A Vectorial DEVS Extension for Large Scale System Modeling and Parallel Simulation

Federico Bergero*, Ernesto Kofman

Laboratorio de Sistemas Dinámicos. FCEIA - UNR. CIFASIS–CONICET.
27 de febrero 210 bis - (S2000EZP) Rosario, Argentina
Phone: +54 (341) 4237248 Ext. 336
bergero@cifasis-conicet.gov.ar kofman@fceia.unr.edu.ar
∗ Corresponding author

Date Received: December 11th 2012
Date Revised: July 25th 2013 (Major) and January 31th 2014 (Minor)
Date Accepted: March 5th 2014

Abstract

In this article we introduce an extension to the Discrete Event System (DEVS) formalism called Vectorial DEVS (VECDEVS) that allows to represent large scale systems in a graphic block diagram way. A pure VECDEVS model basically consist in an array of identical classic DEVS models that may differ in their parameters. The interconnection of VECDEVS models with some special classic DEVS models that can handle VECDEVS events allows to easily represent large systems of arbitrary structure.

A noticeable feature of this extension is that VECDEVS models can be easily split for parallel simulation. For that purpose, we developed an algorithm that automatically splits VECDEVS models into an arbitrary number of sub-models for parallel simulation.

The implementation of VECDEVS and the partitioning algorithm in a DEVS simulation tool is also described and its usage is illustrated through some application examples.

Keywords: DEVS, Large Scale Modeling, Parallel Simulation, Load Balancing

1 Introduction

Most modeling tools provide graphical interfaces that allow to build models creating and connecting different subsystems. The usage of modular hierarchical coupling of subsystems is the typical way to compose systems in most technical domains.

However, in presence of large scale systems, graphical modular hierarchical coupling is not enough since it becomes impractical (or even impossible) to create and connect thousands of models in a graphical way.

For that purpose, different modeling languages and tools allow the usage of *vectorial models*, where components with a simple graphical representation can contain several identical instances of elementary subsystems.

DEVS[1] is the most general formalism for discrete event system modeling as it allows representing any system provided that it performs a finite number of changes in finite intervals of time. This includes any type of discrete event systems, discrete time systems and even continuous time and hybrid systems through the usage of Quantized State Systems (QSS) approximation algorithms[2].

An important feature of QSS algorithms is that they exploit sparsity and handle discontinuities in a very efficient manner. Thus, they are particularly suitable for the simulation of large scale hybrid systems as it was shown in several applications[3,4,5], where noticeable advantages over classic algorithms for continuous system simulation have been demonstrated. A remarkable fact is that, in those applications, QSS methods show a linear growth of the computational load with the system size, while in classic integration methods it grows at least quadratically.

In spite of these advantages, DEVS does not have appropriate extensions to easily represent these type of large scale models.

The main DEVS extension to support modeling and simulation of large scale systems is Cell–DEVS[6]. However, cellular models cannot easily represent large systems with complex structures and a vectorial extension appears as a necessary tool.

An alternative way to build large scale DEVS models is by means of automatic generators, that following certain rules, construct complex structures of several components[7]. However, this approach involves the extra work of having to code and test the generator software. Also, the description of the constructed model may result too large to be handled by the simulation tool. Thus, in many applications a direct methodology for large scale modeling may provide a better solution.

Motivated by this need, we propose here the Vectorial DEVS[a] formalism, which provides a way to represent large scale models in a simple manner.

An atomic VECDEVS model consists in an array of identical atomic classic DEVS models (that can differ only in the values of certain parameters). Atomic VECDEVS can be coupled in a hierarchical way and the coupling is equivalent to an array of identical coupled DEVS models not connected between them. Thus, in principle, pure VECDEVS models allow to easily represent several copies of eventually complex coupled DEVS models.

The VECDEVS modeling technique is completed with some special atomic DEVS models that can manipulate the VECDEVS events to provoke connections between the different copies and also to connect VECDEVS models with regular DEVS models. That way, the new formalism allows to represent large scale systems of complex structure.

The simulation of large scale systems usually requires using parallelization techniques. One of the main features of VECDEVS is that models can be easily split for parallel simulation and we developed an algorithm for automatic partitioning of Vectorial DEVS models. This algorithm can be used to simulate large scale models in parallel using techniques such as *Adaptive Scaled Real Time Synchronization* (ASRTS)[3].

Besides defining the VECDEVS formalism, we have implemented a Vectorial DEVS library and programmed the automatic partitioning in the DEVS simulation tool PowerDEVS[9], providing also some test cases.

It is worth mentioning that the initial PowerDEVS implementation of VECDEVS was previous to the definition of the formalism itself. Therefore, the formalization of Vectorial DEVS is not mandatory for its implementation. However, by defining the VECDEVS formalism we provide a common abstract representation that can be seamlessly implemented in any other DEVS simulation tool. Moreover, without the formalization of VECDEVS it would have been impossible to propose and implement the automatic partitioning algorithm.

[a]There is a DEVS extension named Vector-DEVS[8] for specifying geographic information system (GIS) models. However, in Vector-DEVS the concept of *vector* refers to the data representing spatial coordinates meanwhile in Vectorial DEVS the concept of *vector* refers to arrays of models.

The paper is organized as follows: Section 2 introduces the main concepts used in the rest of the article and reviews some similar approaches from the literature.

Then Section 3 introduces the VECDEVS formalism and develops the tools for interconnecting VECDEVS models, and Section 4 presents the PowerDEVS VECDEVS library implementation. Those main results are completed with the automatic partitioning algorithm described in Section 5.

The usage of the new tools is then illustrated with some examples in Section 6 and finally, Section 7 concludes the article with a brief discussion about the current results and some future ideas.

2 Background and Related Work

In this section we introduce the concepts used along the rest of the article and we analyze some related work from the literature. We first introduce the DEVS formalism and then the Quantized State System (QSS) Methods, that allow to approximate continuous and hybrid systems with DEVS models. Then we describe some DEVS simulation tools and we focus on PowerDEVS, the DEVS simulation environment in which the actual implementation of our work is based on. Finally, we analyze some related work on large scale DEVS models (focusing in Cell–DEVS) and we discuss about different languages with vectorial modeling capabilities.

2.1 DEVS Formalism

DEVS (Discrete EVent System specification) is a formalism introduced first by Bernard Zeigler[10,1].

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:

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

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.
- $ta : S \to \Re_0^+$ is the time advance function, which says how long the system remains in a given state until it performs an *internal transition.*
- $\delta_{\text{int}} : S \to S$ is the internal transition function that defines the new state after the time advance expires.
- $\lambda : S \to Y$ is the output function defining the output event value produced before each internal transition.
- $\delta_{\text{ext}} S \times \Re_0^+ \times X$ is the external transition function, which defines the new state after an input event arrives.

This formalism is also called *classic DEVS* to distinguish it from *Parallel DEVS*[1] (an extension of classic DEVS 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 coupling DEVS models 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.

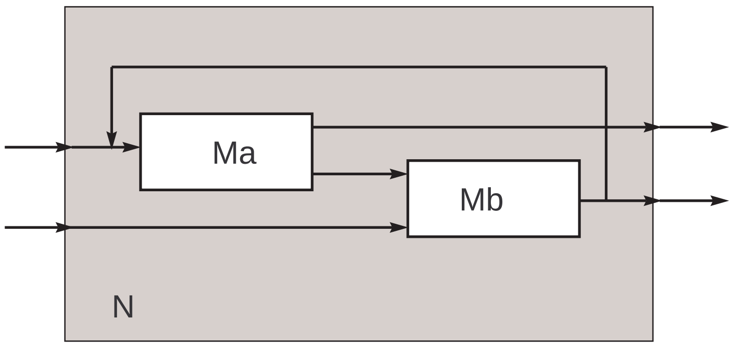

Figure 1: Coupled DEVS model

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

DEVS coupled models can be easily simulated. The simplest way of doing it is writing a program with a hierarchical structure equivalent to the hierarchical structure of the model to be simulated[1].

2.2 Quantized State System Methods

The DEVS formalism allows to represent any type of discrete systems (discrete event and discrete time). Moreover, numerical approximations of continuous time and hybrid systems can be also represented by DEVS models. While most numerical integration methods produce discrete time approximations (which can eventually be represented by DEVS), there are some algorithms that directly approximate continuous time systems by DEVS models.

These algorithms are called *Quantized State Systems* (QSS) methods[11,2] and they replace the time discretization by the *state quantization.*

A continuous time system can be written as a set of ordinary differential equations (ODEs):

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

where $\mathbf{x} \in \Re^n$ is the state vector. The different QSS methods approximate the ODE by

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

where $\mathbf{q} \in \Re^n$ is the quantized state vector. The state and the quantized state vectors are componentwise related by a *quantization function.* The usage of different quantization functions leads to different QSS algorithms such as QSS1[11], QSS2[12], QSS3[13], LIQSS(1,2,3)[14,5], BQSS and CQSS[15].

QSS methods have nice stability and error bound properties[2], and they are particularly efficient to simulate continuous systems that exhibit frequent discontinuities.

2.3 DEVS Simulation Tools

There are numerous tools for modeling and simulation of DEVS models. Some of them consist in simulation libraries where the user must describe the model programatically. Other tools offer graphical user interfaces.

The first category includes ADEVS[16], a C++ library that allows the use of two extension of the DEVS formalism: Parallel DEVS[17] and Dynamic DEVS[18]. DEVSJAVA and DEVS/HLA[1] are in this category also. Both tools, written in JAVA, support the execution of the simulation in a multi–processor system and in real time.

CD++[19] is based in Cell–DEVS, which supports real–time simulation and it constitutes one of the most extended and elaborated DEVS–based simulation tools.

Finally PowerDEVS is an environment focused on continuous and hybrid system simulation based on QSS approximations. Since the main applications of this article are related to hybrid system simulation, we developed the VECDEVS extension and libraries in PowerDEVS. Thus, we shall focus on this simulation tool and provide a more detailed description of that tool below.

2.4 PowerDEVS

PowerDEVS[9] is a general purpose tool for DEVS simulation. It consists of two main modules: The main graphic user interface (GUI) that allows the user to describe DEVS models in a block diagram fashion as shown in Figure 2, and the simulation engine which is in charge of the simulation.

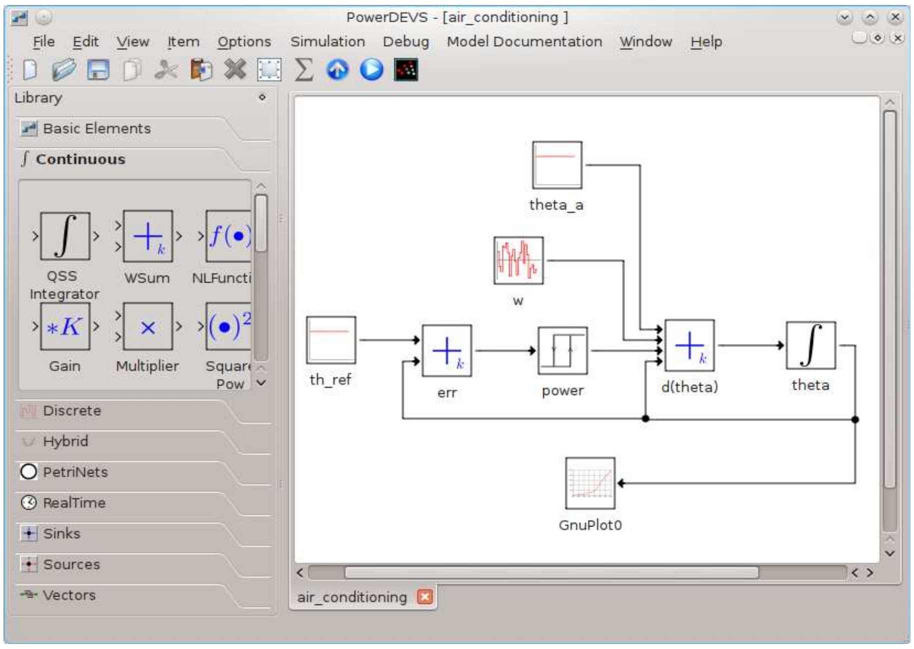

Figure 2: PowerDEVS Model Window.

An auxiliary tool (PowerDEVS pre–processor) translates the models into C++ descriptions that are compiled together with the simulation engine producing and executable simulation code.

PowerDEVS can also simulate continuous (and hybrid) models through the use of different QSS methods. For that goal, PowerDEVS distribution has a complete library for hybrid system simulation based on these algorithms.

End users can take the blocks from the libraries, build the block diagrams using the main GUI, and invoke the simulation in a straightforward way (without any knowledge about the DEVS formalism).

DEVS–aware users can easily build new blocks by defining the dynamics of the corresponding atomic DEVS model in C++ language using a simple GUI (called *atomic model editor*).

PowerDEVS Simulation Engine

Each simulation in PowerDEVS is executed by a C++ program that contains the simulation engine, the model structure (obtained from the graphical block diagram representation) and the C++ classes corresponding to the atomic models used in the system.

Atomics models associated to the same code belong to the same class. For example the *Integrator* model in Figure 2 belongs to the *Integrator* class defined in the files `integrator.h` and `integrator.cpp`. These files contain the code associated to the atomic model, that is, its internal and external transition functions, the time function, the output function and the state definition.

All atomic model classes are inherited from the *simulator* class. This is an abstract class that acts as an interface to deal with atomics models.

The external transition and and output functions take and return instances of the class *Event*, used to represent the corresponding event values. The *Event* class has the following properties:

- Event.port: an integer that refers to the corresponding input or output port through which the event is transmitted.
- Event.value: a `void *` (pointer to void) that allows the value of the event to take any arbitrary type.

The hierarchical structure of coupled models is implemented in the *Coupling* class. Each object of this class is associated with a DEVS coupled model that contains a list with internal/external connections, atomic and coupled child models. Following the closure property of DEVS, the *Coupling* class is inherited form the *Simulator* class.

The objects that form the hierarchical structure are instantiated at initialization. This initialization function is generated automatically by PowerDEVS since it depends on the model that must be simulated. In fact the only difference between the code that executes the simulation of different models is this initialization function and their connections.

2.5 Related Work

We now discuss some work related with the results presented in this article.

2.5.1 Large Scale DEVS Models

To the best of our knowledge, the only DEVS extension suitable for large scale modeling is Cell-DEVS[20], a DEVS–specialized Cellular Automata (CA).

A CA is a regular n-dimensional lattice in which each of the cells can take a finite value. States in the lattice are updated according to a local rule in a simultaneous, synchronous way, and cell states change in discrete time steps.

Cell–DEVS is a combination of DEVS and CA with explicit timing delays where each cell is defined as an atomic DEVS model, and a procedure to couple cells is defined.

Cell–DEVS models have been widely used by the DEVS community in several applications including (fire spreading, ecology, partial differential equations, etc).

However, its CA formulation restricts the usage of Cell–DEVS to large scale systems with regular inter–connection structures. Thus, it is not the most appropriate tool for modeling large scale systems with arbitrary structures.

2.5.2 Vectors of Models

The are several modeling languages and tools that allow defining vectors of models. That way, several copies of a single model can be easily defined and interconnected producing large scale models that can have more complex structures than those provided by cellular automatas.

Among the different modeling languages that permit defining vector of models we can mention:

- Statecharts[21]: This graphical modeling language is an extension of state machines and state diagrams. It is used to specify and design complex Discrete Event Systems. Statechart users can describe vectorial models to represent, for instance, N models of type M. For that goal, a three-dimensional box can be drawn where each layer is a model of type M (see Fig 3). Anyway, the connections between models is restricted. All the models are connected in a regular structure since they all share the same description except for a parameter value.

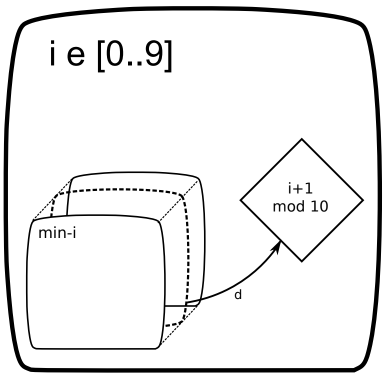

Figure 3: StateChart representation of vectorial models

- Modelica[22,23] is an object-oriented, equation-based language which allows the representation of continuous as well as hybrid models using sets of non-causal equations. The Modelica language provides a standardized way to model complex physical systems containing mechanical, electrical, hydraulic, thermal, and general multi–domain subcomponents. Being an object-oriented language, the Modelica user can define an array of models and connect them programatically. Modelica graphical interfaces such as Dymola[24] and OMEdit (by OpenModelica[25]), enable to connect vectorial models in a graphical way.

- Mathwork's Simulink[26] is an environment for multi-domain simulation and Model-Based Design for dynamic and embedded systems, which works as part of Matlab. This block–diagram modeling tool supports vectorial modeling. It has the concept of *Bus* where multiple signals can be transmitted in the same "wire". When a *Bus* is connected to a scalar block (for example an integrator), the block is replicated as many times as the bus size, thus obtaining a vectorial version of the block. Simulink has blocks to multiplex and demultiplex scalar signals to bus signals and vice-versa and blocks to select a specified subset of the elements of the bus. This gives Simulink the ability to model vectorial models while keeping the graphical description.

An alternative way to the construction of vectors of models is the use of single models handling multidimensional input and output signals. In fact, this is the way Simulink works (at least from an end user point of view).

However, in the context of the DEVS formalism this approach does not make sense as each event represents a change in a particular place of the system. Thus, a multidimensional signal at each instant of time would only contain a single value at one component, with the remaining vector entries having a null value (i.e., *NO_EVENT*).

3 The Vectorial DEVS Formalism

In this section we introduce the new DEVS extension called Vectorial DEVS. We first discuss a motivating example to show the need of the new formalism and then we formally define the *parametrized DEVS models* (which constitute the basis of VECDEVS) and then the Vectorial DEVS formalism. After analyzing the semantics of the new formalism, we introduce some special *Interface* DEVS models that allow to represent more complex structures.

3.1 Motivating Example

The following example was taken from Perfumo et al.[27], and it is a model proposed to study the power consumption of a large population of Air Conditioners (AC).

The idea is that each AC controls the temperature of a different room. Each room has a different temperature, thermal resistance and thermal capacity, and it suffers from different disturbances.

Each AC keeps the room temperature close to a common temperature reference, turning on and off the cooling system. To this end, each AC unit follows a hysteretic on–off law.

The evolution of one room temperature $\theta(t)$ is described by a differential equation:

$$\frac{d\theta(t)}{dt} = -\frac{1}{C \cdot R}[\theta(t) - \theta_a + R \cdot P \cdot m(t) + w(t)], \tag{3}$$

where R and C are the thermal resistance and capacity of the room, respectively. P is the power of the air conditioner when it is in its *on* state, θ_a is the outside temperature, and $w(t)$ is a noise term representing thermal disturbances. The term $m(t)$ represents the on–off control of the AC, i.e. $m(t) = 1$ when the AC is on and $m(t) = 0$ when it is off.

This system is a large scale hybrid model. As it was mentioned earlier, the QSS algorithms are very efficient to simulate systems of this type.

Figure 2 shows the model of this AC unit in PowerDEVS. There, each block is a DEVS atomic model resulting from the usage of QSS methods to approximate Eq.(3).

The model used in the article[27] is in fact composed by 10000 AC units, where every room is characterized by different parameters R and C, receiving a different disturbance $w(t)$ and where every AC unit has a different power consumption P. Then, the total power is the result of adding the individual power consumption of all the AC units (the power is at the output of hysteresis block labeled 'power' in Fig. 2).

Thus, if we want to build the whole model, we should replicate the model of Fig. 2 10000 times, changing the parameters accordingly and then we should create a sum block with 10000 input ports to obtain the total power.

Of course, we can simplify the problem using hierarchical coupling and forming clusters of 10 AC units, and then clusters of clusters until we get a whole model with 10000 ACs. However, that solution does not help much, as in the end we will be dealing with a huge model anyway (although better organized than before).

To take into account that in vectorial models the different components can have different parameters, we shall define first a formal extension of DEVS called *Parametrized* DEVS.

3.2 Parametrized DEVS Models

Parametrized DEVS models attempts to provide a tool to formally define sets of identical DEVS models that differ in some given parameters.

An atomic model $M(p)$:

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

where p is a *parameter* belonging to an arbitrary *parameter set* P, such that functions $\delta_{\text{int}}, \delta_{\text{ext}}, \lambda$ and ta may depend also on p will be called *Parametrized DEVS Model.*

Notice that two DEVS models $M(p_1)$, $M(p_2)$ with $p_1 \neq p_2$ can exhibit different behavior. However, they share the same input, output and state sets (X, Y, and S, respectively).

In the motivating AC example, we can define the parameter set P as:

$$P = \Re^+ \times \Re^+ \times \Re^+$$

so that each parameter $p \in P$ has the form (R, C, P_w), i.e., the values for the resistance, capacitance and AC power.

After the definition of parametrized DEVS models, we can define the Vectorial DEVS formalism.

3.3 Vectorial DEVS Definition

Given the set of scalar parametrized DEVS models:

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

a Vectorial DEVS model is defined as a structure:

$$V_D = \{N, X_V, Y_V, P, \{M_i\}\}$$

In this structure,

- $N \in \mathbb{N}$ is the vector dimension.
- $X_V = X \times \mathit{Index} \cup \{-1\}$ is the set of vector input events. X is the set of scalar input events and $\mathit{Index} = \{1, \cdots, N\}$ is the set of indexes saying which is the scalar atomic DEVS model that receives the event.
- $Y_V = Y \times \mathit{Index}$ is the set of vector output events. Y is the set of scalar output events and $\mathit{Index} = \{1, \cdots, N\}$ is the set of indexes saying which is the scalar atomic DEVS that emits the event.
- P is a parameter set.
- For each $i \in \mathit{Index}$, $p(i) \in P$ is a parameter and $M_i = M(p_i)$ is the corresponding scalar parametrized DEVS model.

3.4 Semantics of Vectorial DEVS Models

Input Events

When a Vectorial DEVS model V_D receives an input event with value $(x \in X, i \in Index)$, it executes an external transition in the scalar atomic DEVS model M_i with value x.

Additionally, if the VECDEVS model receives an input event with values $(x \in X, -1)$, then it executes the external transition of all the scalar atomic models with value x.

Output Events

When the scalar model M_i produces an output event with value $y \in Y$, the Vectorial DEVS model V_D sends an output event with value (y, i).

If two (or more) scalar models, M_i and M_j schedule an output event for the same time, the Vectorial DEVS model sends the event from the model with the smallest index.

3.5 Coupling Vectorial DEVS models

Vectorial DEVS models can be coupled among them like regular DEVS models. Of course, the corresponding input and output sets and the size (N) of two interconnected Vectorial DEVS models must be consistent.

When some Vectorial DEVS models $(V_{D_1}, V_{D_2}, \ldots, V_{D_r})$ of dimension N are coupled, the structure obtained is equivalent to that of N independent coupled DEVS scalar models $(M1_1, M2_1, \ldots, Mr_1)$, $(M1_2, M2_2, \ldots, Mr_2)$, $\cdots$, $(M1_N, M2_N, \ldots, Mr_N)$. Here, Mk_i is the scalar DEVS model with index i corresponding to the VECDEVS model V_{D_k}.

The N coupled scalar DEVS models will be identical to each other except for their parameters.

As we mentioned above, the coupling of r Vectorial DEVS models of dimension N represent N disconnected systems since an event generated in a scalar model with index i will only be propagated to other scalar models with the same index.

The interconnection of components with different indexes need the usage of some special DEVS models that we shall describe later on.

Going back to our motivating example, if we want to model a population of 10000 AC units, we can define a VECDEVS model of dimension 10000 for each block in Fig 2 and then we can couple the Vectorial DEVS models following the same structure of that Block Diagram.

Figure 4 shows a PowerDEVS a graphical representation of the Vectorial DEVS model which looks almost identical to that of Fig 2. Here each green block is actually a Vectorial DEVS model of size 10000, that is, each one contains 10000 instances of the underlying scalar block. We see also that there are two scalar blocks (the source blocks for the reference temperature and the outside temperature) connected to Vectorial DEVS through special *scalar to vector* blocks.

We shall introduce those blocks below.

3.6 Interface Models for Vectorial DEVS

Pure Vectorial DEVS models are limited to replicate an arbitrary number of instances of an atomic or coupled scalar DEVS model.

In order to represent large models with more complex structures we need tools that allow to interconnect scalar models with different indexes and, eventually, to interconnect Vectorial DEVS models with scalar DEVS models.

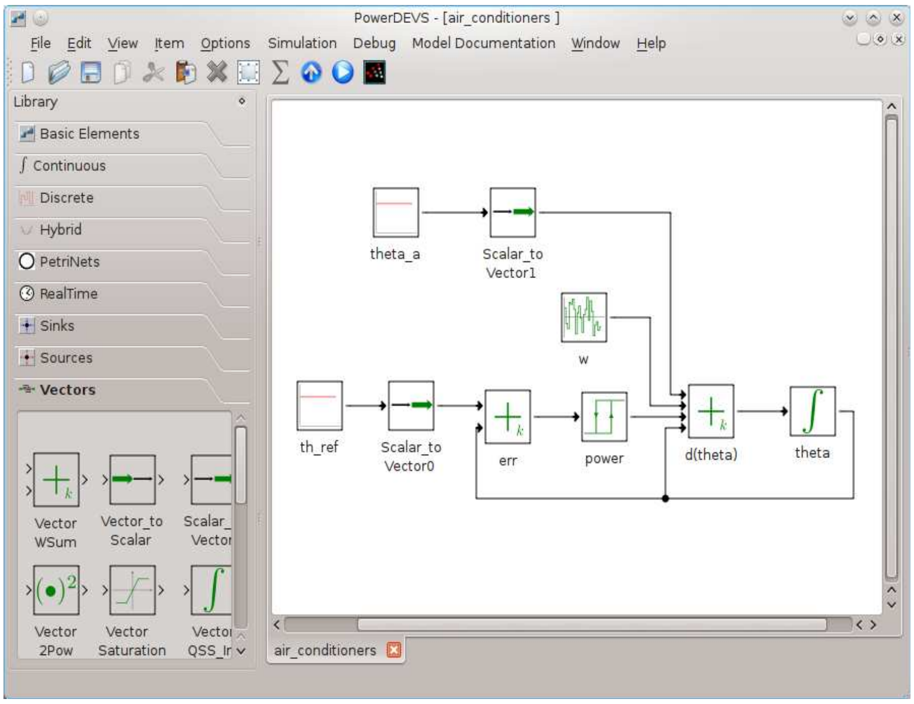

Figure 4: PowerDEVS Model of 10000 AC units using Vectorial DEVS

Thus, we introduce next some interface DEVS models that accomplish that goal. These blocks are not Vectorial DEVS models. They are simple scalar DEVS models that can receive and/or send vector events.

Event Routing Blocks

If we want to interconnect two Vectorial DEVS models so that the events emitted in the first one with index i arrive to the second one with index $j \neq i$, we need to use an *interface* DEVS model that modifies the vector event index.

We introduce then three different DEVS models with this functionality.

- Index Shift: The simplest DEVS model to manipulate vector event indexes is the *Index Shift*. When it receives a vector event with value (x, i), it immediately sends an output event with value $(x, i + sh)$ where sh is an integer parameter.

  The index shift block has in fact two output ports. It normally sends the output events through the first port. However, when $i + sh$ is greater than the dimension N, the event is sent through the second port with index $i+sh-N$. Similarly, when $i+sh$ is less than 1, the event is also sent by the second port with index $i+sh+N$. This allows the user to decide discarding or using the events that become out of range after the index shifting. The usage of this block is illustrated in the model of a transmission line described in Section 6.4.

- Index Map: A more general event routing mechanism can be defined by a binary incidence matrix $IM_{N\times N}$. There, $IM_{i,j} = 1$ indicates that all events generated with index i must be retransmitted with index j. Using this block between two Vectorial DEVS models, an arbitrary connection structure can be obtained.

The usage of the Index Map is illustrated in the model of a spiking neural network discussed in Section 6.3

- Index Selector: This model acts as an event filter, forwarding only the events that satisfy a given predicate on their index. The transmitted events do not change their index nor value and the remaining events are just discarded.

  For instance, given the predicate:

  $$Pred(i) = i > 5 \ \wedge \ i < 200$$

  the *Index Selector model* will only propagate the events with index between 5 and 200. Notice that if the predicate is static (i.e., it does not change with the time) the *Index Selector* is a specialized version of an *Index Map* with a diagonal matrix IM where $IM_{i,i} = 1$ if and only if i satisfies the predicate.

  The usage of the Index Map is also illustrated in the model of the spiking neural network of Section 6.3

Using the *Index Map* block any connection structure can be defined, including those obtained using *Index Shift* or *Index Selector* blocks. However, the *Index Map* block requires defining a complete incidence matrix and its usage has a greater computational cost than that of the other blocks. Thus, whenever it is possible to use an *Index Shift* block or an *Index Selector* block they should be preferred over the *Index Map* block.

Interfacing Vectorial DEVS with regular DEVS models

If we want to connect a scalar DEVS model with a scalar component of a Vectorial DEVS model, we need to add or remove the index information from the event value. That goal is accomplished by the *Scalar to Vector* and the *Vector to Scalar* model interfaces.

- Scalar to Vector: This block simply adds the index i to the scalar event it receives, transforming it into a vector event. This is, when it receives an event with value x it sends an event with value (x, i) where i is the block parameter.

  This block is used, for instance, in the model of Fig 4 in order to propagate the same reference and outside temperature to all the scalar components representing each AC unit. There, the parameter is $i = -1$.

- Vector to Scalar: This block has a parameter i containing the index of the vector events to retransmit. When it receives a vector event with index $j = i$, it removes the index and sends the scalar event through its output port. If the index received is $j \neq i$, then the event is just discarded.

  The usage of this block is illustrated in the example of the transmission line in Section 6.4, where it is used to plot the trajectory produced by a scalar component of a vector integrator.

Throughout the description of event routing and interfacing blocks we have assumed that the connection parameters remain constant during the simulation, which results in a classic fixed structure DEVS model. Dynamical structure DEVS models[28] can be implemented by changing these parameters during the simulation. This idea is not developed in this work, but will be considered for future research.

4 PowerDEVS Implementation of Vectorial DEVS

A library of Vectorial DEVS models, including also the interface DEVS models described above, was implemented in PowerDEVS. In this section, we describe this implementation.

4.1 Vector Events

As we described in Section 2.4, PowerDEVS events are instances of the C++ class *Event*, formed by an integer representing the corresponding input or output port and a pointer to void representing the value which can belong to any user defined class.

As PowerDEVS libraries are intended to implement QSS methods, their events represent sections of polynomial trajectories. Thus, in spite of the fact that PowerDEVS allows to use any type of event values, all the library blocks use an array of 10 doubles. This is, the event values point to an array `double y[10]` where `y[0]` contains the first coefficient of the current polynomial section, `y[1]` contains the second coefficient and so on.

The PowerDEVS vector library was built in order to be compatible with the rest of the libraries, and so, the vector event values belong to a class `vector` defined as follows

```
class vector {
  double value[10];
  int index;
}
```

As C++ arrays are zero-indexed, the `index` values range from 0 to $N-1$ rather than from 1 to N as the formal definition states.

It is important to stress the difference between the port number and the index number. Both, scalar and vector events, are characterized by a port number (the component *Event.port* of class *Event* described in Sec. 2.4). The port number says through which input or output port the event is received or sent. On the other hand, the index number is only present in vector events and it refers to the scalar component of the Vectorial DEVS model that receives or sends the event.

4.2 Vectorial DEVS Models

Atomic Vectorial DEVS Models in PowerDEVS were implemented by regular atomic models that contain an array of atomic models as part of their state.

Formally, given a set of parametrized scalar models

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

where $p_i \in P$ is a parameter with $i \in Index = \{1, \cdots, N\}$, the associated atomic Vectorial DEVS model of size N can be described as follows:

$$V_M = (V_X, V_Y, V_S, V_{\delta_{\text{int}}}, V_{\delta_{\text{ext}}}, V_\lambda, V_{ta})$$

where:

- $V_X = X \times Index$
- $V_Y = Y \times Index$
- $V_S = \{(s_1, e_1), (s_2, e_2), \ldots, (s_N, e_N)\}$
- $V\delta_{\text{int}}(s) = \{(s'_1, e'_1), (s'_2, e'_2), \ldots, (s'_N, e'_N)\}$
  where $(s'_i, e'_i) = \begin{cases} (s_j, e_j + V_{ta}(s)) \text{ if } i^* \neq j \\ (\delta_{\text{int}}(s_j, p_j), 0) \text{ if } i^* = j \end{cases}$
- $V\delta_{\text{ext}}(s, (x, i), e) = \{(s'_1, e'_1), (s'_2, e'_2), \ldots, (s'_N, e'_N)\}$
  where $(s'_i, e'_i) = \begin{cases} (s_j, e_j + e) \text{ if } i^* \neq j \\ (\delta_{\text{ext}}(s_j, x, e_j + e, p_j), 0) \text{ if } i^* = j \end{cases}$
- $V_\lambda(s) = (\lambda(s_{i^*}, p_{i^*}), i^*)$
- $V_{\text{ta}}(s) = \min_i(\text{ta}(s_i) - e_i)$

where $i* = argmin_i(\text{ta}(s_i) - e_i)$, i.e., the index of the scalar component that performs the next transition.

Notice that most functions are just simple calls to the scalar component functions, so its implementation is straightforward. There are only two non–trivial issues:

- An algorithm is required to find the imminent scalar component $i*$.
- The scalar components must be parametrized during the initialization.

The algorithm to find the imminent component must be efficient. The problem is that VECDEVS is meant to deal with large scale models and something like a simple linear search cannot be accepted.

PowerDEVS engine already implements a heap structure inside coupled models in order to efficiently find which is the imminent child.

In order to reuse this feature, we formed the state of the Vectorial DEVS atomic model with a coupled model which contains the N atomic scalar components inside (without connections among them). The atomic scalar components are instantiated at runtime, during the initialization.

As an example, the code below implements in PowerDEVS an atomic Vectorial DEVS model where each scalar component is an atomic model that computes x^2 (it corresponds to a block of PowerDEVS continuous library associated to the class *xpower2*).

```
void vector_pow2::init(double t,...) {
  va_list parameters;
  va_start(parameters,t);
  char *fvar= va_arg(parameters,char*);
  N=getScilabVar(fvar ); //obtain the parameter N (dimension) defined in the GUI
  D0 = new Coupling("CoupledPow2"); //instatiate the coupled model
  Connection **EIC1 = new Connection* [0]; //connections are empty
  Connection **EOC1 = new Connection* [0];
  Connection **IC1 = new Connection* [0];
  Simulator **D1 = new Simulator* [N]; //create an array of N atomic models
  for (int i=0;i<N;i++){
        D1[i] = new xpower2("xpower2i"); //instantiate the atomic models
        D1[i]->init(t);                  //initialize the atomic models
  }
  D0->setup(D1,N,IC1,0,EIC1,0,EOC1,0); //build the coupled model
  D0->init(t);                         //initialize the coupled model
}
double vector_pow2::ta(double t) {
  return D0->ta(t); //this returns the time advance of the coupled model
}
void vector_pow2::dint(double t) {
  D0->dint(t); //this performs dint on the imminent child
}
void vector_pow2::dext(Event x, double t) {
  vector vec1=*(vector*)x.value; //cast the event value as a vector
  int index=vec1.index;
  if ((index>-1)&&(index<N)){
        D0->D[index]->dextmessage(x,t); //send the event to the scalar component
        D0->heap.update(index); //update the heap structure
} else if (index==-1) {
  for (int ind=0;ind<N;ind++){
        D0->D[ind]->dextmessage(x,t); //send the event to all scalar components
        D0->heap.update(ind);
 }
};
}
Event vector_pow2::lambda(double t) {
y= D0->D[D0->transitionChild]->lambdamessage(t); //obtain the imminent child's
output event
vec=*(vector*)y.value; //convert it to vector format
vec.index=D0->transitionChild; //add the right index
y.value=&vec;
return y;
}
```

4.3 Parametrization of Scalar Components

Vectorial DEVS offers the possibility of defining scalar components with different parameters.

In PowerDEVS, the block parameters are defined in the GUI (by double clicking on each block). Most blocks in PowerDEVS libraries, during their initialization routine, ask Scilab about the values they receive. That way, if the parameter is an expression like $a + 32$, Scilab will return the right value for the expression (assuming that variable a is defined in Scilab workspace).

This mechanism was exploited also for Vectorial DEVS models. Here, the vector atomic model asks Scilab about their parameter values. Then, it initializes their scalar components according to the following rule:

- If a parameter value is scalar, all the scalar component receive the same value for that parameter.
- If a parameter value is a Scilab vector V of size N, then the vector model initializes the i–th scalar component with value V_i for that parameter.

That way, if we want to create a vector model of dimension N where the scalar components have a different parameter value, we must use as parameter for the Vectorial DEVS model a Scilab vector of dimension N.

4.4 Vectorization of Scalar Models

Taking into account what was discussed before, a vector of previously defined scalar models in PowerDEVS can be created with the following procedure:

1. Define a new empty atomic model with the same number of input and output ports than the original scalar model and with the same parameters, adding one parameter to indicate the vector size N.
2. In the initialization function, instantiate a Coupled model and, inside it, create N scalar atomic models, initializing its parameters as it was explained before.
3. At the internal transition and time advance functions invoke the corresponding functions of the Coupled model.
4. At the external transition, decompose the received vector event value (x, i) and execute the external transition of scalar model i with event value x. If i is equal to -1, then execute the external transition of all children (with value x).
5. At the output function, compose the vector event value (x, i) with the output event value x and the index i of the imminent child.

This was the procedure used, for instance, to convert the scalar model *xpower2* into the corresponding vectorial model explained in Sec. 4.2. For other examples, including the usage of vector parameters, we refer the reader to the source code of PowerDEVS library available at SourceForge[29].

5 Automatic Partitioning of Vectorial DEVS Models for Parallel Simulation

Parallel simulation of DEVS models is usually based on splitting a large model into p sub-models (where p is the number of processing units or processors) so that every sub-model has a similar workload and the event traffic between different sub-models is minimized to avoid the resulting communication and synchronization overhead. Techniques such as those presented by Bergero et al.[3] for parallelization of DEVS models require a previous stage of model partitioning which in presence of complex connection structures can be very complicated.

Pure Vectorial DEVS models consist in N identical disconnected scalar DEVS models. Thus, in this case the partition is straightforward. We can assign to the first processor all the scalar blocks with indexes from 1 to N/p. Then, we put in the second processor all the blocks with indexes from $N/p+1$ to $2*N/p$ and so on. That way, no events are transmitted between processors and the parallel simulation is straightforward.

However, in most applications we shall deal with non–pure Vectorial DEVS models. There, the presence of *index shift* and other interface DEVS models, produces more complex structures and the partitioning technique is no longer that simple.

In this Section, we study the problem of automatic partitioning of Vectorial DEVS models, starting with the trivial case of pure VECDEVS and then extending the results for the presence of *Index Shift* and *Index Map* blocks. We also discuss the implementation in PowerDEVS.

In all cases, the resulting split model is a coupled DEVS model that can be simulated by the regular DEVS simulation algorithm or using parallelization techniques like ASRTS[3].

There are alternative parallelization techniques like Map-Reduce[30] that are based on the parallel calculation of a single function with different input data. However, except for pure VECDEVS models, the different sub-models dynamically interact with each other and these kind of strategies cannot be applied in general cases.

5.1 Partition of Pure Vectorial DEVS Models

As we mentioned above, the basic idea to split a pure Vectorial DEVS model into p sub-models is to place the scalar components with indexes in a given range on the same processor.

This is equivalent to convert a vector DEVS model of dimension N into p identical vector DEVS models of dimension N/p. For simplicity, we shall assume that N can be divided by p. [b]

More formally, given a model M that is the result of coupling K Vectorial DEVS models $V_1, V_2, \cdots, V_K$ of dimension N, the partition algorithm consist in forming p disconnected coupled models identical to M except that each V_i have now dimension N/p.

This procedure for $p = 2$ is illustrated by Figure 5.

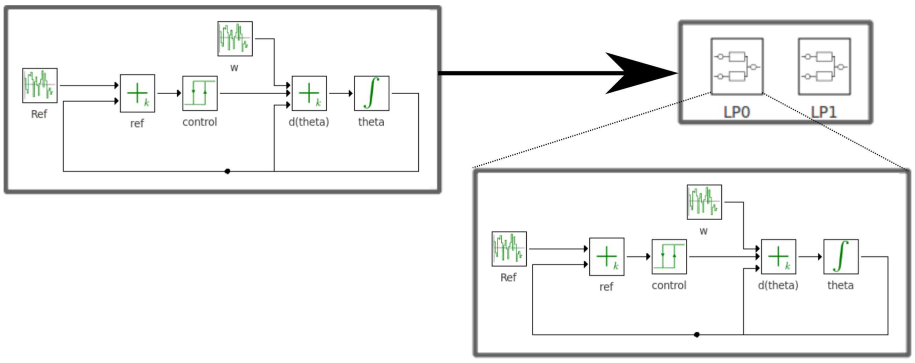

Figure 5: Partition of a coupled model with Vectorial DEVS.

5.2 Partitioning in Presence of Interface DEVS Models

Event routing DEVS models provoke connections between scalar components with different indexes. Thus, in presence of these blocks, the partitioned model will look like the model of Fig 5 but with connections between the different coupled models.

In addition, when Vectorial DEVS models and regular DEVS models are interconnected through the corresponding interface models (vector to scalar and scalar to vector), the partitioning algorithm must also place the scalar DEVS models in one or more processors.

Here, we describe different rules that extend the partitioning algorithm to deal with *Event Routing* blocks and interface DEVS models.

Partitioning with Index Shift Blocks

The Index Shift model is the simplest Event Routing block. It interconnects scalar components of index i with scalar components of index $i + sh$ (where sh can be a positive or a negative integer).

We shall assume that $|sh| > n/p$, i.e., that the index shift is smaller than the number of scalar components in one processor. In large scale models, this assumption will be always satisfied.

Thus, scalar components placed in the j–th processor can be only connected to scalar components placed in the processors $j - 1$, j, or $j + 1$.

[b]If N is not a multiple of p, then some of the resulting vector DEVS models will have different dimension.

Thus, for each *index shift* block we simply add an input port and an output port to each coupled model. When $sh > 0$ we connect the output port of each coupled model with the input port of the next one. Otherwise, we connect the output port with the input port of the previous one.

Internally, the new coupled input port is connected to all the models that were connected to the index shift's first outport. The new coupled outport, on the other hand, is connected with the second output port of the index shift block. That way, the events with index greater than N/p or smaller than 1 are transmitted to the neighbor coupled model and received by the correct scalar component.

For instance, the model of Figure 6 contains tow *index shift* blocks with parameters $sh = 1$ (*Index Shift 0*) and $sh = -1$ (*Index Shift 1*) connecting a vector integrator and a vector sum block.

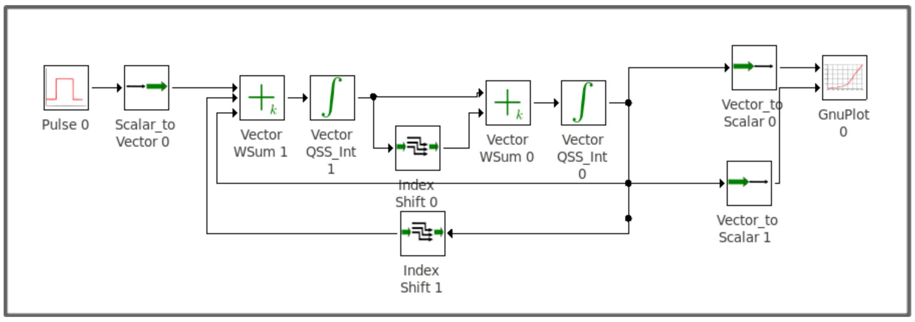

Figure 6: A Model with index routing blocks.

This model can be split as it is shown in Fig 7. In this new model, both sub-models are identical to the original model of Fig 7 except that the dimension of the Vectorial DEVS models is now $N/2$. We see that the new ports corresponding to *Index Shift 0* (the first ports in LP0 and LP1) connect the j-th processor with the $j-1$ (LP1 with LP0) because the *Index Shift 0 sh* parameter was -1. On the contrary the ports corresponding to *Index Shift 1* (the second ports in LP0 and LP1) connect the j-th processor with the $j+1$ (LP0 with LP1) because the *Index Shift 1 sh* parameter was 1.

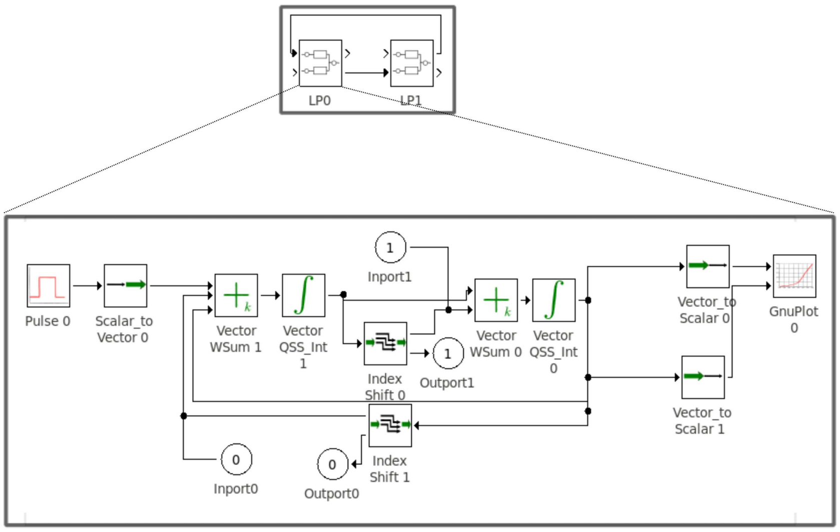

Figure 7: Split model with index routing blocks.

It is not difficult to realize that the models of Figs.6 and 7 have identical behavior.

Partitioning with Index Map Blocks

The Index Map is a more advanced event routing block, where the user can implement all the possible connection schemes (See Section 3.6).

Let us suppose that we want to partition a model that connects Vectorial DEVS components of dimension N with an Index Map block characterized by a binary matrix IM of dimension $N \times N$, as shown at the left side of Fig.8. Then, we proceed as follows:

- We generate p sub-models containing the VECDEVS models but now with dimension N/p.
- At every sub–model, we add p *Index Map* blocks.
- We split the matrix IM into $p \times p$ sub-matrices $IM_{i,j}$ of size $N/p \times N/p$ so that

$$IM = \begin{bmatrix} IM_{1,1} & \cdots & IM_{1,p} \\ \vdots & \ddots & \vdots \\ IM_{p,1} & \cdots & IM_{p,p} \end{bmatrix}$$

  and we characterize the j–th *Index Map* block of the i–th sub–model with the binary sub-matrix $IM_{i,j}$.
- Then, at every sub-model we add p external input and output ports and we connect the j–th *Index Map* to the j–th output port with $j = 1, \ldots, p$.
- For each connection of the original model from a component to the *Index Map* block, at each sub-model we add p connections from the component to the p *Index Map* new blocks.
- For each connection of the original model from the *Index Map* block to a component, at each sub-model we add p connections from the p external input blocks to the component.
- Then, we add the connections between sub-models so that the j–th output of the i–th sub–model is connected to the i–th input port of the j–th sub-model.
- Taking into account that DEVS forbids the connection of a sub-model with itself, the i–th input and output port of the i–th sub–model must be removed and the corresponding connections must be replaced by direct connections from the i–th *Index Map* to the components.

In Figure 8 we see on the left the original model with an Index Map block and on the right the second sub-model resulting of the partitioning algorithm here described with $p = 4$.

Note that the *Index Selector* block is a special case of the Index Map, thus the same rules apply to its partitioning.

Interconnection with Scalar Blocks

The interconnection of Vectorial DEVS models and scalar DEVS models is performed through the interface *Scalar to Vector* and *Vector to Scalar* blocks.

In presence of these blocks, the partitioning algorithm proceeds replicating all the scalar and interface blocks in all the processors, and modifying the interface block parameter to obtain an equivalent behavior.

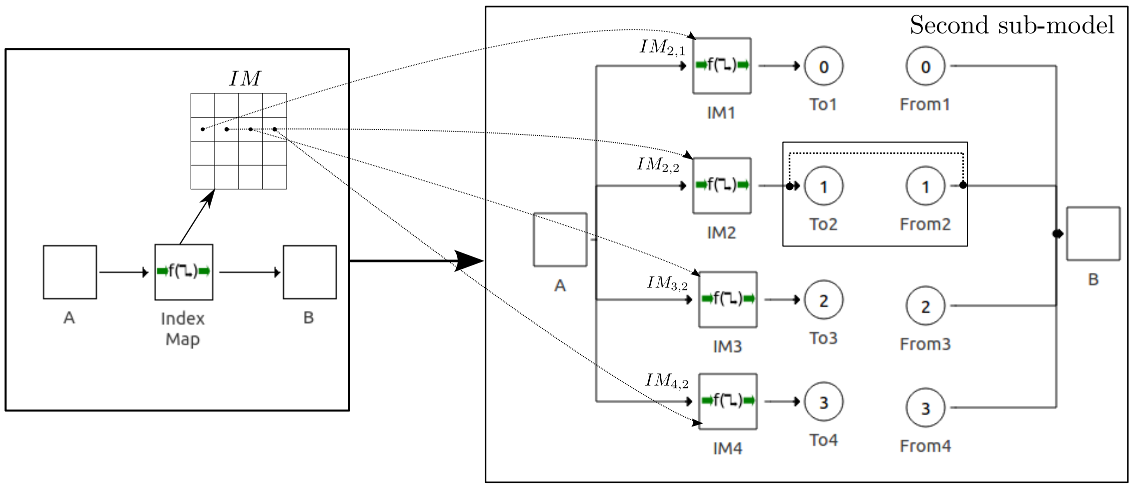

Figure 8: Split model with index routing blocks.

Let us analyze first the *Scalar to Vector* interface, which converts the scalar events to vectorial events with index j being j the block parameter.

When the parameter is $j = -1$, the block acts as a broadcaster that sends the received scalar events to all the scalar components of the vectorial model. Thus, in this case, we do not need to modify the parameter.

When the parameter is $j \neq -1$ all the scalar events received are only sent to the j–th scalar component. Then, we modify the parameter of the replicated *Scalar to Vector* block at the k–th processor as:

$$j_k = j - (k-1) \cdot N/p$$

That way, at first processor the parameter is $j_1 = j$, at the second the parameter is $j_2 = j - N/p$, etc. That parameter has only a valid value when $1 \geq j_k \geq N/p$, which only occurs at the processor where the original j–th scalar component is placed. In the remaining processors, the value is either negative or greater than N/p and the events are discarded. Thus, the scalar events are only transmitted to the correct scalar component.

For the *Vector to Scalar* interface the procedure is analogous and the parameters are modified following the same rule as before.

5.3 Model Partitioning Algorithm

We summarize here the algorithm for partitioning a coupled model M with Vectorial DEVS models of size N into p sub–models.

1. We create p disconnected coupled models $M_1, M_2, \cdots, M_p$, identical to the original coupled model M.

2. We resize each vectorial model in M_i from N to N/p (we assume that p divides N).

3. For each *Index Shift* block, we add an output port and an input port to all the coupled models M_i. Then, we create a connection from the second output port of the *Index Shift* to the new coupled model output port and another connection from the new coupled model input port to the input of the *Index Shift* block.

4. If the parameter of the *Index Shift* block is $sh > 0$, then we add a connection from the new output port of M_i to the new input port M_{i+1}, for all $i < N$. Otherwise, if $sh < 0$, we add a connection from the new output port of M_i to the new input port M_{i-1} for all $i > 1$.

5. For each *Index Map* block we replicate p instances in each of the p sub–models and connect them following the rules described in Section 5.2.

6. For each *ScalarToVector* or *VectorToScalar* interface inside coupled model M_k with index parameter $i \neq -1$, we change the index according to

$$i_k = i - (k-1) \cdot N/p$$

.

6 Applications and Examples

In this section we present some examples that illustrate the usage and different applications of the Vectorial DEVS extension.

The simulation results reported below were obtained using a CPU with an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz processor running under Linux–Ubuntu Operating System.

6.1 Queue-Processor System

In this example we study a pure Discrete Event System, namely a Queue-Processor system. The model consists of N identical queue-processor sub-systems with an admission control mechanism.

Each sub–system contains a queue that stores the jobs received and a processor that computes the jobs and notifies the queue whenever a job is completed so the queue sends a new job. The admission control block receives jobs and decides whether to send them to the Queue or to discard them according to the number of jobs currently stored in the Queue.

The jobs are generated by a *Job Generator* block and sent to the first queue-processor sub-system. When the queue of the first sub-systems gets full, the jobs are discarded and sent to the second queue-processor sub–system. The model is connected as a chain, i.e. the jobs discarded in the n-th queue-processor sub-system are sent to the $n+1$ th subsystem. The discarded jobs from the last queue-processors are lost.

Figure 9 shows the hand–made (non vectorial) PowerDEVS model with $N = 10$.

We can easily model this system using VECDEVS. First, we have to develop vectorial versions of the blocks (as described in Sec. 4.4) and then connect the discard port to the input port of the *Admission Control* block using an index shift (with a shift of +1). Figure 10 shows the resulting VECDEVS model.

Figure 11 shows the simulation results of the first queue-processor system, where we can see the action of the admission control block (in blue), discarding jobs when the queue gets full (green pulses), and sending more jobs when the queue has space (red pulses).

Table 1 compares the compilation and simulation time between the VECDEVS and the non vectorial model for dimension $N = 10, 100, 500$. The simulation times of the VECDEVS model are slightly larger than those of the non–vectorial model in this case. This can be explained by the presence of an extra *Index Shift* block, which increases the number of events. On the other hand, the compilation time remains constant in the VECDEVS model (in fact, the VECDEVS model is always the same,

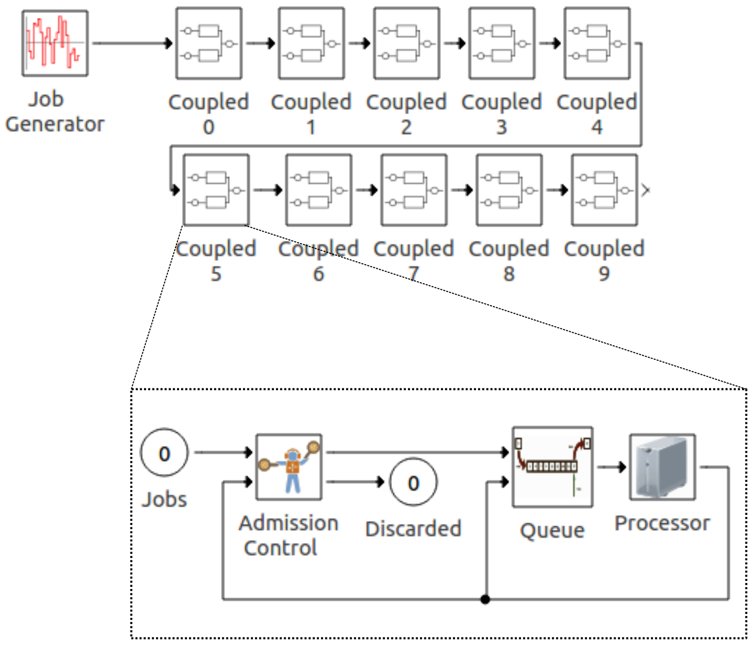

Figure 9: The Queue-Processor sub-system

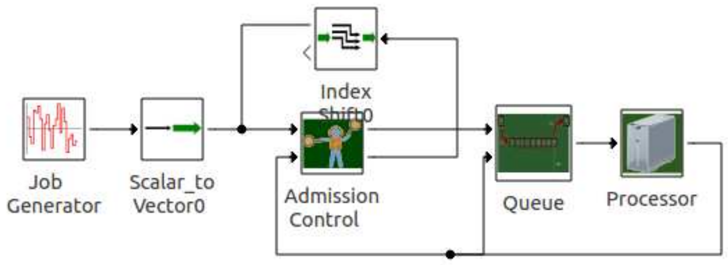

Figure 10: The VECDEVS Queue-Processor sub-system

independent of the size given by the parameter N), while it grows considerably with N in the non vectorial case.

It is worth mentioning that while the size of the file describing the VECDEVS model is independent of the dimension (9.8 KB in the three cases), in the non vectorial version it increases with N (138 KB, 7.8 MB, and 185 MB respectively).

Also, the modeling process of the VECDEVS version is much simpler and faster and the system dimension can be changed without modifying the model, just by changing the parameter N.

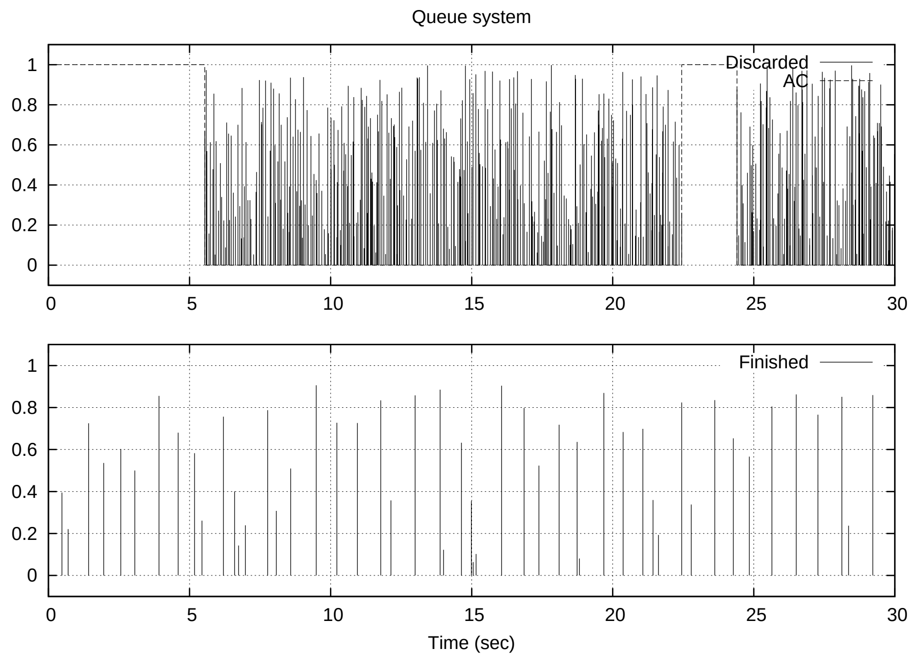

Figure 11: Simulation results of a queue–processor system with admission control.

Table 1: Compilation and Simulation times (in seconds) vs. model dimension (N) for the Queue–Processor system.

	10		100		500	
	Comp.	Sim.	Comp.	Sim.	Comp.	Sim.
VECDEVS	0.18	0.85	0.18	4.69	0.18	17.6
Hand-Made	0.2	0.59	1.24	3.91	7.3	14

6.2 State Space model

Continuous time linear time invariant systems play a key role in Automatic Control applications. These systems are usually represented in the following *state–space* form:

$$\dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t)$$

where $\mathbf{x}(t)$ and $\mathbf{u}(t)$ are the state and input vector, respectively. A and B are the evolution and input matrices.

Representing a QSS approximation of a system like this using scalar DEVS models can be very uncomfortable when the dimension of $\mathbf{x}(t)$ is large. The reason is that the model requires using a large number of integrators and gain components.

However, using Vectorial DEVS the model becomes straightforward. Figure 12 shows the corresponding PowerDEVS model build with blocks from the vectorial library.

In this model, besides standard, vectorial and interface DEVS models, there is a new block called *Matrix Gain* that processes vectorial events multiplying a matrix by the input event value.

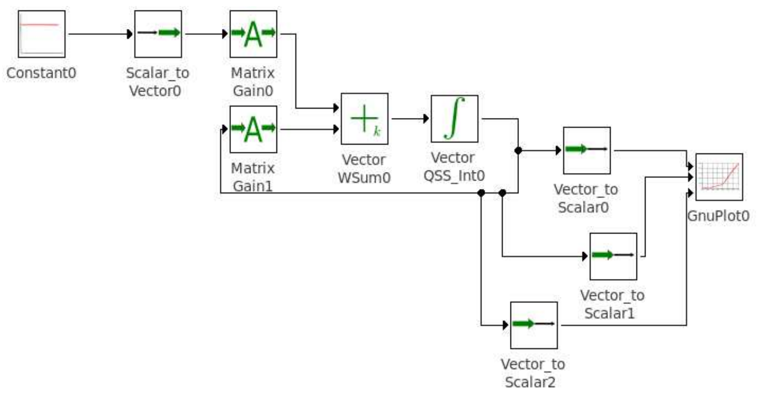

Figure 12: PowerDEVS generic State Space Model using VECDEVS

The different blocks (integrators, weighted sum, and matrix gain) can take their parameters from Scilab workspace. In this example we have defined in Scilab the matrices:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -2 & -4 & -3 \end{bmatrix}, \; B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{4}$$

for a constant input $u = 1$ we get the trajectories of Figure 13 for the three state variables.

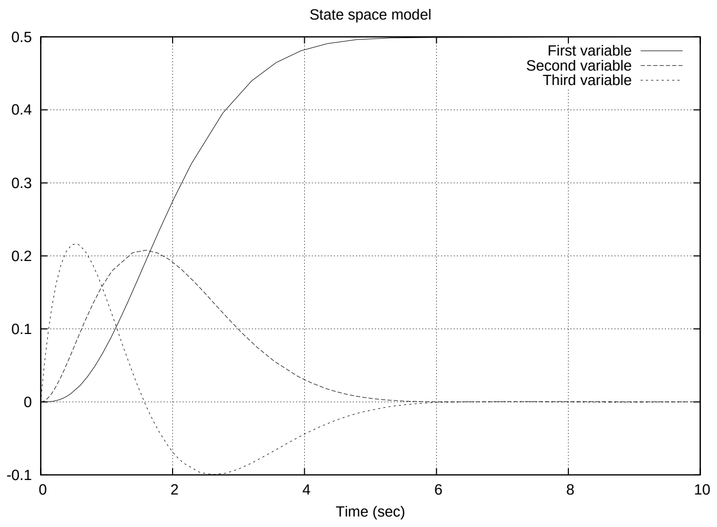

Figure 13: State variables

This example has a low dimension and performance comparisons do not show differences between vectorial and non–vectorial approaches. However, the modeling tasks are greatly simplified with the usage of VECDEVS. Here, the user only has to provide the matrices A and B to simulate any state space model, while in a non–vectorial approach, a complex and large block diagram must be constructed for each model.

6.3 Spiking Neural Network

This model, attempts to reproduce a model[31], consisting in a Spiking Neural Network (SNN) composed by 1000 leaky integrate-and-fire neurons.

Each neuron follows the dynamics:

$$\frac{dv}{dt} = (V_{rest} - V) - g_{ex}(E_{ex} - V_{rest}) + g_{inh}(E_{inh} - V_{rest}) \tag{5a}$$

$$\tau_{ex}\frac{dg_{ex}}{dt} = -g_{ex} \tag{5b}$$

$$\tau_{inh}\frac{dg_{inh}}{dt} = -g_{inh} \tag{5c}$$

where $V(t)$ is the membrane potential, g_{ex} and g_{inh} are the excitatory and inhibitory synaptic conductances, respectively, and the remaining variables are the neuron parameters.

The neuron fires when it cross the threshold $V_{th} = -50mV$ and resets the membrane potential to $V_{rest} = -60mv$. When a neuron fires, an impulse is transmitted to the neurons to which it is connected. That impulse, instantaneously changes the values of τ_{ex} or τ_{inh} depending on the excitatory or inhibitory feature the firing neuron.

In this model, the connections were defined randomly, so that every neuron is connected to about 20 neurons (i.e., a 2% probability of connection). The excitatory or inhibitory feature of each neuron was also randomly chosen, so that about 800 neurons are of excitatory type and about 200 neurons are of inhibitory type.

We modeled the SNN in PowerDEVS using Vectorial DEVS blocks, as can be seen in Figure 14. The blocks *map_exc* and *map_inh* are instances of Index Map and they represent the neuron interconnections. The blocks *excitatory* and *inhibitory* are instances of Index Selector and they split the network into excitatory and inhibitory neurons.

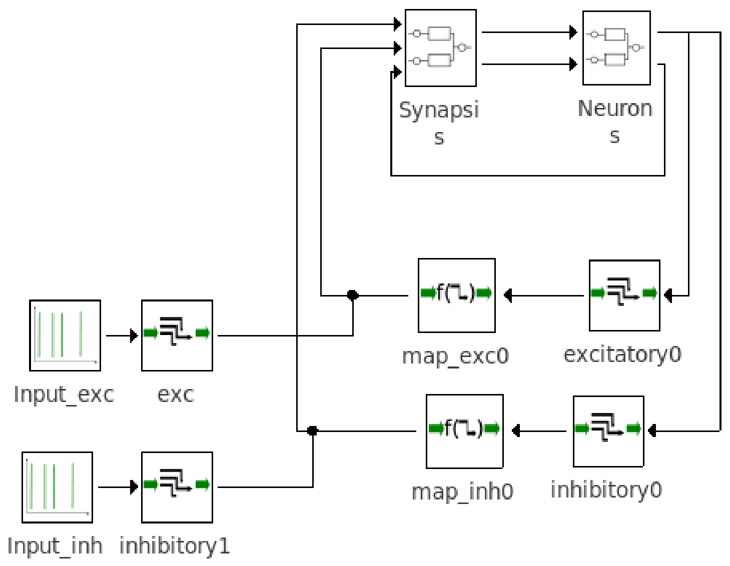

Figure 14: Model of a Spiking Neural Network

The *input_exc* and *output_inh* blocks are Poisson vectorial sources. The coupled model *Neurons* contains a vectorial model for 1000 instances of Eq.(5a), as shown in Fig 15. Similarly, the coupled model *Synapsis* contains a vectorial model for 1000 instances of Eqs.5b–5c, as shown in Fig 16.

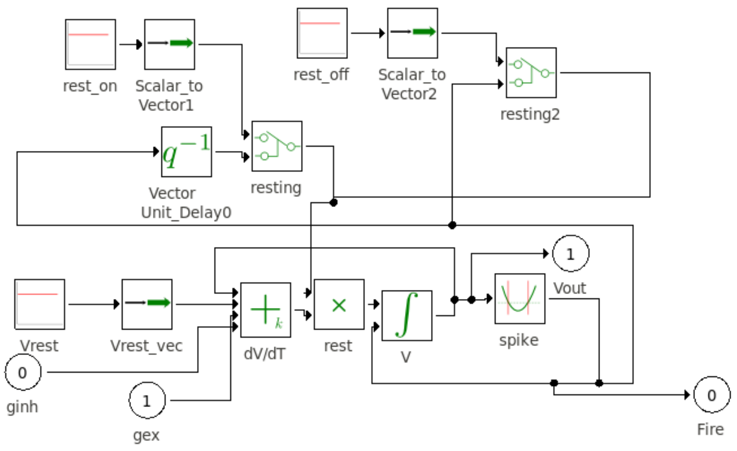

Figure 15: Model of the neurons

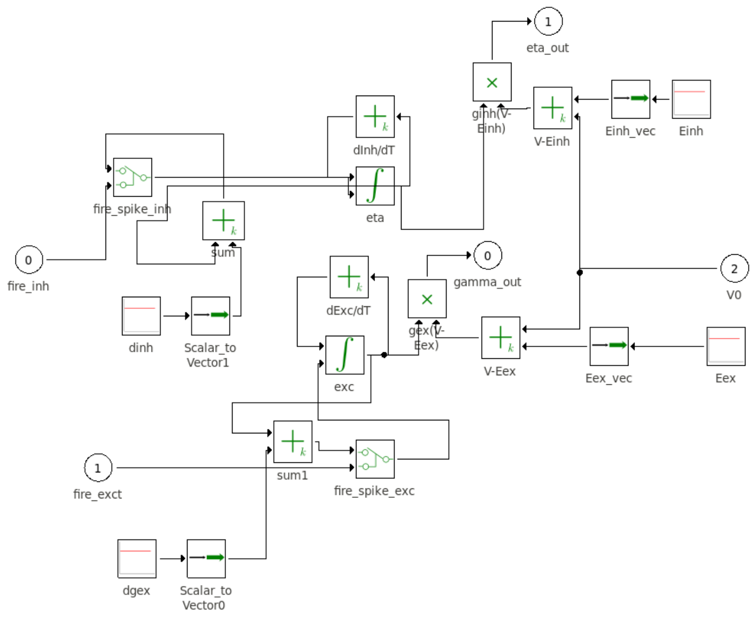

Figure 16: Model of the synapses

The model was simulated using the QSS3 method, obtaining similar results to those of the article[31]. Figure 17 shows the simulated membrane potential at one randomly chosen neuron.

In this case, building a non vectorial model is almost impossible. Each neuron is described by about 25 interconnected blocks, and the whole model contains 1000 randomly interconnected neurons.

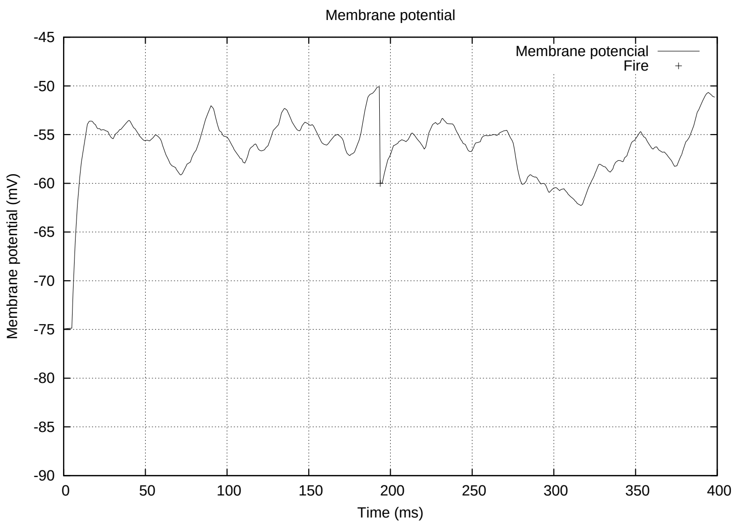

Figure 17: Membrane potential of a randomly chosen neuron

Thus, building the corresponding block diagram may take several days of work. Thus, in this case we were unable to compare VECDEVS results against non vectorial ones.

A system similar to this one was used in[4]. That time around, as VECDEVS was not available, the standard DEVS model was built programming an ad–hoc automatic model builder, in a similar manner to what is described in[7]. The process of programming the model builder took a couple of days and the compilation of the resulting large scale DEVS model took several minutes. In contrast, the usage of VECDEVS allows to build the model immediately and the compilation of the resulting large scale model is almost instantaneous.

6.4 LC Transmission Line

The following system of equations represents a lumped model of a lossless transmission line, formed by N sections of LC (inductor and capacitor) circuits:

$$\begin{aligned}\dot{v}_j &= \frac{i_j - i_{j+1}}{C}\\ \dot{i}_j &= \frac{v_{j-1} - v_j}{L}\end{aligned}$$

for $i = 1, \ldots, N$.

We consider also a input step:

$$v_0(t) = \begin{cases} 1 & \text{if } t < 1 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

This system is easily represented using Vectorial DEVS in PowerDEVS as we have seen in Figure 6.

Taking $N = 600$ and using $L = C = 1$ and null initial conditions we get a marginally stable system of order 1200.

We simulated the system with QSS3. Figure 18 shows the voltage at the end of the transmission line $v_{600}(t)$.

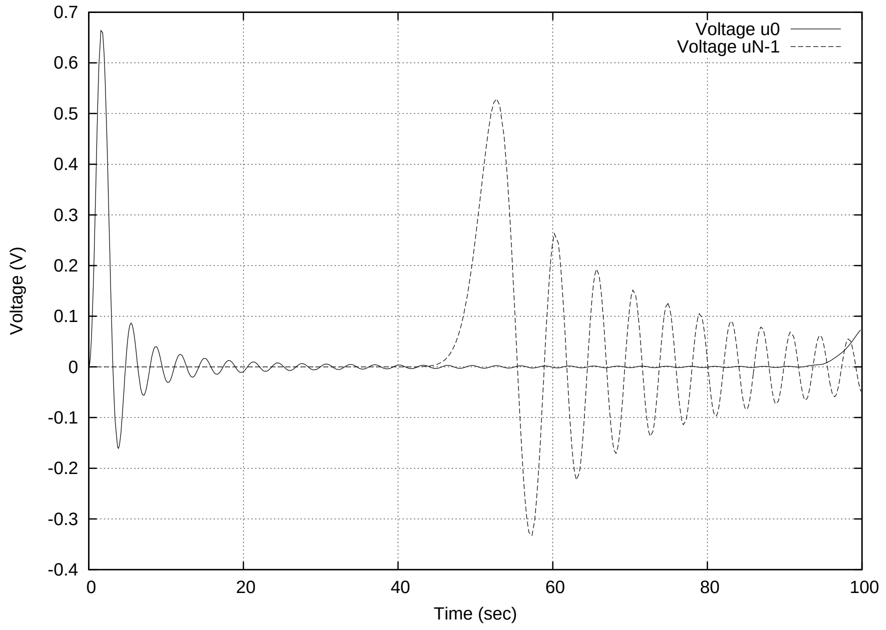

Figure 18: Simulation Result – Voltage at the end of the line.

Table 2 shows a comparison of compilation and simulation time between the VECDEVS version and the non vectorial hand-made version of this model for a final simulation time of $TF = 1200$ sec. In this case, the simulation times are similar between both models. However, as in the first example, the compilation times are constant in Vectorial DEVS while it grows noticeably in the non vectorial case. As before, the file size of the VECDEVS description is constant (9 KB) while the file size of the hand-made version for $N = 50, 500, 1000$ is 13 MB, 297 MB and 1.2 GB respectively.

Table 2: Compilation and Simulation times (in seconds) vs. model dimension (N) for the LC Line.

	50		500		1000	
	Comp.	Sim.	Comp.	Sim.	Comp.	Sim.
VECDEVS	0.2	2.52	0.2	17.9	0.2	28.6
Hand-Made	1.82	4.31	10.7	18.3	24.65	27.9

We have used the automatic partitioning algorithm for this example and simulated the resulting model with the parallelization techniques presented by the authors[3]. The model is similar to the one of Fig. 7 but split into 12 sub-models. Here we choose 12 sub-models because the target platform for simulation is an Intel i7-970 processor running at 3.20GHz with 12 cores. These sub-models were simulated in parallel with the ASRTS technique were we found an acceleration 5.5 times over the the sequential simulation. The application of the ASRTS would have not been possible without the partitioning algorithm here presented.

6.5 Power control of an air conditioner population

Here, we recall the motivating example of Section 3.1, consisting in a large population of air conditioners. The corresponding PowerDEVS model of Fig. 4 was modified adding a *vector sum* block that calculates the sum of the event values received with different indexes. This *vector sum* does a reduction operation on the events it receives. It has p vectorial inports and one scalar outport and computes the sum of all its input signals. Notice that an vectorial event with value (x, i) coming from the first inport represents a different input than a vector with the same value but coming from the second inport. Formally the block computes:

$$P(t) = \sum_{i=1}^{N} m_i(t) \cdot P_i$$

which represents the power consumption of the total group of AC.

A global control system regulates it, so it follows a desired power profile $P_r(t)$. In order to accomplish this, a Proportional Integral (PI) control law is used to compute the reference temperature:

$$\theta_r(t) = K_P \cdot [P_r(t) - P(t)] + K_I \cdot \int_{\tau=0}^{t} [P_r(\tau) - P(\tau)]\mathrm{d}\tau$$

where K_P and K_I are the parameters of the PI controller.

In this example we chose $N = 2400$ air conditioner units, and the set of parameters given in cited article[27]. The resulting model can be seen in Figure 19, where we see the addition of the *vector sum* in the AC model.

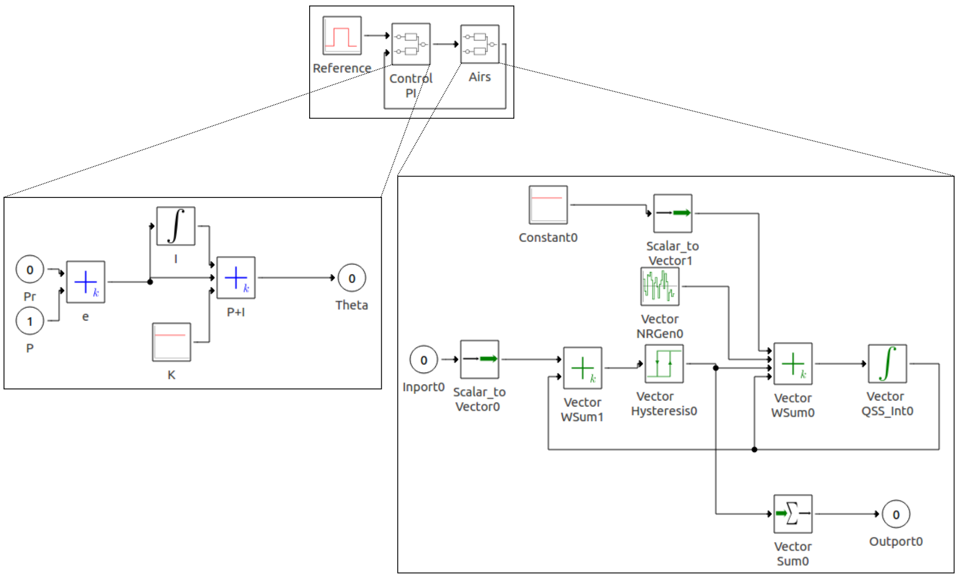

Figure 19: Model of the AC population with a PI Control

The results of the simulation using QSS3 can be seen in Fig. 20.

Table 3 shows a comparison of compilation and simulation times between the VECDEVS and the non–vectorial versions of this model. As before, the simulation times are similar. Also, the compilation

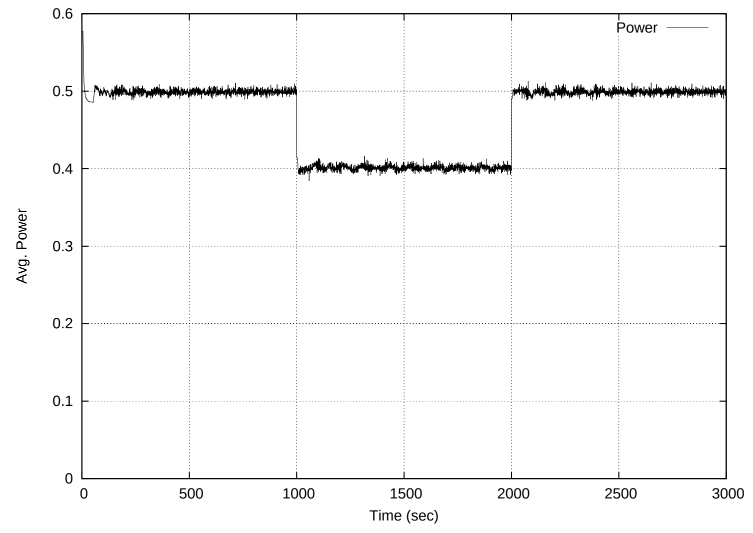

Figure 20: Power consumption in the AC system

time of Vectorial DEVS remains constant while it grows with N in the non-vectorial case. The file size of the VECDEVS description is also constant (30 KB) while the size of the hand-made versions for $N = 100, 500, 1000$ is 18 MB, 348 MB and 1.6 GB respectively.

Table 3: Compilation and Simulation times (in seconds) vs. model dimension (N) for the Air Conditioners models.

	100		500		1000	
	Comp.	Sim.	Comp.	Sim.	Comp.	Sim.
VECDEVS	0.212	0.7	0.212	5.84	0.212	12.2
Hand-Made	2.73	0.79	15.63	5.32	40.19	14.2

Here we again used the automatic partitioning algorithm to split the model of Fig. 4 into 11 sub-models. The resulting model has 11 *Airs* blocks with out any connections. We then added manually the *Control PI* (the 12-th block) and its connection. This is done because the algorithm for automatic partitioning does not take into account blocks like the *vector sum*. This block is neither pure vectorial, nor does event routing or acts as an interface. The kind of computation it performs (a reduction) needs special treatment in the algorithm that is not yet included.

The split model can bee seen in Fig. 21. There we see that each AC sub-model sends its power consumption to the control and the control sends back the reference temperature to all the AC sub-models.

Once split, we simulated the model again using ARSTS on the same platform than before and found acceleration of 9 times over the sequential simulation. Again, the use of the ASRTS technique would have been impossible without the split model.

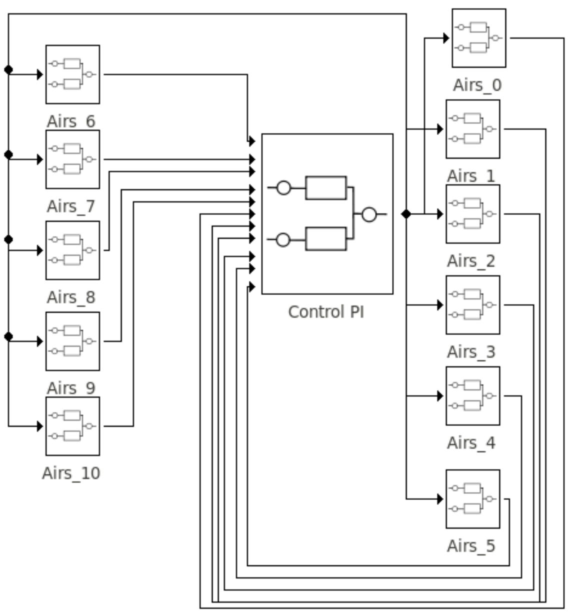

Figure 21: Split model of the AC population with a PI Control

7 Conclusions and Future Work

In this article we have introduced an extension to the DEVS formalism called Vectorial DEVS for modeling and parallel simulation of large scale continuous and hybrid systems. We have also developed a set of interface DEVS blocks that allow to interconnect VECDEVS models in order to build systems with complex structures. Additionally, we have proposed an algorithm that automatically splits Vectorial DEVS models for parallelization purposes.

The new formalism was implemented in PowerDEVS, a graphical tool for DEVS modeling and simulation, developing a new library and developing some application examples. The automatic partitioning algorithm was also implemented in PowerDEVS.

In the examples analyzed, the usage of VECDEVS shows several advantages, reducing significantly the time for building large scale systems and the compilation time of the resulting models, while preserving almost unchanged the execution time of the simulations.

Developing non vectorial models of large scale systems requires at least replicating several blocks and carefully interconnecting them, tasks which are time consuming and error prone. Also, the modeling GUIs become slower by the burden of having to deal with these huge model descriptions. All these issues are elegantly solved with the usage of Vectorial DEVS.

For future work, we are considering exploiting some features of Vectorial DEVS models to develop a dynamic partitioning algorithm, where the vector size of the sub–models at different processors is periodically changed in order to keep a balanced workload. Taking into account the way VECDEVS is defined, the implementation of such algorithm is not very difficult as a new partitioning only affects the parameters related to the indexes of event routing and interfacing blocks and the size of the Vectorial DEVS models.

As mentioned before, Dynamical DEVS structures[28] can be implemented by changing the parameters of the event routing blocks during the simulations. This opens several lines of work in structure optimization applications[32,33].

Simulation–Based Optimizations can be implemented in a straightforward manner using VECDEVS. One can set different parameters in every index of the VECDEVS model and process the output of each sub-simulation to find optimal parameters. This encourages us to investigate optimization applications[34]. In the same way stochastic analysis[35,36] can be made by setting different conditions to each sub-model.

The extension of the partitioning algorithm is also promising. For example including support for blocks like the *vector sum* would broaden the application of the Vectorial DEVS formalism.

The application of the partitioning algorithm with other methods of parallel or distributed simulations is also an interesting problem to study. Once the model is split into several sub-models, the usage of specific communication protocols[37] may extend the usage of techniques like ASRTS for distributed simulation. It is also worth analyzing the partitioning algorithm presented here in a formal framework[38] comparing different model partitioning techniques.

So far, Vectorial DEVS considers only one dimensional arrays of models. Another future line is to develop a multi–indexed version of the formalism, for example, having two indexes in the events one could model matrix-type models.

PowerDEVS is an Open Source project developed at the National University of Rosario (UNR) and the French-Argentine International Center for Information and Systems Sciences (CIFASIS) . The software with all the features and models described in this article can be downloaded from `http://sourceforge.net/projects/powerdevs/`.

REFERENCES

1. Bernard Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation - Second Edition.* Academic Press, 2000.

2. François Cellier and Ernesto Kofman. *Continuous System Simulation.* Springer, New York, 2006.

3. Federico Bergero, Ernesto Kofman, and François Cellier. A novel parallelization technique for devs simulation of continuous and hybrid systems. *SIMULATION*, 89(6):663–683, 2013.

4. Guillermo L. Grinblat, Hernán Ahumada, and Ernesto Kofman. Quantized state simulation of spiking neural networks. *Simulation*, 88(3):299–313, 2012.

5. Gustavo Migoni, Mario Bortolotto, Ernesto Kofman, and François E. Cellier. Linearly implicit quantization-based integration methods for stiff ordinary differential equations. *Simulation Modelling Practice and Theory*, 35:118 – 136, 2013.

6. G. Wainer and N. Giambiasi. Application of the Cell-DEVS paradigm for cell spaces modelling and simulation. *Simulation*, 76(1):22–39, 2001.

7. T. G. Kim and B.P. Ziegler. Knowledge-based environment for investigating multicomputer architectures. *Inf. Softw. Technol.*, 31(10):512–520, December 1989.

8. Jean-Baptiste Filippi, Frédéric Morandini, Jacques Henri Balbi, and David RC Hill. Discrete event front-tracking simulation of a physical fire-spread model. *SIMULATION*, 86(10):629–646, 2010.

9. Federico Bergero and Ernesto Kofman. PowerDEVS: A Tool for Hybrid System Modeling and Real Time Simulation. *Simulation*, 87:113–132, January 2011.

10. B. Zeigler. *Theory of Modeling and Simulation.* John Wiley & Sons, New York, 1976.

11. E. Kofman and S. Junco. Quantized State Systems. A DEVS Approach for Continuous System Simulation. *Transactions of SCS*, 18(3):123–132, 2001.

12. E. Kofman. A Second Order Approximation for DEVS Simulation of Continuous Systems. *Simulation: Transactions of the Society for Modeling and Simulation International*, 78(2):76–89, 2002.

13. E. Kofman. A Third Order Discrete Event Simulation Method for Continuous System Simulation. *Latin American Applied Research*, 36(2):101–108, 2006.

14. G. Migoni and E. Kofman. Linearly Implicit Discrete Event Methods for Stiff ODEs. *Latin American Applied Research*, 39(3):245–254, 2009.

15. G. Migoni, E. Kofman, and F. Cellier. Quantization-Based New Integration Methods for Stiff ODEs. *Simulation: Transactions of the Society for Modeling and Simulation International*, 88(4):387–407, 2012.

16. James J. Nutaro. *Building Software for Simulation: Theory and Algorithms, with Applications in C++.* Wiley Publishing, 2010.

17. Alex Chung Hen Chow and Bernard Zeigler. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation*, WSC '94, pages 716–722, San Diego, CA, USA, 1994. Society for Computer Simulation International.

18. Monageng Kgwadi, Hui Shang, and Gabriel Wainer. Definition of dynamic DEVS models: Dynamic Structure CD++. In *Proceedings of the 2008 Spring simulation multiconference*, SpringSim '08, pages 10:1–10:4, San Diego, CA, USA, 2008. Society for Computer Simulation International.

19. Gabriel Wainer. CD++: a toolkit to develop DEVS models. *Software: Practice and Experience*, 32(13):1261–1306, 2002.

20. Gabriel Wainer. *Discrete-Event Modeling and Simulation: a Practitioner's approach.* CRC Press. Taylor and Francis, 2009.

21. David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, June 1987.

22. Peter Fritzson and Peter Bunus. Modelica-A General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation. In *Annual Simulation Symposium*, pages 365–380, 2002.

23. Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. In *ECOOP*, pages 67–90, 1998.

24. M. Otter, H. Elmqvist, and F. Cellier. Modeling of multibody systems with the object-oriented modeling language Dymola. *Nonlinear Dynamics*, 9:91–112, 1996. 10.1007/BF01833295.

25. Peter Fritzson, Peter Aronsson, Hakan Lundvall, Kaj Nystrom, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Development Environment. *Proceedings of the 46th Conference on Simulation and Modeling (SIMS'05)*, pages 83–90, 2005.

26. Harold Klee. *Simulation of Dynamic Systems with MATLAB and Simulink*. CRC, 2007.

27. Cristian Perfumo, Ernesto Kofman, Julio Braslavsky, and John K. Ward. Load management: Model-based control of aggregate power for populations of thermostatically controlled loads. *Energy Conversion and Management*, 55:36–48, 2012.

28. Fernando J. Barros. A Formal Definition of Dynamic Structure Hybrid Simulation Model. In *PADS*, page 149, 2008.

29. PowerDEVS site at SourceForge - http://sourceforge.net/projects/powerdevs/.

30. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

31. Tim P. Vogels and L. F. Abbott. Signal Propagation and Logic Gating in Networks of Integrate-and-Fire Neurons. *The Journal of Neuroscience*, 25(46):10786–10795, 2005.

32. Olaf Hagendorf, Thorsten Pawletta, and Roland Larek. An approach to simulation-based parameter and structure optimization of MATLAB/Simulink models using evolutionary algorithms. *SIMULATION*, 89(9):1115–1127, 2013.

33. Juan-Ignacio Latorre and Emilio Jiménez. Simulation-based optimization of discrete event systems with alternative structural configurations using distributed computation and the Petri net paradigm. *SIMULATION*, 89(11):1310–1334, 2013.

34. Paulo Salem da Silva and Ana Cristina Vieira de Melo. On-the-fly verification of discrete event simulations by means of simulation purposes: Extended version. *SIMULATION*, 2013.

35. Rodrigo Castro, Ernesto Kofman, and Gabriel Wainer. A Formal Framework for Stochastic Discrete Event System Specification Modeling and Simulation. *SIMULATION*, 86(10):587–611, 2010.

36. Ali Khalili, Mohammad Abdollahi Azgomi, and Amir Jalaly Bidgoly. SimGine: A simulation engine for stochastic discrete-event systems based on SDES description. *SIMULATION*, 2013.

37. Dylan Pfeifer, Jonathan Valvano, and Andreas Gerstlauer. Simconnect and simtalk for distributed cyber-physical system simulation. *SIMULATION*, 2013.

38. Adedoyin Adegoke, Hamidou Togo, and Mamadou K Traoré. A unifying framework for specifying DEVS parallel and distributed simulation architectures. *SIMULATION*, 89(11):1293–1309, 2013.