u[2], *sigma*; //states
float y; //output
float inf; //parameter

Init Function:
$inf = 1e10$;
$u[0] = 0$;
$u[1] = 0$;
$sigma = inf$;
$y = 0$;

Time Advance Function:
return *sigma*;

Internal Transition Function:
$sigma = inf$;

External Transition Function:
float xv;
xv =*(float*)($x.value$);
$u[x.port] = xv$;
$sigma = 0$;

Output Function:
$y = u[0] - u[1]$;
return Event(&y,0);

It is clear that the translation from the atomic DEVS into the PowerDEVS code is quite straightforward.

With that code, the Atomic Editor automatically produces the .cpp and .h files which are then used by the Preprocessor to generate the simulation of the whole system.

3.1.3. The preprocessor.

The *Preprocessor* takes a .pdm (or .pds) file produced by the *Model Editor* and produces the simulation program.

It basically translates the .pdm file into a header .h file (called *model.h*) which binds the simulators and coordinators according to the coupling structure passing also the block parameters.

The preprocessor also produces a makefile (*Makefile.include*) which is then invoked to compile and generate the program which implements the simulation (that program is called *model*).

As we mentioned before, the *Preprocessor* can be invoked in a transparent way using the *Quick Simulation* command. Anyway, it can also be called from the command line (it is also a stand-alone application).

3.1.4. The simulation interface.

The generated program *model*, when executed, simulates the associated DEVS model.

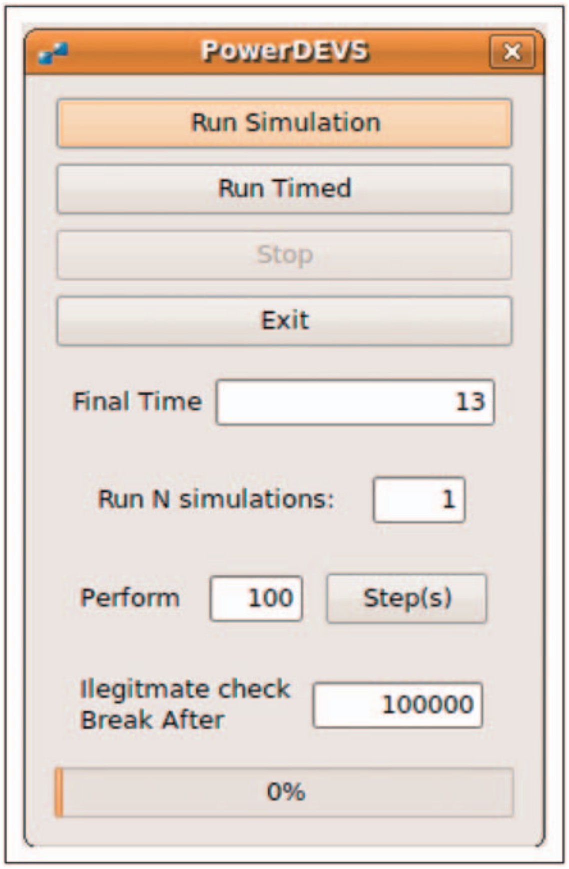
PowerDEVS provides a graphical interface for running the simulation (Figure 8). The interface also allows some parameters to be changed to set up the experiment:

- Final Time: states how long the model should be simulated.
- Simulations to run: multiple simulation runs can be executed at once. This can be useful when statistics from simulation results are to be calculated.
- Illegitimate check break: PowerDEVS stops the simulation if the time does not advance after a selected number of steps. This avoids hang-ups in illegitimate models.
- Simulate step-by-step: the simulation can be advanced performing one step (or many) at the time, and results can be analyzed in between.
- Synchronize time: The simulation can be run synchronized with the real clock (with the precision of the underlying OS).

Also, the *model* program can be invoked from the command line, and executed in an interactive shell. All of the simulation parameters can be changed in the same way they are changed from the graphical interface.

3.2. Internal implementation

Having described the main functional aspects of PowerDEVS, we now explain the way in which the simulation algorithm described in Section 2.2 is implemented inside the program that executes the simulation.



Final Time 13

Run N simulations: 1

Perform 100 Step(s)

Ilegitmate check Break After 100000

Figure 8. Simulation Interface.

As *PowerDEVS* performs an object-oriented simulation, we start by describing the internal class structure.

Atomic models with the same associated code belong to a particular class (defined by that code). For instance, the *Integrator* atomic models in the model of Figure 4 belong to the *Integrator* class, defined in the files integrator.h and integrator.cpp which contain the code associated with that atomic model.

When the code of an atomic model is written using the Atomic Editor, the code related to the class definition is generated automatically.

All of the atomic classes are derived from the *simulator* class. The simulator class is an abstract class which acts as an interface to deal with different atomic model implementations. The variables representing the state of the model and the functions that operate on it (time advance, transition and output functions) are member variables and methods.

The external transition and output functions of the simulator class receive and return, respectively, objects belonging to the *Event* class. Events have the following properties:

- Event.port: is an integer number indicating the input or output port where the event is received or sent.

- Event.value: is a pointer to void. That way, the values carried by the events can belong to arbitrary types.
- Event.realTimeMode: is an integer that can take the following values: 0 (indicating that the event is not synchronized with the real time), 1 (with normal synchronization) and 2 (with precise synchronization). When an event has its mode set to 1 or 2 (the difference between them are explained in Section 4.2.1), the simulation engine waits until the physical time reaches the simulation time to propagate it.

To provide the capability of initializing and stopping devices that interact with the hardware, the *simulator* class has two extra methods which are not included in the definition of Zeigler et al.[1] As we already mentioned, for real-time simulation purposes *PowerDEVS* also inverts the time managing features. Thus, when an atomic model receives an interrupt request from an external device, it informs the change in its time to next event (*tn*) to its parent.

The hierarchical coupling structure is implemented by the *coupling* class. This class is similar to the coordinator in Figure 2. Each object of this class is associated with a coupled DEVS model and it contains a list of references to the corresponding connections, atomic and coupled models.

The difference with the coordinator is that, following the *simulator* class behavior, the *coupling* objects are able to receive the messages coming from their children notifying changes in their time to next event. Similarly, the *coupling* objects have also the possibility of informing their own parent about changes in their *tn*. Coherently with the closure property, the *coupling* class is derived from the *simulator* class.

The *coupling* and *simulator* objects also contain objects belonging to the classes *connection* and *event* (the first is only in the *coupling* class). The *connection* class is formed by four integer numbers representing two pairs of models and ports. The *event* objects are formed by a pointer to void and an integer (identifying the input or output port). Thus, PowerDEVS models can produce output events which belong to different types.

Having defined the model structure in terms of components and functionality, we now describe the framework developed to actually execute the simulation.

The *root-simulator* class is in charge of running the simulation. Basically, this class manages the simulation advance interacting with the object representing the coupling at the top of the structure. Thus, the *root-simulator* plays the role of the root-coordinator in Figure 2.

Before following the description we should mention that the behavior of the top coupling object has a small difference to the others. In terms of the simulation execution, there is no coupling above to notify the timing changes but these changes should be notified directly to the *root-simulator*. Thus, a new class denominated *root-coupling* has been introduced. This class, derived from the *coupling* class, has the same functionality as the latter but it differs in the hierarchical superior object.

Finally, the *main* class of the simulation program is called *model*. This class provides the interactive shell for interacting with the user (or with the graphical simulation interface of Figure 8) and running the simulation. This class talks directly with the *root-simulator* allowing the user to start the simulation in different modes.

The objects which form the coupling structure are instantiated at the model.h file. This file is automatically generated by the Preprocessor and it declares the simulators, couplings and connections according to the structure file.

In fact, the only difference between the codes that execute the simulation of two different models is in that file (model.h). Obviously, the fact that both files are different also implies that they may include different *simulators* corresponding to different atomic models.

3.3. Interconnection with Scilab

Scilab[24] is a numerical computational package developed by the Institut National de Recherche en Informatique et Automatique (INRIA) and it is released under the GPL license. Scilab is an open-source alternative to the widely used software Matlab.

Scilab contains an interactive interface where the user can define variables and matrices and perform complex operations between them. It also has a programming language that can be used to define new functions and algorithms, and some graphical tools to plot data. The Scilab distribution includes several toolboxes (i.e. set of functions developed by different people) to solve problems related to linear algebra, signal processing, automatic control, optimization, filtering design, etc.

To make the communication between PowerDEVS and Scilab, some modifications to the source code of Scilab were made. These modifications open a ‘back door’ into Scilab’s workspace, where variables can be read or written from PowerDEVS. The communication is performed through UDP messages over the port 27015 in a client–server model, with Scilab acting as server and PowerDEVS as client.

3.3.1. The Scilab Side.

As Scilab is an open-source tool, all of its source code is available online to view, edit and enhance. We used Scilab 4.1.2 version and made modifications in both, Linux and Windows source versions. These modifications were made in the files routines/wsci/WScilex/WScilex.c (for Windows) and routines/xsci/x_main.c (for Linux).

As the idea was to use Scilab as PowerDEVS *workspace* (that is, PowerDEVS runs the simulation and Scilab responds to PowerDEVS commands) it was necessary to modify the normal behavior of Scilab in order to establish the communication.

To keep running the Scilab GUI, while communicating with PowerDEVS a new thread is created in Scilab address space. This new thread is in charge of receiving, executing, and responding to PowerDEVS requests.

The actual communication between PowerDEVS and this Scilab thread was implemented in a client–server model over UDP. Under this architecture, Scilab (acting as the server) waits for requests on the UDP port 27015 and PowerDEVS sends requests to that port (see Figure 9).

The PowerDEVS requests sent to Scilab consist of string representations of the commands to execute in Scilab workspace (for example 'freq=pi*345'). Each request starts with a character indicating whether Scilab must notify PowerDEVS once the command execution is finished.

The Scilab notification to PowerDEVS (when the first character indicates so) consists of a UDP message containing the value of the 'ans' variable in the Scilab workspace. This permits us, indirectly, to evaluate any Scilab expression or variable from PowerDEVS, sending the correct command sequence.

3.3.2. The PowerDEVS side.

The PowerDEVS simulation engine has a set of services to communicate with Scilab. These services can be called from any atomic model written with the atomic editor:

```
void    getAns(double *ans, int r, int c);
void    putScilabVar(char *varname, double v);
double  getScilabVar(char *varname);
double  executeScilabJob(char    *job,    bool
        blocking);
```

The function getAns(double *ans, int r, int c) returns the answer of the last calculation in the Scilab workspace (i.e. the 'ans' variable). The result is

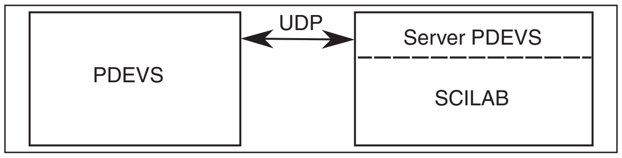


Figure 9. PowerDEVS–Scilab communication.

put in the double **pointed** by ans. The r and c arguments indicate the size of the result to be retrieved (Scilab works with matrices and so does the PowerDEVS interface). If a scalar value is to be retrieved r and c must be 1.

The functions putScilabVar(char *varname, double v) and getScilabVar(char *varname) are the basic methods to interact with Scilab variables. putScilabVar creates (or updates) the Scilab variables named varname with value v. On the other hand, getScilabVar reads the variable named varname from Scilab workspace. If the variable does not exist this function returns 0.0 and writes a warning message in the simulation log.

Finally, the function executeScilabJob(char *job, bool blocking) runs the command job in the Scilab workspace. It works just like when one types the sentence in the Scilab command window. The second argument blocking indicates whether the function should return immediately or should wait for the command to finish. The function returns the 'ans' variable (like getAns) which, according to the command, can represent the result of the executed command.

3.4. Library

The PowerDEVS distribution comes with a set of atomic and coupled models already programmed, that can be used to build new models. These basic models form the PowerDEVS library (any advanced user can easily extend the library including their own developed models).

The library is divided into the following categories:

Basic Elements: This is the only PowerDEVS core library. Based on these models, all of the other libraries were developed. It contains four basic models, an atomic (a skeleton for all the atomics), a coupled (an empty coupled model), and two special objects called inport and outport. The last two objects represent external input and output interfaces for coupled models.

Continuous: This library contains models that process continuous signals based on the QSS methods to simulate continuous systems.[a] It includes integrators, gains, non-linear static functions, multipliers, etc.

Discrete: This library contains models for simulating discrete-time systems.

Hybrid: Under this category, a set of models are included combining continuous and discrete features to simulate hybrid systems: quantizers, switches, comparators, samplers, etc.

Real Time: This library contains specific blocks to make use of different real-time features of PowerDEVS. They are described in Section 4.3.

Source: Here we can find several sources for continuous-time signals approximated by QSS methods, such as sine waves, ramps, pulses, square waves, etc.

Sink: This library contains different sink models where simulation results are sent for visualization or future processing. Some models included in this library are GnuPlot (which plots its input signals), ToDisk (which writes its input signals to a CSV (Comma Separated Values) file), toWorkspace (that writes in Scilab workspace the received signal), etc.

All of the models in the library accept Scilab expressions as parameter values. For instance, one could use '*freq*2+0.5*' as the gain parameter in a gain block, and this expression will be evaluated in Scilab workspace (in this example *freq* must be a defined Scilab variable).

4. PowerDEVS in real time

As we anticipated earlier, PowerDEVS was extended to run on a real time operating system. For some reasons that will be explained soon in this section, we choose Linux RTAI as the real-time OS for PowerDEVS.

The simulation engine of PowerDEVS is an implementation in C++ of the simulator described in Section 2.2. As RTAI is an extension of Linux, in principle, the PowerDEVS simulations (that run fine under Linux) should also run under RTAI.

Although this is true, the goal of running a simulation under RTAI is making use of its services to ensure real-time performance. So several modifications were made in the PowerDEVS simulation engine which enable the user to easily access basic services such as synchronizing events, capturing and handling hardware interrupts, obtaining file support and performing real-time measurement.

Also, the PowerDEVS library was extended with a set of blocks to make use of these services in a simple drag-and-drop way.

This section, after introducing some concepts related to real-time systems and real-time simulation, describes the extensions made to the simulation engine, the new library blocks and some experiments performed to measure different performance indicators of PowerDEVS in real time.

4.1. Real-time systems and real-time OS

4.1.1. Real-time systems. A real-time system is a system in which computations are subject to *real-time constraints*. That is, responses to stimuli have a dead-line that must be met, regardless of system load, in order for the system to be considered correct.

Real–Time systems can be classified in two categories:

Hard real-time systems: in which the completion of a computation after its deadline is considered useless, and may cause critical failures in the system;

Soft real-time systems: in which a missed deadline can be accepted, or ignored or even corrected.

4.1.2. Real-time operating systems. Real-time OSs are a special class of OSs which provide the basic tools and services to implement systems with time constraints.

This kind of OS should be expressive enough to represent the time constraints of each task, the way these tasks communicate, and must have some control over the low-level hardware of the computer (memory, interrupts, ports).

Some commonly used real-time OSs are QNX,[26] VxWorks,[27] Drops,[28] RT-Linux,[29] RTAI,[14] etc.

Among them, we decided working using RTAI (Real Time Application Interface) because it consists of an extension of the Linux Kernel[30] that permits running all the software that runs on Linux. This was a crucial requirement in the context of this work since we wanted to run not only PowerDEVS, but also the external applications that communicate with it (GNUPlot and Scilab).

Also, the fact that RTAI is distributed under the GPL License allows us to freely distribute the real time version of PowerDEVS included with the OS installer.

4.1.3. RTAI. RTAI is a real-time OS that supports several architectures (i386, PowerPC, ARM). RTAI is an extension of the Linux kernel to support real-time tasks to run concurrently with Linux processes. This extension uses a method to enable this, first used in RT-Linux.[29]

Linux kernel is not itself a real-time OS. It does not provide real-time services, and in some parts of the kernel, interrupts are disabled as a method of synchronization (to update internal structures in an atomic way). These periods of time where interrupts are disabled, lead to a scenario where the response time of the system is unknown, and time deadlines could be missed.

To avoid this problem RTAI inserts an abstraction layer beneath the Linux kernel. In this way, Linux never disables the *real* hardware interrupts. The Linux kernel is run under another micro-kernel

(RTAI + Adeos[b]) as user processes. All hardware interrupts are captured by this micro-kernel and forwarded to Linux (if Linux has interrupts enabled).

Another problem with running real-time tasks under Linux is that the Linux scheduler can take control of the processor from any running process without restrictions. This is unacceptable in a real-time system. Thus, in RTAI real-time tasks are run in the micro-kernel, without any supervision of Linux. Moreover, these processes are not seen by the Linux scheduler.

Real-time tasks are managed by the RTAI scheduler. Clearly, there are two different kinds of running processes, Linux processes and RTAI processes. RTAI processes cannot make use of Linux services (such as File system) and vice versa. To avoid this problem, RTAI offers various IPC mechanisms (Pipes, Mailboxes, Shared Memory, etc.).

RTAI provides the user with some basic services to implement real-time systems:

Deterministic interrupt response time: An interrupt is an external event to the system. The response time to this event varies from computer to computer, but RTAI guarantees a boundary (on each computer), which is necessary to communicate with external hardware, for example a data acquisition board.

Inter-process Communication (IPC): RTAI offers various methods of IPC, such as semaphore, pipes, mailboxes and shared memory. These IPC mechanisms can be used to connect processes running in real time with normal Linux processes or vice versa.

High precision timers: When developing real-time systems, the time handling accuracy is very important. RTAI offers clocks and timers with nanosecond (1×10^{-9} s) precision. On i386 architectures, these timers use the processor's time stamp, therefore, they are very precise.

Interrupt Handling: RTAI allows the user to capture hardware interrupts and treat them with custom user handlers. Normal OSs (such as Linux, Windows) hide hardware interrupts from the user, making the development of communication with external hardware a bit more difficult (you have to write a kernel driver). As said before, the response time is bounded.

RTAI tasks can be run in two different modes:

Loadable kernel module: in this mode executable files are not stand-alone programs but loadable kernel modules. The RTAI process is *inserted* into the running Linux kernel and from then on it switches to the RTAI scheduler. Running in kernel mode (as modules do) gives the process certain privileges and rights that normal user processes do not have (accessing kernel objects, writing/reading ports, etc).

LXRT: RTAI offers a library called LXRT for developing real-time systems in user space. LXRT is a 'bridge' between the Linux user mode and RTAI. LXRT brings to the user space all RTAI services. LXRT works with a concept called the *angel process*. LXRT creates a RTAI counterpart (the angel process) in charge of executing all of the real-time services on behalf of the user space process.

Having described the main features of RTAI, we now focus on the extensions made to PowerDEVS.

4.2. PowerDEVS engine support

Four new modules were added to the simulation engine to make use of the real-time services provided by RTAI. The modules consist of new functions that can be used by any atomic model.

4.2.1. Synchronization module. This module is the cornerstone of PowerDEVS in real time. It allows the simulation engine to synchronize the events with the RTAI real clock.

RTAI has two methods of synchronizing, both with nanosecond precision:

- *rt_sleep*: waits for an amount of time, releasing the processor.
- *rt_busy_sleep*: waits for an amount of time by busy-waiting, that is, burning CPU cycles until the wait is over. A more precise synchronization is obtained in this mode.

Based on these two methods, the PowerDEVS simulation engine implements the function `waitFor(unsigned long int nanoseconds, int mode)` which waits the specified time in one of two modes: *Normal* or *Precise*.

This function has the following logic:

- If the amount of time to wait is *too small* (smaller than a constant that defines the synchronization accuracy), the sentence is ignored.
- If the function is called in *Normal* mode, the non-busy *rt_sleep* is invoked (where the CPU is released).
- If the function is called in *Precise* mode, the behavior depends on the time to wait:
  - If the period is less that a given constant (which defines the accuracy of the non-busy *rt_sleep*), a busy waiting is performed invoking *rt_busy_sleep*.
  - Otherwise, the waiting is split into two periods. The first period is computed as the whole period minus twice the non-busy accuracy, and it is performed in a non-busy way invoking *rt_sleep*.

After finishing this period, the time is measured again (to eliminate the error introduced by the non-busy waiting) and the remaining waiting time is done with a busy-waiting. In this way, the synchronization has the accuracy of a busy waiting, but the processor is released most of the time.

The function `waitFor` can be invoked by any atomic model, but it is automatically called by the simulation engine when an output event has its realTimeMode property (see Section 3.2) set to 1 or 2.

In this way, in the same model, some events can be propagated immediately (when their realTimeMode is set to 0) and others can be synchronized either in normal or precise mode. This gives the user the possibility of choosing which events must be synchronized (typically those that involve communication with external hardware) and which should be computed as fast as possible (those involving intermediate calculations).

4.2.2. Interrupt handling module. This module is in charge of managing the interaction between the PowerDEVS simulation engine and the hardware interrupts.

Any atomic model can express interest in a specific hardware interrupt by calling:

```
requestIRQ(int irq);
```

When a hardware interrupt `irq` occurs, the simulation engine sends an external transition message to all of the atomic models that have expressed interest in that interrupt. The message arrives to a non-existing port (denoted by -1).

Since these atomic models will possibly change their time advances in the external transition, the engine automatically transmits from the bottom to the top of the hierarchy about these changes. This is the mechanism that we mentioned in Section 2.2, and it is the only modification of the abstract simulator of Zeigler et al.[1] introduced by PowerDEVS.

4.2.3. File support module. When a task is running under RTAI, it cannot make use of Linux services (such as the Linux File System). This is because a system call to the Linux kernel does not have a deterministic response time, so a system call could lead to missing a deadline in the real-time system. In fact, RTAI tasks are not scheduled by the Linux scheduler which does not know they exist. Thus, to enable the simulations to use files, a File Support Module was developed. This module has two parts:

Real Time Interface: This is a real-time task (running under RTAI) that accepts the requests of the real--time simulation (such as open, read, write files). It communicates with a program running under Linux user space (which has access to the Linux File System) using a real-time FIFO (one of the IPC provided by RTAI).

User Space Angel: This is a normal Linux program that accepts commands from the Real Time Interface. This program (which is launched together with the simulation) makes all of the system calls on behalf of the simulation.

The interface provides similar functions to those of the C library (stdio) that can be directly used by any atomic model.

4.2.4. Real-time measurement module. This module allows the atomic models to ask for the value of the real-time clock. This is useful, for instance, when a user wants to change the behavior of an atomic model depending on the difference (in either way) between the real time and the simulated time (to handle overruns or to compute more precisely if enough time is available).

4.3. Real-time library

To make use of all of the real-time features in a simple way, the PowerDEVS library was extended to include several atomic blocks that can be included directly in the models.

RTWait: This atomic block receives events and outputs them synchronized with the real-time clock. In this way, a normal model (i.e. non-real-time) can be transformed into a real-time model by inserting these blocks on the connections carrying events that need to be synchronized. The block has a parameter to choose what kind of synchronization must be made, *precise* or *normal* (see Section 4.2.1).

RTClock: When this block receives an event (no matter what the value of the event is), it emits an event whose value is equal to the real-time clock value.

ToDisk: This model writes the input signal in a CSV file. It makes use of the File Support Module to run in real time.

IRQDetector: This model is the user interface for the Interrupt Handling Module. It emits an event whenever the hardware interrupt occurs. From the point of view of the DEVS formalism (and from PowerDEVS too) this is a passive model (its time advance is ∞), but when the associated interrupt is triggered, the model changes its own time advance to 0 (and the engine automatically notifies its parent of this change). The block has a parameter to indicate which interrupt it waits for.

4.4. Real-time performance

One of the most important features to characterize a real-time system is the *error* (or latency) of synchronization.

We conducted several experiments to see how the system responds (in relation to the load of the system). All of the experiments where run on a AMD Athlon 1.8 GHz PC.

First, we performed the following experiment: we simulated a source block which generates a QSS approximation of a sine wave at 440 Hz (this model generates 20 events per cycle, to a total of 8800 events per second). The model was simulated in real time measuring the maximum and average latency. Also the two different modes (see Section 4.2.1) of waiting were tested. The results shown in Table 1 were obtained.

Next, a simulation of a more complex model was performed: a PI control of a DC motor using Pulse Width Modulation (PWM). In this example, several simulations were run varying the PWM carrier from 1000 to 20,000 Hz (Table 2).

As can be seen in Table 2, using a frequency of 1000 Hz there are no *overruns*. With frequencies between 15,000 and 17,000 Hz, there are more *overruns* and the Linux system (not the real-time system) experiences some delays, which indicates that the system load is high.

With a frequency of 20,000 Hz, the system stops responding (the Linux part) and we see that all events are emitted after their time deadline.

This last case reflects a limitation of the hardware platform, not a limitation of the system synchronization. What happens with a frequency of 20,000 Hz is that the hardware cannot complete the calculations in time.

Table 1. Synchronization error with low load.

Wait	Maximum	Average	Overruns
Normal	4800 ns	1500 ns	—
Precise	450 ns	180 ns	—

Table 2. Synchronization error with varying load.

f (Hz)	Wait mode	Maximum	Average	Overruns
1000	Normal	5786 ns	1512 ns	—
	Precise	1330 ns	180 ns	—
15,000	Normal	5622 ns	1622 ns	372/19,905 events
	Precise	1000 ns	512 ns	305/19,905 events
17,000	Normal	4547 ns	1648 ns	6119/17,616 events
	Precise	973 ns	454 ns	5924/17,616 events
20,000	Normal	0 ns	0 ns	18,292/18,292 events
	Precise	0 ns	0 ns	18,292/18,292 events

4.5. Interrupt latency

To measure the interrupt latency the following procedure was used.

A PowerDEVS model was created (Figure 10) which generates hardware interrupts and then captures back these interrupts, measuring the period of time between the generation of the signal provoking the interrupt and the capture of the interrupt by the handler.

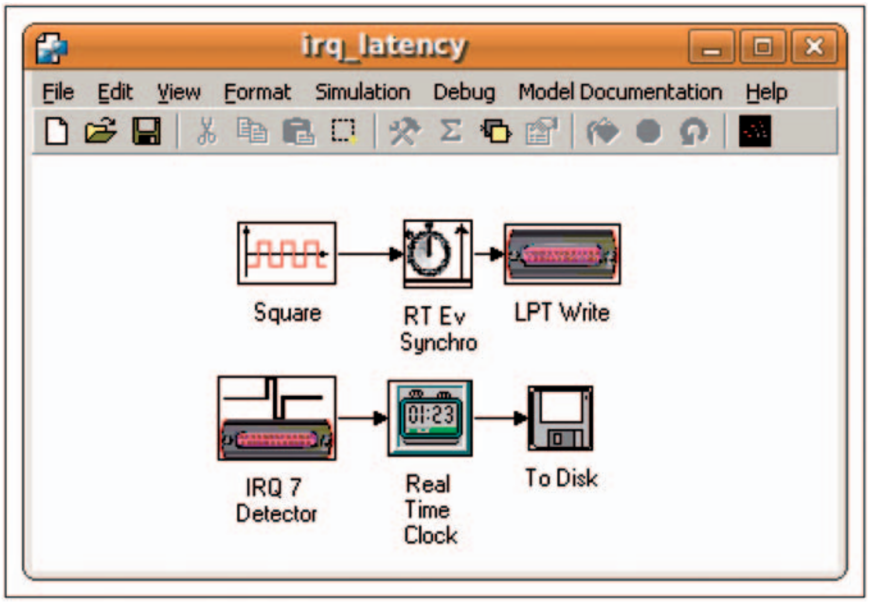


Figure 10. Interrupt Latency tester model.

For that purpose, we used the PC Parallel port. One of the pins on the parallel port (*STO*) is in charge of generating interrupts. This pin was wired to a data pin, thus generating a hardware interrupt every time that the data pin goes from a low state (0 V) to a high state (5 V). The model stores the time at which a 1 is written to the data pin (high state), and the time at which the interrupt handler of the parallel port is run. The difference between these two times is an upper bound of the interrupt latency of the system (in fact, that difference also includes the time needed to write on the parallel port and the electrical transients).

Running this experiment on the same hardware (AMD 1.8 GHz PC), we obtained an upper bound on the interrupt latency of about 20 μs.

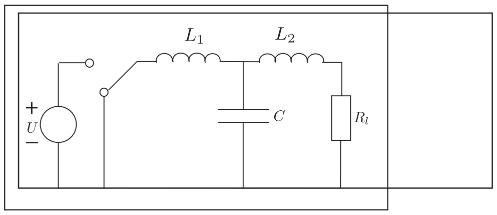


Figure 11. Buck circuit.

5. Examples

This section describes two examples that show different features of PowerDEVS.

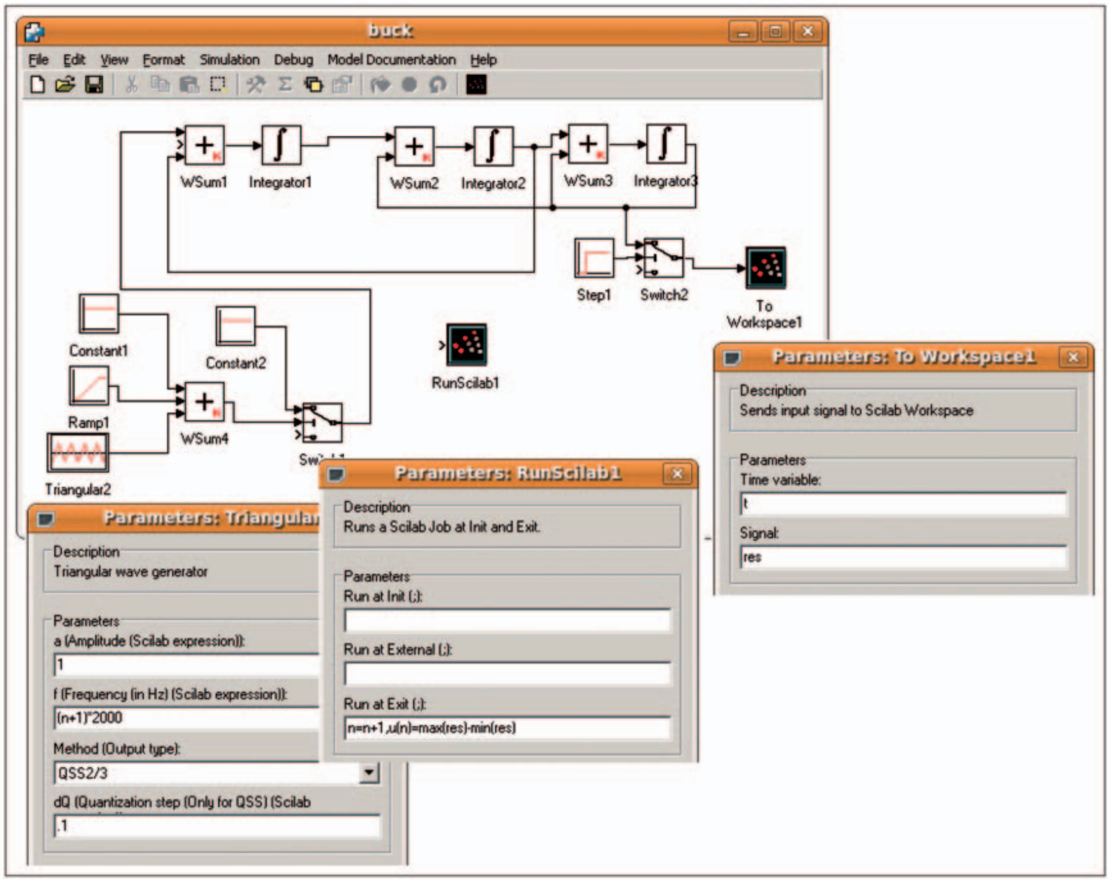
5.1. *Ripple versus frequency in a buck circuit*

Figure 11 represents a DC–DC converter circuit known as *Buck Circuit*. The presence of the switch introduces hybrid behavior to the system.

The goal of the experiments is to analyze the dependence of the ripple amplitude on the switching frequency. In this example we used the following parameters $L_1 = L_2 = 0.1$ mH, $C = 20$ μF, $Rl = 10$ Ω and $U = 12$ V.

Figure 12 shows the PowerDEVS model of the circuit, made entirely with atomic blocks from the PowerDEVS library. The switch is commanded by a PWM signal. The remaining blocks implement source, static functions and integrators based on QSS methods.

The frequency of the triangular carrier was chosen in a way that increments 2000 Hz in each simulation. The 'RunScilab Job' block increments the Scilab variable n after each simulation and calculates the ripple



Description
Sends input signal to Scilab Workspace
Parameters
Time variable: t
Signal: res

Description
Runs a Scilab Job at Init and Exit.
Parameters
Run at Init (;):
Run at External (;):
Run at Exit (;): n=n+1,u(n)=max(res)-min(res)

Description
Triangular wave generator
Parameters
a (Amplitude (Scilab expression)): 1
f (Frequency (in Hz) (Scilab expression)): (n+1)*2000
Method (Output type): QSS2/3
dQ (Quantization step (Only for QSS) (Scilab: .1

Figure 12. Buck circuit model.

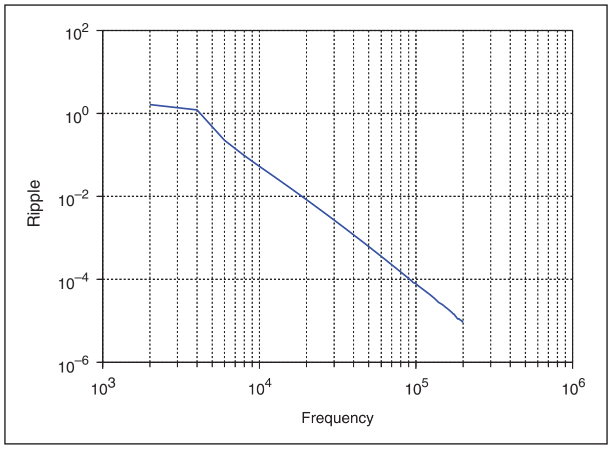


Figure 13. Ripple versus frequency.

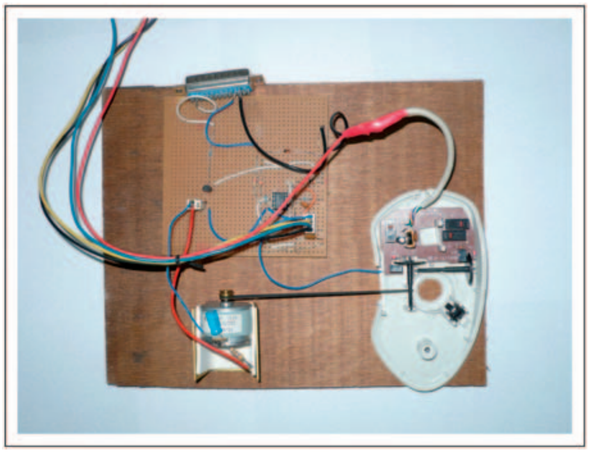


Figure 14. Motor and mouse (encoder).

amplitude in steady state. This amplitude is saved in the array `u(n)`.

After 100 simulations (in which the frequency goes from 2000 to 200,000 Hz), we can plot the results directly in Scilab (see Figure 13).

It should be noted that 100 simulations of this hybrid system (with very quick commutations) were run, and thanks to the efficiency of the QSS methods in treating this kind of problem, the experiment only took about 13 seconds (while in Matlab/Simulink it takes about 1 minute).

5.2. DC motor control in real time

This example shows the real-time asynchronous control of a DC motor. For this purpose, we used a small DC drive moving an old PC mouse wheel acting as an encoder (see Figure 14). Each time the wheel moves a small angle, a pulse is sent to the bit *STO* of the parallel port, which triggers an interrupt in the PC. The motor is fed by an amplifier, implemented with a transistor and a filter taking the on–off voltage from a data pin of the parallel port.

The control system is composed of the following subsystems (see Figure 15):

- The *Motor Speed* block detects and counts the interrupts generated by the encoder to estimate the motor speed.
- This estimation is compared with the reference speed (the block *WSum4* calculates the difference or error).
- In the base of this error a Proportional–Integral (PI) control is applied, using QSS discretization.

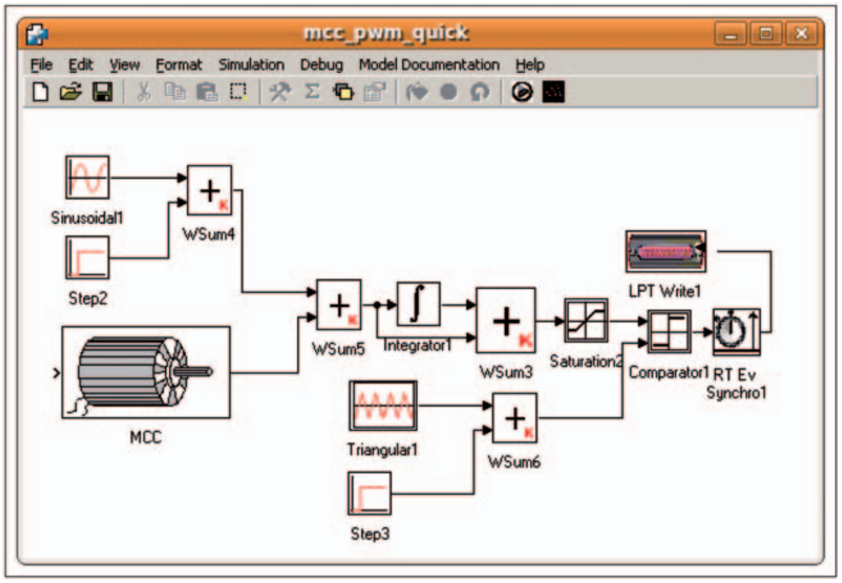


Figure 15. Proportional control model.

- The control signal (previously saturated to avoid over-modulation) is pulse width modulated.
- The PWM signal (calculated in the *Comparator1* block), is sent *synchronized* to the parallel port.

The whole control system of Figure 15 was built using blocks from the PowerDEVS library, except the model that estimates the motor speed based on the number of interrupts it receives per second.

Figure 16 shows the reference speed signal while Figure 17 shows the speed measured by the control system.

6. Conclusions and future work

We have introduced and described a general-purpose tool for DEVS simulation. We have illustrated its use in hybrid system simulation, where it shows the most important facilities and advantages compared with existing simulation software.

In addition to the user-friendly environment and the simplicity it offers to different kinds of users, *PowerDEVS* has a new way of managing the simulation time advance which allows the implementation of

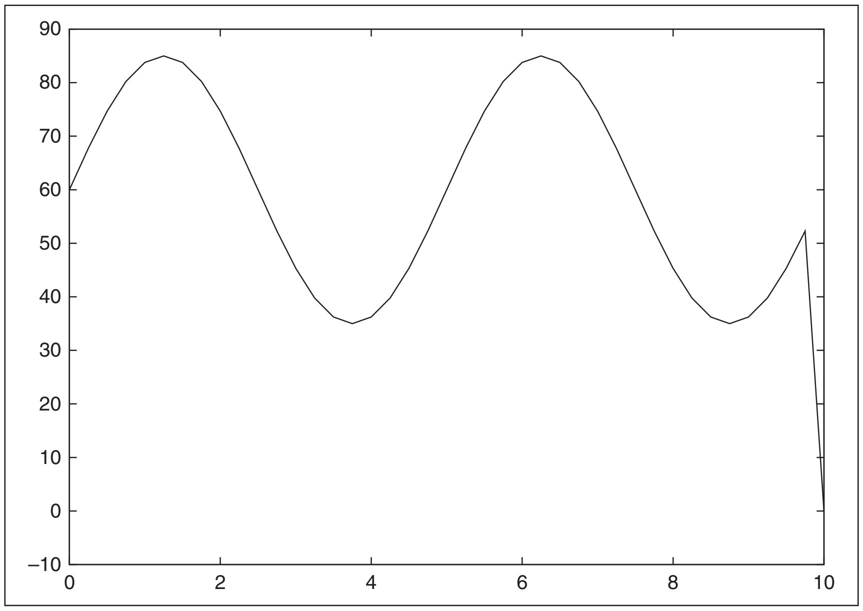


Figure 16. Reference speed.

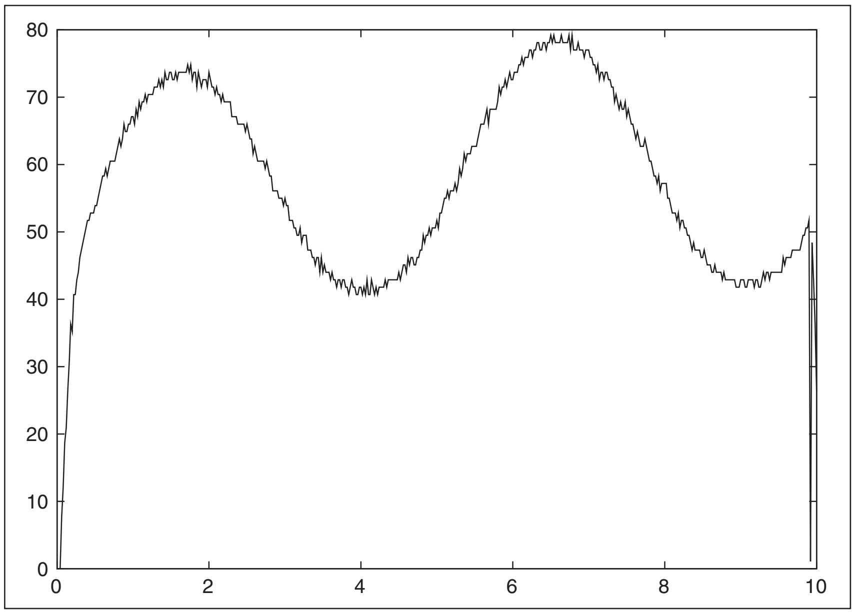


Figure 17. Measured speed.

simulations synchronized with a real-time clock and with the possibility of handling interrupts in an easy fashion. In that way, it can directly implement hybrid asynchronous QSC controllers (as was done in the example of the DC motor control).

However, *PowerDEVS* is not only a tool for hybrid system simulation and control. As we mentioned before, it is a general-purpose DEVS simulator and taking into account that its use is very simple, it is also appropriate for educational use.

Compared with other existing DEVS simulation environments, the main advantage of PowerDEVS is the convenient user interface and the simplicity of implementing continuous and hybrid system simulations based on the family of QSS methods. Also the possibility to running simulations under a real-time OS (RTAI) and the communication with Scilab are remarkable features.

When it comes to future work, an immediate goal is to finish the migration of PowerDEVS source code from Visual Basic to C++ with QT libraries. At the moment, only the model editor needs to be translated. After that, the complete environment would run under different platforms without the need for Windows emulators (the current version of the PowerDEVS model editor uses the Wine Windows Emulator to run under Linux and RTAI).

The incorporation of visual programming tools can also improve the atomic model edition. At the moment, users can build complex models by coupling simpler models (unless they are good programmers). An alternative way to build complex atomic models is with the help of graphical tools, such as Simulink Stateflows. A tool that translates graphical formalisms into atomic PowerDEVS C++ code might facilitate the generation of new atomic blocks. If such a tool is developed (as a separated program) it can be integrated with PowerDEVS in a straightforward manner because of its user-configurable menu features.

Finally, a project is being carried out at ETH Zurich to automatically translate Modelica models into PowerDEVS models (based on QSS approximations). The completion of that goal will allow the modeling capability of PowerDEVS to be increased drastically, also giving Modelica users the possibility of implementing QSS simulations.

Notes

(a) The Continuous library was conceived to simulate ODEs. Differential algebraic equations can be simulated with the use of the Implicit Function Block, that implements the methodology developed by Kofman.[25] If an algebraic loop is included in the model instead of using that block, the simulation will be illegitimate.

(b) Adeos[31,32] is the abstraction layer used in RTAI. Adeos implements the concept of *interrupt domain*, see http://home.gna.org/adeos/

References

1. Zeigler BP, Praehofer H, and Kim TG. *Theory of Modeling and Simulation*, 2nd ed. New York: Academic Press, 2000.
2. Zeigler B, Vahie S. DEVS formalism and methodology: unity of conception/diversity of application. *Proceedings of the 25th Winter Simulation Conference*. Los Angeles, CA. 1993, p.573–579.
3. Vangheluwe H. DEVS as a common denominator for multi-formalism hybrid systems modelling. *IEEE International Symposium on Computer Aided Control System Design*, Anchorage, AK, 2000, p.129–134.
4. Cellier F, Kofman E. *Continuous System Simulation*. New York: Springer, 2006.
5. Klee H. *Simulation of Dynamic Systems with MATLAB and Simulink*. Boca Raton, FL: CRC Press, 2007.
6. Campbell S, Chancelier J, and Nikoukhah R. *Modeling and Simulation in Scilab/Scicos*. Berlin: Springer, 2006.
7. Zeigler B, Sarjoughian H. *Introduction to DEVS Modeling and Simulation with JAVA: A Simplified Approach to HLA-Compliant Distributed Simulations*, Arizona Center for Integrative Modeling and Simulation, Available at http://www.acims.arizona.edu/.
8. Kim TG. *DEVSim++ User's Manual. C++ Based Simulation with Hierarchical Modular DEVS Models*. Korea Advance Institute of Science and Technology, 1994.
9. Cho H, Cho Y. *DEVS–C++ Reference Guide*. The University of Arizona, 1997.
10. Wainer G, Christen G, and Dobniewski A. Defining DEVS models with the CD++ toolkit. *Proceedings of ESS2001*, Marseille, France, 2001, p.633–637.
11. Filippi J, Delhom M, and Bernardi F. The JDEVS environmental modeling and simulation environment. *Proceedings of IEMSS 2002*, 2002, Vol. 3, p.283–288.
12. Kofman E. Discrete event simulation of hybrid systems. *SIAM J Sci Comput* 2004; 25(5): 1771–1797.
13. Pagliero E, Lapadula M. *Herramienta Integrada de Modelado y Simulación de Sistemas de Eventos Discretos*. Argentina: Diploma Work, FCEIA, UNR, 2002.
14. Mantegazza P, Dozio EL, and Papacharalambous S. RTAI: Real Time Application Interface. *Linux J* 2000; 72: 10.
15. Kofman E. Quantized-state control. a method for discrete event control of continuous systems. *Latin Amer Appl Res* 2003; 33(4): 399–406.
16. Zeigler B. *Theory of Modeling and Simulation*. New York: John Wiley & Sons, 1976.
17. Kofman E. A third order discrete event simulation method for continuous system simulation. *Latin Amer Appl Res* 2006; 36(2): 101–108.
18. Migoni G, Kofman E. Linearly implicit discrete event methods for stiff ODEs. *Latin Amer Appl Res 2009*; 39(3): 245–254.

19. Cellier F, Kofman E, Migoni G, and Bortolotto M. Quantized state system simulation. *Proceedings of SummerSim 08 (2008 Summer Simulation Multiconference)*, Edinburgh, Scotland, 2008.
20. Nutaro JJ. ADEVS (A Discrete EVent System simulator) C++ Library. 2005. http://www.ece.arizona.edu/~nutaro/index.php.
21. Wainer G. CD++ : a toolkit to develop DEVS models. *Softw Pract Exp* 2002; 32(13): 1261–1306.
22. DEVSJava. http://www.acims.arizona.edu/SOFTWARE/devsjava3.0/setupGuide.html.
23. The Mathworks. Real-time workshop 7.0. http://www.mathworks.com/products/rtw/
24. Gomez CE. *Engineering and Scientific Computing with Scilab*. Boston, MA: Birkhäuser, 1999.
25. Kofman E. Quantization-based simulation of differential algebraic equation systems. *Simulation* 2003; 79(7): 363–376.
26. Hildebrand D. An architectural overview of QNX. *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*. Berkeley, CA: USENIX Association, 1992, p.113–126.
27. Wehner C. *Tornado and VxWorks*. Books on Demand GmbH, 2006.
28. Härtig H, Baumgartl R, Borriss M, and Haman C-J. Drops: OS support for distributed multimedia applications. *EW 8: Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*. New York: ACM Press, 1998, p.203–209.
29. Barabanov M. *A Linux-based Realtime Operating System*. New Mexico: Master's thesis, New Mexico Institute of Mining and Technology, 1997.
30. Bovet D, Cesati M. *Understanding the Linux Kernel*. O' Reilly, 2002.
31. Yaghmour K. Adaptive Domain Environment for Operating Systems 2002. http://www.opersys.com/adeos/
32. Yaghmour K. Building a real-time operating systems on top of the adaptive domain environment for operating systems 2003. http://www.opersys.com/adeos/.